Level Up Your Code With Game Progamming Simplified-Chinese

介绍设计模式

当在Unity中工作时，你不需要重新发明轮子。很可能有人已经为你发明了一个。

对于你遇到的每一个软件设计问题，都有成千上万的开发者在你之前遇到过。虽然你不能总是直接向他们寻求建议，但你可以通过设计模式来学习他们的决策。

设计模式是解决软件工程中常见问题的通用解决方案。这些并不是你可以复制粘贴到代码中的完成的解决方案，但你可以把设计模式看作是工具箱中的额外工具。有些比其他的更明显。

这份指南汇集了Unity开发中的知名设计模式。本指南中的示例已经简化，技术术语也已经减少，使得它们更易于接触，尽管你在开始使用它们之前应该具备一些C#基础知识。

如果你对设计模式还很新，或者需要快速复习，本指南还提供了一些你可以在游戏开发中应用它们的常见场景。对于那些从其他面向对象语言（如Java、C++等）转到C#的人，这些样例将展示如何将模式特别地适应到Unity中。

归根结底，设计模式只是一些想法。它们并不适用于所有情况。但是，如果正确使用，它们可以帮助你构建规模更大的应用程序。将它们整合到你的项目中，可以提高代码的可读性，使你的代码库更加清晰。当你对模式有了更多的经验，你会认识到它们何时可以加快你的开发过程。

然后，你就可以停止重新发明轮子，开始着手于一些新的事情。

贡献者

本指南由Wilmer Lin撰写，他是一位有超过15年电影和电视行业经验的3D和视觉效果艺术家，现在作为一名独立游戏开发者和教育者工作。高级技术内容营销经理Thomas Krogh-Jacobsen和高级Unity工程师Peter Andreasen和Scott Bilas也作出了重要贡献。

>**使用本指南和KISS原则** > >本指南旨在向你展示关于思考和组织你的代码的新方法。本指南中突出显示的几种软件设计模式都已适应于Unity开发。 >包含的[示例项目](https://github.com/Unity-Technologies/game-programming-patterns-demo/)显示了一些上下文中的代码。使用相应的场景来探索这些设计模式及其基础原则。 >然而，在审查这些例子时，请记住并没有一种全面的“正确方式”来处理问题。示例代码只是众多解决方案中的一种。 >在有疑问的时候，通过KISS原则来过滤本指南中的所有内容：“保持简单、愚蠢。”只有在必要的时候才增加复杂性. >每种设计模式都带有权衡，无论是意味着需要维护额外的结构，还是在开始时需要更多的设置。在实施之前，决定是否利益能够证明额外的工作是值得的。 >如果你不确定某个模式是否适用于你的特定问题，你可能最好等待一个情况，让它感觉更自然地适合。不要因为一个模式对你来说是新的或者是新颖的就使用它；只有在你需要的时候才使用它。 >然后，设计模式将发挥其预期的目的：帮助你开发更好的软件。 > >让我们开始吧。

SOLID原则

在深入设计模式本身之前，让我们看一下一些影响它们工作方式的设计原则。
SOLID是五个软件设计的核心基础的助记符：

- 单一职责
- 开闭原则
- 里氏替换
- 接口隔离
- 依赖反转

让我们分别检查每个概念，看看它们如何帮助你让你的代码更易于理解、更灵活和更易于维护。

单一职责原则

一个类应该只有一个变化的理由，也就是其单一的职责。

最重要的SOLID原则是单一职责原则(SRP)，它规定每个模块、类或函数只负责一件事，并且只封装那部分逻辑。

你应该把你的项目从许多小的项目中组合起来，而不是构建庞大的类。较短的类和方法更容易解释、理解和实现。

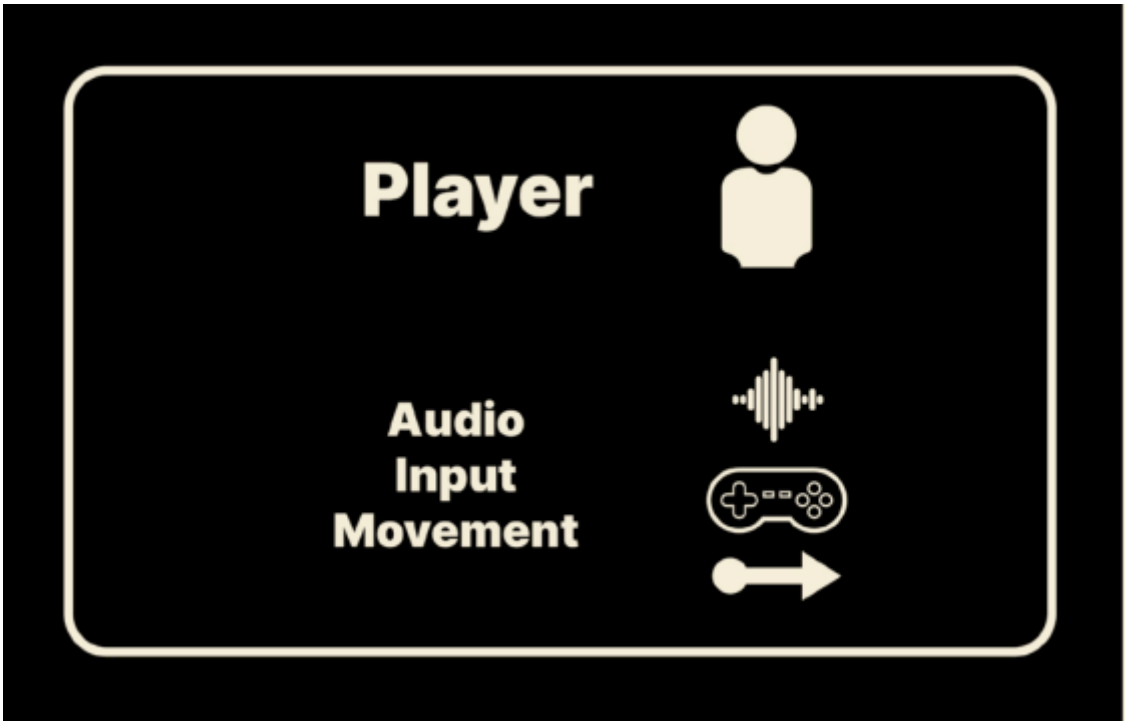
如果你在Unity中工作了一段时间，你可能已经熟悉了这个概念。当你创建一个GameObject时，它会包含各种小的组件。例如，它可能会带有：

- 一个MeshFilter，存储对3D模型的引用
- 一个Renderer，控制模型表面在屏幕上的显示方式
- 一个Transform组件，存储缩放、旋转和位置
- 如果需要与物理模拟交互，还会有一个Rigidbody

每个组件只做一件事，并且做得很好。你从GameObject构建整个场景。它们的组件之间的互动是游戏可能性的体现。

你将以同样的方式构建你的脚本组件。设计它们使得每一个都可以清楚地被理解。然后让它们协同工作，产生复杂的行为。

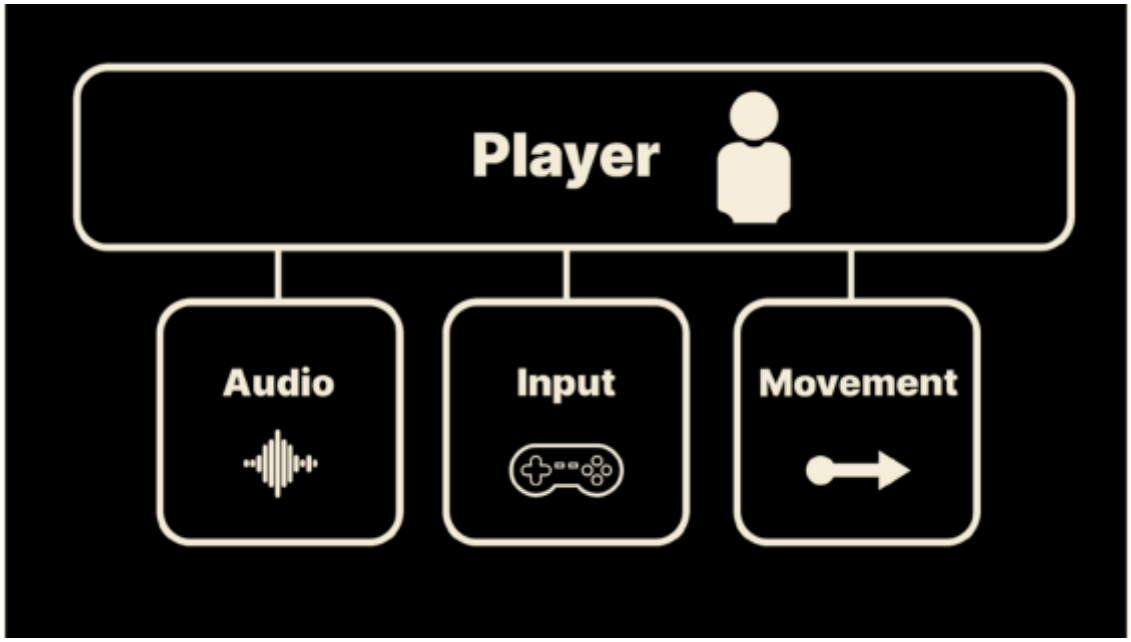
如果你忽视了单一职责，你可能会创建一个自定义组件，如下所示：




```
public class UnrefactoredPlayer : MonoBehaviour
{
    [SerializeField] private string inputAxisName;
    [SerializeField] private float positionMultiplier;
    private float yPosition;
    private AudioSource bounceSfx;

    private void Start()
    {
```

```
        bounceSfx = GetComponent<AudioSource>();
    }

    private void Update()
    {
        float delta = Input.GetAxis(inputAxisName) * Time.deltaTime;
        yPosition = Mathf.Clamp(yPosition + delta, -1, 1);
        transform.position = new Vector3(transform.position.x, yPosition *
positionMultiplier, transform.position.z);
    }

    private void OnTriggerEnter(Collider other)
    {
        bounceSfx.Play();
    }
}
```

这个未重构的Player类的职责混杂。它在玩家与某物碰撞时播放声音，管理输入，并处理移动。即使这个类目前相对较短，但随着你的项目的发展，它将变得难以维护。考虑将Player类拆分为更小的类。




```
[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
{
    // ...
}
```

Player脚本仍然可以管理其他的脚本组件，但每个类只做一件事。这种设计使得修改代码变得更容易，特别是随着你的项目需求随时间的变化。另一方面，你需要用一定的常识来平衡单一职责原则。不要过度简化到创建只有一个方法的类的极端。

另一方面，你需要在单一职责原则和常识之间找到平衡。不要过度简化到只创建一个方法的类的极端。

在使用单一职责原则时，要牢记以下目标：

- 可读性：短的类更容易阅读。没有硬性规定，但许多开发者设置了200-300行的限制。自己或作为一个团队确定什么构成“短”。当超过这个阈值时，决定是否可以将其重构成更小的部分。
- 可扩展性：你可以更容易地从小的类中继承。修改或替换它们，而不用担心破坏意外的功能。

- 可重用性：设计你的类使其小而模块化，这样你可以在游戏的其他部分重用它们。

在重构时，考虑重新排列代码将如何改善自己或其他团队成员的生活质量。一开始的一些额外努力可以在后来为你节省很多麻烦。

> **简单并不等于容易**
> 在软件设计中，我们经常讨论简单性，它是可靠性的前提。你的软件设计能够处理生产中的变化吗？随着时间的推移，你能扩展和维护你的应用程序吗？
>
> 本指南中介绍的许多设计模式和原则都帮助你实现简单性。这样做使你的代码更具可扩展性、灵活性和可读性。然而，它们需要一些额外的工作和规划。"简单"并不等于"容易"。
>
> 尽管你可以在不使用模式的情况下创建相同的功能（并且通常更快），但快速和容易不一定导致简单。使某事简单意味着使其更加集中。设计它来做一件事，不要用其他任务过度复杂化它。
>
> 请查看Rich Hickey的讲座《简单即易》，以了解简单性如何帮助你构建更好的软件。

开闭原则

在SOLID设计中的开闭原则（OCP）规定类必须对扩展开放，但对修改封闭。组织你的类，使得你可以在不修改原始代码的情况下创建新的行为。

这方面的经典例子是计算形状的面积。你可以创建一个名为AreaCalculator的类，其中包含返回矩形和圆形面积的方法。为了计算面积，Rectangle类有Width和Height。

而Circle只需要Radius和pi的值。

```
public class AreaCalculator
{
    public float GetRectangleArea(Rectangle rectangle)
    {
        return rectangle.width * rectangle.height;
    }

    public float GetCircleArea(Circle circle)
    {
        return circle.radius * circle.radius * Mathf.PI;
    }
}
```

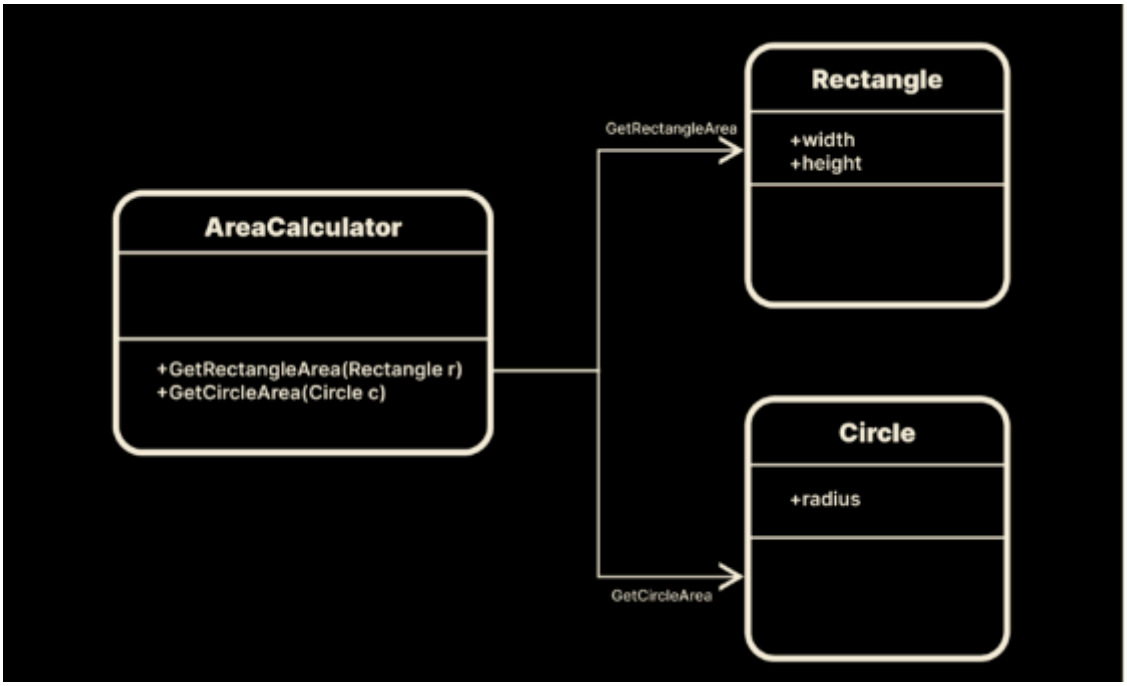
```
public class Rectangle
{
    public float width;
    public float height;
}

public class Circle
{
    public float radius;
}
```

这种方式运行得很好，但是如果你想在你的AreaCalculator中添加更多的形状，你需要为每一个新的形状创建一个新的方法。

假设你以后想要传递一个五边形或者八边形呢？如果你需要20个更多的形状呢？AreaCalculator类将会迅速失控。

你可以创建一个名为Shape的基类，并创建一个方法来处理形状。然而，这样做将需要在逻辑内部有多个if语句来处理每一种形状。这样的扩展性不好。

你希望在不修改原始代码（AreaCalculator的内部）的情况下，打开程序进行扩展（使用新的形状的能力）。虽然它是功能性的，但当前的AreaCalculator违反了开闭原则。



相反，考虑定义一个抽象的Shape类：

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

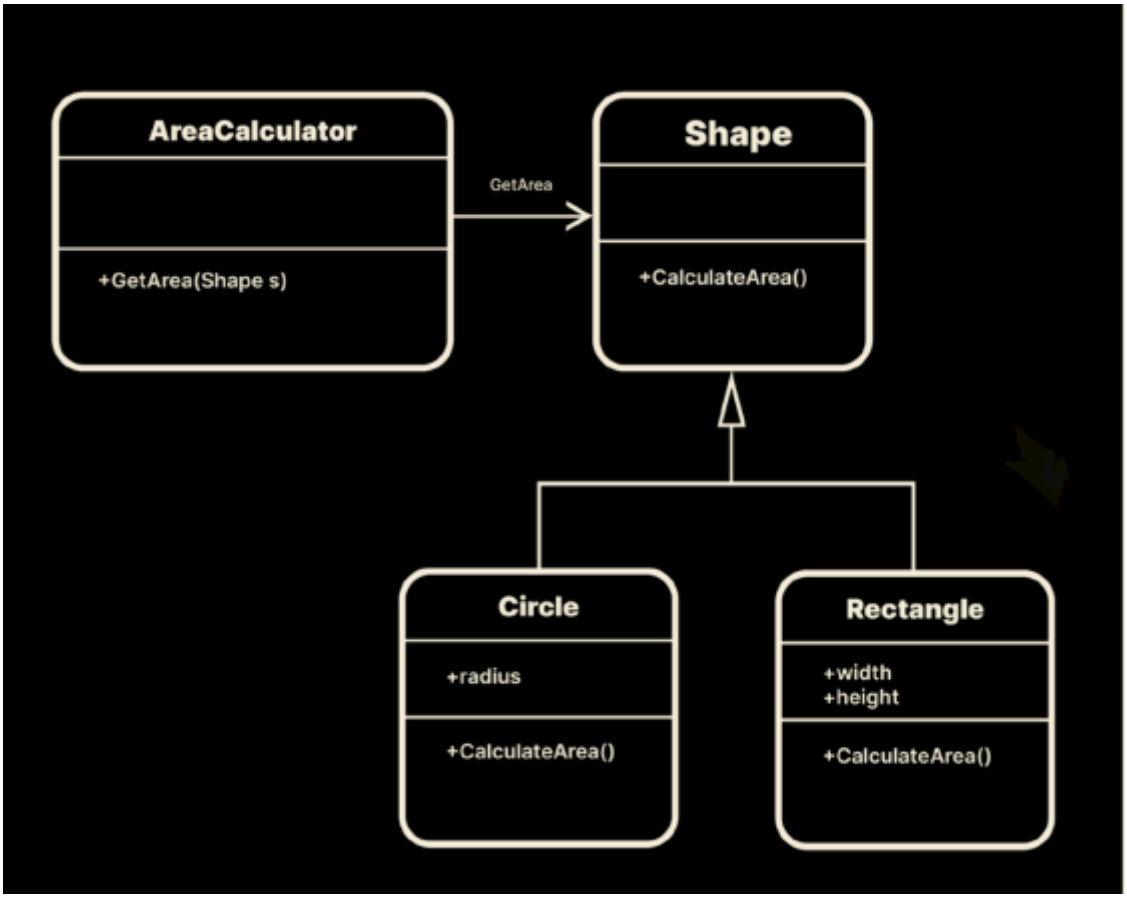
这包括一个名为CalculateArea的抽象方法。如果你让Rectangle和Circle从Shape继承，每个形状都可以计算自己的面积，并返回以下结果：

```
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;
    }
}
```

AreaCalculator可以简化为这样：

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

修订后的AreaCalculator类现在可以获取任何正确实现抽象Shape类的形状的面积。你可以扩展AreaCalculator的功能，而无需改变任何原始源代码。



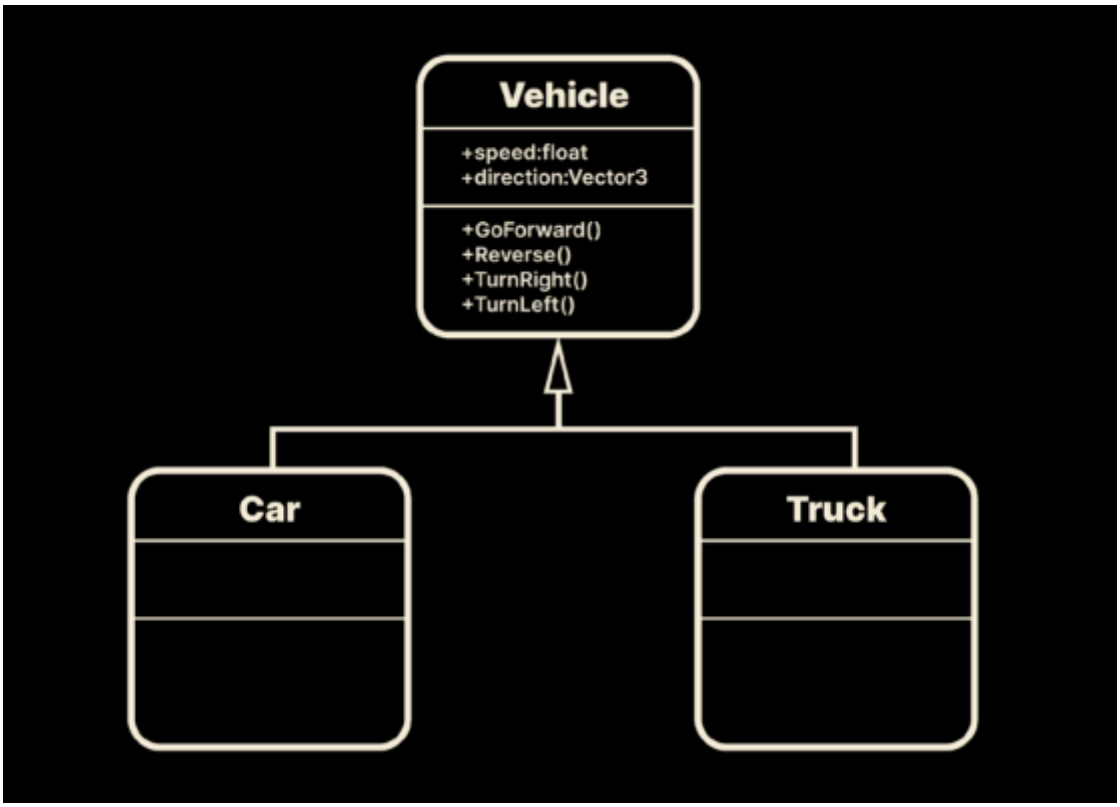
每当你需要一个新的多边形，只需定义一个从Shape继承的新类。每个子类的形状然后覆盖CalculateArea方法来返回正确的面积。
这种新的设计使得调试更容易。如果新形状引入了错误，你不必重新访问AreaCalculator。旧代码保持不变，所以你只需要检查新代码中是否有任何逻辑错误。
在Unity中创建新的类时，利用接口和抽象。这有助于避免在你的逻辑中出现难以后期扩展的繁琐的switch或if语句。一旦你习惯了设置你的类来尊重OCP，长期添加新的代码变得更简单。

里氏替换原则

里氏替换原则（LSP）规定，派生类必须能够替换它们的基类。面向对象编程中的继承允许你通过子类增加功能。然而，如果你不小心，这可能导致不必要的复杂性。

里氏替换原则，SOLID的第三个支柱，告诉你如何应用继承，使你的子类更加健壮和灵活。

想象一下，你的游戏需要一个叫做Vehicle的类。这将是你为应用创建的车辆子类的基类。例如，你可能需要一辆汽车或卡车。



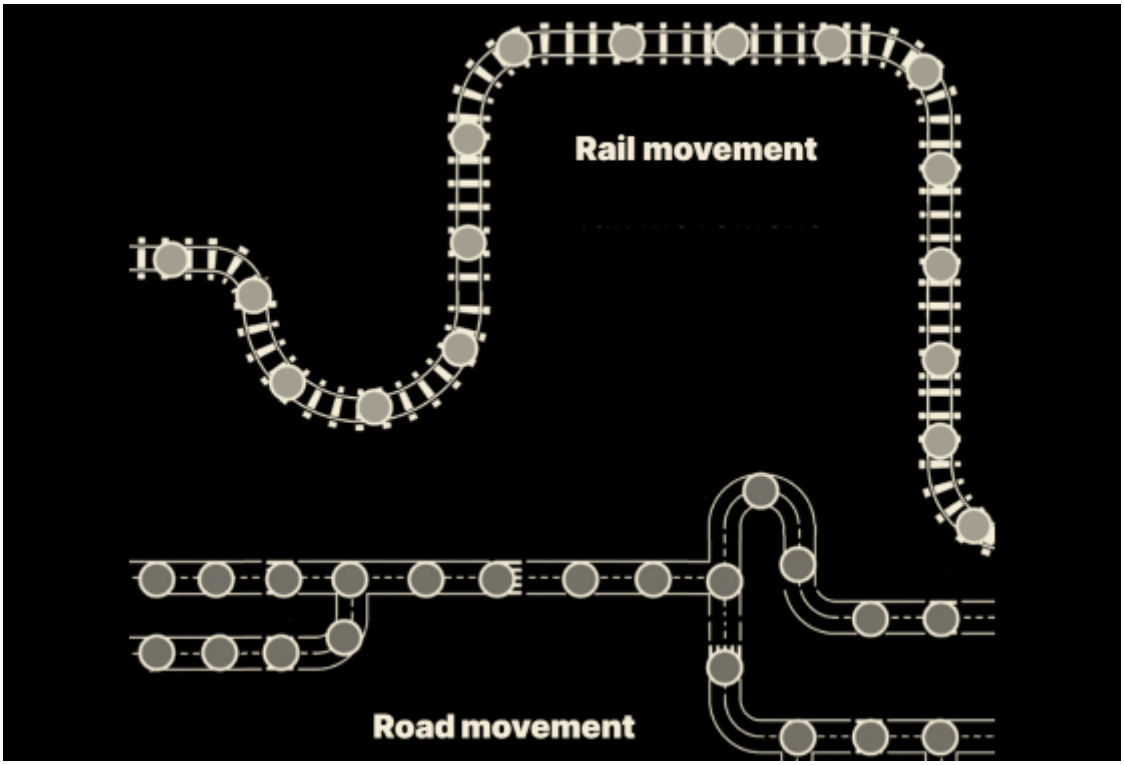
在你可以使用基类（Vehicle）的任何地方，你应该能够使用像Car或Truck这样的子类，而不会破坏应用程序。

你的Vehicle类可能看起来像这样：

```
public class Vehicle
{
    public float speed = 100;
    public Vector3 direction;

    public void GoForward()
    {
        // ...
    }

    public void Reverse()
    {
        // ...
    }

    public void TurnRight()
    {
        // ...
```

```
    }

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

假设你正在构建一个回合制的游戏，你在棋盘上移动车辆。

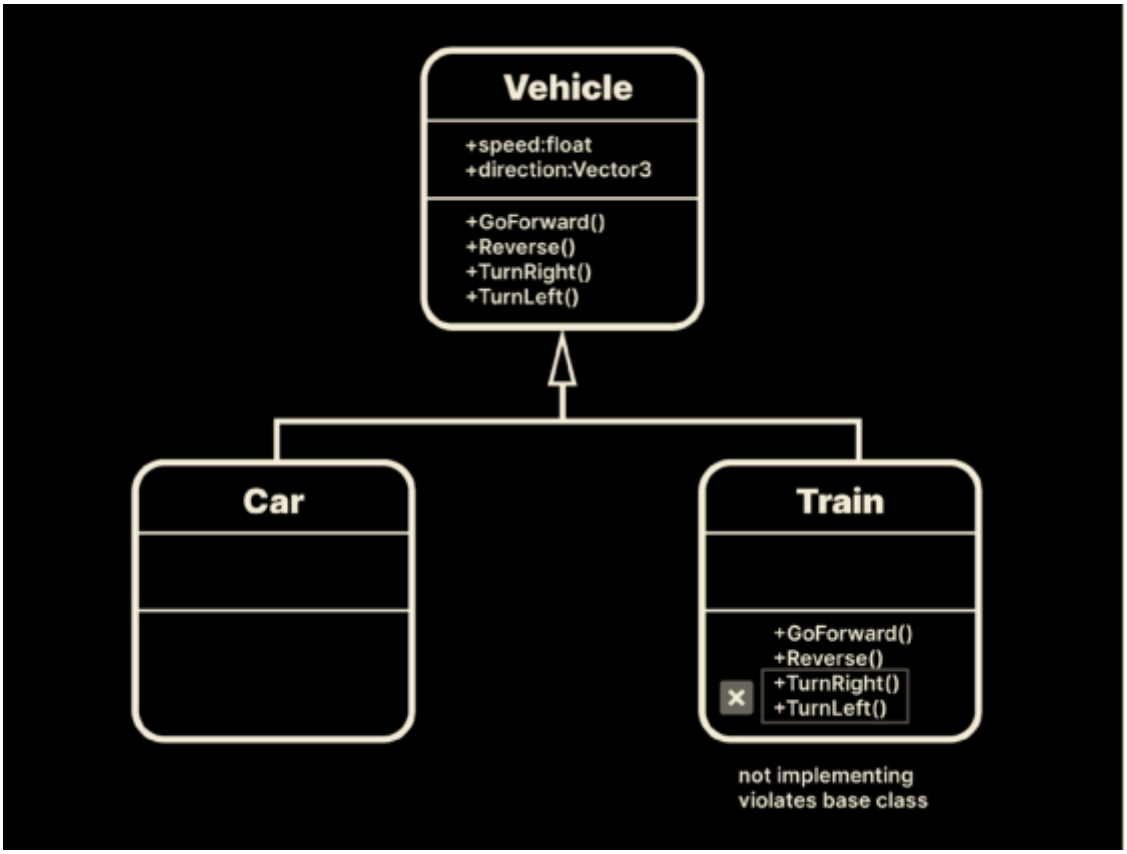


你可以有另一个叫做Navigator的类，来沿着预设的路径驾驶车辆：

```
public class Navigator
{
    public void Move(Vehicle vehicle)
    {
        vehicle.GoForward();
        vehicle.TurnLeft();
        vehicle.GoForward();
        vehicle.TurnRight();
        vehicle.GoForward();
    }
```

```
}
```

有了这个类，你期望能够将任何车辆传递给Navigator的Move方法，这对于汽车和卡车都会很好地工作。但是，当你想实现一个叫做Train的类时，会发生什么？



TurnLeft和TurnRight方法在Train类中不会工作，因为火车不能离开它的轨道。如果你确实将一辆火车传递给Navigator的Move方法，那么当你到达那些行时，它将抛出一个未实现的异常（或者什么也不做）。如果你不能用子类型替代类型，你就违反了里氏替换原则。

由于Train是Vehicle的子类型，你会期望在任何接受Vehicle类的地方使用它。否则，可能会使你的代码行为不可预测。

考虑一些提示以更加符合里氏替换原则：

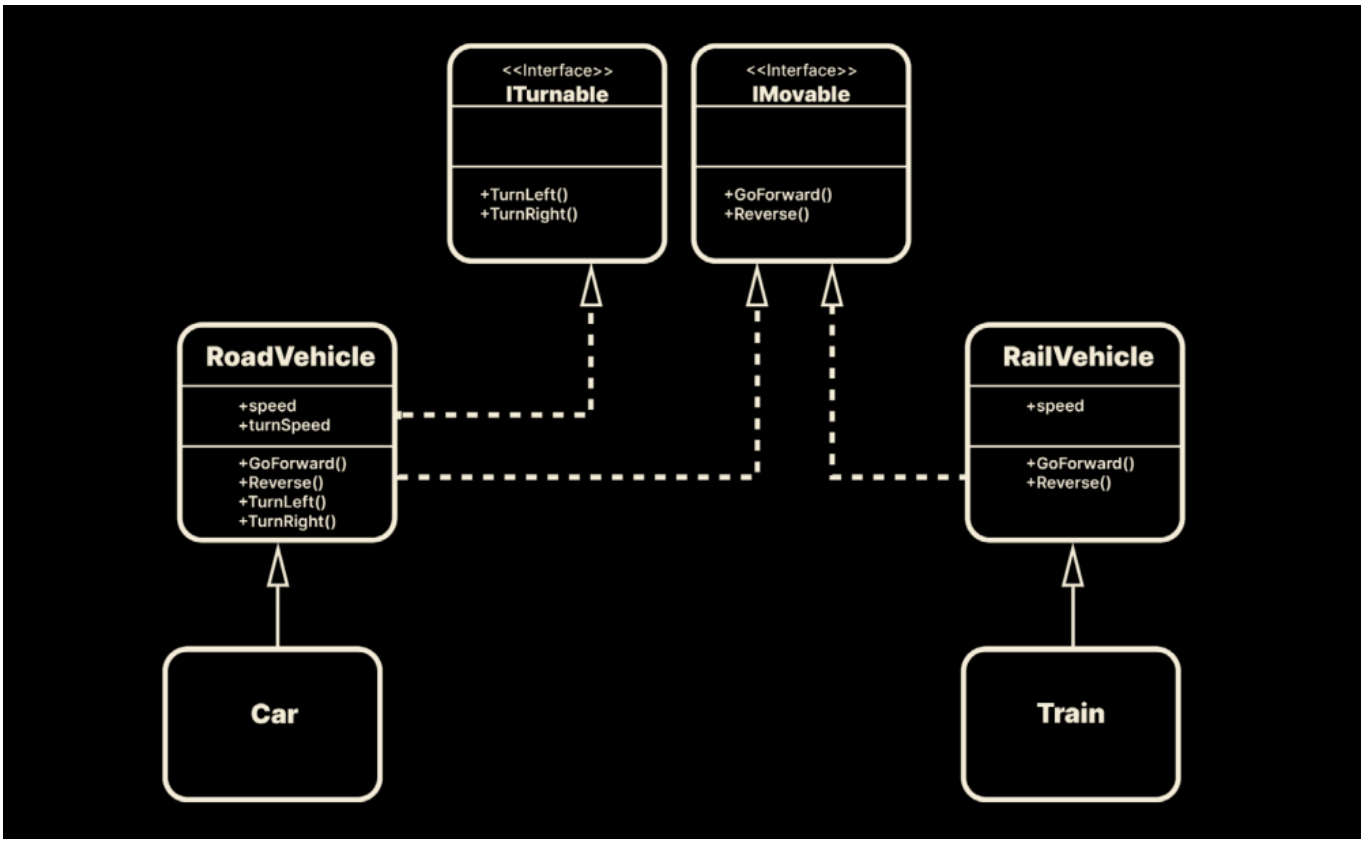
- **如果你在子类化时移除了功能，你可能正在破坏里氏替换：** NotImplementedException是你违反了这个原则的明显迹象。留下一个空的方法也是这样。如果子类的行为不像基类，那么你就没有遵循LSP——即使没有明显的错误或异常。
- **保持抽象简单：** 你在基类中放入的逻辑越多，就越可能破坏LSP。基类应该只表达派生子类的公共功能。
- **子类需要拥有与基类相同的公有成员：** 调用它们时，这些成员也需要具有相同的签名和行为。

- **在建立类层次结构之前考虑类API：** 尽管你把它们都看作是车辆，但是Car和Train继承自不同的父类可能更有意义。现实中的分类并不总是转化为类层次结构。
- **倾向于组合而不是继承：** 而不是试图通过继承传递功能，创建一个接口或单独的类来封装特定的行为。然后通过混合和匹配来构建不同功能的“组合”。

为了修复这个设计，废弃原来的Vehicle类型，然后将大部分功能移动到接口中：

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

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

通过创建RoadVehicle类型和RailVehicle类型，更加接近地遵循LSP原则。然后Car和Train将从各自的基类中继承。




```
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
{
    // ...
}
```

这样，功能通过接口而不是继承来实现。Car和Train不再共享同一基类，这符合了LSP。虽然你可以从同一基类派生RoadVehicle和RailVehicle，但在这种情况下没有太大的必要。

这种思考方式可能是违反直觉的，因为你对现实世界有一些假设。在软件开发中，这被称为圆-椭圆问题。并不是每一个实际的“是一个”关系都可以转化为继承。记住，你希望你的软件设计驱动你的类层次结构，而不是你对现实的先验知识。

遵循里氏替换原则，限制你如何使用继承，保持你的代码库可扩展和灵活。

接口隔离原则

接口隔离原则（ISP）规定，没有客户应该被迫依赖于它不使用的方法。

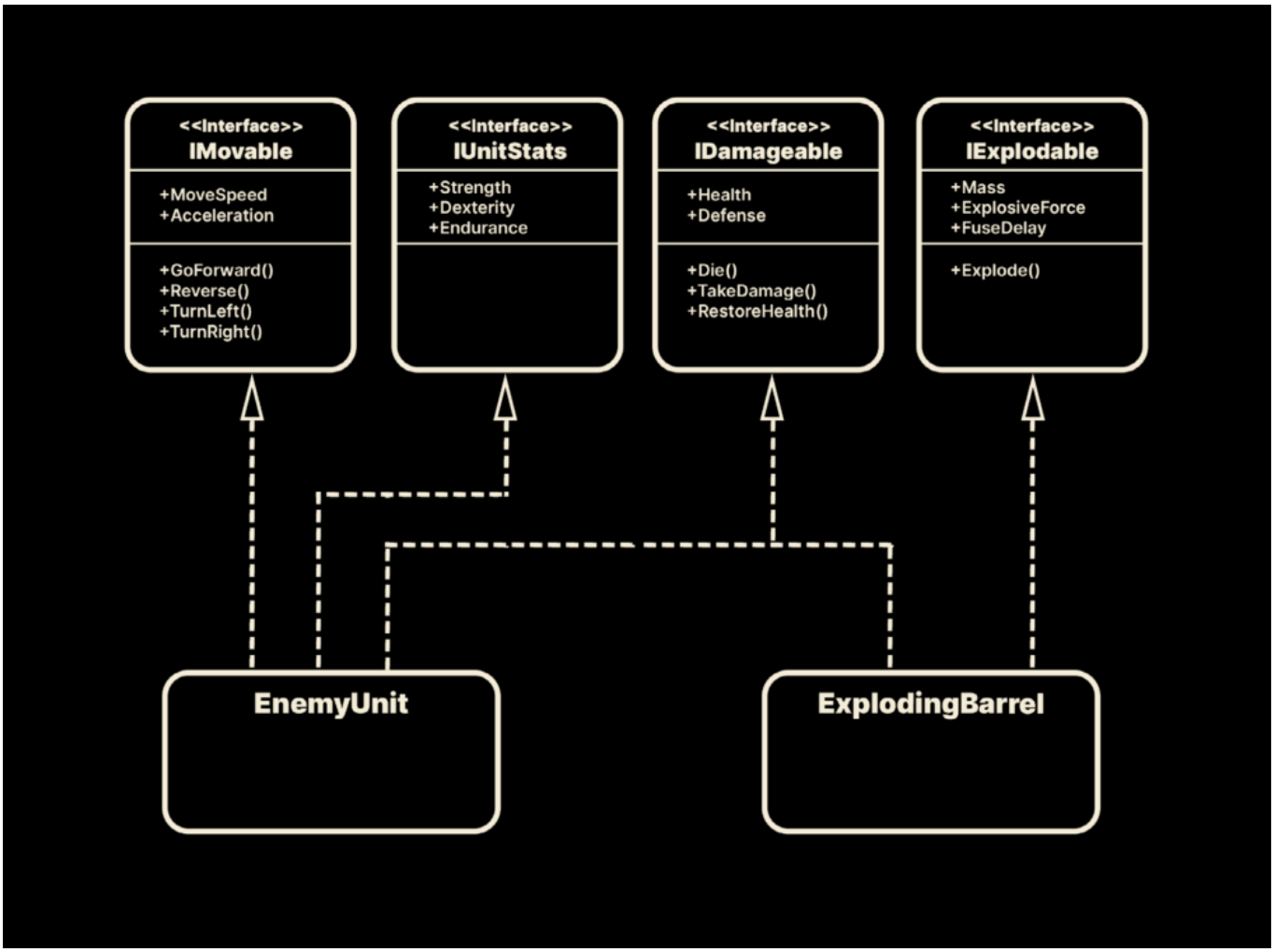
换句话说，避免大的接口。遵循与单一职责原则相同的理念，即保持类和方法短小。这给你提供了最大的灵活性，保持接口紧凑和专注。

假设你正在制作一个有不同玩家单位的策略游戏。每个单位都有不同的属性，如健康和速度。你可能想要创建一个接口，以保证所有的单位都实现类似的功能：

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

假设你想要制作一个可破坏的道具，如可破的桶或箱子。这个道具也需要健康的概念，尽管不会移动。一个箱子或桶也不会有与游戏中的其他单位相关的许多能力。

将它分解为几个较小的接口，而不是创建一个给可破坏道具提供太多方法的接口。实现它们的类将只需要混合和匹配它所需要的东西。




```
public interface IMovable
{
    float MoveSpeed { get; set; }
    float Acceleration { get; set; }
    void GoForward();
    void Reverse();
    void TurnLeft();
    void TurnRight();
}

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

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

你还可以为爆炸桶添加一个IExplodable接口：

```
public interface IExplodable
{
    float Mass { get; set; }
    float ExplosiveForce { get; set; }
    float FuseDelay { get; set; }
    void Explode();
}
```

因为一个类可以实现多个接口，你可以从IDamageable、IMoveable和IUnitStats组合一个敌人单位。

一个爆炸桶可以使用IDamageable和IExplodable，而不需要其他接口的不必要的开销。

```
public class ExplodingBarrel : MonoBehaviour, IDamageable, IExplodable
{
        ...
}
public class EnemyUnit : MonoBehaviour, IDamageable, IMovable, IUnit-Stats
{
        ...
}
```

再次，这更倾向于组合而不是继承，类似于里氏替换的例子。接口隔离原则有助于解耦你的系统，使它们更易于修改和重新部署。因为一个类可以实现多个接口，你可以从IDamageable、IMoveable和IUnitStats组合一个敌人单位。一个爆炸桶可以使用IDamageable和IExplodable，而不需要其他接口的不必要的开销。

依赖反转原则

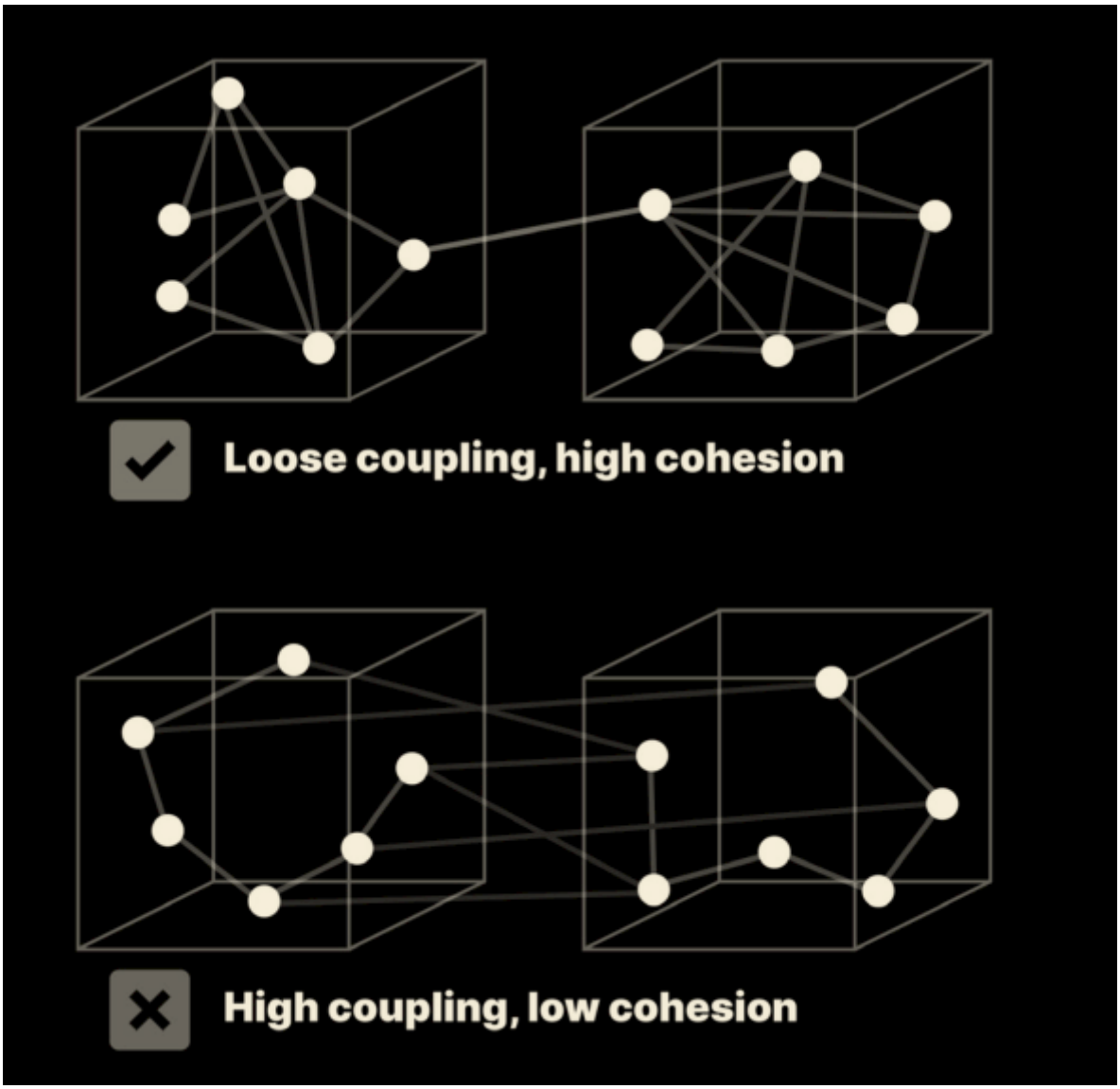
依赖反转原则（DIP）规定，高级模块不应直接从低级模块导入任何东西。两者都应依赖于抽象。

让我们解释一下这意味着什么。当一个类与另一个类有关系时，它有一个依赖或耦合。软件设计中的每一个依赖都带来一些风险。

如果一个类对另一个类的工作方式了解得太多，修改第一个类可能会损害第二个，反之亦然。高度的耦合被认为是不良的代码实践。一个部分的应用程序中的错误可能会雪球般变大。

理想情况下，应尽可能减少类之间的依赖。每个类也需要其内部部分一致地协同工作，而不是依赖于外部连接。当对象基于内部或私有逻辑进行操作时，被认为是有凝聚力的。

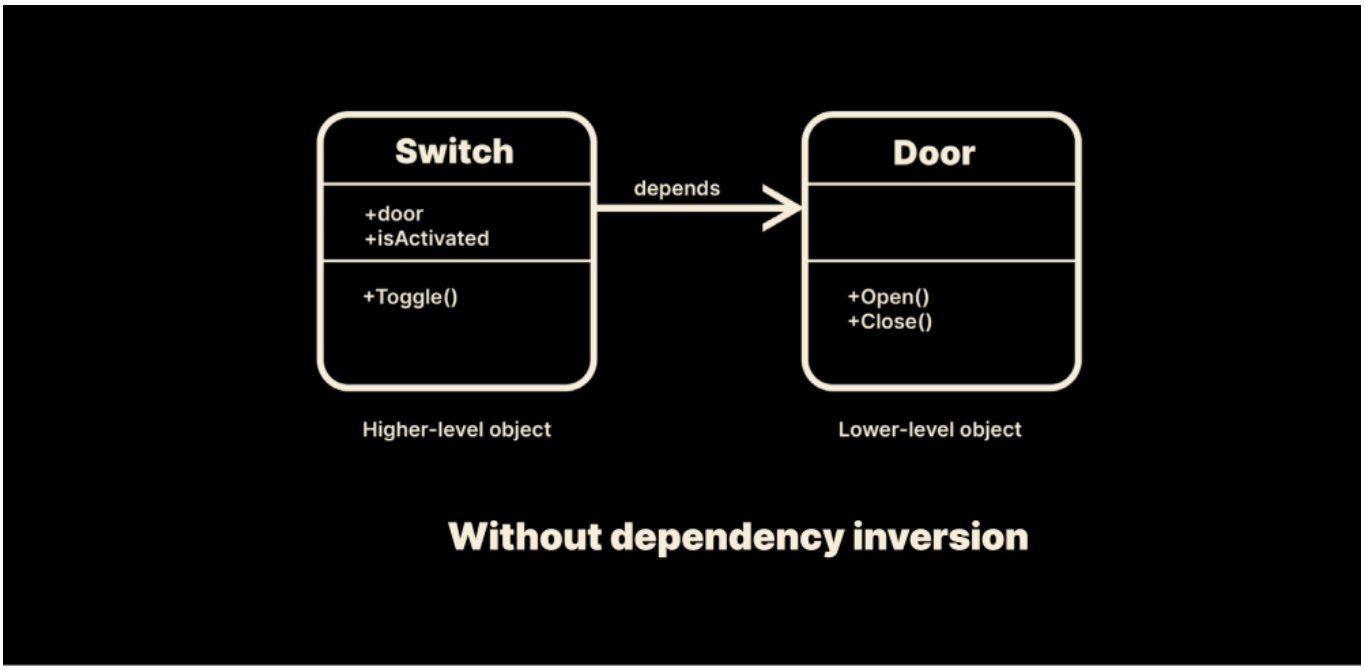
在最佳情况下，应寻求松散的耦合和高度的凝聚。



你需要能够修改和扩展你的游戏应用程序。如果它脆弱且抵制修改，那么就调查一下它目前的结构。

依赖反转原则可以帮助减少类之间的紧密耦合。在构建应用程序中的类和系统时，有些自然是“高级”的，有些是“低级”的。高级类依赖于低级类来完成某些工作。SOLID告诉我们要改变这种情况。

假设你正在制作一个游戏，角色在关卡中探索并触发门打开。你可能想要创建一个叫做Switch的类和另一个叫做Door的类。



在高层次上，你希望角色移动到一个特定的位置，并且发生一些事情。Switch将负责这个。

在低层次上，有另一个类Door，包含了如何打开门几何体的实际实现。为了简化，添加了一个Debug.Log语句来表示开门和关门的逻辑。

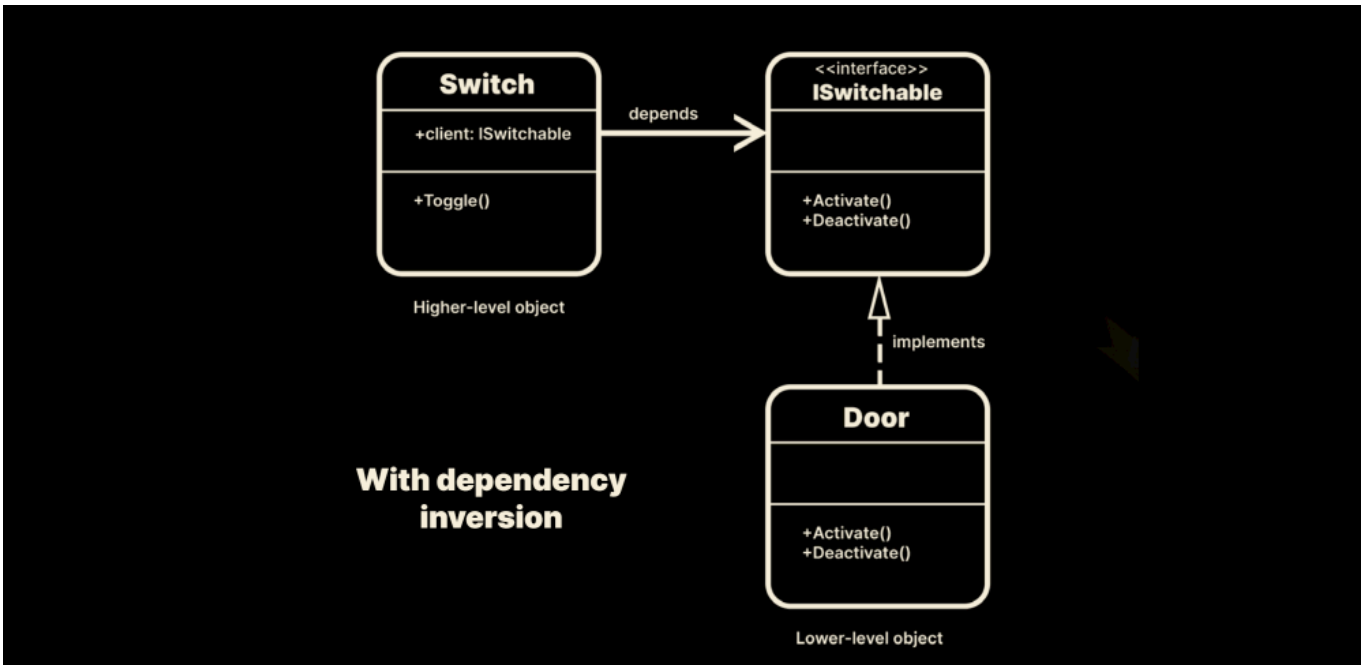
```
public class Switch : MonoBehaviour
{
    public Door door;
    public bool isActivated;

    public void Toggle()
    {
        if (isActivated)
        {
            isActivated = false;
            door.Close();
        }
        else
```

```
        {
            isActivated = true;
            door.Open();
        }
    }
}

public class Door : MonoBehaviour
{
    public void Open()
    {
        Debug.Log("The door is open.");
    }

    public void Close()
    {
        Debug.Log("The door is closed.");
    }
}
```

Switch可以调用Toggle方法来打开和关闭门。这是可行的，但问题是一个依赖直接从Door连线到Switch。如果Switch的逻辑需要作用于不仅仅是Door，例如，激活一个灯或巨型机器人，会怎么样?

你可以在Switch类中添加额外的方法，但你将违反开闭原则。每次你想扩展功能，都需要修改原始代码。

再一次，抽象拯救了你。你可以在你的类之间加一个叫做ISwitchable的接口。



ISwitchable只需要一个公共属性，所以你知道它是否活跃，再加上一对激活和停用它的方法。

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

然后Switch变成了这样，依赖于一个ISwitchable客户端，而不是直接依赖于门。

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

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

```
    }
}
```

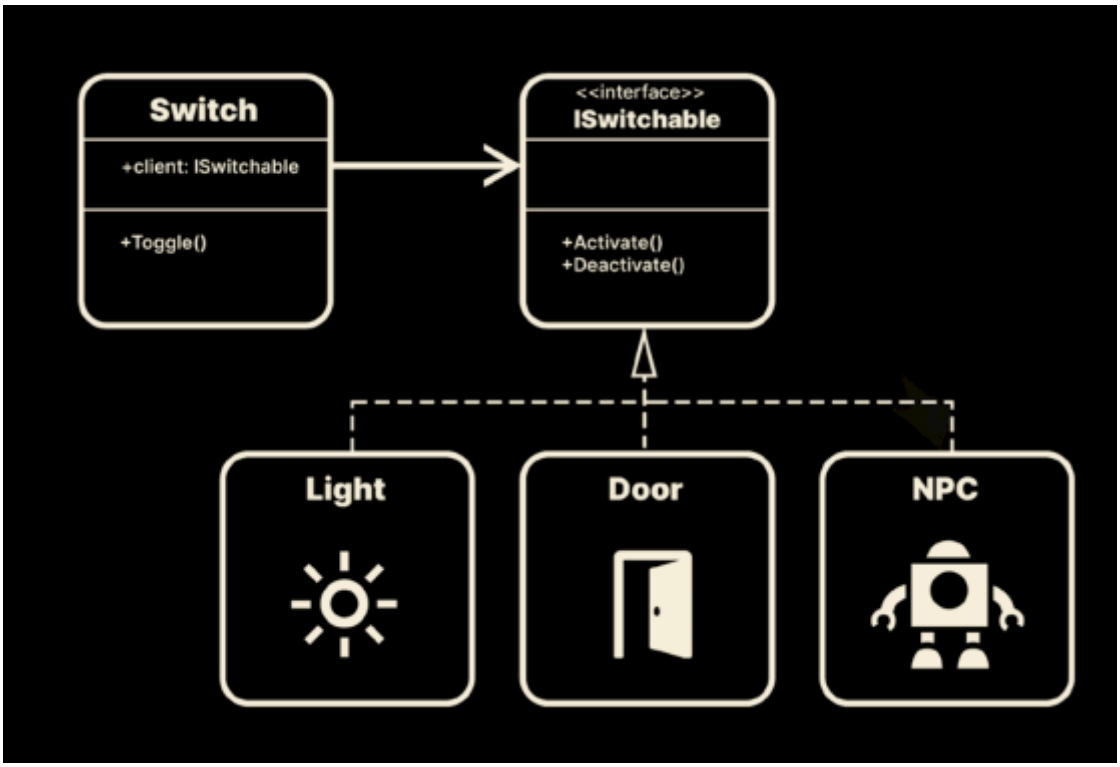
另一方面，你需要改造Door以实现ISwitchable：

```
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.");
    }
}
```

现在你已经反转了依赖。接口在它们之间创建了一个抽象，而不是硬编码Switch专门连接到门。Switch不再直接依赖于门特定的方法（Open和Close）。相反，它使用ISwitchable的Activate和Deactivate。

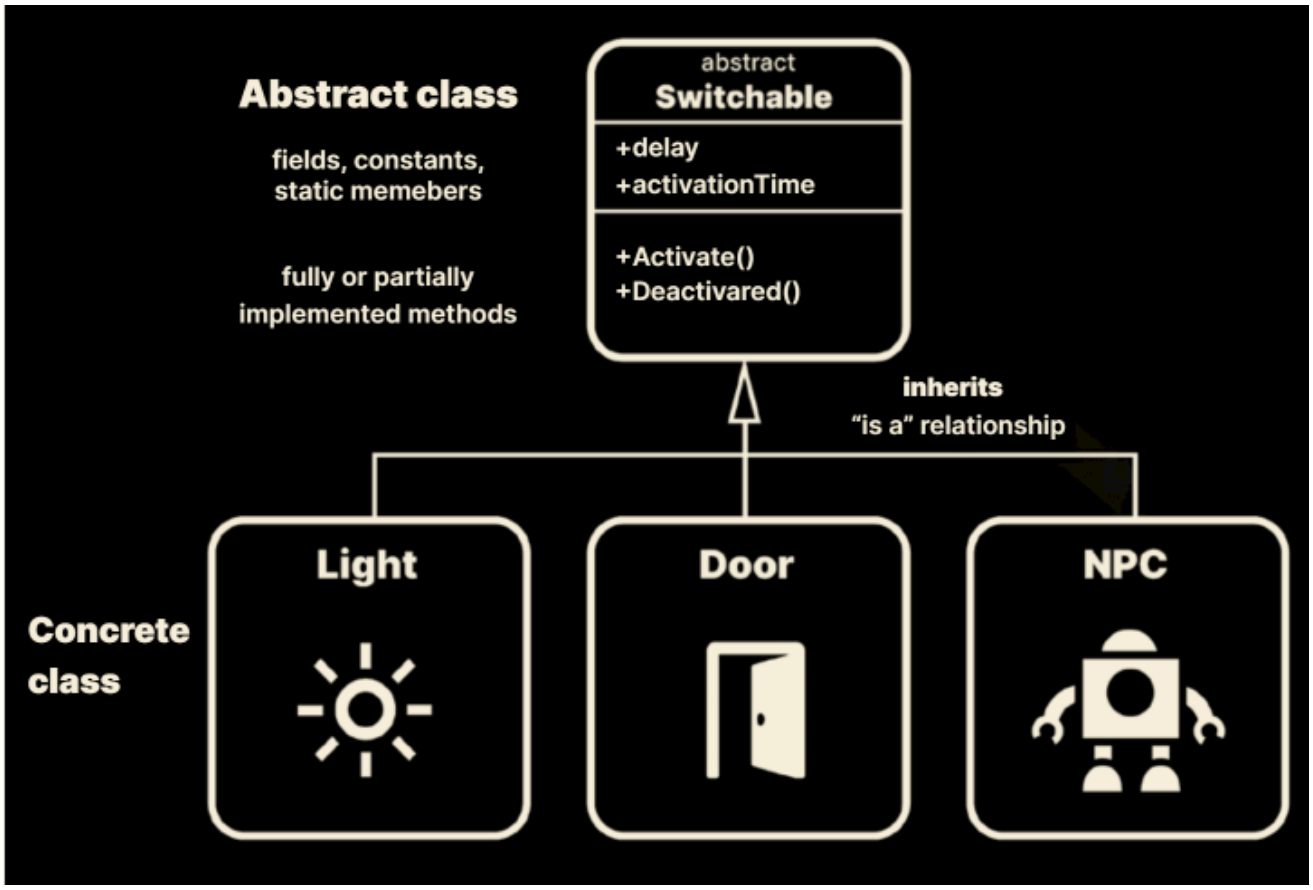
这个小而重要的改变促进了可重用性。虽然Switch以前只能和Door一起工作，但现在它可以和实现ISwitchable的任何东西一起工作。

这使你能够创建更多Switch可以激活的类。无论是陷阱门还是激光束，高级Switch都可以工作。它只需要一个实现ISwitchable的兼容客户端。



和SOLID的其余部分一样，依赖反转原则要求你检查你通常如何设置类之间的关系。方便地通过松散耦合来扩展你的项目。

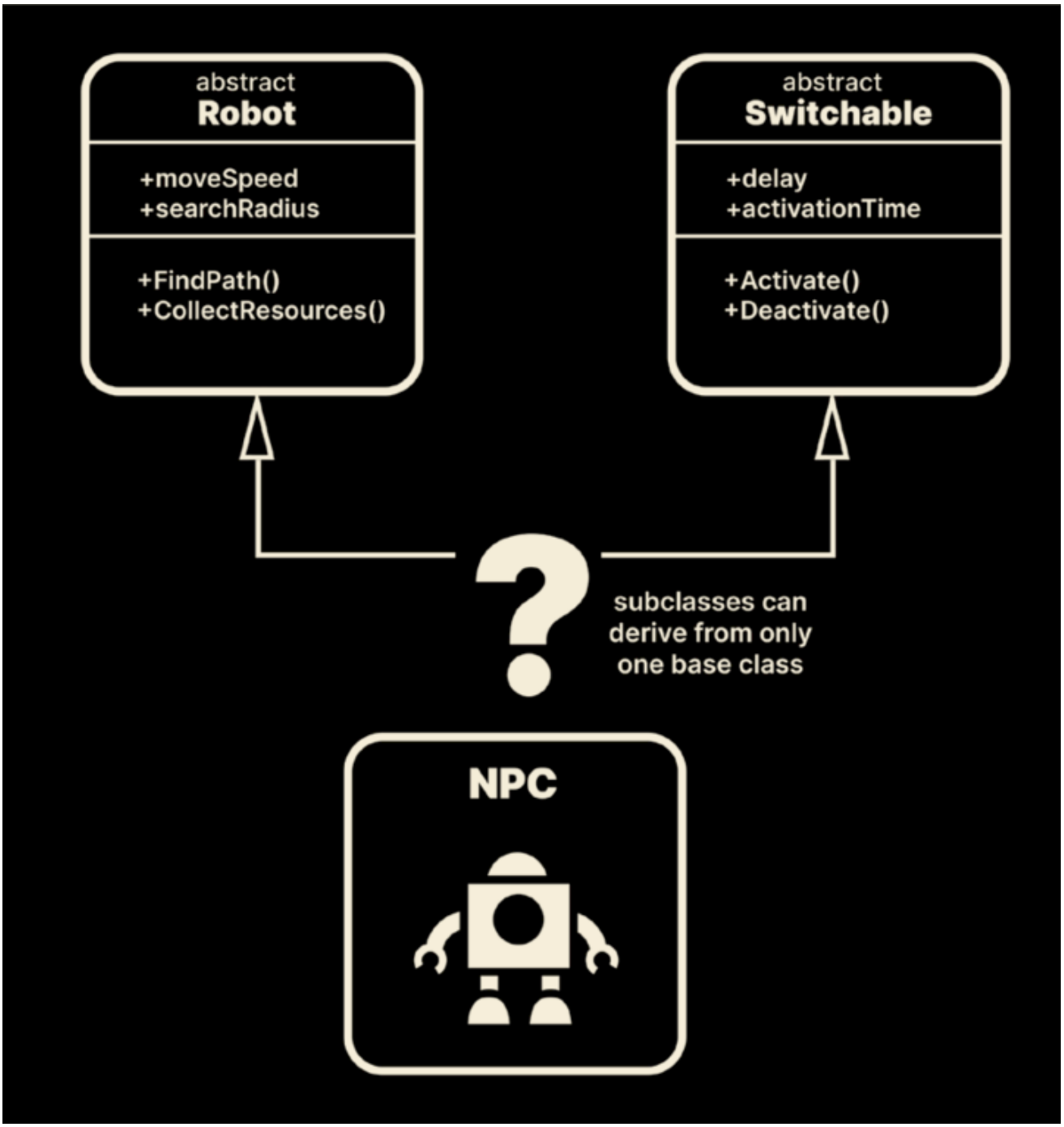
> **接口与抽象类**
>
> 遵循“优先考虑组合而不是继承”的哲学，在本指南中的许多示例使用接口。然而，你也可以使用抽象类来遵循许多设计原则和模式。
>
> 这两种都是在C#中实现抽象的有效方法。你使用哪种取决于你的实际需求。
>
> **抽象类**
>
> 抽象关键字允许你定义一个基类，这样你可以通过继承将公共功能（方法、字段、常量等）传递给子类。
>
> 你不能直接实例化一个抽象类。相反，你需要派生一个具体的类。
>
> 在前面的例子中，一个抽象类可以实现相同的依赖反转，只是用了不同的方法。所以，而不是使用接口，从一个叫做Switchable的抽象类派生一个具体的类（例如，Light或Door）。



继承定义了一个“是一个”关系。上图显示的都是可以打开和关闭的“可切换”的事物。

抽象类的优点是它们可以有字段和常量以及静态成员。它们也可以应用更严格的访问修饰符，如protected和private。与接口不同，抽象类允许你实现逻辑，让你在具体类之间共享核心功能。

继承在你想创建一个具有两个不同基类特性的派生类时表现得很好。在C#中，你不能从多个基类继承。



如果你有另一个抽象类，用于游戏中的所有机器人，那么决定从哪个基类派生就比较困难了。你是使用Robot还是Switchable基类?

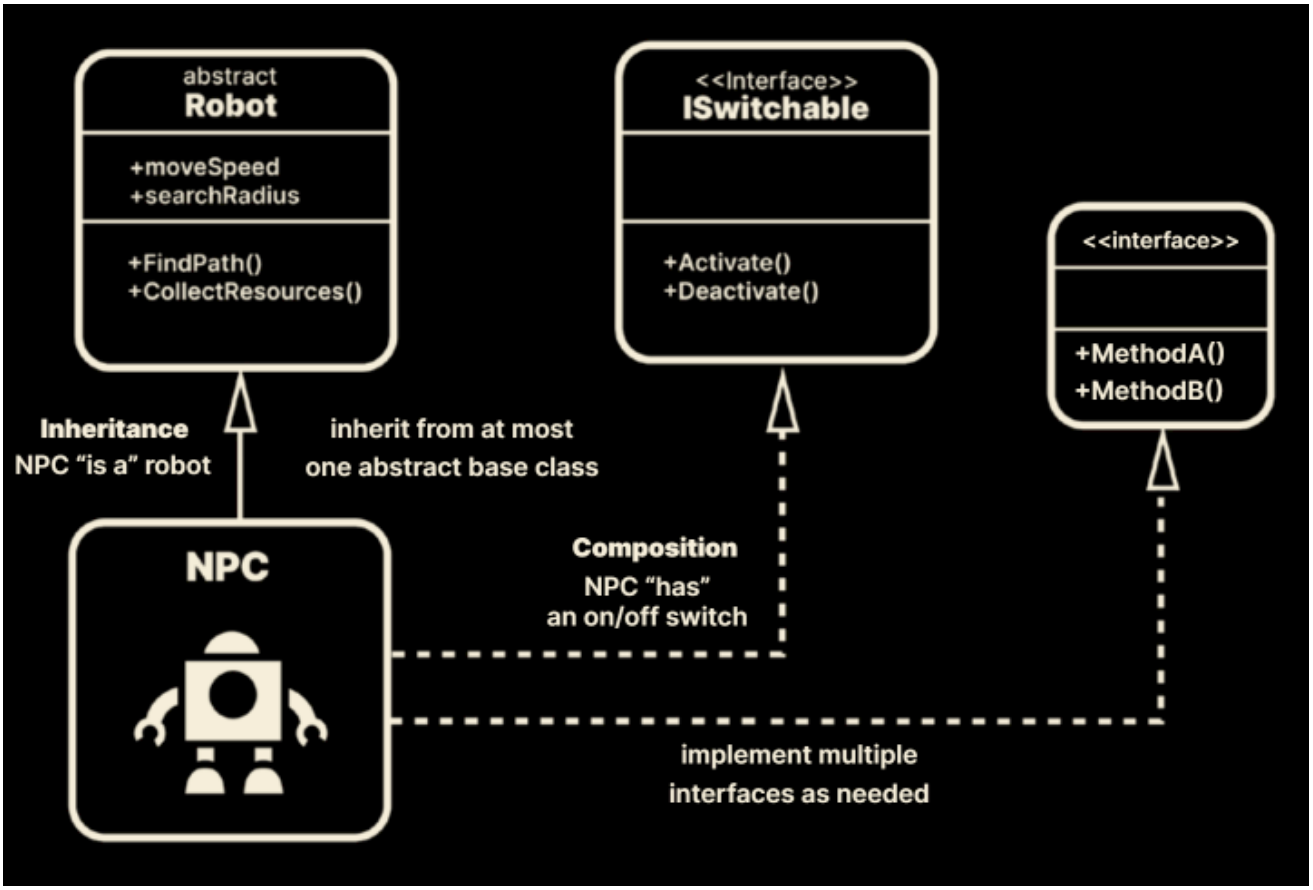
接口

如接口隔离原则所示，当某些东西不能整齐地适应继承范式时，接口可以给你更多的灵活性。你可以更容易地用“有一个”关系进行选择。

然而，接口只包含它们成员的声明。实际实现接口的类将负责制定具体的逻辑。

因此，这不总是一个非此即彼的决定。在你想共享代码的地方，使用抽象类定义基础功能。在你需要灵活性的地方，使用接口定义周边能力。

在这个例子中，你可以从Robot基类派生出NPC以继承其核心功能，但然后使用ISwitchable接口来增加打开和关闭NPC的能力。



记住抽象类和接口之间的以下区别

抽象类	接口
完全或部分实现方法	声明方法但不能实现它们
声明/使用变量和字段	仅声明方法和属性（但不包括字段）
具有静态成员	不能声明/使用静态成员
使用构造函数	不能使用构造函数
使用所有访问修饰符（protected，private等）	不能使用访问修饰符（所有成员都隐含为公开的）

记住：一个类最多只能继承自一个抽象类，但是它可以实现多个接口。

SOLID理解

了解SOLID原则是日常练习的问题。将它们视为编码时始终要记住的五个基本规则。这是一个方便的回顾：

- **单一责任：** 确保类只做一件事并且只有一个改变的原因。
- **开放封闭：** 你应该能够在不改变它已经如何工作的情况下扩展一个类的功能。
- **里氏替换：** 子类应该可以替换其基类。
- **接口隔离：** 保持你的接口简洁，带有少量的方法。客户端只实现它们所需的。
- **依赖反转：** 依赖于抽象。不要直接从一个具体类依赖另一个。

SOLID原则是帮助你编写更清晰代码的指导方针，这样它更容易维护和扩展。SOLID原则在企业级别已经主导了近二十年的软件设计，因为它们非常适合需要扩展的大型应用。

在某些情况下，坚持SOLID可能会导致前期的额外工作。你可能需要将一些功能重构为抽象或接口。然而，通常会有长期节省的回报。

自己决定你将如何严格地将原则应用到你的项目中；它们并不是绝对的。有许多细微差别，并且有许多没有在这里介绍的实现每一个的方法。记住：原则背后的思考比任何特定语法都更重要。

如果不确定如何使用它们，回头参考KISS原则。保持简单，不要试图为了做而将原则强加到你的脚本中。让它们通过必要性自然地找到自己的位置。

想要了解更多信息，一定要查看Unite Austin的Unity SOLID演示。

为游戏开发设计模式

一旦你理解了SOLID原则，你就会想要深入研究设计模式。

设计模式让你可以重新利用解决日常软件问题的知名解决方案。然而，模式并不是一个现成的库或框架。也不是算法，算法是实现结果的特定步骤集。

相反，将设计模式看作更像是一个蓝图。它是一个通用的计划，将实际的建设留给你。两个程序可以遵循相同的模式，但有非常不同的代码。

当开发者在实际中遇到相同的问题时，许多人将不可避免地提出类似的解决方案。一旦这样的解决方案被重复足够多的次数，有人可能会“发现”一个模式并正式地给它命名。

<四人帮

今天的许多软件设计模式都源自开创性的作品，Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides的《设计模式：可复用面向对象软件的元素》。这本书描述了在各种日常应用中识别的23种模式。

原作者常被称为“四人帮”（GoF），你也会听到原始模式被称为GoF模式。虽然引用的例子大多是C++（和Smalltalk），但你可以将它们的思想应用到任何面向对象的语言，比如C#。

自从四人帮在1994年首次发布《设计模式》以来，开发者们在各种领域发现了更多的面向对象模式。许多工程专业都有经过良好建立的模式。游戏开发也不例外。

学习设计模式

虽然你可以在没有研究设计模式的情况下作为一个游戏程序员工作，但学习它们只会帮助你成为更好的开发者。毕竟，设计模式被标记为这样，因为它们是众所周知的问题的常见解决方案。

软件工程师在正常的开发过程中总是重新发现它们。你可能已经无意中实现了其中的一些模式。

训练自己找到它们，做这些可以帮助你：

- **学习面向对象编程：** 设计模式不是隐藏在一个神秘的StackOverflow帖子中的秘密。它们是克服开发中日常难题的常见方法。它们可以告诉你许多其他开发者是如何解决同样的问题的。记住，即使你不使用模式，其他人也在使用。
- **和其他开发者交流：** 模式可以在试图作为一个团队进行交流时作为速记。提到“命令模式”或“对象池”，经验丰富的Unity开发者将会知道你正在尝试实现什么。
- **探索新框架：** 当你从Asset Store导入一个内置的包或者其他东西时，你必然会偶然遇到这里讨论的一个或多个模式。识别设计模式将帮助你理解新框架是如何工作的，以及创造它的思考过程。

当然，不是所有的设计模式都适用于每一个游戏应用。不要带着马斯洛的锤子去寻找它们，否则你可能只会找到钉子。

像任何其他工具一样，设计模式的有用性取决于上下文。每一个都在某些情况下提供了一个好处，并且也带来了它自己的一份缺点。每一个在软件开发中的决定都带有妥协。

你是否正在飞行中生成大量的GameObject？这是否影响你的性能？重构你的代码是否可以修复这个问题？

当时机合适时，要了解这些设计模式，并从你的gamedev技巧袋中提取它们来解决手头的问题。

拓展阅读

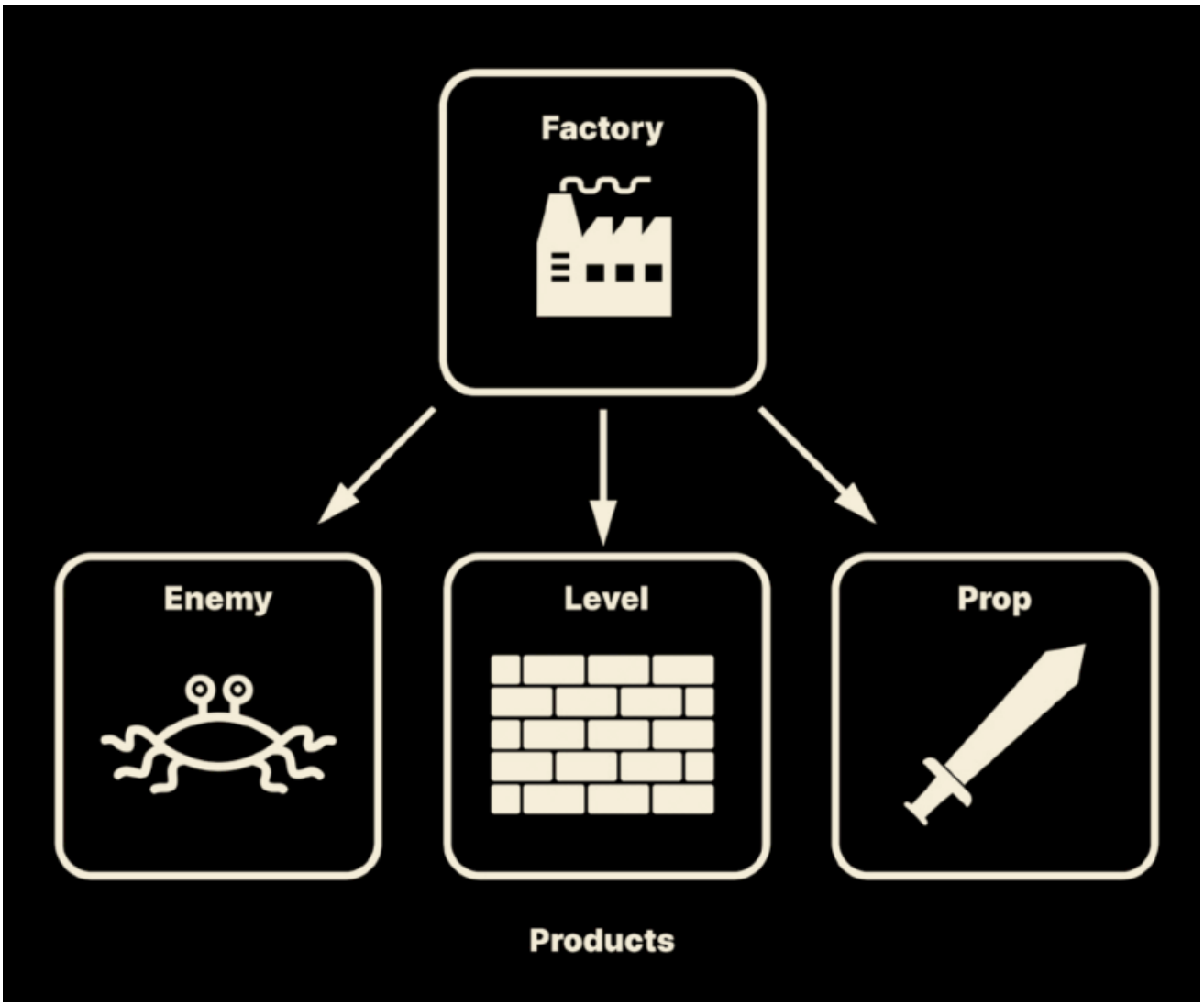
> 除了Gang of Four的《设计模式：可重用的面向对象软件的元素》之外，另一个突出的卷子是Robert Nystrom的《游戏编程模式》。作者以简明扼要的方式详细介绍了各种软件模式。基于网络的版本可以在gameprogrammingpatterns.com上免费获取。

Unity已经处理了这个，所以你不需要自己实现它。你只需要使用MonoBehaviour方法来管理游戏玩法，比如Update、LateUpdate和FixedUpdate。

Patterns within Unity

- 游戏循环：所有游戏的核心都是一个无限循环，它必须独立于时钟速度运行，因为支持游戏应用程序的硬件可能会有很大的差异。为了考虑到不同速度的计算机，游戏开发者通常需要使用固定的时间步长（设定每秒帧数）和变量时间步长，其中引擎测量自上一帧以来已经过去的时间。
  Unity已经处理了这一点，所以你不需要自己实现它。您只需要使用MonoBehaviour方法，如Update、LateUpdate和FixedUpdate来管理游戏玩法。
- **更新：** 在你的游戏应用中，你通常会一帧一帧地更新每个对象的行为。虽然你可以在Unity中手动重建这个，但是MonoBehaviour类会自动做这个。简单地使用适当的Update、LateUpdate或FixedUpdate方法来修改你的GameObject和组件以适应游戏时钟的一个刻度。
- **原型：** 你经常需要复制对象而不影响原来的对象。这个创造模式解决了复制和克隆一个对象来制作类似自己的其他对象的问题。这样你就避免了为你的游戏中的每种类型的对象定义一个单独的类。
  Unity的Prefab系统实现了GameObject的一种原型形式。这允许你复制一个模板对象，包括它的组件。重写特定的属性来创建Prefab变体或者在其他Prefab内部嵌套Prefab来创建层次结构。使用一个特殊的Prefab编辑模式来在隔离或上下文中编辑Prefab。
- **组件：** 在Unity中工作的大多数人都知道这个模式。不要创建带有多个责任的大类，而是构建更小的组件，每个组件只做一件事。
  如果你使用组合来挑选和选择组件，你可以组合它们进行复杂的行为。添加Rigidbody和Collider组件用于物理。添加MeshFilter和MeshRenderer用于3D几何。每个GameObject只是和其组件集合一样丰富和独特。
  当然，Unity不能为你做所有的事。你不可避免地会需要其他内置的模式。让我们在下一章节探索其中的一些。

FACTORY PATTERN



有时候，拥有一个特殊的对象来创建其他对象是有帮助的。许多游戏在游戏过程中会生成各种事物，你通常不知道在实际需要之前运行时需要什么。

工厂模式为此目的指定了一个特殊的对象，你猜对了，就是一个工厂。在一个层面上，它封装了生成其“产品”的许多细节。直接的好处是简化你的代码。

然而，如果每个产品遵循一个公共的接口或基类，你可以更进一步，并使其包含更多的自己的构造逻辑，隐藏它远离工厂本身。因此，创建新的对象变得更加可扩展。

你还可以对工厂进行子类化，制作多个专门用于特定产品的工厂。这样做有助于在运行时生成敌人、障碍物或其他任何事物。

示例：一个简单的工厂

假设你想要创建一个工厂模式来实例化游戏级别的项目。你可以使用Prefab来创建GameObject，但是你可能也想在创建每个实例时运行一些自定义行为。

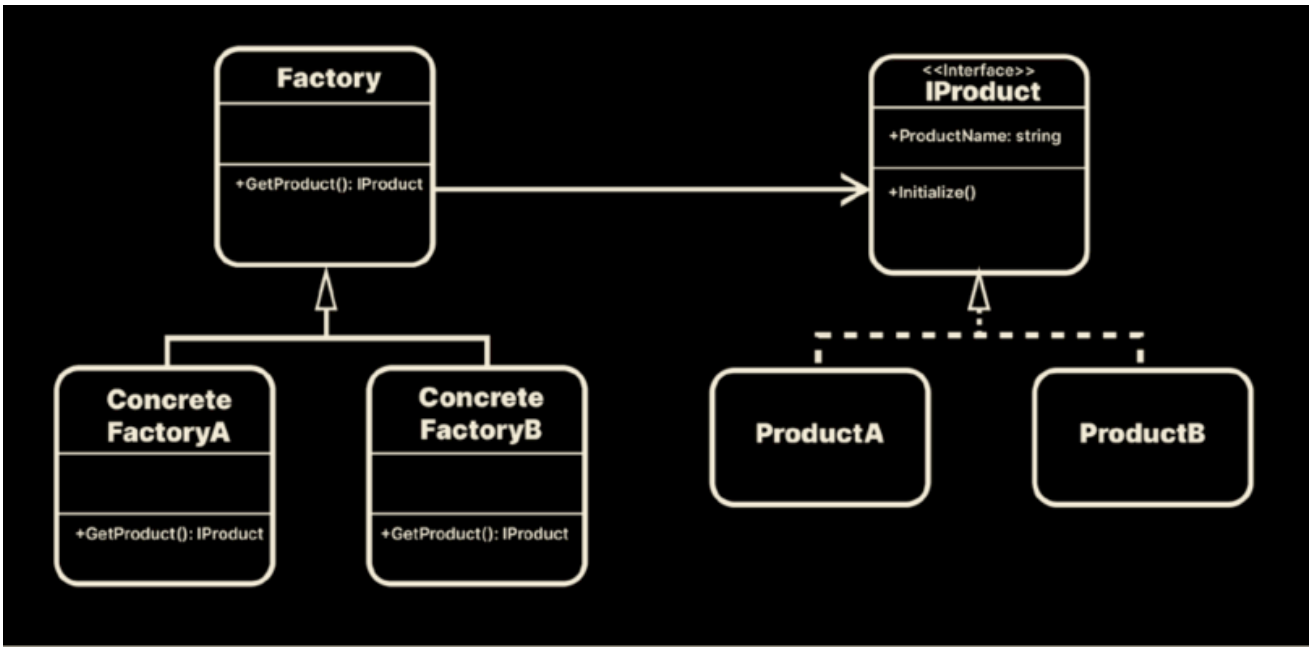
而不是使用if语句或switch来维护这个逻辑，创建一个名为IProduct的接口和一个名为Factory的抽象类。

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

public abstract class Factory : MonoBehaviour
{
    public abstract IProduct GetProduct(Vector3 position);

    // shared method with all factories
    // ...
}
```

产品需要遵循他们方法的特定模板，但他们并不共享任何功能。因此，你定义了IProduct接口。

工厂可能需要一些共享的公共功能，所以这个示例使用了抽象类。当使用子类时，只需记住SOLID原则中的里氏替换。

他们可以产生这样的结构：



IProduct接口定义了你的产品之间的共同点。在这种情况下，你只有一个ProductName属性和产品在Initialize时运行的任何逻辑。

然后你可以定义你需要的尽可能多的产品（ProductA，ProductB等），只要他们遵循IProduct接口。

基类Factory有一个返回IProduct的GetProduct方法。它是抽象的，所以你不能直接实例化Factory。你派生出一对具体的子类（ConcreteFactoryA和ConcreteFactoryB），它们实际上会得到不同的产品。

在这个示例中，GetProduct接受一个Vector3位置，这样你可以更容易地在特定位置实例化一个Prefab GameObject。每个具体工厂的一个字段也存储了相应的模板Prefab。

这里是示例ProductA和ConcreteFactoryA。

```
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;
    }
}
```

在这里，你已经使产品类成为实现IProduct的MonoBehaviours，利用工厂中的Prefab。

注意每个产品如何可以有自己的Initialize版本。示例ProductA Prefab包含一个ParticleSystem，当ConcreteFactoryA实例化一个副本时播放。工厂本身不包含触发粒子的任何特定逻辑；它只调用Initialize方法，这是所有产品的共有的。

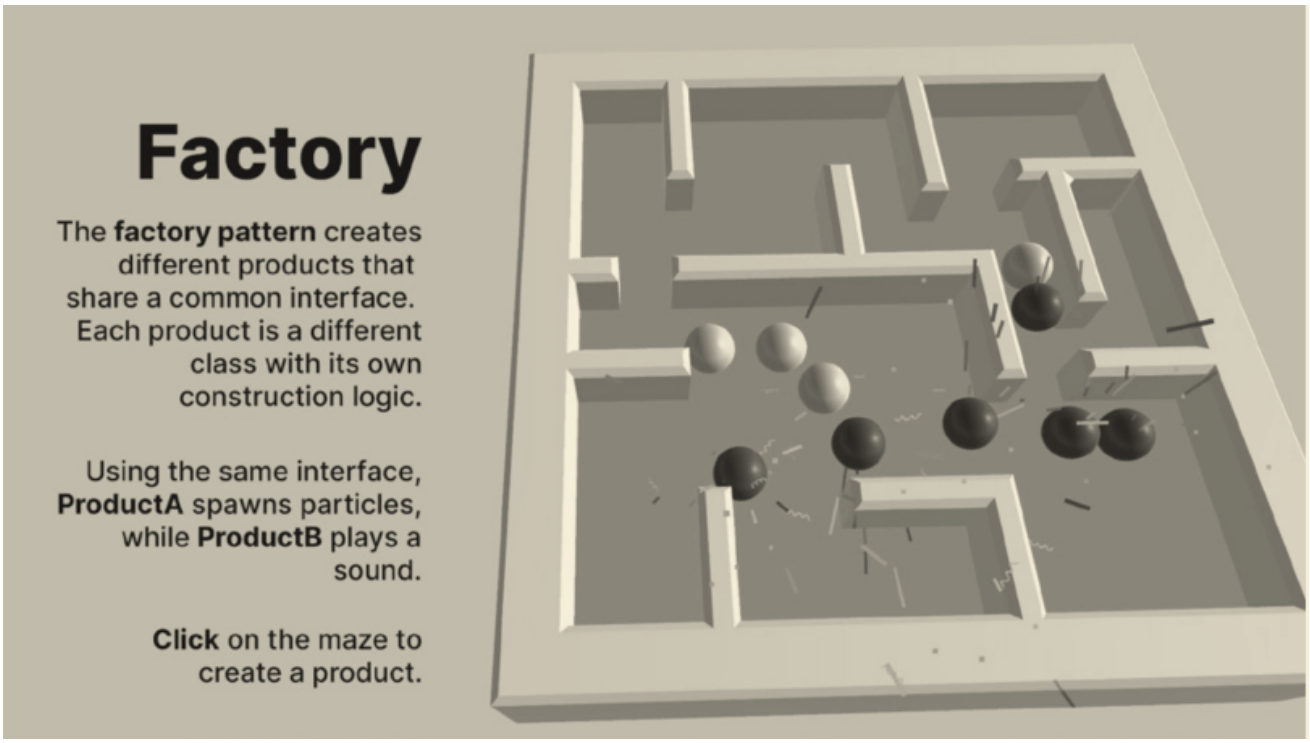
探索示例项目，看看ClickToCreate组件如何切换工厂来创建ProductA和ProductB，它们有不同的行为。ProductB在生成时播放一个声音，而ProductA则启动一个粒子效果。

优点和缺点

当设置许多产品时，你将从工厂模式中受益最多。在你的应用中定义新的产品类型并不会改变你现有的产品或需要你修改之前的代码。

将每个产品的内部逻辑分离到它自己的类中，保持工厂代码相对较短。每个工厂只知道调用每个产品上的Initialize，而不了解底层的细节。

缺点是你需要创建许多类和子类来实现这个模式。和其他模式一样，这引入了一些开销，如果你没有大量的产品种类，这可能是不必要的。



一个产品播放声音，而另一个播放粒子。两者都使用相同的接口。

改进

工厂的实现方式可以从这里显示的内容大相径庭。在构建你自己的工厂模式时，考虑以下调整：

- **使用字典搜索产品：** 你可能想要将你的产品作为键值对存储在字典中。使用一个唯一的字符串标识符（例如，名称或某个ID）作为键，类型作为值。这可以使检索产品和/或其对应的工厂更方便。
- **让工厂（或工厂管理器）成为静态的：** 这使其更容易使用，但需要额外的设置。静态类不会出现在检查器中，所以你需要让你的产品集合也成为静态的。
- **将其应用到非GameObject和非MonoBehaviour：** 不要将自己限制于Prefabs或其他Unity特定的组件。工厂模式可以用任何C#对象工作。
- **与对象池模式结合：** 工厂并不一定需要实例化或创建新的对象。它们也可以检索层次结构中的现有对象。如果你一次实例化许多对象（例如，从武器发射的弹头），使用对象池模式进行更优化的内存管理。

工厂可以根据需要生成任何游戏元素。然而，注意，创建产品通常并不是它们唯一的目的。你可能正在使用工厂模式作为另一个更大任务的一部分（例如，设置对话框中的UI元素或游戏级别的部分）。

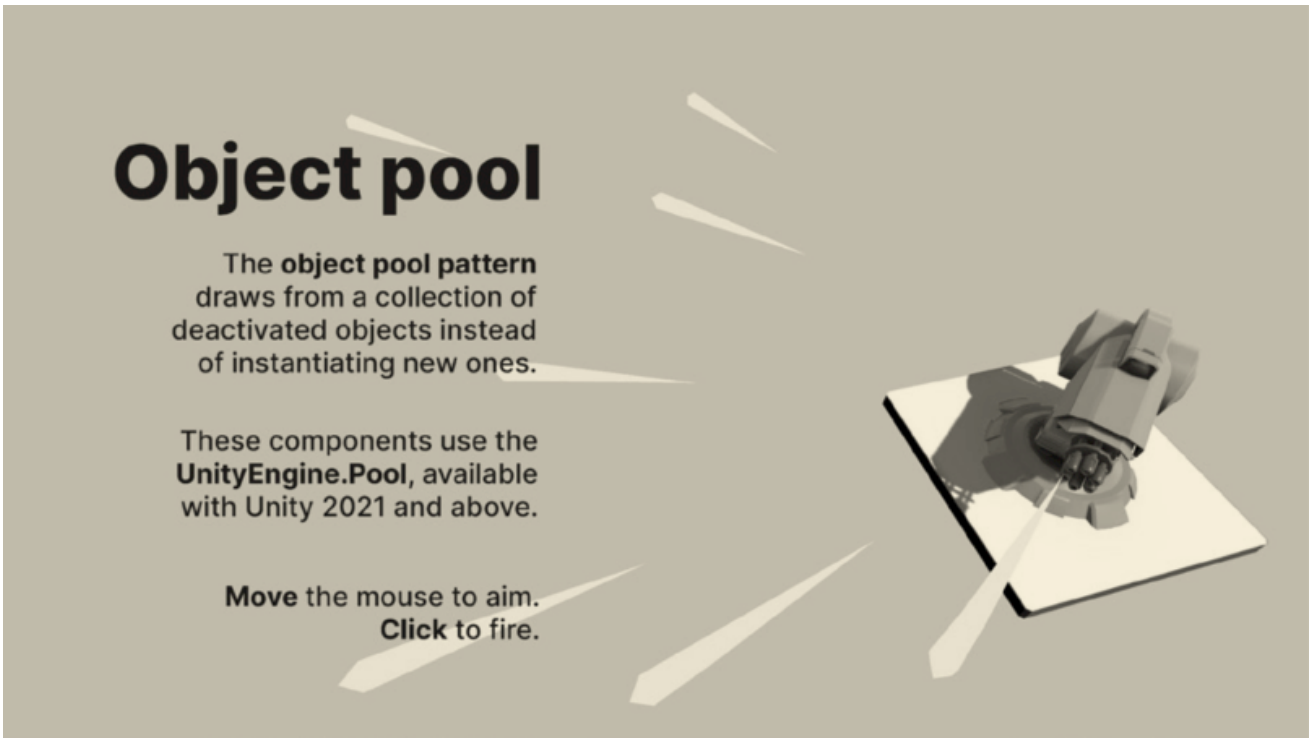
OBJECT POOL

对象池是一种优化技术，用于在创建和销毁大量GameObject时减轻CPU的负担。

对象池模式使用一组初始化的对象，这些对象在被停用的“池”中保持就绪和等待。当你需要一个对象时，你的应用程序不会实例化它。相反，你从池中请求GameObject并启用它。

使用完后，你会停用对象并将其返回到池中，而不是销毁它。

对象池可以减少可能由于垃圾收集峰值而导致的卡顿。GC峰值经常伴随着创建或销毁大量对象，这是由于内存的分配。你可以在合适的时候预先实例化你的对象池，比如在加载屏幕上，这样用户就不会注意到卡顿。



示例：简单的池系统

考虑一个简单的池系统，定义了两个MonoBehaviours：

- 一个ObjectPool，它持有要提取的GameObject的集合
- 添加到Prefab的PooledObject组件。这帮助每个克隆的项目保持对池的引用

在ObjectPool中，你设置了描述池大小的字段，你想要存储的PooledObject Prefab，以及将形成池本身的集合（在这个例子中是一个堆栈）。

```
public class ObjectPool : MonoBehaviour
{
    [SerializeField] private uint initPoolSize;
    [SerializeField] private PooledObject objectToPool;

    // store the pooled objects in a collection
    private Stack<PooledObject> stack;

    private void Start()
    {
        SetupPool();
    }

    // creates the pool (invoke when the lag is not noticeable)
    private void SetupPool()
    {
        stack = new Stack<PooledObject>();
        PooledObject instance = null;

        for (int i = 0; i < initPoolSize; i++)
        {
            instance = Instantiate(objectToPool);
            instance.Pool = this;
            instance.gameObject.SetActive(false);
            stack.Push(instance);
        }
    }
}
```

SetupPool方法填充对象池。创建一个新的PooledObjects堆栈，然后实例化objectToPool的副本，用initPoolSize元素填充它。在Start中调用SetupPool，以确保它在游戏过程中运行一次。

你还需要方法来检索一个池项目（GetPooledObject）和返回一个到池（ReturnToPool）：

```
// returns the first active GameObject from the pool
public PooledObject GetPooledObject()
{
    // if the pool is not large enough, instantiate a new PooledObjects
    if (stack.Count == 0)
```

```
    {
        PooledObject newInstance = Instantiate(objectToPool);
        newInstance.Pool = this;
        return newInstance;
    }

    // otherwise, just grab the next one from the list
    PooledObject nextInstance = stack.Pop();
    nextInstance.gameObject.SetActive(true);
    return nextInstance;
}

public void ReturnToPool(PooledObject pooledObject)
{
    stack.Push(pooledObject);
    pooledObject.gameObject.SetActive(false);
}
```

GetPooledObject只在池为空时创建一个新的PooledObject。否则，它只是返回下一个可用的元素。如果池的大小足够，大部分时间你应该只获取一个现有GameObject的引用。

调用GetPooledObject的客户端然后需要移动/旋转池对象到位。

每个池的元素将有一个小的PooledObject组件，只是为了引用ObjectPool：

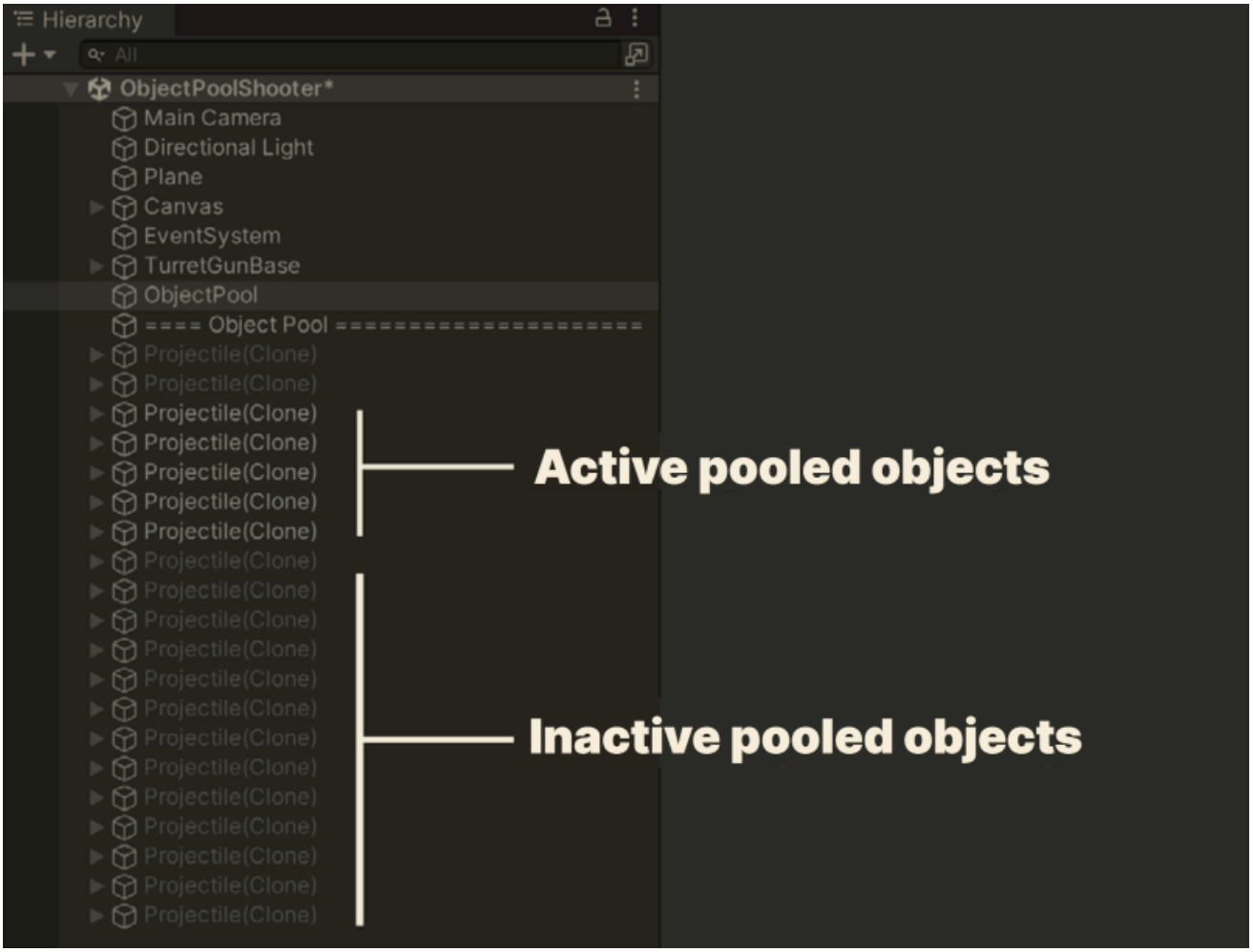
```
public class PooledObject : MonoBehaviour
{
    private ObjectPool pool;

    public ObjectPool Pool
    {
        get => pool;
        set => pool = value;
    }

    public void Release()
    {
        pool.ReturnToPool(this);
    }
```

```
}
```

调用Release会禁用GameObject并将其返回到池队列。

伴随的项目显示了一个基本的使用例子。这里，一个ExampleGun脚本被附加到一个GameObject。它存储了对对象池的引用。当用户射击时，武器脚本调用它的GetPooledObject方法，而不是调用Object.Instantiate。

在弹头本身上有一个ExampleProjectile脚本和一个PooledObject脚本。ExampleProjectile有一个Deactivate方法，可以在几秒钟后禁用每个发射的子弹GameObject，将其返回到可用的池。



这样，你可以看起来在屏幕外发射数百发子弹，实际上，你只是禁用并回收它们。只需确保你的池大小足够大，以显示同时活动的对象。

如果你需要超过池大小，池可以实例化额外的对象。然而，大部分时间它只是从现有的非活动对象中拉取。

如果你使用过Unity的ParticleSystem，那么你就有了使用对象池的亲身体验。ParticleSystem组件包含一个最大粒子数的设置。这只是回收可用的粒子，防止效果超过最大数量。对象池的工作

方式类似，但是可以使用你选择的任何GameObject。

改进

以上的示例是一个简单的例子。当为实际项目部署对象池时，考虑以下升级：

- 使其静态或单例：如果你需要从各种源生成池对象，考虑使对象池静态化。这使得它可以在你的应用程序的任何地方访问，但阻止了使用检查器的使用。或者，将对象池模式与单例模式结合使用，使其全局可访问，便于使用。
- 使用字典管理多个池：如果你有多个你想要池化的Prefabs，将它们存储在单独的池中，并存储一个键值对，以便你知道哪个池要查询（Prefab的InstanceID可以作为唯一的键）。
- 创造性地移除未使用的GameObject：有效地使用对象池的一部分是隐藏未使用的对象并将它们返回到池。利用每一个机会停用一个池对象（例如，屏幕外，被爆炸隐藏，等等）
- 检查错误：避免释放已经在池中的对象。否则，这可能会在运行时导致错误。
- 添加最大大小/上限：大量的池对象会消耗内存。你可能需要移除超过某个限制的对象，以便池不会使用过多的资源。

你如何使用对象池会因应用而异。这种模式常见于枪支或武器需要发射多个弹头的情况，比如在一个弹幕射击游戏中。

每当你实例化大量的对象时，你都有可能因为垃圾收集峰值而引起小的停顿。一个对象池可以减轻这个问题，使你的游戏流畅。

如果你正在使用2021年及以后版本的Unity，它包含一个内置的对象池系统，所以没有必要像前面的例子那样创建你自己的PooledObject或ObjectPool类。

UnityEngine.Pool

对象池模式在Unity 2021中现在支持自己的UnityEngine.Pool API。这为您提供了一个基于堆栈的ObjectPool来跟踪您的对象池模式对象。根据您的需要,您也可以使用CollectionPool(List、HashSet、Dictionary等)。

在示例项目中(请参见场景),您不再需要自定义池组件。相反,使用using UnityEngine.Pool在顶部更新枪脚本;这允许您使用内置的ObjectPool创建一个弹丸池:

```
using UnityEngine.Pool;
```

```
public class RevisedGun : MonoBehaviour
{
    // ...
    // 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 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()
    {
        // ...
    }
}
```

大部分脚本适用于原始的ExampleGun脚本。然而,ObjectPool构造函数现在包括在以下情况下设置一些逻辑的有用功能:

- 首先创建一个池化物品来填充池
- 从池中获取一个物品
- 将一个物品返回到池中
- 销毁一个池化对象(例如,如果您达到最大限制)

然后,您必须定义一些相应的方法来传递给构造函数。

请注意,内置的ObjectPool还包括默认池大小和最大池大小的选项。超过最大池大小的项目会触发自毁操作,以保持内存使用量在检查之下。

弹丸脚本进行了小的修改,以保持对ObjectPool的引用。这使得将对象释放回池变得更加方便。

```
public class RevisedProjectile : MonoBehaviour
{
    // ...
    private IObjectPool<RevisedProjectile> objectPool;

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

    // ...
```

```
}
```

UnityEngine.Pool API使设置对象池更快,现在您不必从头重新构建该模式。这是一个轮子不必重新发明。

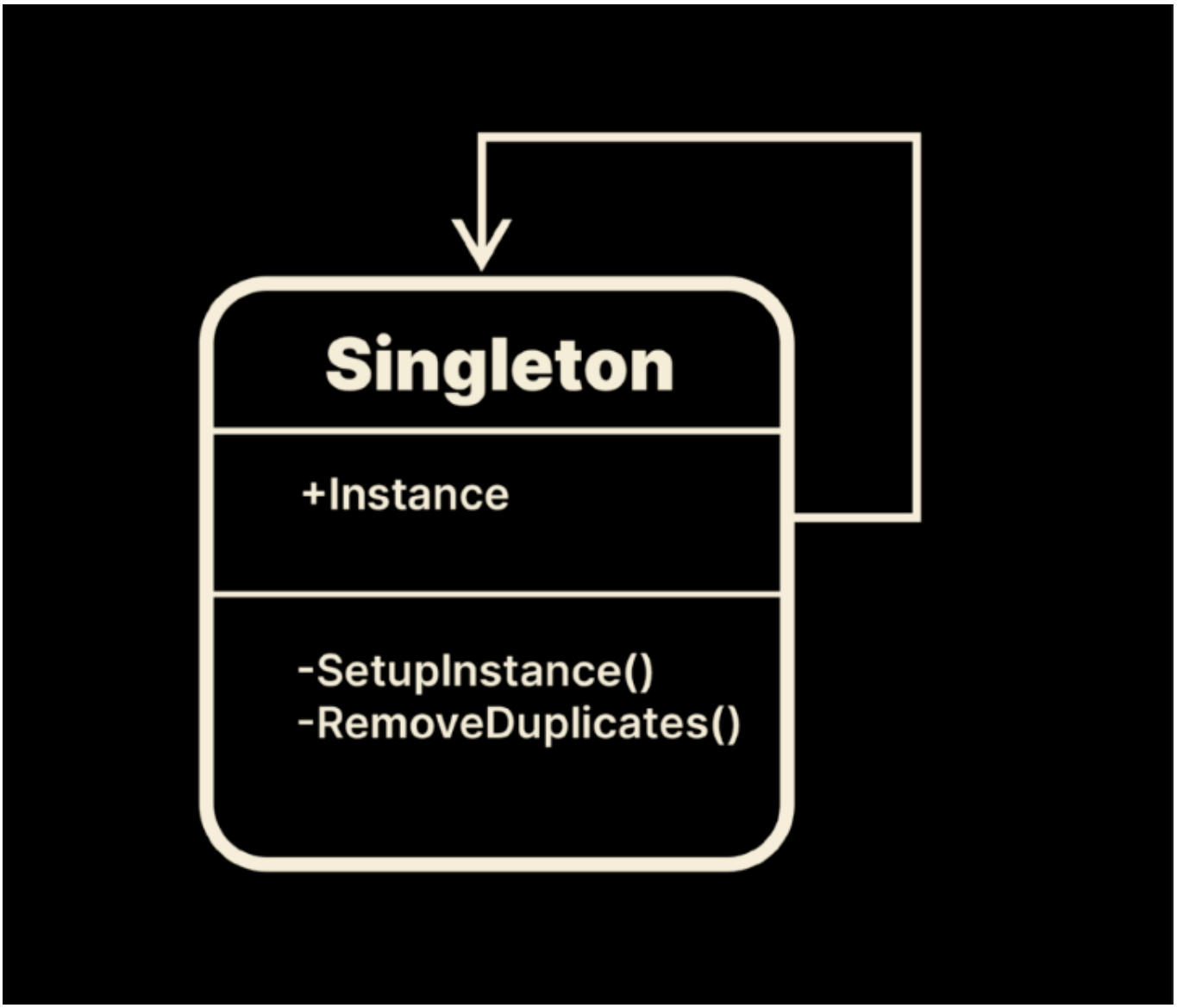
SINGLETON PATTERN

单例模式经常受到批评。如果你是 Unity 开发的新手，单例可能是你在实际编程中遇到的第一个可以识别的模式。但它也是最容易被批评的设计模式之一。

根据最初的四人帮 (Gang of Four)，单例模式：

- 确保一个类只能实例化一个实例
- 提供全局轻松访问该单一实例

如果你需要有一个确切的对象来协调整个场景的行动，这就很有用。例如，你可能希望在你的场景中有一个精确的游戏管理器来指导主游戏循环。你也可能只想有一个文件管理器一次写入你的文件系统。这样的中心，管理级别的对象往往是单例模式的好候选对象。



在《游戏编程模式》中，它说单例模式带来的坏处多于好处，并将其列为反模式。这个糟糕的声誉是因为该模式的使用容易导致滥用。开发人员倾向于在不适当的情况下应用单例模式，引入不必要的全局状态或依赖性。

让我们来看看如何在 Unity 中构建一个单例，并权衡其优点和缺点。然后你可以决定是否值得将其纳入你的应用程序中。

示例：简单的单例

一个最简单的单例可能看起来像这样：

```
using UnityEngine;
```

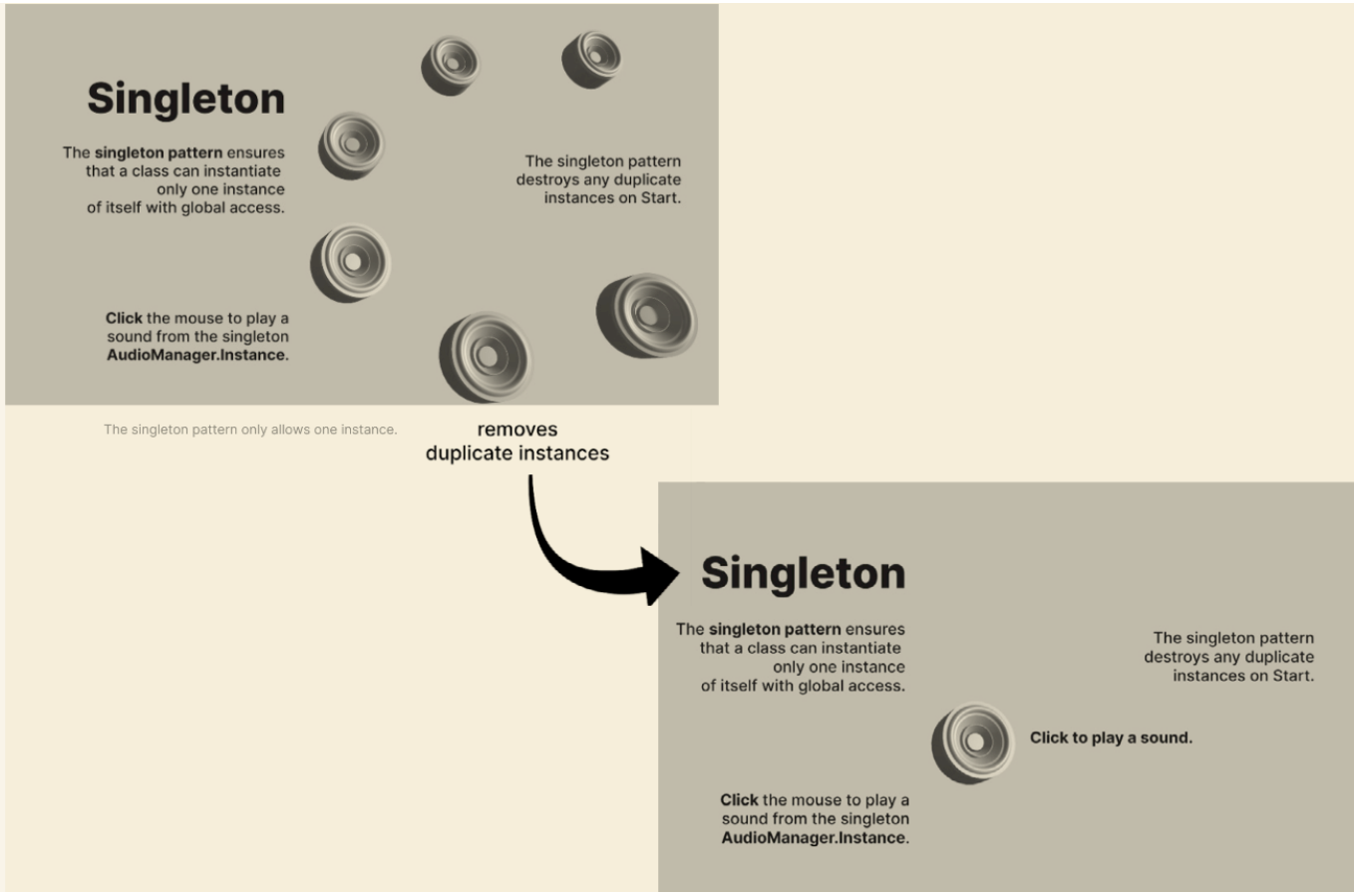
```
public class SimpleSingleton : MonoBehaviour
{
    public static SimpleSingleton Instance;

    private void Awake()
    {
        if (Instance == null)
        {
            Instance = this;
        }
        else
        {
            Destroy(gameObject);
        }
    }
}
```

公共静态 Instance 将保存场景中 Singleton 的一个实例。

在 Awake 方法中，检查它是否已经设置。如果 Instance 当前为 null，那么 Instance 被设置为这个特定的对象。这必须是场景中的第一个单例。

否则，这个实例必定是一个重复项；你需要调用 Destroy(gameObject) 来确保你的单例在场景中只有一个这样的组件。

如果你在运行时的层次结构中将脚本附加到多个 GameObject 上，Awake 中的逻辑将保留第一个对象，然后丢弃其余的。



Instance 字段是公开的和静态的。任何组件都可以从场景中的任何地方全局访问唯一的单例。

持久性和延迟实例化

SimpleSingleton 按原样工作。然而，它确实有两个问题：

- 加载新场景会销毁 GameObject。
- 在使用它之前，你需要在层次结构中设置单例。

因为单例通常作为一个全能的管理脚本，你可以使其持久化，如使用 DontDestroyOnLoad。

此外，你可以使用延迟实例化在你首次需要时自动构建单例。你只需要一些逻辑来创建一个 GameObject，然后添加适当的 Singleton 组件。

改进的单例看起来像这样：

```
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);
        }
    }
}
```

Instance 现在是一个公共属性，引用私有 instance 备份字段。第一次你引用单例时，检查 Instance 是否存在于 getter 中。如果不存在，SetupInstance 方法创建一个带有适当组件的

GameObject。

DontDestroyOnLoad(gameObject) 阻止场景加载从层次结构中清除单例。现在，单例实例是持久的，即使你在游戏中改变场景，它也会保持活动。

使用泛型

以上版本的脚本都没有解决如何在同一场景中创建不同的单例。例如，如果你想要一个表现为 AudioManager 的单例和另一个作为 GameManager 的单例，它们现在无法共存。你需要复制相关的代码并将逻辑粘贴到每个类中。

相反，制作一个通用版本的脚本，像这样：

```
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);
        }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);
        }
    }
}
```

这允许你将任何类变成一个单例。当你声明你的类时，只需继承通用的单例。例如，你可以通过这样声明，将一个名为 GameManager 的 MonoBehaviour 变成一个单例：

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

然后你可以随时引用公共静态的 GameManager.Instance，每当你需要它。

优点和缺点

单例模式与本指南中的其他模式不同，因为它们在多个方面都违反了 SOLID 原则。许多开发人员出于各种原因不喜欢它们：

- 单例需要全局访问：因为你将它们用作全局实例，它们可以隐藏许多依赖关系，使得故障难以排查。
- 单例使测试变得困难：单元测试必须彼此独立。因为单例可以改变场景中许多 GameObject 的状态，它们可能会干扰你的测试。
- 单例鼓励紧密耦合：本指南中的大部分模式都试图解耦依赖关系。单例则相反。紧密耦合使得重构变得困难。如果你改变一个组件，你可能会影响到任何与它连接的组件，导致代码不清晰。

反对单例的声音很多。如果你正在构建一个企业级的游戏，你期望在未来几年内进行维护，你可能会希望避开单例。

但是，许多游戏并不是企业级的应用程序。你不需要像商业软件那样不断地扩展它们。

事实上，单例提供了一些好处，你可能会发现它们很有吸引力，如果你正在构建一个不需要扩展性的小游戏：

- 单例相对容易学习：核心模式本身相当直接。
- 单例用户友好：要从另一个组件使用你的单例，只需引用公共和静态的实例。单例实例始终在你的场景中的任何对象需要时可用。
- 单例有良好的性能：因为你总是有全局访问静态的单例实例，你可以避免缓存 GetComponent 或 Find 操作的结果，这些操作往往比较慢。

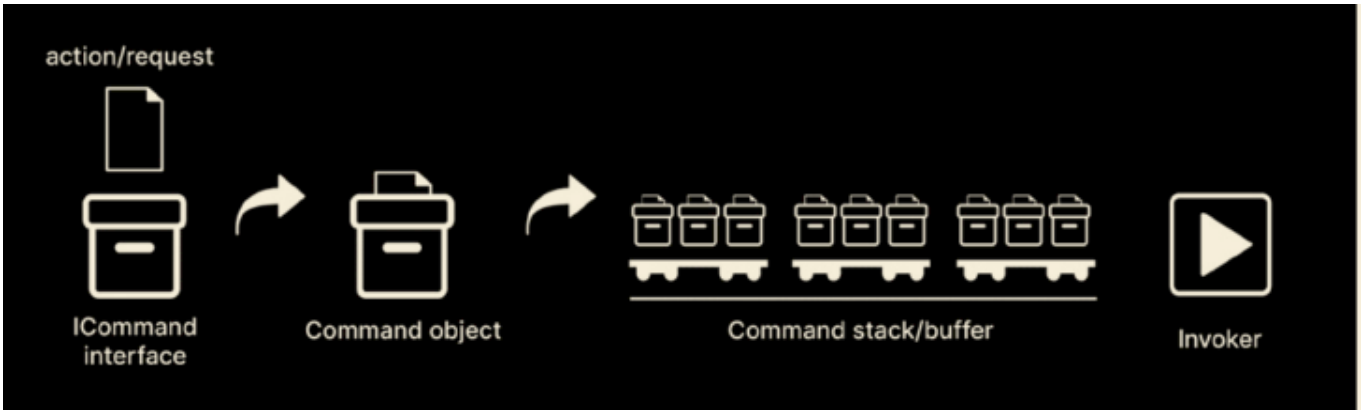
通过这种方式，你可以创建一个管理对象（例如，游戏流程管理器或音频管理器），它始终可以从场景中的每个其他 GameObject 访问。此外，如果你已经实现了对象池，你可以将你的池系统设计为一个单例，以便更容易地获取池对象。

如果你决定在你的项目中使用单例，要将它们保持在最小限度。不要随意使用它们。仅将单例留给那些可以从全局访问中受益的一小部分脚本。

COMMAND PATTERN

原型之一的 Gang of Four 模式，命令模式在你想要跟踪一系列特定操作时非常有用。如果你玩过一个使用撤销/重做功能或在列表中保存你的输入历史的游戏，你可能已经看到了命令模式的作用。想象一下一个策略游戏，用户可以在实际执行它们之前计划几个回合。那就是命令模式。

与直接调用方法不同，命令模式允许你将一个或多个方法调用封装为一个“命令对象”。



使用命令模式存储操作

将这些命令对象存储在队列或堆栈等集合中，可以让你控制它们的执行时机。这可以作为一个小缓冲区。然后，你可能可以延迟一系列的操作以便稍后回放，或者撤销它们。

要实现命令模式，你需要一个通用的对象来包含你的操作。这个命令对象将保存要执行的逻辑以及如何撤销它。

命令对象和命令调用者

有许多实现方法，但这里有一个使用接口的版本：

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

在这种情况下，每个游戏操作都将应用 ICommand 接口（你也可以用抽象类来实现）。

每个命令对象将负责自己的 Execute 和 Undo 方法。因此，向游戏中添加更多命令不会影响现有的任何命令。

你还需要另一个类来执行和撤销命令。创建一个 CommandInvoker 类。除了 ExecuteCommand 和 UndoCommand 方法外，它还有一个撤销堆栈来保存命令对象的序列。

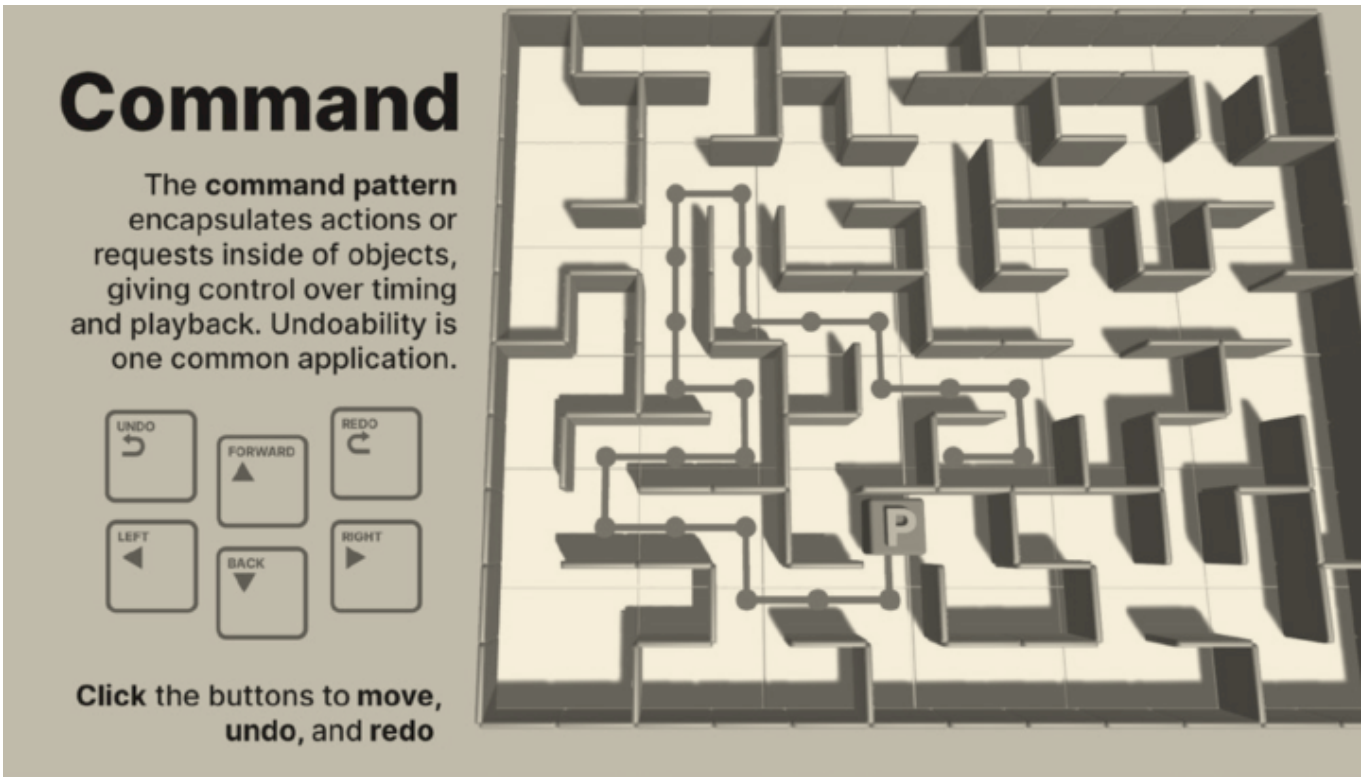
```
public class CommandInvoker
{
    private static Stack<ICommand> undoStack = new Stack<ICommand>();

    public static void ExecuteCommand(ICommand command)
    {
        command.Execute();
        undoStack.Push(command);
    }

    public static void UndoCommand()
    {
        if (undoStack.Count > 0)
        {
            ICommand activeCommand = undoStack.Pop();
            activeCommand.Undo();
        }
    }
}
```

示例：可撤销的移动

假设你想要在迷宫中移动你的角色。你可以创建一个 PlayerMover 来负责改变玩家的位置：

```
public class PlayerMover : MonoBehaviour
{
    [SerializeField] private LayerMask obstacleLayer;
    private const float boardSpacing = 1f;

    public void Move(Vector3 movement)
    {
        transform.position = transform.position + movement;
    }

    public bool IsValidMove(Vector3 movement)
    {
        return !Physics.Raycast(transform.position, movement, boardSpacing,
obstacleLayer);
    }
}
```

你将向 Move 方法中传入一个 Vector3，以引导玩家沿着四个罗盘方向移动。你还可以使用射线投射来检测适当 LayerMask 中的墙壁。当然，你想要应用到命令模式的实现与模式本身是分开的。



样本中的玩家移动

要遵循命令模式，捕获 PlayerMover 的 Move 方法作为一个对象。不要直接调用 Move，而是创建一个新的类，MoveCommand，实现 ICommand 接口：

```
public class MoveCommand : ICommand
{
    PlayerMover playerMover;
    Vector3 movement;

    public MoveCommand(PlayerMover player, Vector3 moveVector)
    {
        this.playerMover = player;
        this.movement = moveVector;
    }

    public void Execute()
    {
        playerMover.Move(movement);
    }

    public void Undo()
    {
        playerMover.Move(-movement);
```

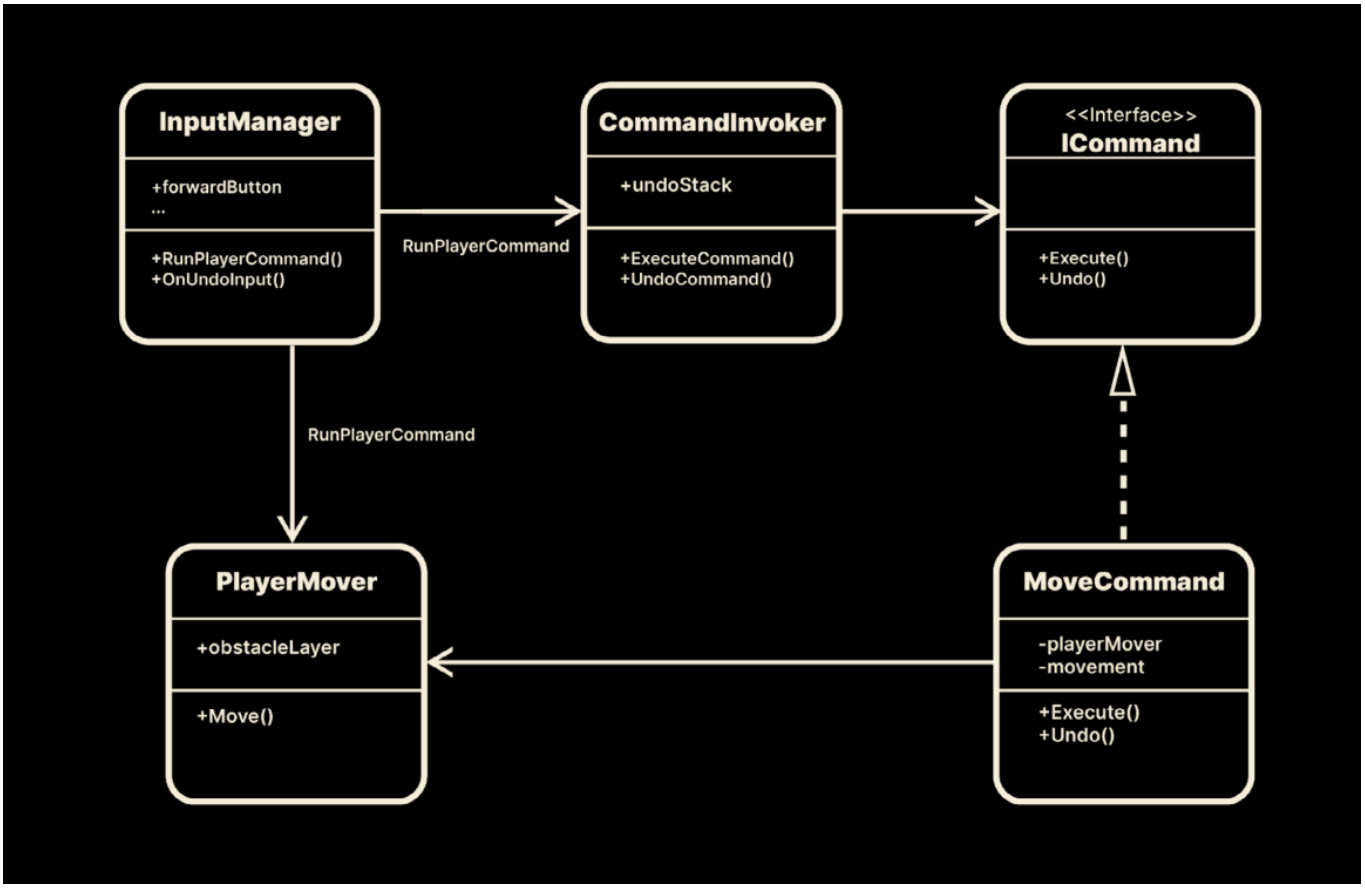
```
    }
}
```

ICommand 需要一个 Execute 方法来存储你想要完成的操作。你想要完成的任何逻辑都放在这里，所以用移动向量调用 Move。

ICommand 还需要一个 Undo 方法来恢复场景到以前的状态。在这种情况下，Undo 逻辑减去了移动向量，本质上是将玩家推到相反的方向。

MoveCommand 存储它需要执行的任何参数。通过构造函数进行设置。在这种情况下，你保存适当的 PlayerMover 组件和移动向量。

一旦你创建了命令对象并保存了所需的参数，就使用 CommandInvoker 的静态 ExecuteCommand 和 UndoCommand 方法传入你的 MoveCommand。这将运行 MoveCommand 的 Execute 或 Undo，并在撤销堆栈中跟踪命令对象。



InputManager不直接调用PlayerMover的Move方法。相反，添加一个额外的方法RunMoveCommand，以创建一个新的MoveCommand并将其发送到CommandInvoker。

```
private void RunPlayerCommand(PlayerMover playerMover, Vector3 movement)
{
```

```
    if (playerMover == null)
    {
        return;
    }

    if (playerMover.IsValidMove(movement))
    {
        ICommand command = new MoveCommand(playerMover, movement);
        CommandInvoker.ExecuteCommand(command);
    }
}
```

然后，设置UI按钮的各种onClick事件，以四个移动向量调用RunPlayerCommand。

查看示例项目以获取InputManager的实现细节，或使用键盘或游戏手柄设置您自己的输入。您的玩家现在可以导航迷宫。点击Undo按钮，您可以回退到起始方块。

优点和缺点：

实现可重播性或可撤销性就像生成命令对象的集合一样简单。您还可以使用命令缓冲区按照特定的控制顺序回放操作。

例如，想想一个格斗游戏，其中一系列特定的按钮点击触发一个组合动作或攻击。使用命令模式存储玩家操作使得设置这样的组合变得更加简单。

另一方面，命令模式引入了更多的结构，就像其他设计模式一样。您必须决定这些额外的类和接口是否为在应用程序中部署命令对象提供了足够的好处。

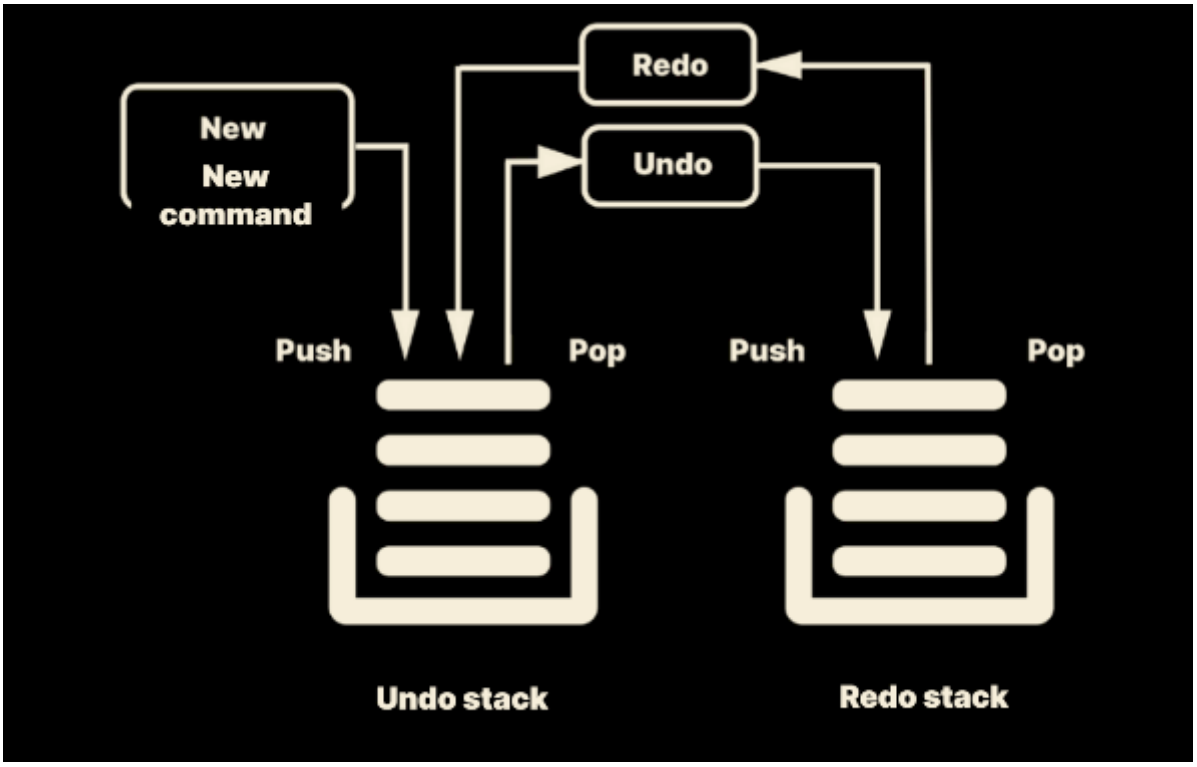
改进

一旦您掌握了基础知识，您就可以根据上下文影响命令的时序，并按顺序或反向播放它们。

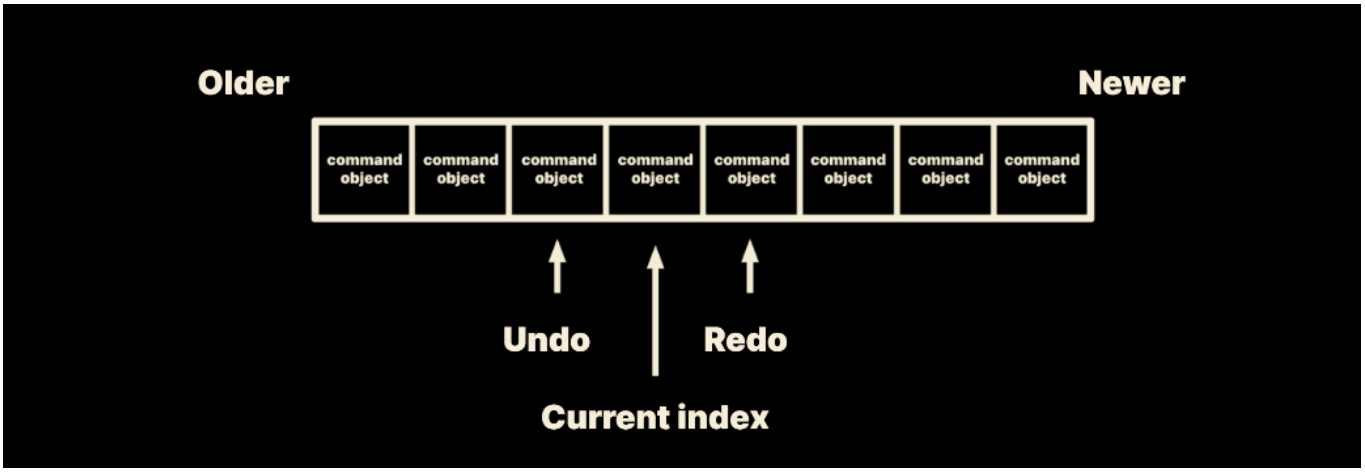
在融入命令模式时，请考虑以下几点：

- 创建更多的命令：示例项目只包含一种类型的命令对象，即MoveCommand。您可以创建任意数量的实现ICommand的命令对象，并使用CommandInvoker跟踪它们。
- 添加重做功能只是添加另一个堆栈的问题：当您撤消一个命令对象时，将其推到跟踪重做操作的单独堆栈上。这样，您可以快速循环浏览撤消历史记录或重做这些操作。当用户调用一

个全新的移动时，清除重做堆栈（您可以在附带的示例项目中找到一个实现）。



- 为您的命令对象缓冲区使用不同的集合：如果您想要先进先出（FIFO）的行为，队列可能更方便。如果您使用列表，跟踪当前活动的索引；活动索引之前的命令是可以撤销的。索引之后的命令是可以重做的。



列表或其他集合充当命令缓冲区。

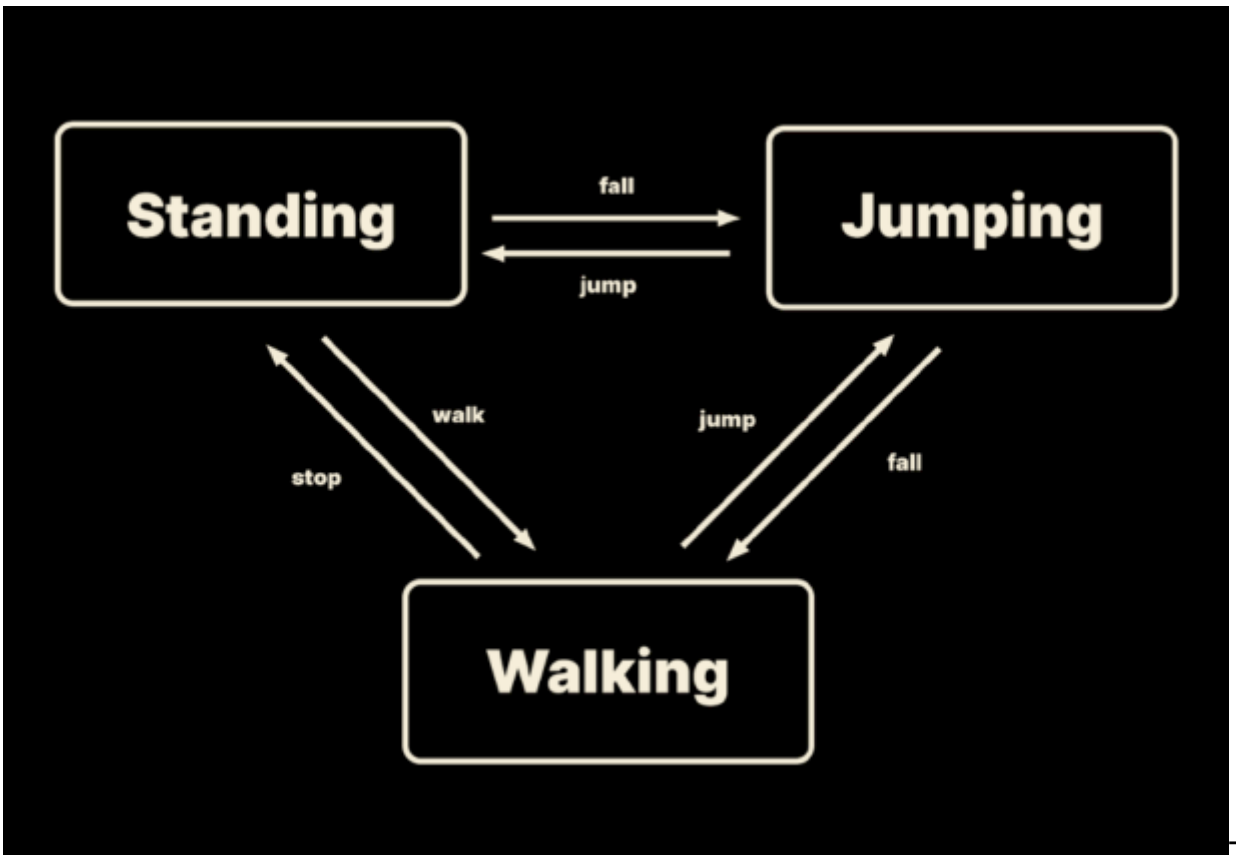
- 限制堆栈的大小：撤销和重做操作可能会迅速失控。将堆栈限制为最后一些命令。
- 将任何必要的参数传递给构造函数：这有助于封装MoveCommand示例中看到的逻辑。CommandInvoker与其他外部对象一样，不看到命令对象的内部工作，只调用Execute或Undo。在调用构造函数时，给命令对象提供任何需要工作的数据。

STATE PATTERN

想象你正在构建一个可玩的角色。在某一时刻，角色可能站在地面上。移动控制器，它看起来像是在跑或走路。按跳跃按钮，角色跳入空中。几帧后，它降落并重新进入其空闲、站立的位置。

状态和状态机

游戏是互动的，它们迫使我们跟踪许多在运行时改变的系统。如果你绘制一个表示角色不同状态的图表，你可能会得到这样的东西：



一个简单的状态图

它类似于一个流程图，但有一些不同：

- 图表由多个状态组成（如：空闲/站立、行走、跑步、跳跃等），在给定的时间只有一个当前状态是活动的。
- 每个状态可以基于运行时的条件触发到另一个状态的转换。
- 当转换发生时，输出状态成为新的活动状态。

这个图表描述了所谓的有限状态机（FSM）。在游戏开发中，一个典型的用例是跟踪游戏角色或道具的内部状态。

要在代码中描述一个基本的FSM，你可能会使用一个enum和一个switch语句。

```
public enum PlayerControllerState
{
    Idle,
    Walk,
    Jump
}

public class UnrefactoredPlayerController : MonoBehaviour
{
    private PlayerControllerState state;

    private void Update()
    {
        GetInput();

        switch (state)
        {
            case PlayerControllerState.Idle:
                Idle();
                break;
            case PlayerControllerState.Walk:
                Walk();
                break;
            case PlayerControllerState.Jump:
                Jump();
                break;
        }
    }

    private void GetInput()
    {
        // process walk and jump controls
    }

    private void Walk()
    {
        // walk logic
    }
```

```
    private void Idle()
    {
        // idle logic
    }

    private void Jump()
    {
        // jump logic
    }
}
```

这样做是可行的，但PlayerController脚本很快就会变得混乱。增加更多的状态和复杂性每次都需要我们重新查看PlayerController脚本的内部。

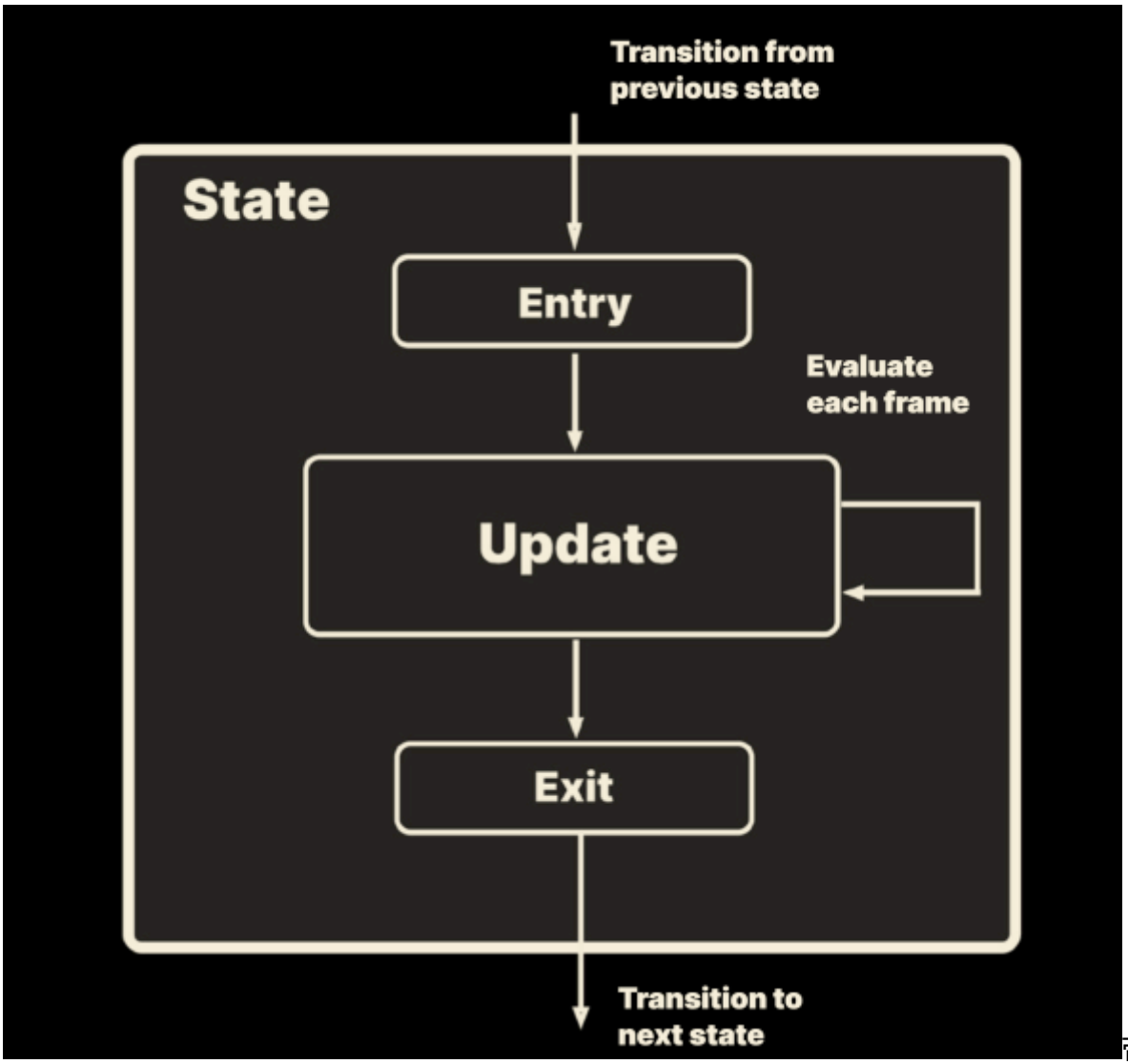
示例：简单的状态模式

幸运的是，状态模式可以帮助你重新组织逻辑。根据原始的四人帮，状态模式解决了两个问题：

- 当对象的内部状态发生变化时，它应该改变其行为。
- 状态特定的行为是独立定义的。添加新的状态不会影响现有状态的行为。

虽然上面的例子中UnrefactoredPlayerController类可以跟踪状态变化，但它不满足第二个问题。当你添加新的状态时，你希望尽量减少对现有状态的影响。相反，你可以将状态作为一个对象封装。

想象每个状态的结构如下：



带有入口、出口和更新的封装状态

这里你进入状态并循环每一帧，直到一个条件导致控制流退出。为了实现这个模式，创建一个接口，IState:

```
public interface IState
{
    void Enter();
    // code that runs when we first enter the state

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

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

你的游戏中的每个具体状态都将实现IState接口：

- 入口：首次进入状态时执行的逻辑。
- 更新：此逻辑每帧运行（有时称为执行或滴答）。你可以进一步细分Update方法，如MonoBehaviour那样，使用FixedUpdate进行物理计算，LateUpdate等。Update中的任何功能在检测到触发状态更改的条件之前每帧都会运行。
- 出口：在离开状态并转换到新状态之前运行的代码。

您需要为每个实现IState的状态创建一个类。在示例项目中，为WalkState、IdleState和JumpState分别设置了独立的类。

另一个类，StateMachine，将管理如何进入和退出状态的控制流。对于上面的三个示例状态，StateMachine可能是这样的：

```
[Serializable]
public class StateMachine
{
    public IState CurrentState { get; private set; }

    public WalkState walkState;
    public JumpState jumpState;
    public IdleState idleState;

    public void Initialize(IState startingState)
    {
        CurrentState = startingState;
        startingState.Enter();
    }

    public void TransitionTo(IState nextState)
    {
        CurrentState.Exit();
        CurrentState = nextState;
        nextState.Enter();
    }

    public void Update()
    {
        if (CurrentState != null)
        {
            CurrentState.Update();
```

```
        }
    }
}
```

为了遵循模式，StateMachine引用了其管理下的每个状态的公共对象（在这种情况下，是walkState、jumpState和idleState）。因为StateMachine不从MonoBehaviour继承，所以使用构造函数来设置每个实例：

```
public StateMachine(PlayerController player)
{
    this.walkState = new WalkState(player);
    this.jumpState = new JumpState(player);
    this.idleState = new IdleState(player);
}
```

你可以传递任何需要的参数给构造函数。在示例项目中，每个状态都引用了一个PlayerController。然后你使用它每帧更新每个状态（参见下面的IdleState示例）。

关于StateMachine的注意事项：

- Serializable属性允许我们在检查器中显示StateMachine（及其公共字段）。另一个MonoBehaviour（如PlayerController或EnemyController）可以使用StateMachine作为一个字段。
- CurrentState属性是只读的。StateMachine本身不显式地设置这个字段。一个外部对象，如PlayerController，可以调用Initialize方法来设置默认状态。
- 每个State对象确定自己调用TransitionTo方法更改当前活动状态的条件。在设置StateMachine实例时，您可以传递任何必要的依赖项（包括State Machine本身）到每个状态。
  在示例项目中，PlayerController已经包含了对StateMachine的引用，所以您只需要传递一个player参数。

每个状态对象将管理其自己的内部逻辑，您可以制作尽可能多的状态来描述您的GameObject或组件。每一个都有自己实现IState的类。遵循SOLID原则，增加更多的状态对任何先前创建的状态的影响都是最小的。

这是IdleState的一个示例：

```
public class IdleState : IState
{
```

```
    private PlayerController player;

    public IdleState(PlayerController player)
    {
        this.player = player;
    }

    public void Enter()
    {
        // code that runs when we first enter the state
    }

    public void Update()
    {
        // Here we add logic to detect if the conditions exist to
        // transition to another state
        ...
    }

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

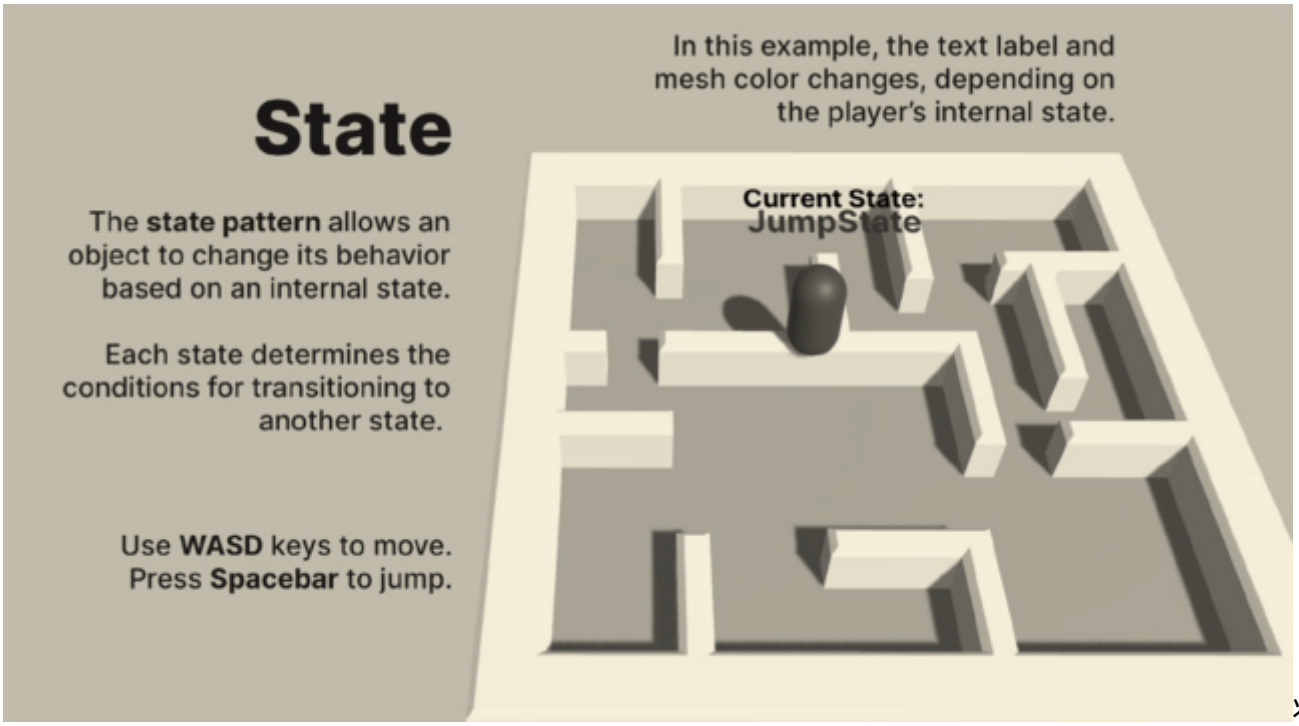
再次使用构造函数传入PlayerController对象。在这个例子中，此player包含了对StateMachine的引用以及Update逻辑所需的所有其他内容。idleState监视Character Controller的速度或跳跃状态，然后适当地调用StateMachine的TransitionTo方法。

也请查看WalkState和JumpState的示例项目实现。与其拥有一个切换行为的大型类，不如让每个状态都有自己的更新逻辑。这样，状态可以彼此独立地工作。

优缺点

状态模式可以帮助您在为对象设置内部逻辑时遵循SOLID原则。每个状态相对较小，只跟踪转换到另一个状态的条件。遵循开放-封闭原则，您可以添加更多的状态，而不影响现有的状态，避免繁琐的switch或if语句。

另一方面，如果你只有几个状态要跟踪，额外的结构可能是过度的。这种模式可能只有在你预期你的状态增长到一定的复杂度时才有意义。

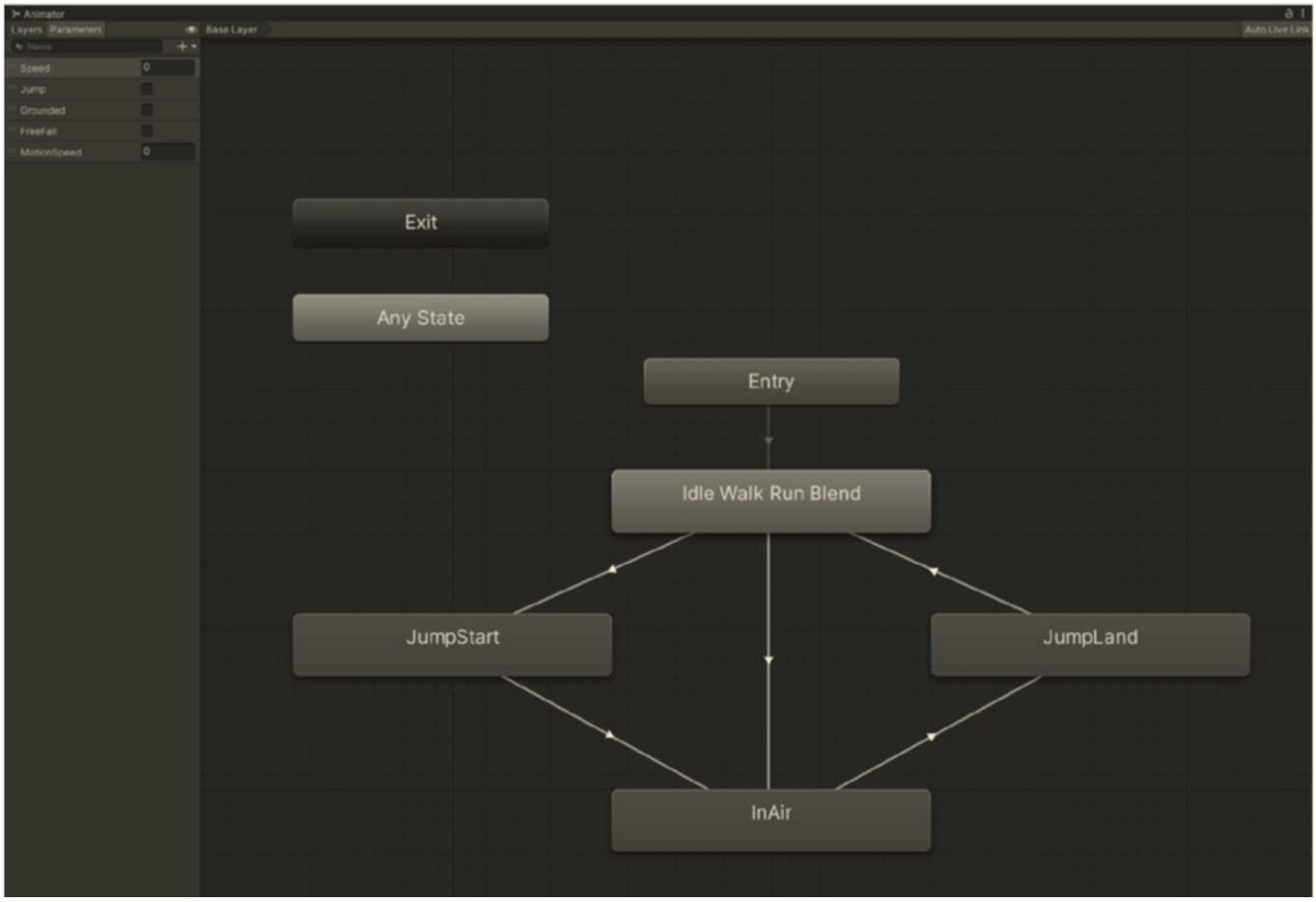


状态模式示例

改进

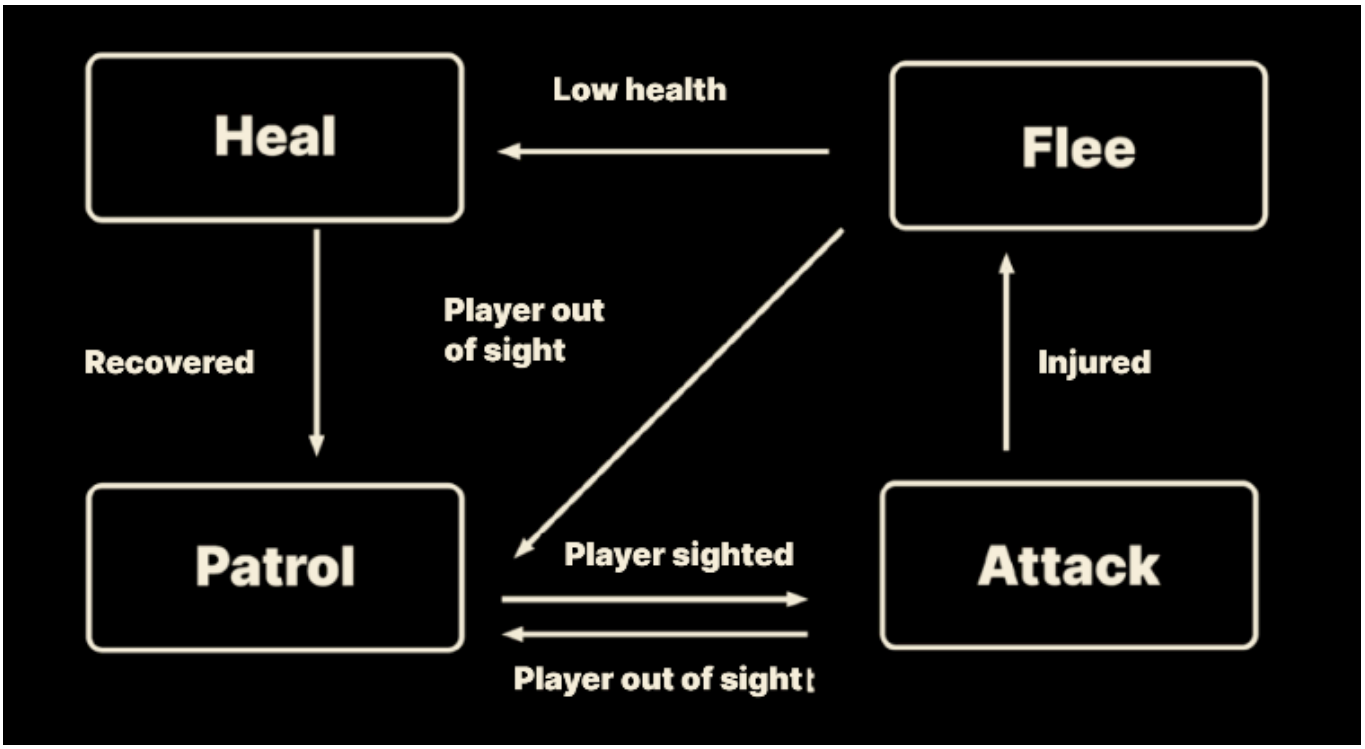
示例项目中的胶囊会改变颜色，UI会随着玩家的内部状态更新。在真实世界的例子中，你可能有更复杂的效果伴随状态的变化：

- 结合状态模式和动画：状态模式的一个常见应用是动画。玩家或敌人角色在宏观层面上经常被表示为基元（一个胶囊）。然后，你可以有动画几何体反应于内部状态的变化，所以游戏角色可以看起来像是在跑步、跳跃、游泳、攀爬等。
  如果你已经使用过Unity的Animator窗口，你会注意到它的工作流与状态模式很好地匹配。每个动画片段占据一个状态，每次只有一个状态是活动的。



Animator状态图的一个示例：与StateMachine的结构进行比较。

- 添加事件：为了将状态变化通知到外部对象，你可能想要添加事件（参见观察者模式）。在进入或退出一个状态时有一个事件可以通知相关的监听器，并让它们在运行时响应。
- 添加层次结构：当你开始用状态模式描述更复杂的实体时，你可能想要实现分层的状态机。一些状态不可避免地会相似；例如，如果玩家或游戏角色在地面上，无论处于WalkingState还是RunningState，都可以蹲下或跳跃。
  如果你实现一个SuperState，你可以将公共行为放在一起。然后使用继承，你可以在子状态中重写任何特定的内容。例如，你可能首先声明一个GroundedState。然后你可以从那里继承RunningState或WalkingState。
- 实现简单的AI：有限状态机在生成基本的敌人AI中也很有用。基于FSM构建NPC大脑的方法可能是这样的：



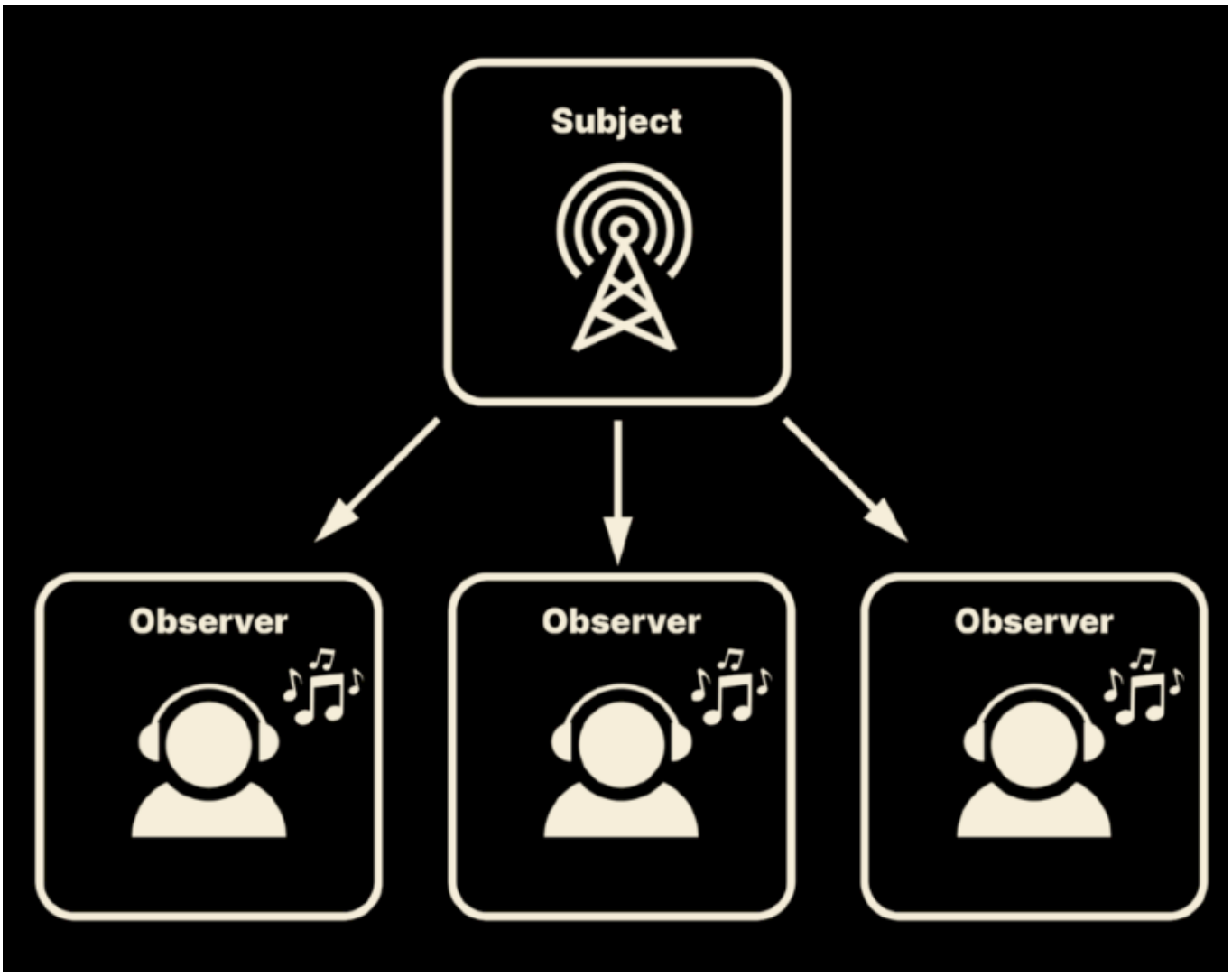
基于状态模式的简单AI

这里再次展示了状态模式在完全不同的上下文中的工作原理。每个状态代表一个动作，如攻击、逃跑或巡逻。每次只有一个状态是活动的，每个状态决定其转换到下一个状态。

OBSERVER PATTERN

在运行时，游戏中可能会发生各种各样的事情。当你消灭一个敌人时会发生什么？当你收集一个能量提升或完成一个任务时呢？你经常需要一个机制，使一些对象在不直接引用它们的情况下通知其他对象，从而避免创建不必要的依赖。

观察者模式是这种问题的一个常见解决方案。它允许你的对象进行通信，但使用“一对多”的依赖关系保持松散耦合。当一个对象更改状态时，所有依赖对象都会自动得到通知。这类似于一个广播塔广播给许多不同的听众。



观察者模式就像一个广播塔。主体向观察者广播。

进行广播的对象被称为主体。正在监听的其他对象被称为观察者。

这种模式松散地解耦了主体，它实际上不真正知道观察者或关心它们在接收到信号后做什么。虽然观察者对主体有依赖，但观察者本身不知道彼此。

事件

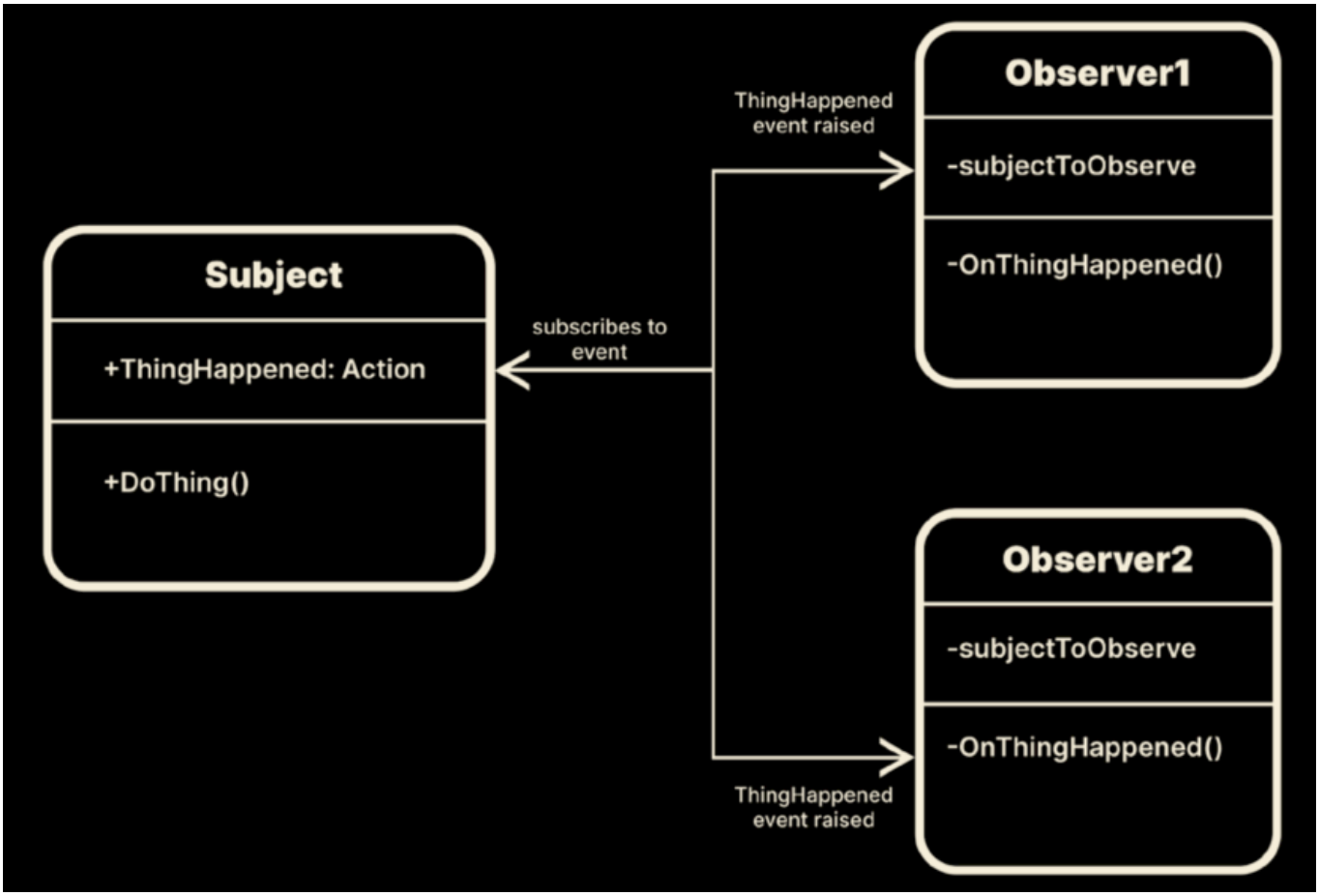
观察者模式如此普遍，以至于它被内置到C#语言中。你可以设计自己的主题-观察者类，但通常是不必要的。还记得关于重复发明轮子的观点吗？C#已经使用事件实现了该模式。

一个事件只是一个通知，表示某事已经发生。它涉及到几个部分：

- 发布者（主体）基于代理创建一个事件，建立一个特定的函数签名。事件只是主体在运行时将执行的某些操作（例如，受到伤害、点击按钮等）。
- 订阅者（观察者）然后各自制作一个称为事件处理程序的方法，该方法必须与代理的签名匹配。

- 每个观察者的事件处理程序订阅发布者的事件。您可以根据需要让尽可能多的观察者加入订阅。所有的观察者都会等待事件触发。
- 当发布者在运行时发出一个事件发生的信号时，你说它引发了事件。这反过来又调用了订阅者的事件处理程序，它们在响应中执行自己的内部逻辑。

通过这种方式，你可以使许多组件对主体的单一事件做出反应。如果主体表示一个按钮被点击，观察者可以播放一个动画或声音、触发一个过场动画或保存一个文件。它们的反应可以是任何事情，这就是为什么你经常会发现观察者模式被用来在对象之间发送消息。



主体引发事件通知观察者。

示例：简单的主体和观察者

例如，您可以定义一个基本的主题/发布者，如下所示：

```
using UnityEngine;
using System;

public class Subject : MonoBehaviour
{
```

```
    public event Action ThingHappened;

    public void DoThing()
    {
        ThingHappened?.Invoke();
    }
}
```

这里，你继承了MonoBehaviour以更容易地附加到GameObject，但这不是必需的。

虽然你可以自由地定义自己的自定义代理，但在大多数情况下System.Action都可以工作。如果您需要随事件发送参数，使用Action代理并将它们作为List传递到尖括号内（最多16个参数）。

ThingHappened是实际的事件，主体在DoThing方法中调用它。

为了监听事件，你可以构建一个示例Observer类。这里你继承了MonoBehaviour以方便使用，但这不是必需的。

```
public class Observer : MonoBehaviour
{
    [SerializeField]
    private Subject subjectToObserve;

    private void OnThingHappened()
    {
        // any logic that responds to event goes here
        Debug.Log("Observer responds");
    }

    private void Awake()
    {
        if (subjectToObserve != null)
        {
            subjectToObserve.ThingHappened += OnThingHappened;
        }
    }

    private void OnDestroy()
    {
        if (subjectToObserve != null)
```

```
        {
            subjectToObserve.ThingHappened -= OnThingHappened;
        }
    }
}
```

将此组件附加到GameObject上，并在检查器中引用subjectToObserver以便监听ThingHappened事件。

OnThingHappened方法可以包含观察者在响应事件时执行的任何逻辑。开发者经常添加前缀“On”来表示事件处理程序（只需使用您的样式指南中的命名约定）。

在Awake或Start中，你可以使用+=运算符订阅事件。这将观察者的OnThingHappened方法与主题的ThingHappened结合在一起。

如果任何事情运行主体的DoThing方法，那么就会引发事件。然后，观察者的OnThingHappened事件处理程序会自动调用并打印调试语句。

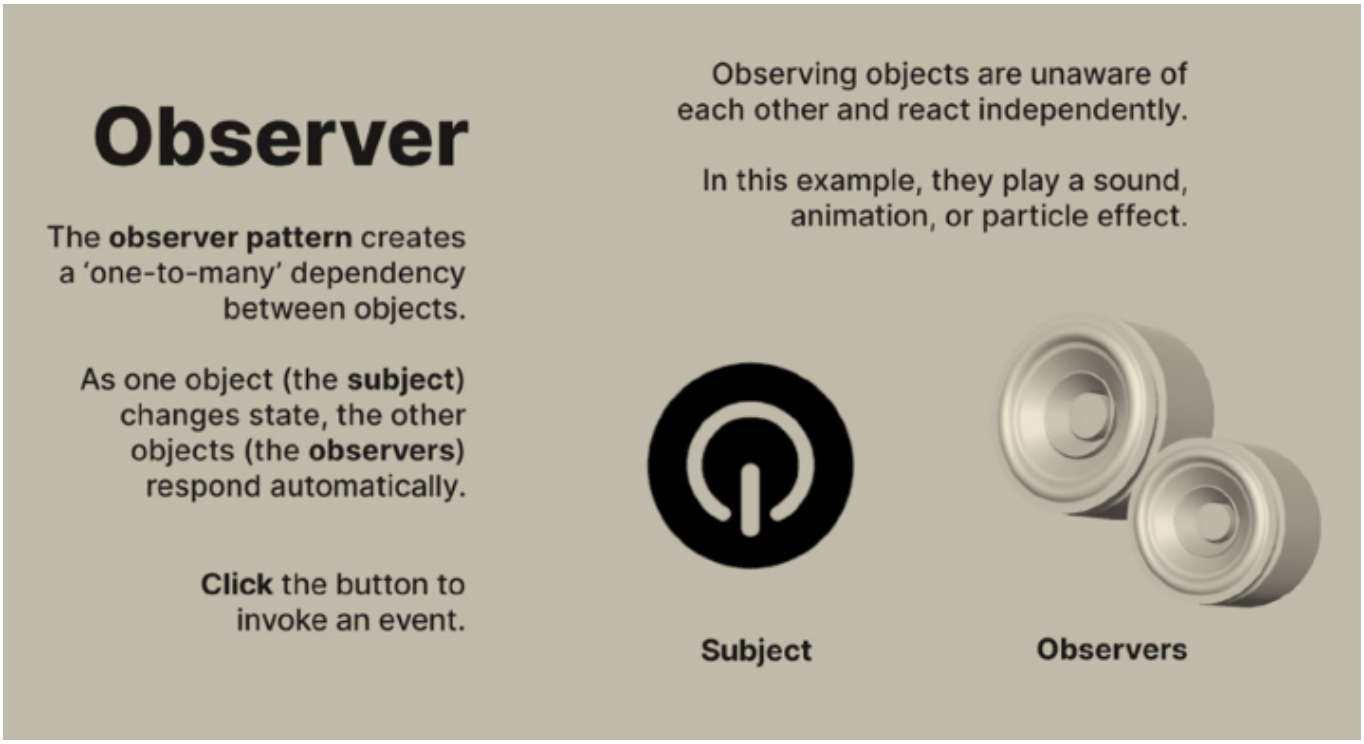
注意：如果在观察者仍然订阅了ThingHappened的情况下在运行时删除或移除观察者，那么调用该事件可能会导致错误。因此，在MonoBehaviour的OnDestroy方法中使用-=运算符取消事件订阅是很重要的。

你可以将观察者模式应用到游戏过程中几乎发生的一切事情。例如，你的游戏可以在玩家每次消灭一个敌人或收集一个物品时引发一个事件。如果你需要一个跟踪分数或成就的统计系统，观察者模式可以让你创建一个，而不影响原始的游戏代码。

许多Unity应用程序将事件应用于：

- 目标或目的
- 获胜/输掉的条件
- PlayerDeath、EnemyDeath或Damage
- 物品拾取
- 用户界面

主体只需要在适当的时候触发一个事件，然后任意数量的观察者都可以订阅。



观察者样本场景

在示例项目中，ButtonSubject允许用户使用鼠标按钮调用Clicked事件。其他带有AudioObserver和ParticleSystemObserver组件的GameObject然后可以以它们自己的方式响应事件。

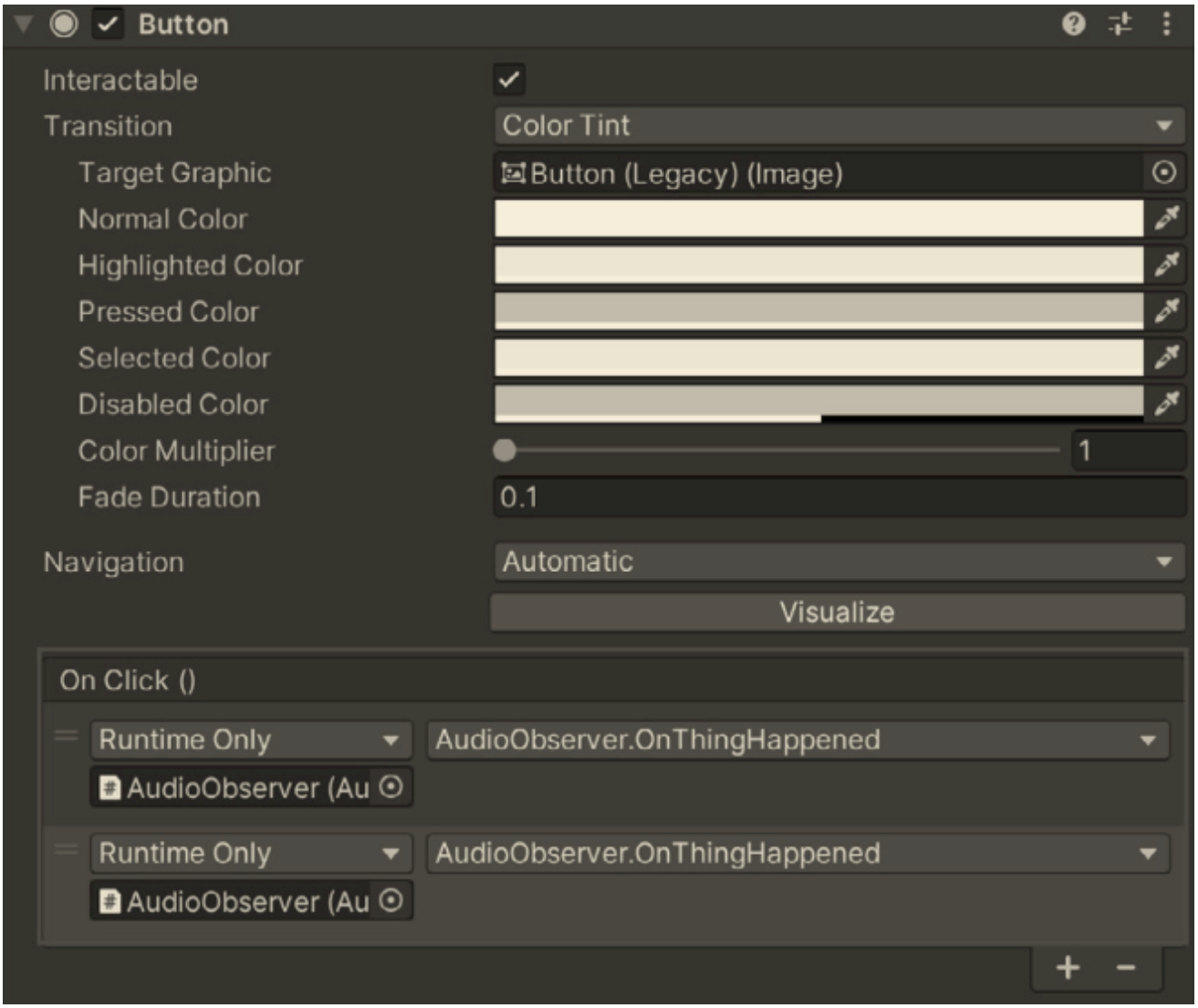
确定哪个对象是“主体”和哪个对象是“观察者”只是根据用途来变化的。任何触发事件的事物都可以作为主体，任何响应事件的事物都是观察者。同一个GameObject上的不同组件可以是主体或观察者。甚至同一个组件在一个上下文中可以是主体，在另一个上下文中可以是观察者。

例如，示例中的AnimObserver在点击时为按钮添加了一点移动。尽管它是ButtonSubject GameObject的一部分，但它作为观察者。

UnityEvents 和 UnityActions

Unity还包括了一个独立的UnityEvents系统，它使用来自UnityEngine.Events API的UnityAction代理。

UnityEvents为观察者模式提供了一个图形界面。如果你已经使用过Unity的UI系统（例如，创建一个UI按钮的OnClick事件），那么你已经有了一些经验。



UnityEvents有图形组件供您设置

在这个例子中，按钮的OnClick事件调用并触发了两个AudioObservers的OnThingHappened方法的响应。因此，你可以在没有代码的情况下设置主体的事件。

如果你想让设计师或非程序员创建游戏事件，UnityEvents非常有用。但是，请注意，它们可能比System命名空间中的等效事件或操作慢。

在考虑使用UnityEvents和UnityActions时，权衡性能与使用方法。查看Unity Learn上的Create a Simple Messaging System with Events模块以获取一个示例。

优缺点

实现一个事件增加了一些额外的工作，但确实提供了优势：

- **观察者模式帮助解耦您的对象：** 事件发布者不需要知道关于事件订阅者本身的任何事情。而不是在一个类和另一个类之间创建直接的依赖关系，主体和观察者在保持一定程度的分离的

同时进行通信。

- **你不必自己构建它：** C#包括了一个已建立的事件系统，你可以使用System.Action代理而不是定义自己的代理。或者，Unity还包括UnityEvents和UnityActions。
- **每个观察者实现其自己的事件处理逻辑：** 这样，每个观察对象都保持了响应所需的逻辑。这使得调试和单元测试更加容易。
- **它非常适合用户界面：** 你的核心游戏代码可以与你的UI逻辑分开。然后，你的UI元素可以监听特定的游戏事件或条件，并适当地作出响应。MVP和MVC模式为此目的使用观察者模式。

观察者模式也有以下注意事项：

- **它增加了额外的复杂性：** 像其他模式一样，创建事件驱动的架构确实需要更多的 upfront 设置。此外，删除主题或观察者时要小心。确保在OnDestroy中取消观察者的注册。
- **观察者需要一个引用到定义事件的类：** 观察者仍然依赖于发布事件的类。使用一个处理所有事件的静态EventManager（如下）可以帮助解开对象之间的关系。
- **性能可能是一个问题：** 事件驱动的架构增加了额外的开销。大场景和许多GameObject可能会妨碍性能。

改进

虽然这里只介绍了观察者模式的基本版本，但你可以扩展它以满足你的游戏应用的所有需求。

在设置观察者模式时，请考虑以下建议：

- **使用ObservableCollection类：** C#提供了一个动态ObservableCollection来跟踪特定的变化。它可以在添加、删除项目或刷新列表时通知您的观察者。
- **作为参数传递一个唯一的实例ID：** 层次结构中的每个GameObject都有一个唯一的Instance ID。如果你触发一个可能适用于多个观察者的事件，将唯一的ID传递到事件中（使用Action类型）。然后只有在GameObject匹配唯一ID时才在事件处理程序中运行逻辑。
- **创建一个静态的EventManager：** 因为事件可以驱动你的大部分游戏，所以许多Unity应用程序使用一个静态或单例的EventManager。这样，你的观察者可以作为主体引用游戏事件的中心来源，使设置更容易。FPS Microgame有一个很好的实现，它实现了自定义的GameEvents并包括了添加或删除监听器的静态帮助方法。Unity Open Project还展示了一个使用ScriptableObjects来传递UnityEvents的游戏架构。它使用事件来播放音频或加载新的场景。
- **创建一个事件队列：** 如果你的场景中有很多对象，你可能不希望一次性触发所有的事件。想象一下，当你调用一个事件时，一千个对象发出的声音会是多么的嘈杂。将观察者模式与命令模式结合起来，允许你将你的事件封装到一个事件队列中。然后，你可以使用一个命令缓

冲区来逐个播放事件，或者根据需要选择性地忽略它们（例如，如果你有一个可以同时发出声音的对象的最大数量）。

观察者模式在模型视图表示器（MVP）架构模式中有很重的分量，这在下一章中将详细介绍。

MODEL VIEW PRESENTER(MVP)

模型视图控制器（MVC）

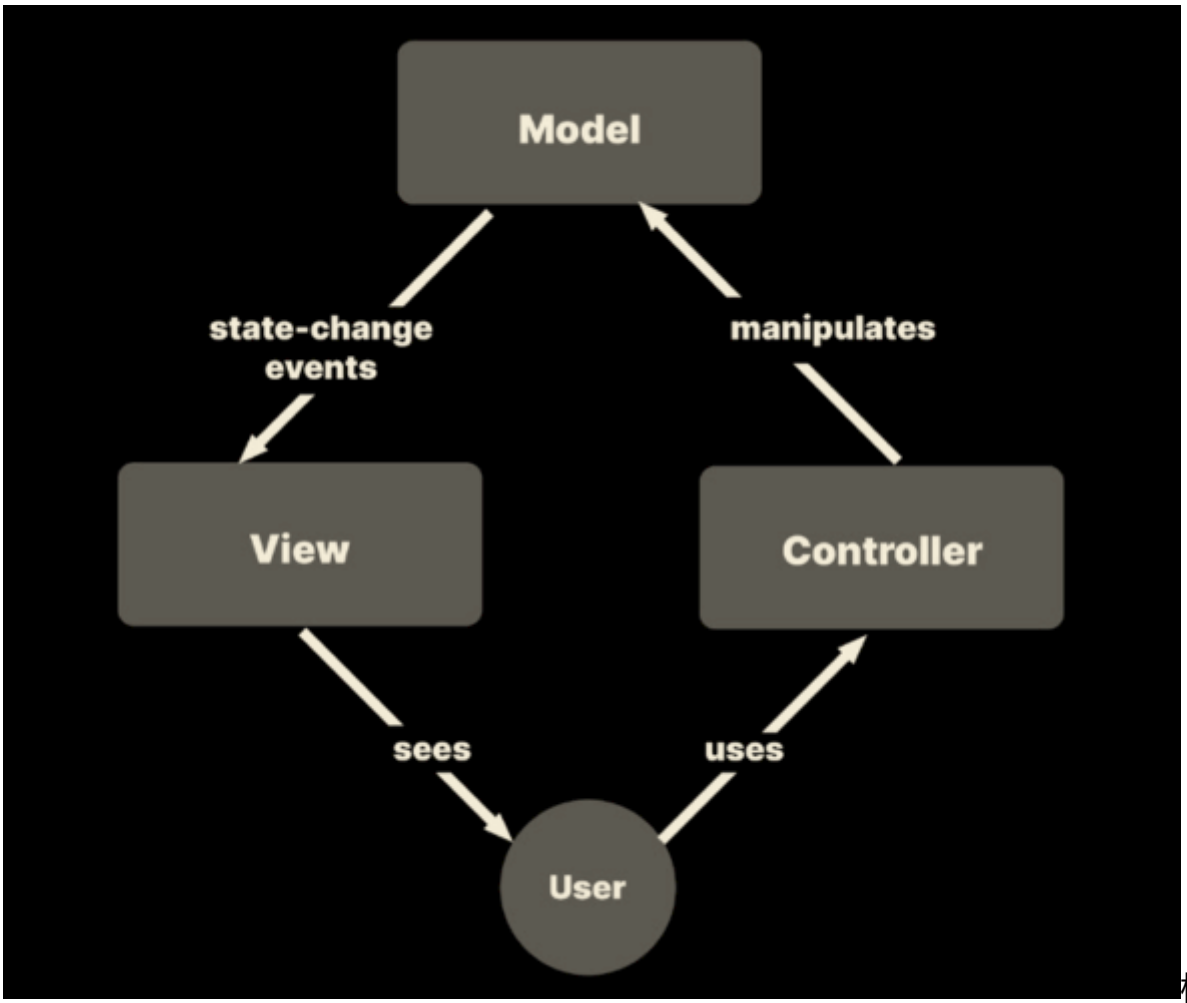
模型视图控制器（MVC）是开发用户界面时常用的设计模式之一。

MVC的基本思想是将您的软件的逻辑部分与数据和展示部分分离。这有助于减少不必要的依赖关系，可能减少意面代码。

MVC设计模式

顾名思义，MVC模式将您的应用程序分为三层：

- **模型 存储数据：** 模型严格地是一个数据容器，保持值。它不执行游戏逻辑或进行计算。
- **视图 是界面：** 视图格式化并在屏幕上呈现您的数据的图形表示。
- **控制器 处理逻辑：** 可以将其视为大脑。它处理游戏数据，并计算运行时的值如何变化。



模型、视图和控制器

这种关注点分离也明确定义了这三个部分如何相互交互。模型管理应用程序数据，而视图将数据显示给用户。控制器处理输入并对游戏数据进行任何决策或计算。然后它将结果发送回模型。

因此，控制器本身不包含任何游戏数据。视图也不包含。MVC设计限制了每个层次的作用。一个部分保持数据，另一个部分处理数据，最后一个部分将数据显示给用户。

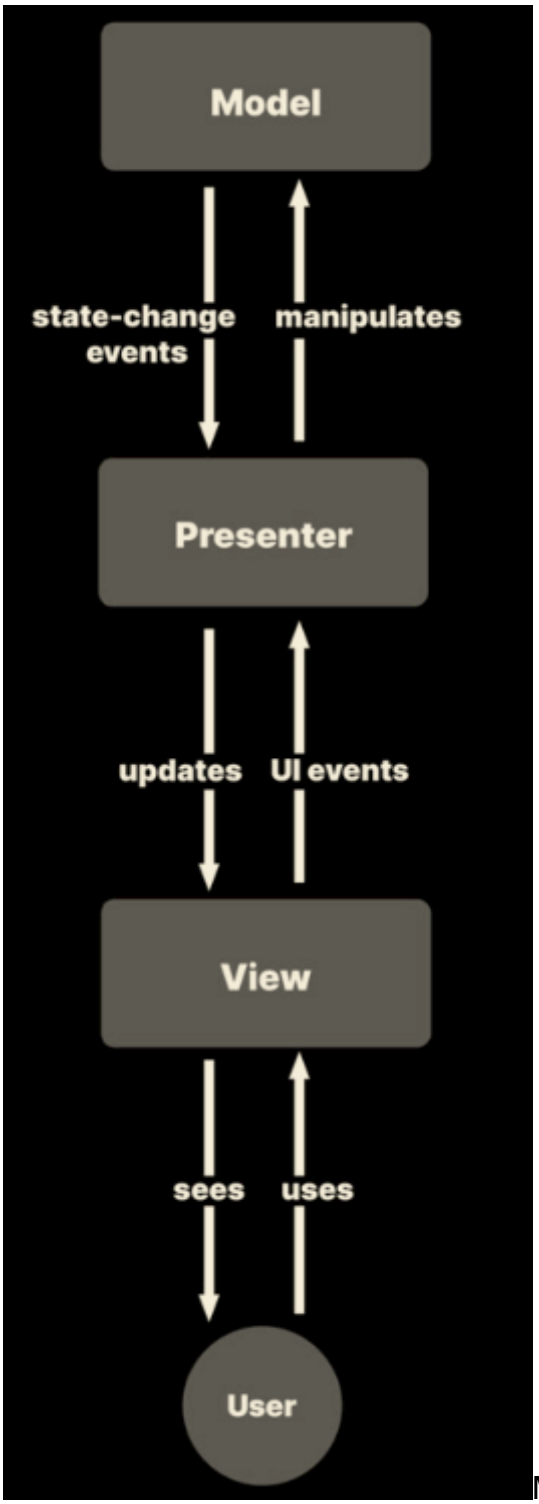
表面上，您可以将其视为单一职责原则的扩展。每个部分都有一个任务，并做得很好，这是MVC架构的一个优点。

Model View Presenter （MVP）和Unity

在使用MVC开发Unity项目时，现有的UI框架（UI Toolkit或Unity UI）自然地充当视图。因为引擎为您提供了一个完整的用户界面实现，所以您不需要从头开始开发单独的UI组件。

但是，遵循传统的MVC模式将需要视图特定的代码在运行时监听模型数据的任何更改。

虽然这是一个有效的方法，但许多Unity开发者选择使用一个MVC的变体，其中控制器充当中介。在这里，视图不直接观察模型。相反，它这样做：



MVP：MVC的一个变种

这种MVC的变种被称为模型视图展示设计，或MVP。MVP仍然保留了使用三个明确的应用层分离关注点的特点。但是，它稍微改变了每个部分的职责。

在MVP中，展示器（在MVC中称为控制器）充当其他层之间的中介。它从模型中检索数据，然后为视图中的显示格式化它。MVP切换了哪个层处理输入。而不是控制器，视图负责处理用户输入。

注意事件和观察者模式如何参与此设计。用户可以与Unity UI的按钮、切换和滑块组件进行交互。视图层通过UI事件将这个输入发送回展示器，展示器反过来操作模型。模型的状态更改事件

告诉展示器数据已更新。展示器将修改后的数据传递给视图，视图刷新UI。

示例：健康界面

为了正式化一个MVP示例，想象一个简单的系统来显示角色或物品的健康状况。您可以将所有内容放入一个混合数据和UI的类中，但这种方式不会很好地扩展。随着功能的增加，扩展它会变得更加复杂。

相反，您可以以更MVP为中心的方式重写您的健康组件。将您的脚本分为Health和HealthPresenter。Health组件可能是这样的：

```
public class Health : MonoBehaviour
{
    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();
    }

    public void Restore()
    {
        currentHealth = maxHealth;
        UpdateHealth();
    }

    public void UpdateHealth()
    {
        HealthChanged?.Invoke();
    }
}
```

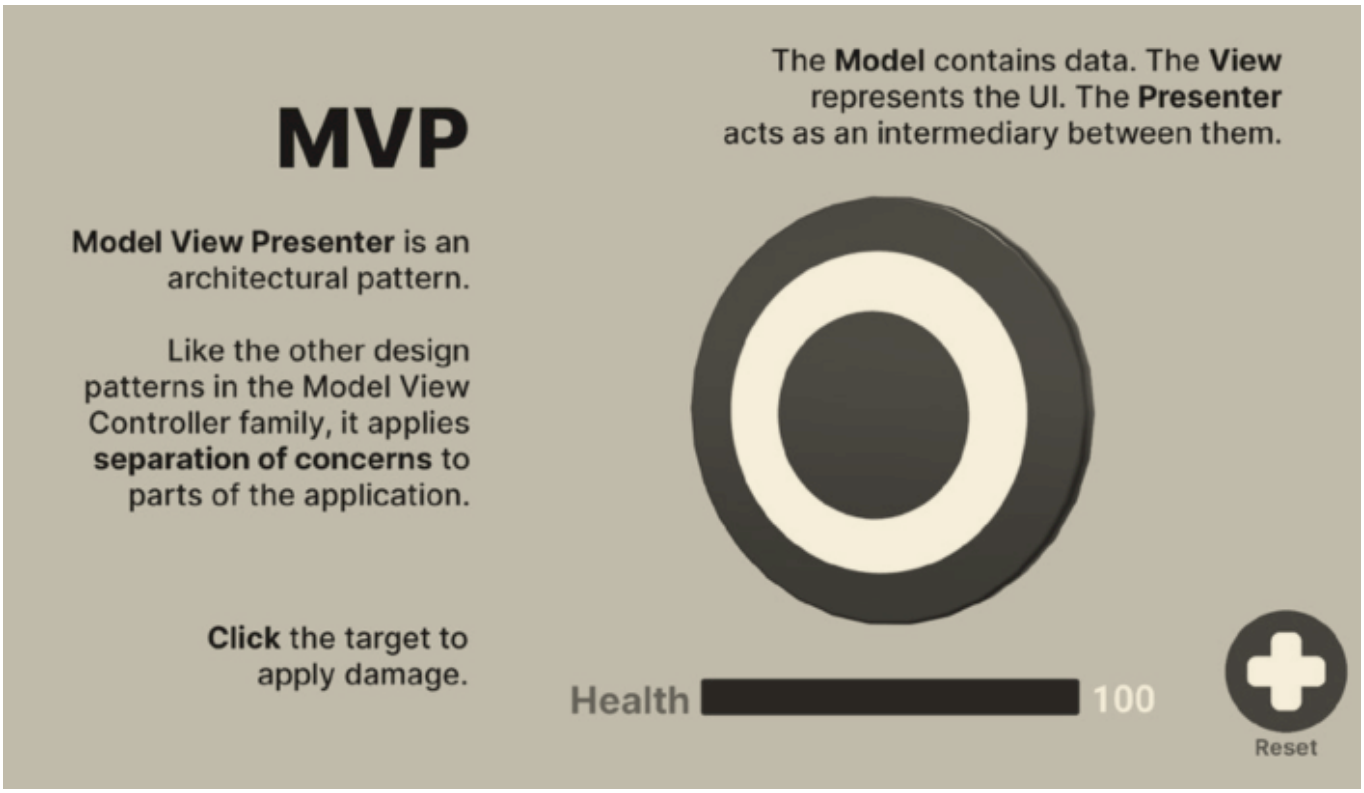
在这个版本中，Health作为模型。它存储实际的健康值，并在该值每次更改时调用一个事件，HealthChanged。Health不包含游戏逻辑，只有增加和减少数据的方法。

但是，大多数对象不会操作Health本身。您将为该任务保留一个HealthPresenter：

```
public class HealthPresenter : MonoBehaviour
{
    [SerializeField]
    Health health;

    [SerializeField]
    Slider healthSlider;

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

    private void OnDestroy()
    {
        if (health != null)
```

```
        {
            health.HealthChanged -= OnHealthChanged;
        }
    }

    public void Damage(int amount)
    {
        health?.Decrement(amount);
    }

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

    public void Reset()
    {
        health?.Restore();
    }

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

        if (healthSlider != null && health.MaxHealth != 0)
        {
            healthSlider.value = (float) health.CurrentHealth /
(float)health.MaxHealth;
        }
    }

    public void OnHealthChanged()
    {
        UpdateView();
    }
}
```

其他GameObject将需要使用HealthPresenter来修改健康值，使用Damage、Heal和Reset。HealthPresenter通常会等待使用UpdateView更新用户界面，直到Health引发其HealthChanged事件。这在设置模型中的值需要一段时间（例如，将值保存到磁盘或存储在数据库中）时很有用。



使用MVP的示例健康界面

在示例项目中，用户可以点击来损坏目标对象或使用按钮重置健康。这些通知HealthPresenter（调用Damage或Reset），而不是直接更改Health。当Health引发事件并通知HealthPresenter其值已更改时，UI文本和UI滑块更新。

优缺点

MVP（和MVC）真正为较大的应用程序发光。如果您的游戏需要一个大小适中的团队来开发，并且您希望在启动后长时间维护它，您可能会从以下方面受益：

- **平滑的工作分割：** 因为您已经将视图从展示器中分离出来，所以可以几乎独立地开发和更新您的用户界面。这样，您可以在专业的开发者之间划分您的劳动。您的团队中有专家前端开发者吗？让他们负责视图。他们可以独立于其他人工作。
- **使用MVP和MVC简化单元测试：** 这些设计模式将游戏逻辑与用户界面分离。因此，您可以模拟对象与您的代码一起工作，而无需实际进入编辑器的播放模式。这可以节省大量的时间。

- **可读的代码可以维护：** 您倾向于使用这种设计模式制作较小的类，这使它们更易于阅读。较少的依赖关系通常意味着您的软件的破裂点较少，可能隐藏的错误也较少。

尽管MVC和MVP在Web开发或企业软件中很受欢迎，但通常在应用程序达到足够的大小和复杂性之前，好处并不明显。在实施这两种模式之前，您需要考虑以下几点：

- **您需要提前计划：** 与本指南中描述的其他模式不同，MVC和MVP是较大的架构模式。要使用其中之一，您需要按职责分割您的类，这需要一些组织和更多的前期工作。
- **不是您的Unity项目中的所有内容都适合该模式：** 在“纯”MVC或MVP实现中，渲染到屏幕的任何内容实际上都是视图的一部分。并非每个Unity组件都可以轻松地在数据、逻辑和接口之间进行分割（例如，MeshRenderer）。此外，简单的脚本可能不会从MVC/MVP中获得很多好处。您需要行使判断，看看哪里可以从模式中获得最大的好处。通常，您可以让单元测试指导您。如果MVC/MVP可以促进测试，请考虑将它们用于应用程序的某个方面。否则，不要尝试将模式强加到您的项目上。

结论

如果您是软件模式的新手，我们希望本指南能帮助您了解Unity开发中您可能遇到的一些最常见的模式。

无论是用于生成预设的工厂还是用于AI的状态模式，当需要时请随手使用这些技巧。识别何时以及如何应用设计模式可以帮助您应对下一个Unity挑战。当然，不要迷失于强制适应特定模式；不使用模式和使用模式同样重要。

正确应用时，设计模式可以加速您的工作流程，并为经常出现的问题提供一个优雅的解决方案。然后，您可以专注于最重要的事情：为玩家创造有趣且独特的体验。

所以，虽然您不需要重新发明轮子，但您绝对可以为其增添自己的特色。

其他设计模式

本指南只是计算和游戏开发中几种众所周知的设计模式的小部分样本。虽然我们不会深入探讨它们的细节，但以下是对您可能觉得有用的其他一些模式的简要概述：

- 适配器：这为两个无关实体之间提供了一个接口（也称为包装器），使它们可以协同工作。
- 享元：如果您有大量的对象，可以在基类中共享常见属性以节省资源。例如，设计森林时，将树的所有通用属性存储在基Tree类中。然后，您不需要在子类中重复它们（例如，PineTree、MapleTree等）。
- 双缓冲：这允许您在计算完成时维护两组数组数据。然后，您可以在处理另一组数据时显示一组数据，这对于程序化模拟（例如，细胞自动机）或仅将内容呈现到屏幕上都很有用。
- 脏标记：这种技术允许您在游戏中的某些内容发生更改时设置一个布尔值，但昂贵的操作尚未运行（例如，保存到磁盘或运行物理模拟）。
- 解释器/字节码：如果您想添加modding支持或允许非程序员扩展您的游戏，您可以创建一个简化的语言，用户可以在外部文本文件中进行编辑。然后，字节码组件可以将该解释语言转换为C#游戏代码。
- 子类沙盒：如果您有具有不同行为的类似对象，您可以在父类中将这些行为定义为受保护的。然后，子类可以混合并匹配以创建新的组合。
- 类型对象：如果您有许多种类的GameObject，而不是为每一个都创建子类，可以在单个抽象或父类中定义所有可能的行为。在单独的数据文件中区分单个对象的特殊特征（例如ScriptableObject），可以在不更改代码的情况下进行定制。例如，这允许您创建一个从同一类派生的看似不同的物品的库存。游戏设计师可以自定义数据文件以使每个物品都是独一无二的（例如，RPG的武器），而不需要程序员的帮助。
- 数据局部性：如果您优化数据，使其在内存中存储得更高效，您可以获得性能的回报。替换类为结构可以使您的数据更易于缓存。Unity的ECS和DOTS架构实施了这种模式。
- 空间划分：对于大型场景和游戏世界，使用特殊结构按位置组织您的GameObject。Grid、Trie（Quadtree、Octree）和二叉搜索树都是帮助您更有效地划分和搜索的技术。
- 装饰器：这允许您在不更改其现有结构的情况下为对象添加职责。装饰器可以赋予特殊能力或修改一个GameObject，例如，添加额外属性到武器而不需要更改基础武器类。
- 外观：这提供了一个简单、统一的接口到一个更复杂的系统。如果您有一个带有单独的AI、动画和声音组件的GameObject，您可能会在这些组件周围添加一个包装类（想象一个玩家控制器类管理PlayerInput、PlayerAudio等）。这个外观隐藏了原始组件的细节并简化了使用。
- 模板方法：这种模式将算法的确切步骤推迟到子类中。例如，您可以在抽象类中定义算法或数据结构的粗略骨架，但允许子类覆盖某些部分而不改变算法的整体结构。
- 策略：这种行为模式（也称为策略模式）帮助您定义算法的家族并将每一个封装在一个类中。这使得每种算法（策略）在运行时都是可互换的。例如，如果您创建了一个寻路系统，您可以使用策略模式来定义多种算法（A*、Dijkstra的最短路径等），并根据上下文在游戏过程中交换它们。
- 组合：使用此结构设计模式将对象组织成树结构，然后像处理单个对象一样处理结果结构。您从简单和复合元素（叶子和容器）构造树。每个元素都实现相同的接口，因此您可以在整棵树上递归地运行相同的行为。