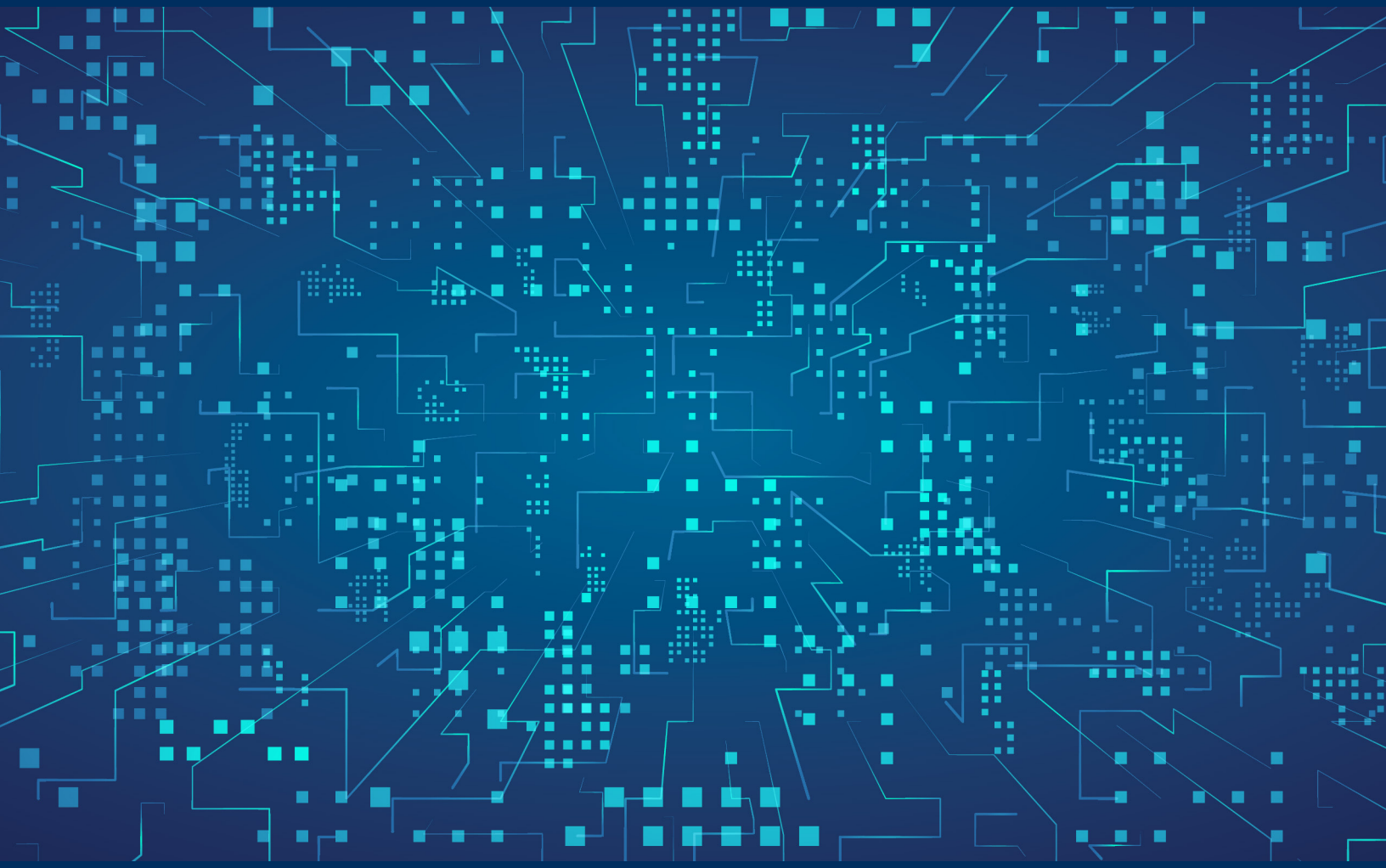
UNITY FOR GAMES

전자책



게임 프로그래밍 패턴으로 코딩 스킬 업그레이드

2021 LTS 에디션

Unity®

Contents

디자인 패턴 소개

Unity를 사용할 때는 새로운 것을 발명하느라 시간을 낭비할 필요가 없습니다. 필요한 기능이 있다면 이미 누군가 제작해 두었을 가능성이 높습니다.

개발자들은 누구나 소프트웨어를 디자인하는 과정에서 유사한 문제에 직면합니다. 직접 조언을 구할 수는 없더라도, 디자인 패턴을 통해 과거의 개발자들이 어떤 결정을 내렸는지 학습할 수는 있습니다.

디자인 패턴을 사용하면 소프트웨어 엔지니어링에서 발생하는 일반적인 문제를 해결할 수 있습니다. 단순히 복사하여 코드에 붙여 넣으면 되는 완성된 솔루션은 아니지만, 필요할 때마다 꺼내 사용할 수 있는 도구와 같다고 보면 됩니다. 다른 패턴에 비해 직관적인 패턴도 있습니다.

본 가이드에서는 Unity 개발에서 잘 알려진 디자인 패턴을 정리해 소개합니다. 본 가이드의 예시는 사용자가 더 쉽게 이해할 수 있도록 간소화되었으며 기술적인 전문 용어도 적게 사용되었지만, 시작하려면 우선 기본적인 C# 실무 지식을 갖추어야 합니다.

아직 디자인 패턴에 생소하거나 학습한 지 오래된 사용자를 위해 본 가이드는 게임 개발에 적용할 수 있는 일반적인 시나리오도 함께 제공합니다. 다른 객체 지향 언어(Java, C++ 등)에서 C#으로 전환하는 사용자라면 그러한 샘플을 통해 Unity에서 구체적으로 패턴을 적용하는 방법을 확인할 수 있습니다.

디자인 패턴은 본질적으로 아이디어일 뿐이며, 모든 상황에 항상 적용되는 것은 아닙니다. 하지만 적절하게 사용할 경우 확장이 필요한 더 큰 규모의 애플리케이션을 제작하는 데 도움이 됩니다. 프로젝트에 통합하면 코드 가독성을 높일 수 있을 뿐만 아니라 코드 베이스를 깔끔하게 개선할 수 있습니다. 패턴을 사용하며 경험을 쌓다 보면 어느새 개발 프로세스가 단축되는 시점이 찾아올 것입니다.

그러면 더 이상 불필요한 시간 낭비를 줄이고 본격적인 창작에 돌입할 수 있습니다.

도움을 주신 분들

본 가이드를 작성한 윌머 린은 영화 및 TV 업계에서 15년 이상의 경력을 쌓았고 현재는 인디 게임 개발자 겸 교육자로 활동하고 있는 3D 및 시각 효과 아티스트입니다. 시니어 테크니컬 콘텐츠 마케팅 매니저인 토마스 크로그 야콥센과 시니어 Unity 엔지니어인 피터 안드레아센, 스콧 빌라스도 큰 도움을 주었습니다.

본 가이드와 KISS 원칙을 활용하는 방법

본 가이드는 코드의 구상과 정리에 대한 새로운 방안을 전달하기 위해 작성되었습니다. 여기에서 중점적으로 소개하는 소프트웨어 디자인 패턴 몇 가지는 Unity 개발에 맞게 조정되었습니다.

가이드에 포함된 샘플 프로젝트는 컨텍스트에 따른 몇 가지 코드를 제시합니다. 해당되는 씬을 사용해 코드의 디자인 패턴과 기반이 되는 원칙을 살펴보세요.

하지만 예시를 검토할 때, 모든 문제에 '정답'이란 없다는 사실을 항상 염두에 두어야 합니다. 샘플 코드는 다양한 해결책 중 하나일 뿐입니다.

확실치 않을 때는 본 가이드의 내용을 KISS 원칙, 즉 'Keep it simple, stupid' (단순하게 유지하기)에 따라 숙지하세요. 복잡도는 필요한 경우에만 높여야 합니다.

디자인 패턴을 적용할 때에는 대가가 따르기 마련인데, 예를 들면 구조체를 더 많이 유지해야 하거나 초기 설정이 더 많이 필요할 수도 있습니다.
수고를 더 들일 가치가 있을지 미리 생각해 보세요.

패턴이 특정한 문제에 적용되는지 확실치 않은 경우에는 더 자연스럽게 적용할 상황이 될 때까지 기다리는 편이 좋을 수 있습니다. 새롭거나 참신하다는 이유만으로 패턴을 사용하지는 말고, 필요할 때만 사용하세요.

그러면 디자인 패턴이 제대로 역할을 수행하고 소프트웨어 개발에 도움이 될 수 있습니다.

이제 시작하겠습니다.

SOLID 원칙

패턴에 대해 이야기하기 전에, 우선 패턴이 작동하는 방식에 영향을 주는 디자인 원칙 몇 가지를 살펴보겠습니다.

SOLID는 소프트웨어 디자인의 다섯 가지 핵심 원칙을 머리글자어로 만든 용어입니다.

— 단일 책임(Single responsibility)

— 개방-폐쇄(Open-closed)

— 리스코프 치환(Liskov substitution)

— 인터페이스 분리(Interface segregation)

— 종속성 역전(Dependency inversion)

하나씩 개념을 살펴보고, SOLID 원칙을 이용해 더 이해하기 쉽고, 유연하며, 유지 관리가 용이한 코드를 작성하는 방법을 알아보겠습니다.

단일 책임 원칙

클래스를 변경해야 한다면 그 이유는 오직 단일 책임 원칙이어야 합니다.

SOLID에서 가장 중요한 첫 번째 원칙은 SRP(단일 책임 원칙)로, 모듈, 클래스 또는 함수가 오직 한 가지만 책임지며 로직의 특정 부분만 캡슐화할 것을 명시합니다.

단일 구조의(monolithic) 클래스를 만들기보다는 여러 개의 작은 클래스로 프로젝트를 조합하세요. 클래스와 메서드는 짧을수록 설명과 이해, 구현이 쉽습니다.

Unity를 어느 정도 사용해 봤다면 이미 익숙한 개념일 것입니다. 게임 오브젝트를 만들면 더 작고 다양한 컴포넌트가 오브젝트 내부에 포함됩니다. 포함될 수 있는 컴포넌트는 다음과 같습니다.

— 3D 모델에 대한 참조가 있는 MeshFilter

— 모델 표면이 화면에 표시되는 방식을 제어하는 Renderer

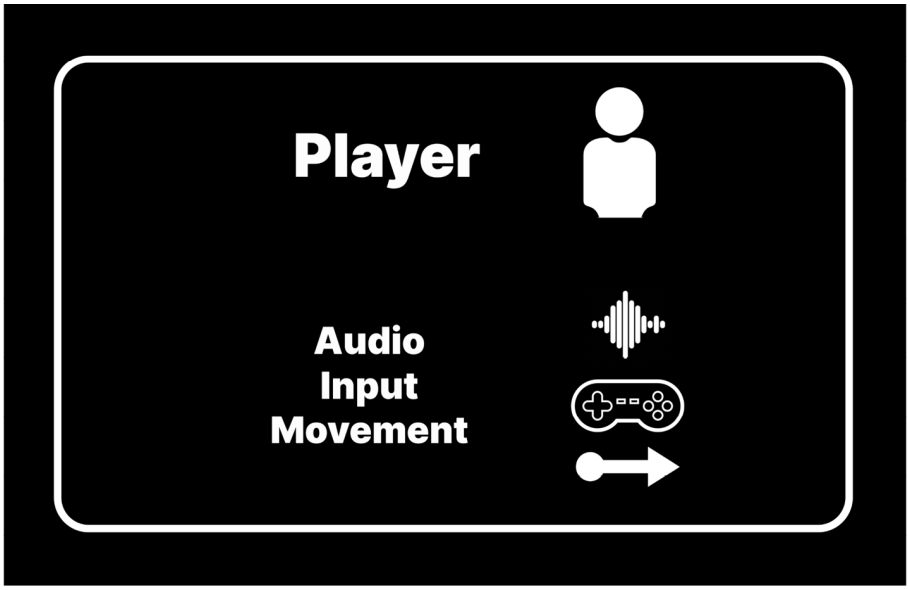
— 스케일, 회전, 위치 정보가 있는 Transform 컴포넌트

— 물리 시뮬레이션과 상호 작용하는 데 필요한 Rigidbody

각 컴포넌트는 한 가지 작업을 올바르게 수행합니다. 게임 오브젝트로 전체 씬을 만들 수 있으며, 컴포넌트 간의 상호 작용으로 게임이 구동됩니다.

스크립팅된 컴포넌트도 같은 방식으로 구성합니다. 각 컴포넌트를 명확하게 이해할 수 있도록 디자인하세요. 그런 다음 여러 컴포넌트가 함께 작동하여 복잡한 동작을 만들도록 합니다.

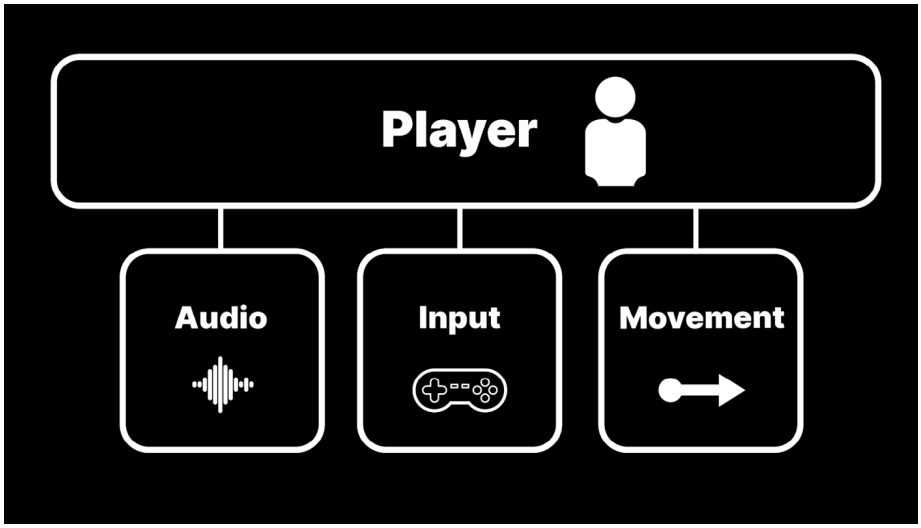
단일 책임을 무시하면 다음과 같이 작동하는 커스텀 컴포넌트를 만들게 될 수 있습니다.



여러 기능을 수행하는 플레이어 스크립트

```
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();
    }
}
```

위 코드에서 UnrefactoredPlayer 클래스는 너무 많은 기능을 수행합니다. 플레이어가 무언가에 충돌하면 소리를 재생하고, 입력을 관리하고, 이동을 처리합니다. 지금은 비교적 짧은 클래스지만 프로젝트를 진행하다 보면 점점 유지하기 어려워질 것입니다. Player 클래스를 더 작은 여러 클래스로 분할해 보세요.



단일 책임 원칙을 따른 클래스로 리팩터링된 Player

```
[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 스크립트가 여전히 스크립팅된 다른 컴포넌트를 관리할 수 있지만, 각 클래스는 오직 한 가지 역할만 수행합니다. 이렇게 디자인하면 코드를 더 쉽게 수정할 수 있으며, 특히 시간이 지나며 프로젝트 요구 사항이 바뀌는 상황에서는 더욱 유용합니다.

하지만, 단일 책임 원칙이라도 합당한 상식선에서 적용해야 합니다. 하나의 메서드만으로 클래스를 만드는 극단적인 간소화는 피하세요.

단일 책임 원칙을 따라 작업할 때 염두에 둘 만한 목표는 다음과 같습니다.

— **가독성:** 클래스가 짧으면 읽기 쉽습니다. 엄격하고 직관적인 규칙은 없지만 많은 개발자가 라인의 수를 200~300개 정도로 제한합니다. 본인 또는 팀 차원에서 어느 정도를 짧다고 규정할지 원칙을 정하세요. 정해진 한도를 초과하면 더 작게 리팩터링할 것인지 결정하세요.

— **확장성:** 작은 클래스에서 상속하기가 더 쉽습니다. 의도치 않은 기능 장애를 걱정할 필요 없이 클래스를 수정하거나 대체하세요.

— **재사용성:** 게임의 다른 부분에 재사용할 수 있도록 클래스를 작은 모듈형으로 디자인하세요.

리팩터링할 때는 코드를 어떻게 재구성해야 자신과 팀원에게 도움이 될지 생각해 보세요. 초반에 약간의 노력을 더 들이면 많은 문제를 미연에 방지할 수 있습니다.

단순함의 중요성

단순성은 소프트웨어 디자인에서 자주 다루는 주제이며 신뢰성을 높이기 위한 전제 조건이기도 합니다. 소프트웨어가 제작 단계에서 변경이 이뤄져도 대응할 수 있도록 디자인되었나요? 앞으로 애플리케이션을 확장하고 유지 관리할 수 있나요?

본 가이드에서 소개하는 디자인 패턴과 원칙 중 상당수가 단순성을 강화하는 데 도움이 됩니다. 그러한 디자인 패턴과 원칙으로 코드의 확장성, 유연성, 가독성을 높일 수 있습니다. 하지만 추가 작업과 계획이 필요합니다. '단순함'과 '쉬움'은 동의어가 아닙니다.

패턴을 사용하지 않아도 같은 기능을 더 빠르게 만들 수 있지만, 빠르고 쉬운 작업이 반드시 단순한 결과물로 이어지지는 않습니다. 단순하게 만들면 결과물의 집중도가 높아집니다. 하나의 작업만 수행하도록 디자인하고, 다른 작업으로 지나치게 복잡도를 높이지 마세요.

리치 히키의 강의 효율성을 이끌어내는 단순함의 힘(영문)에서 더 나은 소프트웨어를 만드는 데 단순성이 중요한 이유를 확인해 보세요.

개방-폐쇄 원칙

SOLID 디자인에서 OCP(개방-폐쇄 원칙)는 클래스가 확장에는 개방적이되 수정에는 폐쇄적이어야 한다고 명시합니다. 원본 코드를 수정하지 않고도 새로운 동작을 생성할 수 있도록 클래스를 구조화하세요.

대표적인 예시가 셰이프의 영역을 계산하는 클래스입니다. 직사각형과 원 영역을 반환하는 메서드가 있는 AreaCalculator라는 클래스를 만들 수 있습니다.

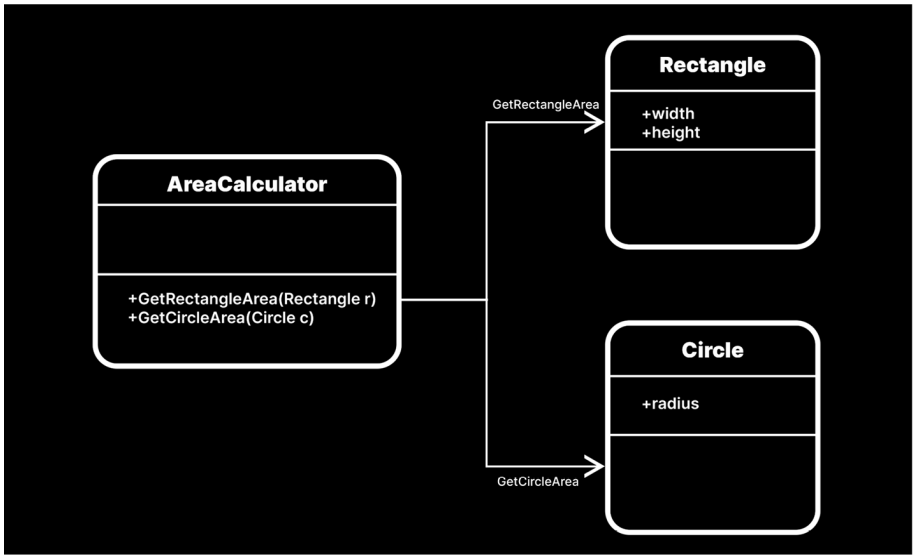
영역을 계산하기 위해 Rectangle 클래스에는 Width와 Height가 있습니다.
Circle 클래스에는 Radius와 파이 값만 필요합니다.

```
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;
}
```

이 코드도 충분히 기능을 수행하지만, AreaCalculator에 더 많은 셰이프를 추가하려면 각각의 새로운 셰이프를 위한 메서드를 생성해야 할 것입니다. 나중에 여기에 오각형이나 팔각형을 보내야 한다면 어떻게 할까요? 셰이프를 20개 더 추가해야 한다면 어떨까요? AreaCalculator 클래스는 지나치게 커져 통제 불능 상태에 빠질 것입니다.

Shape라는 기본 클래스를 만들고 셰이프를 처리할 메서드를 하나 만들 수도 있습니다. 하지만 그렇게 하려면 각 셰이프 형식을 처리하도록 로직 안에 여러 개의 if 문이 있어야 합니다. 그렇게 구현하면 확장성이 떨어집니다.

원본 코드(AreaCalculator의 내부)를 수정하지 않고, 새로운 셰이프를 사용할 수 있도록 확장을 위한 프로그램을 여는 것이 좋습니다. 현재 AreaCalculator는 작동하지만, 개방-폐쇄 원칙을 위반합니다.



AreaCalculator를 새로운 셰이프에 이용할 수 있도록 디자인하려면 어떻게 해야 할까요?

다음과 같이 추상 Shape 클래스를 정의해 보세요.

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

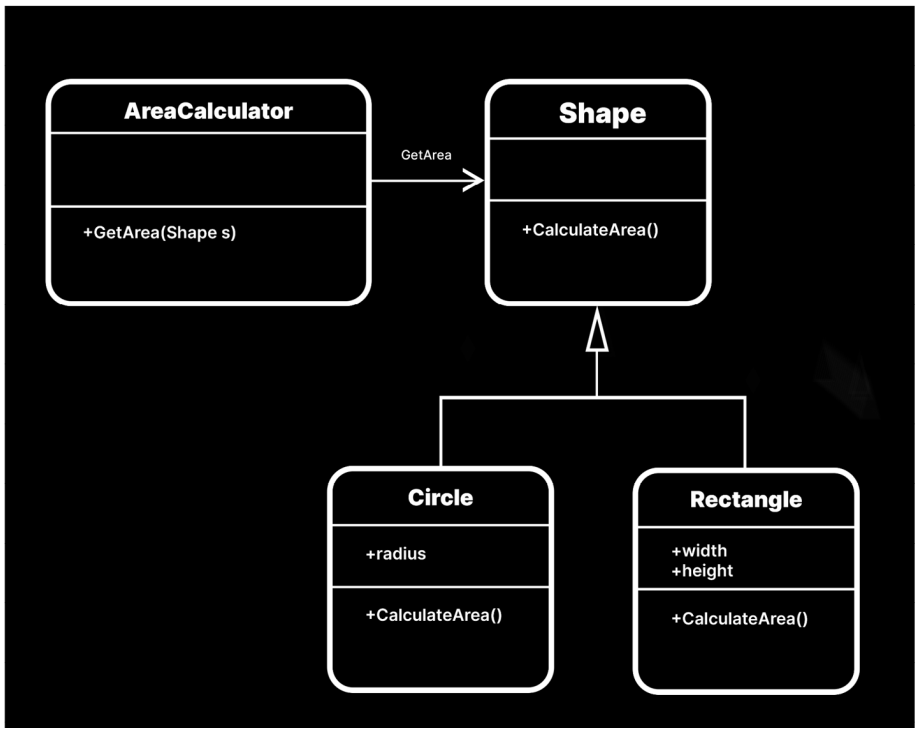
이 코드에는 CalculateArea라는 이름의 추상 메서드가 있습니다. 이제 Rectangle과 Circle이 Shape로부터 상속하도록 하면, 각 셰이프는 각자의 영역을 계산하고 다음과 같은 결과를 반환할 수 있습니다.

```
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;
    }
}
```

AreaCalculator를 다음과 같이 단순화할 수 있습니다.

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

수정된 AreaCalculator 클래스는 이제 추상 Shape 클래스를 적절히 구현하는 셰이프의 영역을 가져올 수 있습니다. 이제 원본 소스를 전혀 변경하지 않고 AreaCalculator 기능을 확장할 수 있습니다.



개방-폐쇄 원칙에 맞게 클래스 수정

새로운 다각형이 필요할 때마다 Shape에서 상속하는 새 클래스를 정의하면 됩니다. 각각의 서브 클래스 셰이프는 CalculateArea 메서드를 오버라이드하여 올바른 영역을 반환합니다.

이 새로운 디자인을 사용하면 디버깅하기도 더 쉽습니다. 새로운 셰이프로 오류가 발생하더라도 AreaCalculator를 재검토할 필요가 없습니다. 기존 코드가 변경 없이 유지되므로, 새 코드에만 잘못된 로직이 있는지 조사하면 됩니다.

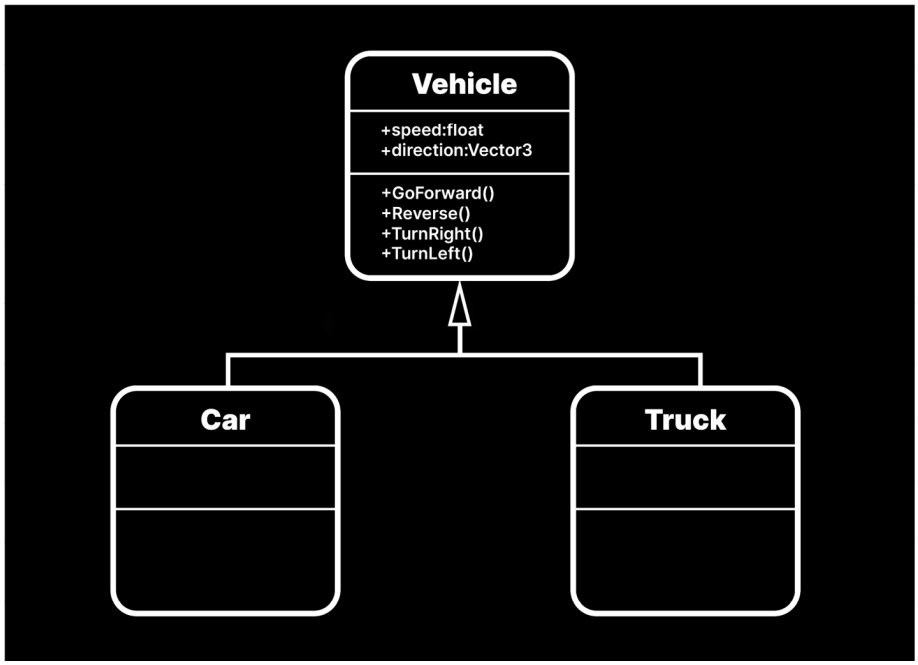
Unity에서 새로운 클래스를 만들 때 인터페이스와 추상화를 활용하세요. 이렇게 하면 나중에 확장하기 까다로운 switch 또는 if 문을 로직에 넣지 않아도 됩니다. OCP에 맞춰 클래스를 설정하는 데 익숙해지면 장기적으로 새로운 코드를 더 간편하게 추가할 수 있게 됩니다.

리스코프 치환 원칙

LSP(리스코프 치환 원칙)는 파생된 클래스가 기본 클래스로 대체될 수 있어야 한다고 명시합니다. 객체 지향 프로그래밍에서 상속을 사용하면 서브 클래스를 통해 기능을 추가할 수 있습니다. 하지만 주의하지 않으면 불필요한 복잡성을 더할 수 있습니다.

SOLID의 세 번째 항목인 리스코프 치환 원칙은 상속을 적용하여 서브 클래스를 더 강력하고 유연하게 만드는 방법을 알려 줍니다.

게임에 Vehicle이라는 클래스가 필요하다고 생각해 보세요. 이 클래스는 애플리케이션에서 사용할 차량이라는 하위 클래스의 기본 클래스가 됩니다. 자동차나 트럭을 사용하려는 경우를 예로 들 수 있습니다.



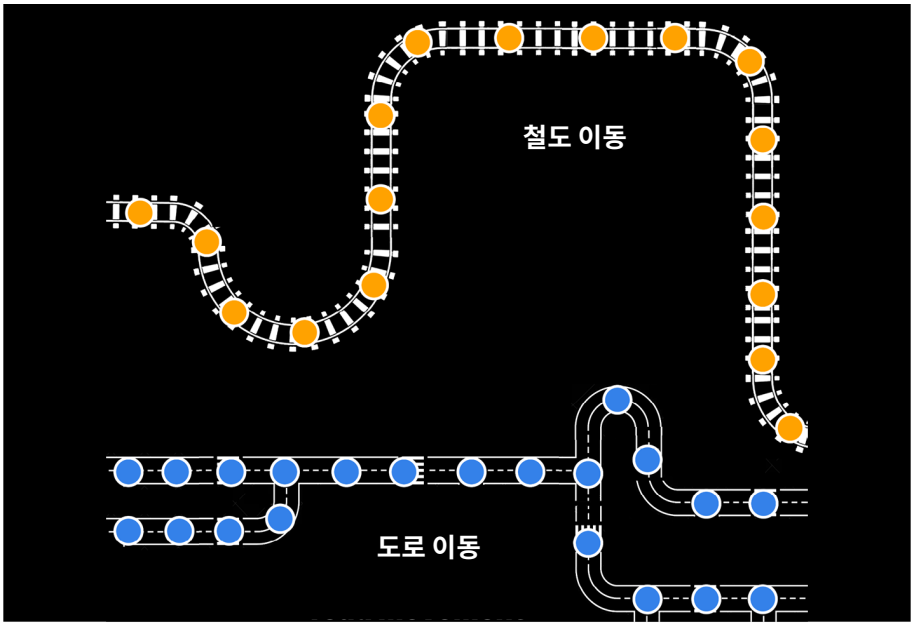
모든 항목은 Vehicle 클래스에서 상속됩니다.

기본 클래스(Vehicle)를 사용할 수 있는 모든 위치에서 애플리케이션에 오류를 발생시키지 않고 Car 또는 Truck과 같은 서브 클래스를 사용할 수 있습니다.

Vehicle 클래스의 예시는 다음과 같습니다.

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

    public void GoForward()
    {
            ...
    }

    public void Reverse()
    {
            ...
    }

    public void TurnRight()
    {
            ...
    }

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

보드에서 차량을 옮기는 턴제 게임을 개발한다고 가정해 보세요.

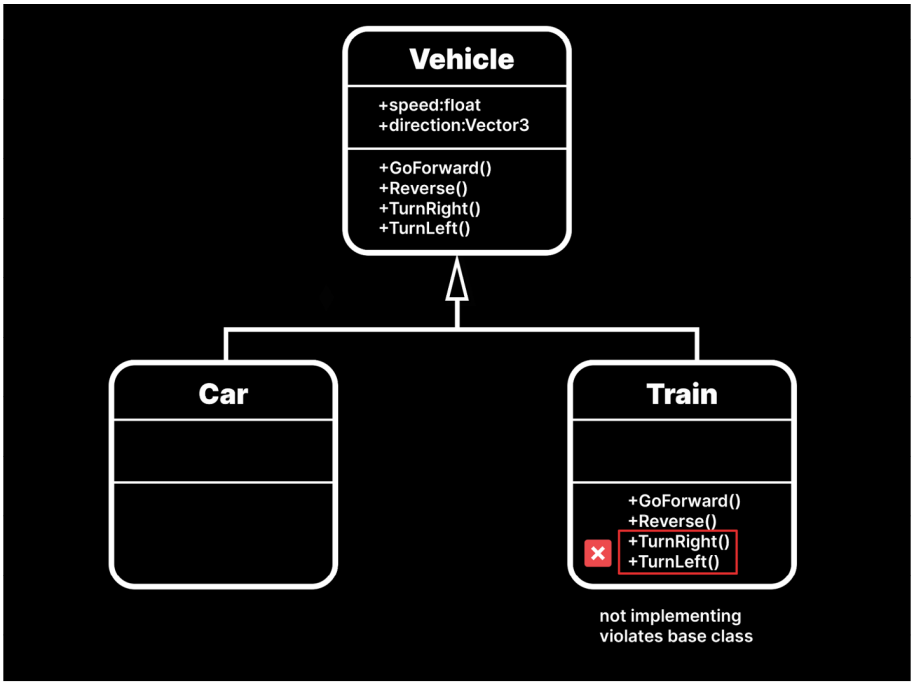


자동차와 기차 게임 비교 예시

사전에 정해진 경로에 따라 차량을 운전하는 Navigator라는 클래스를 추가할 수 있습니다.

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

이 클래스를 사용하면 어떤 차량이든 Navigator 클래스의 Move 메서드로 전달할 수 있으며, 자동차와 트럭에는 이 방법이 잘 통할 것입니다. 하지만 Train이라는 클래스를 구현하려는 경우에는 어떻게 될까요?



Train은 기본 클래스를 위반할 것입니다.

기차는 철도를 이탈할 수 없으므로 TurnLeft와 TurnRight 메서드는 Train 클래스에서 작동하지 않습니다. Navigator의 Move 메서드로 기차를 전달하는 경우, 해당 라인에 도달하면 구현되지 않은 예외가 발생하거나 아무 일도 일어나지 않을 것입니다. 특정 유형을 하위 유형과 교체할 수 없다면 리스코프 치환 원칙을 위반하게 됩니다.

`Train`은 `Vehicle`의 하위 유형이므로 `Vehicle` 클래스가 허용되는 모든 위치에 사용할 수 있어야 합니다. 그러지 않으면 코드가 예측할 수 없는 방식으로 작동할 수 있습니다.

다음은 리스코프 치환 원칙을 더 철저하게 준수하기 위한 몇 가지 팁입니다.

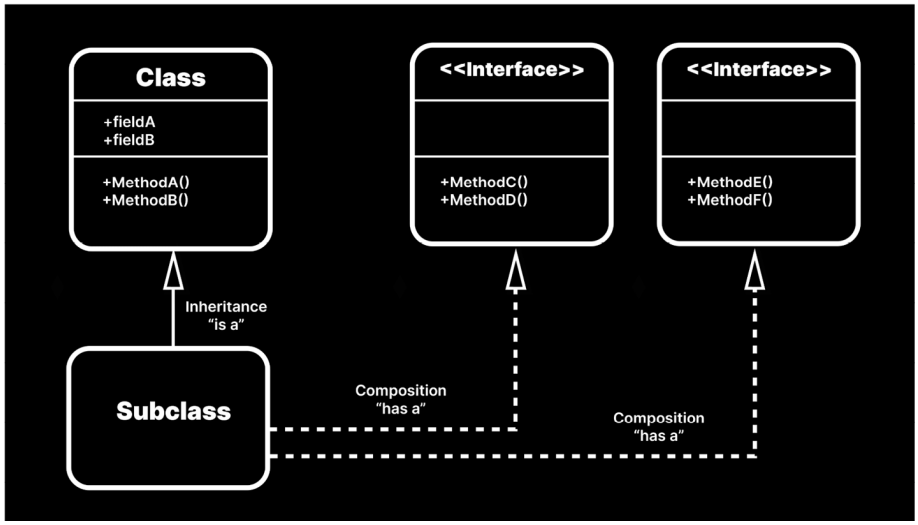
— **서브 클래스를 설정할 때 기능을 제거하면 리스코프 치환을 위반하게 될 가능성이 큽니다.** `NotImplementedException`은 이 원칙을 위반했다는 의미이며, 메서드를 비워 두는 경우도 마찬가지입니다. 서브 클래스가 기본 클래스처럼 동작하지 않는다면 오류나 예외가 명시적으로 보이지 않더라도 LSP를 준수하지 않는 것입니다.

— **추상화를 단순하게 유지합니다.** 기본 클래스에 들어가는 로직이 많을수록 LSP를 위반할 확률도 커집니다. 기본 클래스는 파생 서브 클래스의 일반적인 기능만 표현해야 합니다.

— **서브 클래스는 기본 클래스와 동일한 공용 멤버를 가져야 합니다.** 또한 그러한 공용 멤버는 호출 시 동일한 서명을 갖고 동일한 동작을 취해야 합니다.

— **클래스 계층 구조를 수립할 때 클래스 API를 고려합니다.** 대상을 모두 차량으로 간주하더라도 Car와 Train은 각각 서로 다른 부모 클래스로부터 상속하는 편이 더 나을 수 있습니다. 실질적으로 분류가 항상 클래스 계층 구조와 일치하지는 않습니다.

— **상속보다는 합성을 우선시합니다.** 상속을 통해 기능의 전달을 시도하는 대신, 특정한 동작을 캡슐화할 있도록 인터페이스 또는 별도의 클래스를 만드세요. 그런 다음 믹스 앤 매치를 통해 다양한 기능의 합성물을 생성합니다.

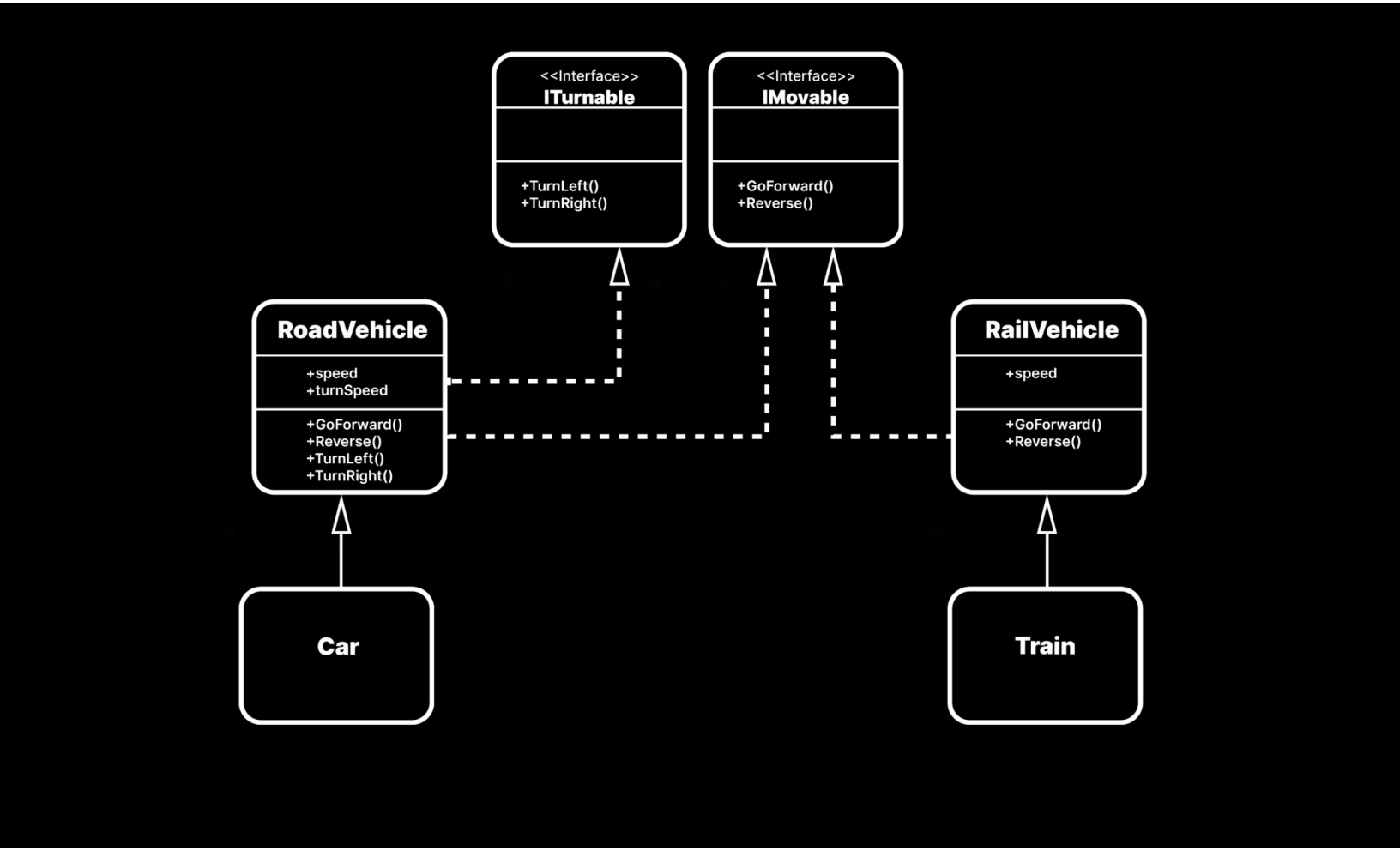


상속보다 합성 우선

이 디자인을 수정하려면 원본 `Vehicle` 유형을 삭제한 다음 대부분의 기능을 인터페이스로 옮깁니다.

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

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

`RoadVehicle` 유형과 `RailVehicle` 유형을 만들면 LSP 원칙을 더 철저하게 따를 수 있습니다. `Car`와 `Train`은 해당하는 기본 클래스로부터 상속합니다.



리스코프 치환 원칙을 따르기 위한 리팩터링

```
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
{
    ...
}
```

이러한 방법에서는 기능이 상속 대신 인터페이스를 통해 실행됩니다. `Car`와 `Train` 클래스가 더 이상 같은 기본 클래스를 공유하지 않으며, 이는 LSP를 준수합니다. 같은 기본 클래스에서 `RoadVehicle`과 `RailVehicle`을 파생시킬 수도 있으나 이 경우에는 크게 필요하지 않습니다.

이러한 사고방식은 직관적이지 않은 것처럼 보일 수 있는데, 사람들이 실제 세상에 대해 가지는 확고한 가정이 있기 때문입니다. 소프트웨어 개발에서는 이를 원-타원 문제(Circle–ellipse problem)라고 합니다. 모든 실제 등가 관계가 상속으로 전환되지는 않습니다. 소프트웨어 디자인으로 진행하려는 것은 실제 세상에 대한 사전 지식이 아닌, 클래스 계층 구조라는 사실을 기억하세요.

리스코프 치환 원칙에 따라 상속 사용 방법에 제한을 두어 코드 베이스를 확장 가능하고 유연하게 유지해야 합니다.

인터페이스 분리 원칙

ISP(인터페이스 분리 원칙)는 어떠한 클라이언트도 자신이 사용하지 않는 메서드에 강제로 종속될 수 없다고 명시합니다.

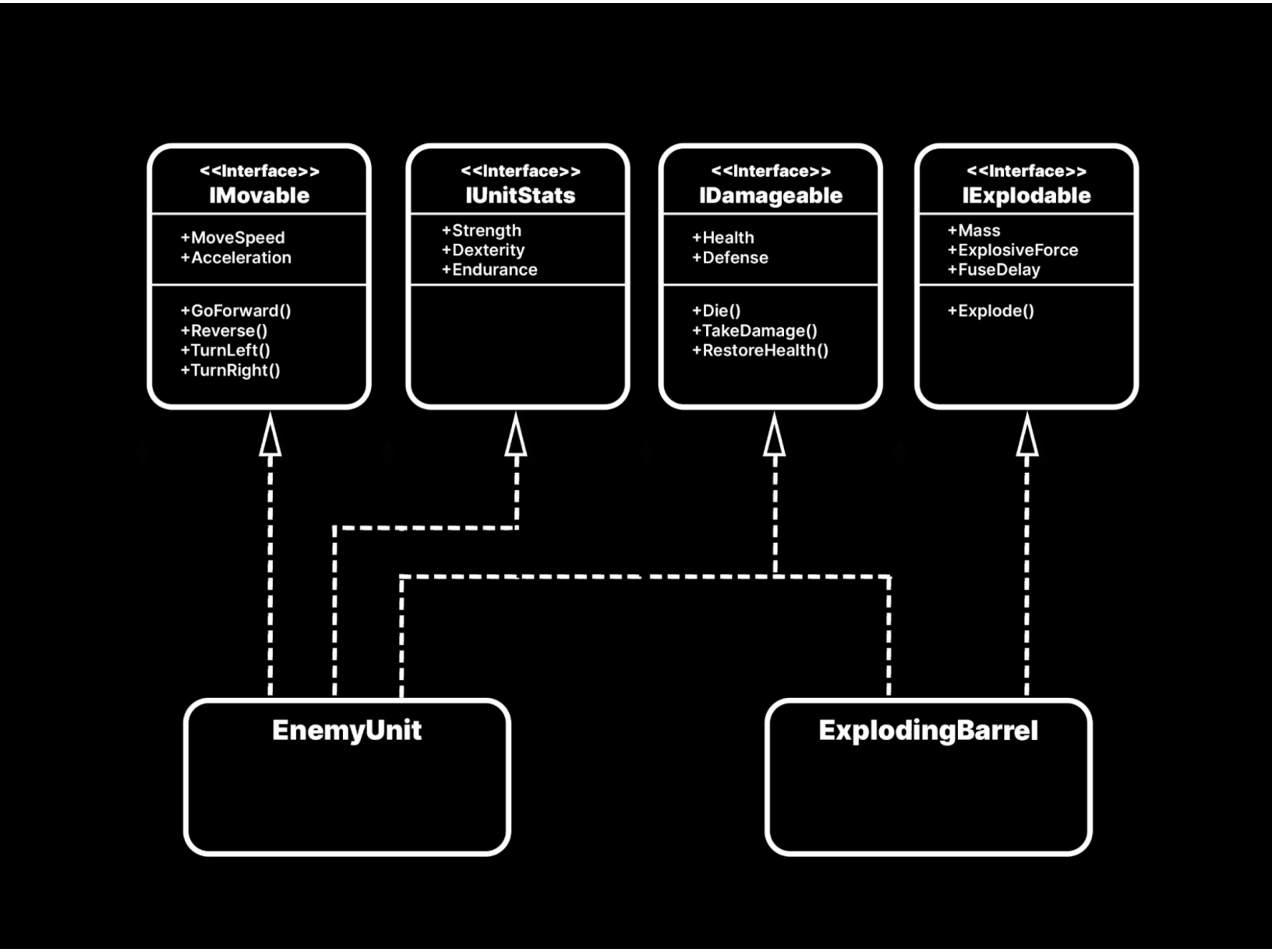
다시 말해 인터페이스의 규모가 커지지 않도록 해야 합니다. 클래스와 메서드의 길이를 짧게 유지하라는 단일 책임 원칙과 같은 맥락으로 이해하세요. 그러면 유연성을 최대로 향상하며, 집중도가 높고 컴팩트한 인터페이스를 유지할 수 있습니다.

다양한 플레이어 유닛이 있는 전략 게임을 만든다고 상상해 보세요. 각 유닛에는 체력과 속도를 비롯한 다양한 스탯이 있습니다. 다음과 같이 모든 유닛이 유사한 기능을 구현하도록 보장하는 인터페이스를 만드세요.

```
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; }

}
```

부술 수 있는 통이나 상자 등 파괴 가능한 프랍을 만든다고 가정해 보겠습니다. 비록 움직이지 않는 프랍이지만 여기에도 체력이라는 개념이 필요합니다. 또한 상자나 통에는 게임 내의 다른 유닛에 부여된 능력 중 상당수가 부여되지 않을 것입니다.

파괴 가능한 프랍에 너무 많은 메서드를 부여하는 인터페이스 한 개를 만드는 대신, 여러 개의 작은 인터페이스로 분할하세요. 그러면 인터페이스를 구현하는 클래스에서 필요한 요소만 선택해 사용할 것입니다.



인터페이스를 보다 작은 여러 인터페이스로 분할합니다.

```
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; }
}
```

폭발하는 통에 `IExplodable` 인터페이스를 추가할 수도 있습니다.

```
public interface IExplodable
{
    public float Mass { get; set; }
    public float ExplosiveForce { get; set; }
    public float FuseDelay { get; set; }

    public void Explode();
}
```

클래스 하나가 둘 이상의 인터페이스를 구현할 수 있으므로 IDamageable, IMoveable, IUnitStats에서 적 유닛 코드를 작성할 수 있습니다.

폭발하는 통은 다른 인터페이스의 불필요한 오버헤드 없이 IDamageable과 IExplodable을 사용할 수 있습니다.

```
public class ExplodingBarrel : MonoBehaviour, IDamageable, IExplodable
{
    ...
}

public class EnemyUnit : MonoBehaviour, IDamageable, IMovable,
IUnitStats
{
    ...
}
```

리스코프 치환의 예시와 유사하게 여기에서도 상속보다 합성을 우선시합니다. 인터페이스 분리 원칙은 시스템을 분리하고 간편하게 수정 및 재배포하는 데 도움이 됩니다.

종속성 역전 원칙

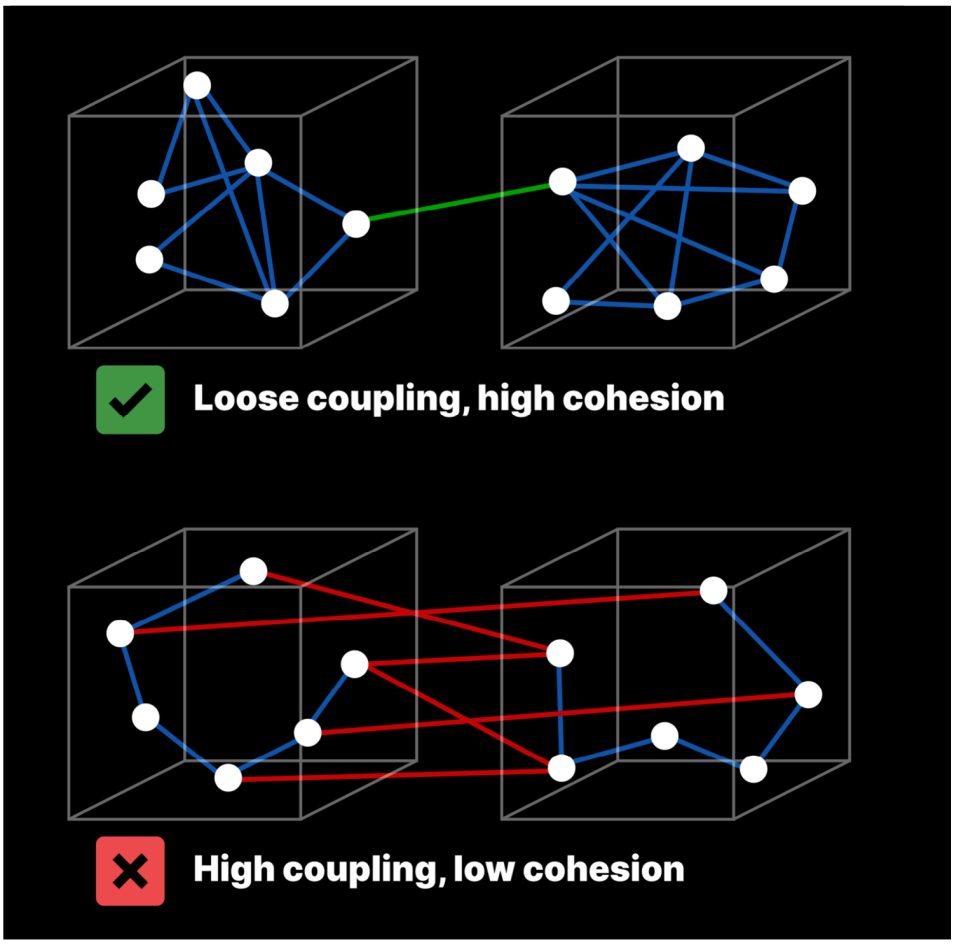
DIP(종속성 역전 원칙)는 상위 수준의 모듈이 하위 수준의 모듈에서 어떤 것도 직접 가져오면 안 된다고 명시합니다. 양측 모두 추상화에 의존해야 합니다.

무슨 의미인지 알아보도록 하겠습니다. 다른 클래스와 관계가 있는 클래스는 종속 또는 결합 관계가 있다고 부릅니다. 소프트웨어 디자인에서 각 종속성은 약간의 위험성을 내포합니다.

한 클래스가 다른 클래스의 작동 방식에 대해 너무 많이 아는 경우, 첫 번째 클래스를 수정하면 두 번째 클래스에 피해를 줄 수 있으며 그 반대의 경우도 마찬가지입니다. 결합도가 높으면 깔끔하지 않은 코드로 간주됩니다. 애플리케이션의 한 부분에서 오류가 발생하면 다른 부분으로 눈덩이처럼 확대될 수 있습니다.

클래스 간 종속성을 가능한 한 최소화하는 것이 이상적입니다. 또한 각 클래스의 내부 요소가 한결같이 작동해야 하며, 외부 연결에 의존하지 않아야 합니다. 내부 로직이나 프라이빗 로직으로 작동하는 객체를 응집도가 높은 것으로 간주됩니다.

최고의 시나리오는 결합도는 낮추고 응집도는 높이는 것을 목표로 삼는 것입니다.

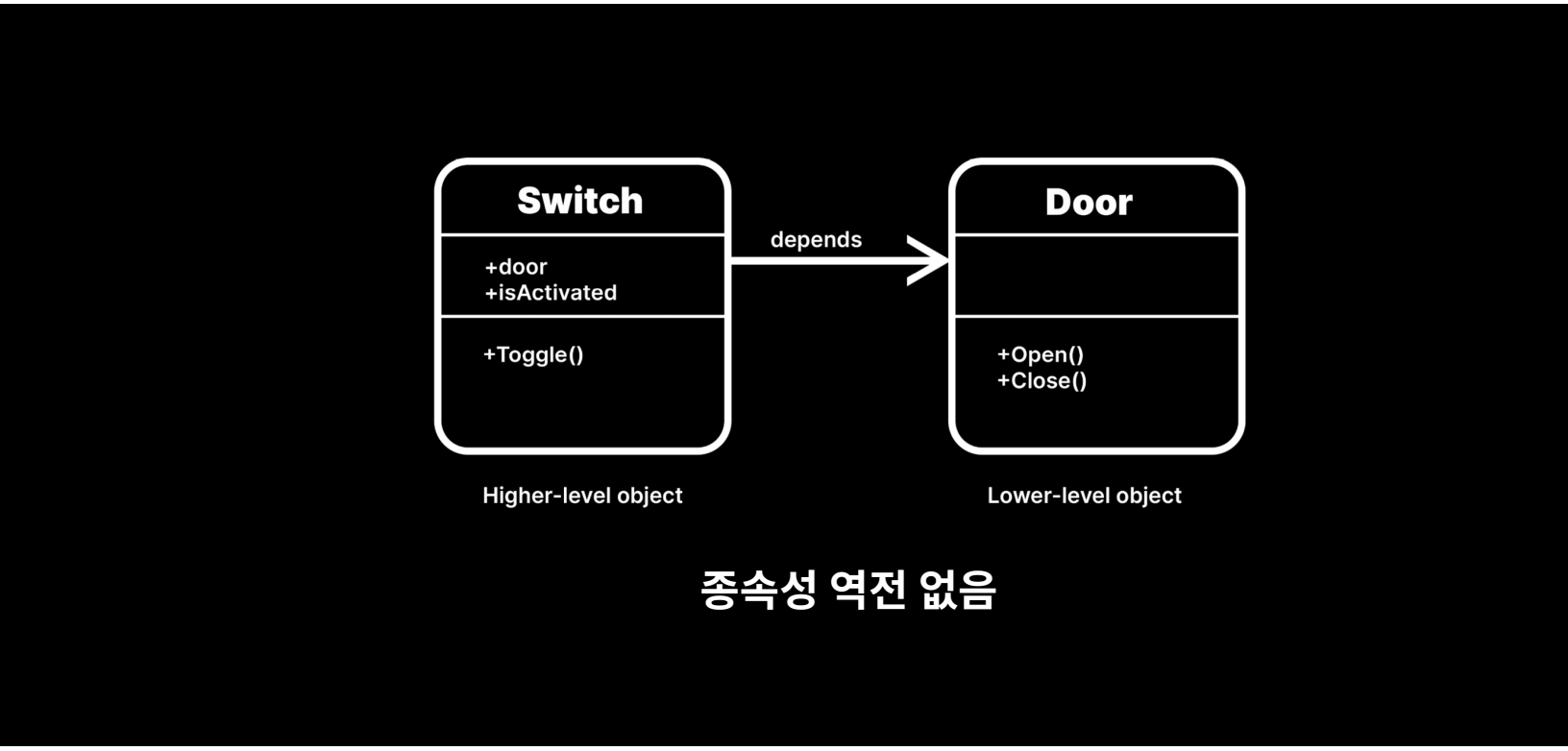


가능한 한 응집도는 높이고 결합도는 낮춥니다.

게임 애플리케이션을 수정하고 확장할 수 있어야 합니다. 수정하기가 까다롭거나 쉽지 않다면 현재 어떻게 구조화되어 있는지 조사하세요.

종속성 역전 원칙은 클래스 간의 결합도를 줄이는 데 도움이 될 수 있습니다. 애플리케이션에서 클래스와 시스템을 만들 때 자연스럽게 일부는 상위 수준이 되고 일부는 하위 수준이 됩니다. 상위 수준 클래스는 하위 수준 클래스에 의존해 작업을 수행하는데, SOLID 원칙에서는 이를 바꿔야 한다고 강조합니다.

캐릭터가 레벨을 탐험하며 문을 트리거해 여는 게임을 개발한다고 가정하겠습니다. `Switch` 라는 클래스와 `Door`라는 클래스를 만들게 될 것입니다.



Switch(상위 수준)는 Door(하위 수준) 클래스에 직접 의존합니다.

상위 수준에서는 캐릭터가 특정 위치로 이동하고 특정 행동이 발생하기를 원하는데, `Switch`가 이러한 행동을 담당합니다.

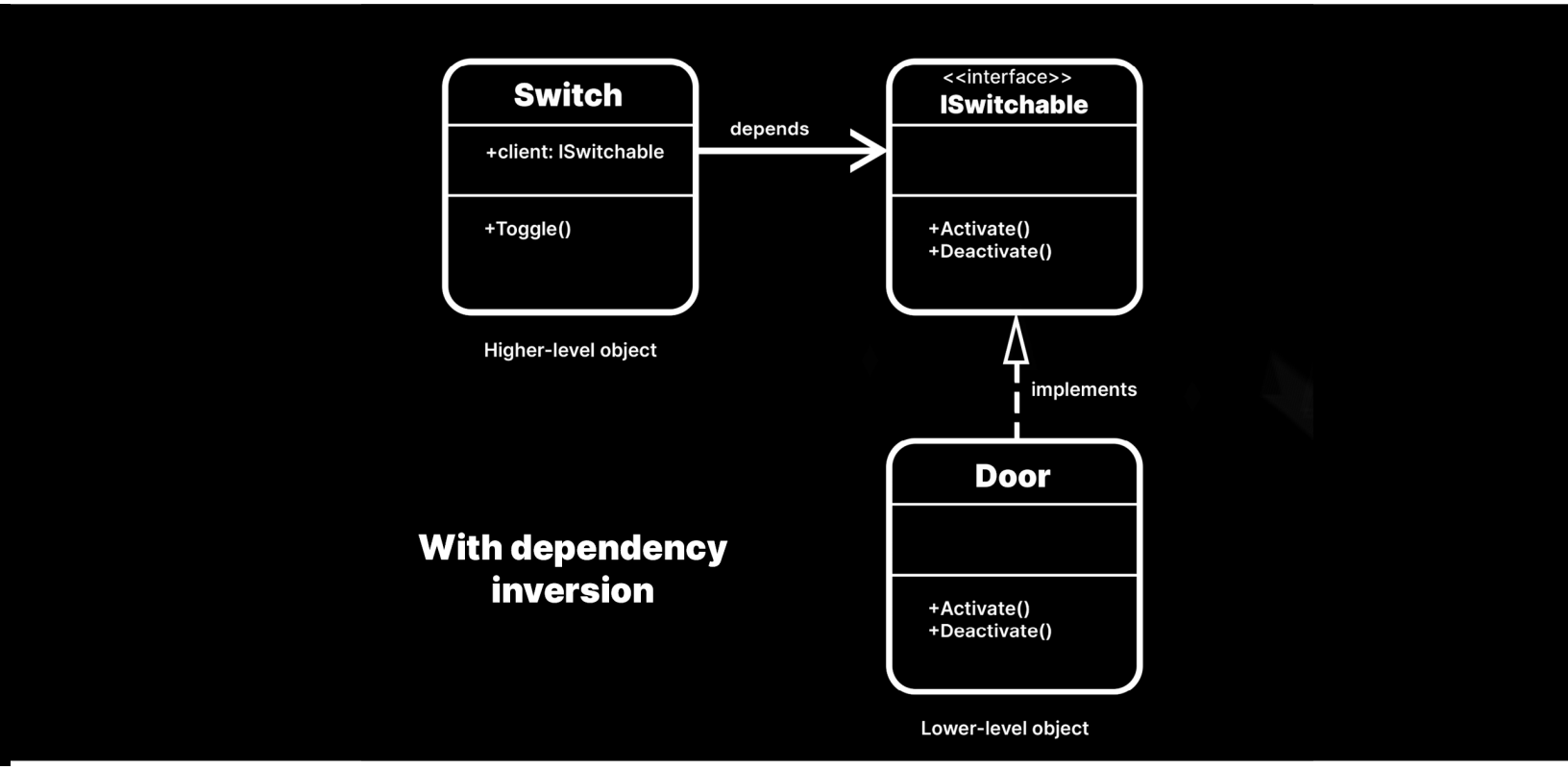
하위 수준에서는 다른 클래스인 `Door`가 있으며, 여기에는 문 지오메트리를 여는 방법의 실제 구현이 포함되어 있습니다. 간소화를 목적으로 `Debug.Log` 문이 추가되어 문을 열고 닫는 로직을 나타냅니다.

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

Switch는 Toggle 메서드를 호출해 문을 열고 닫을 수 있습니다. 작동하기는 하지만 이 경우 Door에서 직접 Switch로 연결되는 종속성이 발생한다는 문제가 있습니다. Switch의 로직이 Door 외의 다른 항목, 이를테면 조명을 켜거나 거대 로봇을 활성화하는 경우에도 사용되어야 한다면 어떨까요?

Switch 클래스에 메서드를 추가할 수 있겠지만, 그러면 개방-폐쇄 원칙을 위반하게 되고, 기능을 확장하려 할 때마다 원본 코드를 수정해야 합니다.

이번에도 추상화로 문제를 해결할 수 있습니다. 클래스 사이에 ISwitchable이라는 인터페이스를 삽입할 수 있습니다.



두 클래스 사이에 있는 ISwitchable 인터페이스

ISwitchable에 필요한 것은 액티브 상태인지 알기 위한 public 프로퍼티, 그리고 Activate 및 Deactivate라는 메서드 2개뿐입니다.

```
public interface ISwitchable
{
    public bool IsActive { get; }

    public void Activate();
    public void Deactivate();
}
```

그러면 Switch는 아래와 같이 구성되며, 문에 직접 의존하지 않고 ISwitchable 클라이언트에 의존합니다.

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

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

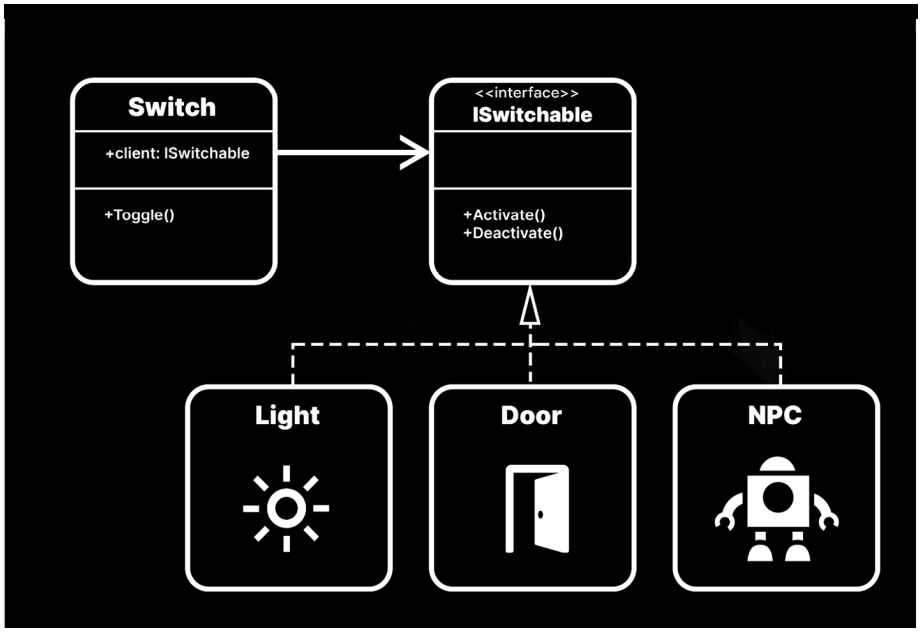
한편 ISwitchable을 구현하려면 Door를 재작업해야 합니다.

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

이제 종속성을 역전했습니다. 인터페이스가 스위치를 문에 배타적으로 고정하지 않고, 둘 사이에 추상화를 형성합니다. Switch는 더 이상 문의 특정한 메서드(Open 및 Close)에 직접 의존하지 않습니다. 대신 ISwitchable의 Activate 및 Deactivate를 사용합니다.

작지만 특별한 이 변화로 재사용성이 향상됩니다. 이전에는 Switch가 Door에만 작동했으나, 이제 ISwitchable을 구현하는 모든 요소에 작동합니다.

따라서 Switch가 활성화할 수 있는 클래스를 더 많이 만들 수 있습니다. 함정 문이든 레이저 빔이든 높은 수준의 Switch가 작동합니다. ISwitchable을 구현하는 호환 가능한 클라이언트만 있으면 됩니다.



Switch가 이제 모든 ISwitchable 오브젝트를 활성화할 수 있습니다.

다른 SOLID 항목들처럼 종속성 역전 원칙에서도 일반적으로 클래스 간 관계를 설정하는 방식을 검토하도록 요구합니다. 결합도를 낮춰 프로젝트를 간편하게 확장/축소하세요.

인터페이스와 추상 클래스 비교

'상속보다 합성을 우선'한다는 철학에 따라 본 가이드에는 인터페이스를 사용하는 예시가 많습니다. 하지만 추상 클래스를 사용하더라도 많은 디자인 원칙과 패턴을 따를 수 있습니다.

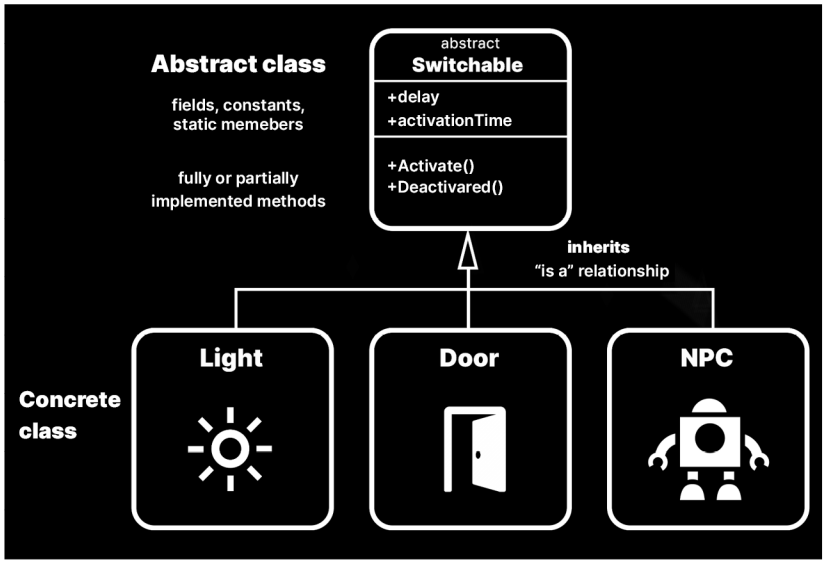
인터페이스와 추상 클래스 모두 C#에서 추상화를 구현하는 유효한 방법입니다. 두 방법 중 무엇을 사용할지는 상황에 따라 다릅니다.

추상 클래스

abstract 키워드를 사용해 기본 클래스를 정의할 수 있으므로, 상속을 통해 일반적인 기능(메서드, 필드, 상수 등)을 서브 클래스로 전달할 수 있습니다.

추상 클래스는 직접 인스턴스화할 수 없으며, 대신 구상 클래스를 파생해야 합니다.

이전 예시에서는 다른 접근법을 사용하는 것만으로 추상 클래스가 동일한 종속성 역전을 달성하도록 했습니다. 따라서 인터페이스를 사용하는 대신, 구상 클래스 (예: `Light` 또는 `Door`)를 `Switchable`이라는 이름의 추상 클래스로부터 파생시킵니다.

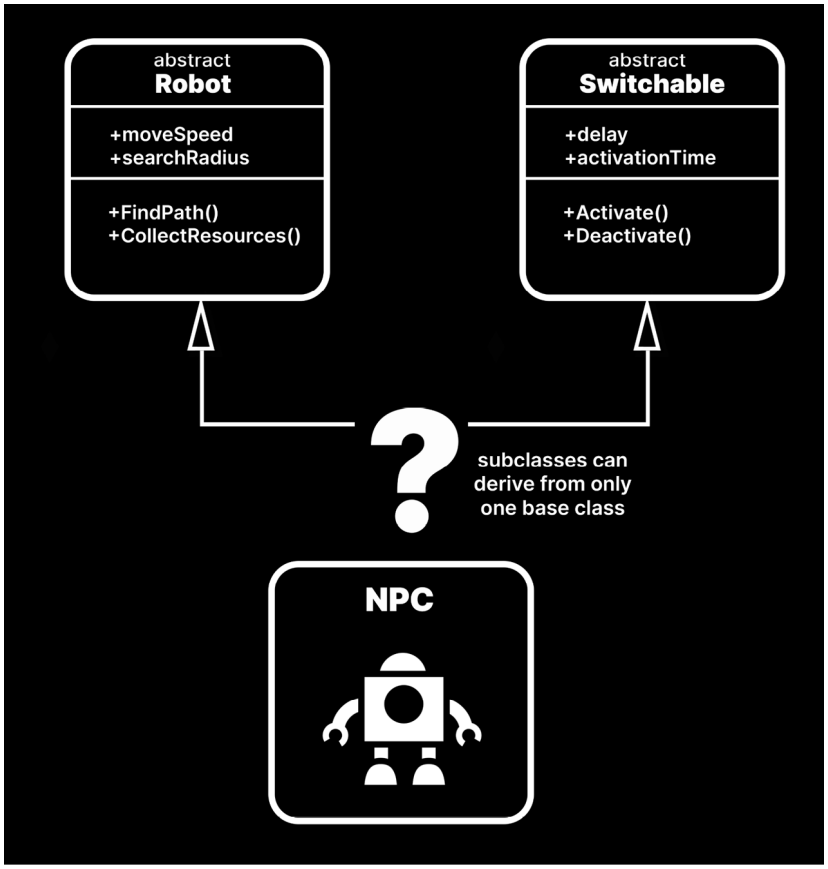


추상 클래스 사용하기

상속은 'is a' 관계를 정의합니다. 위 다이어그램에 보이는 항목은 모두 켜고 끌 수 있는 '전환 가능한'(switchable) 요소들입니다.

추상 클래스에는 정적 멤버뿐 아니라 필드 및 상수도 가질 수 있다는 장점이 있습니다. 또한 protected 및 private처럼 더 제한된 액세스 한정자도 적용할 수 있습니다. 인터페이스와는 달리, 추상 클래스를 사용하면 구상 클래스 간의 핵심 기능을 공유하도록 지원하는 로직을 구현할 수 있습니다.

서로 다른 기본 클래스 2개의 특징을 가지는 파생 클래스를 생성할 필요가 없다면 상속을 문제 없이 이용할 수 있습니다. C#에서는 둘 이상의 기본 클래스로부터 상속할 수 없습니다.



기본 클래스 선택하기

게임의 모든 로봇에 대해 또 다른 추상 클래스를 설정해 두었다면, 어떤 클래스에서 파생시킬지 결정하기가 더 어렵습니다. `Robot` 또는 `Switchable`, 두 기본 클래스 중에서 무엇을 사용해야 할까요?

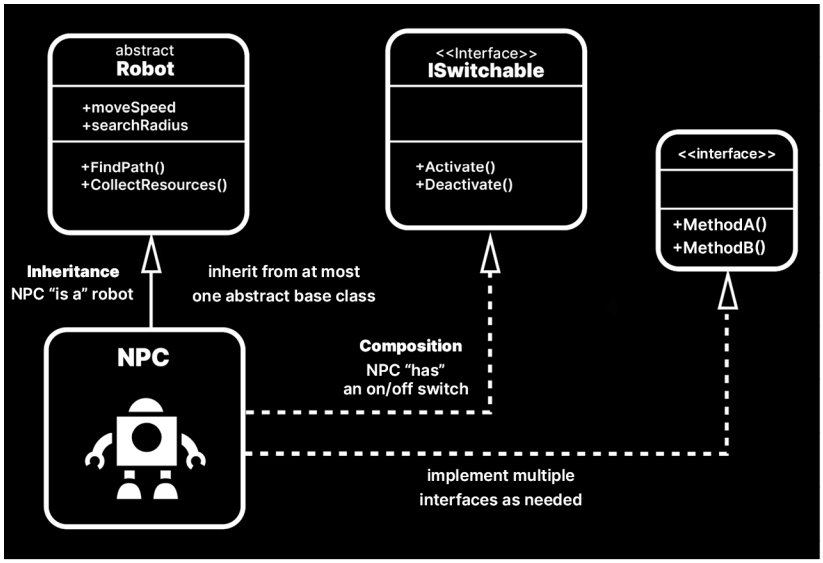
인터페이스

인터페이스 분리 원칙에서 보았듯이, 무언가 상속 패러다임에 제대로 부합하지 않을 때 인터페이스는 더 많은 유연성을 제공합니다. 'has a' 관계를 이용해 더 쉽게 고르고 선택할 수 있습니다.

하지만 인터페이스에는 멤버의 선언만 있습니다. 특정 로직을 구체화하는 책임은 인터페이스를 실제로 구현하는 클래스에 있게 됩니다.

따라서 항상 둘 중 하나를 선택하게 되지는 않습니다. 추상 클래스를 사용해 코드를 공유하려는 기본 기능을 정의하고, 인터페이스를 사용해 유연성이 필요한 주변 기능을 정의하세요.

이 예시에서는 Robot 기본 클래스에서 NPC를 파생시켜 핵심 기능을 상속할 수 있지만, 그런 다음 ISwitchable 인터페이스를 사용해 NPC를 켜고 끄는 기능을 추가합니다.



두 가지를 모두 사용하는 NPC 로봇

다음과 같은 추상 클래스와 인터페이스 간의 차이점을 참고하세요.

추상 클래스	**인터페이스**
메서드 전체 또는 일부를 구현	메서드를 선언하지만 구현할 수는 없음
변수와 필드를 선언/사용	메서드와 프로퍼티만 선언(필드는 제외)
정적 멤버 보유	정적 멤버 선언/사용 불가
생성자 사용	생성자 사용 불가
모든 액세스 한정자 사용(protected, private 등)	액세스 한정자 사용 불가(모든 멤버는 암묵적으로 public)

중요: 클래스는 최대 하나의 추상 클래스에서 상속될 수 있지만, 여러 인터페이스를 구현할 수 있습니다.

SOLID 원칙의 이해

SOLID 원칙은 일상적인 실무를 통해 알아가는 것입니다. 코딩을 하며 항상 참고해야 할 다섯 가지 기본 규칙이라 생각하세요. 간단하게 요약하면 다음과 같습니다.

- **단일 책임**: 클래스는 단 하나의 작업만 수행하고 단 하나의 이유로만 변경할 수 있어야 합니다.
- **개방-폐쇄**: 기존 작동 방식을 변경하지 않고 클래스의 기능을 확장할 수 있어야 합니다.
- **리스코프 치환**: 서브 클래스는 기본 클래스로 대체될 수 있어야 합니다.
- **인터페이스 분리**: 인터페이스는 최소한의 메서드로 짧게 유지하세요. 클라이언트는 필요한 항목만 구현합니다.
- **종속성 역전**: 추상화에 의존하세요. 하나의 구상 클래스에서 다른 구상 클래스로 직접 의존하지 말아야 합니다.

SOLID 원칙은 더 효율적으로 유지하고 확장할 수 있는 깔끔한 코드를 작성하도록 지원하는 가이드라인입니다. SOLID 원칙은 확대/축소가 필요한 대규모 애플리케이션에 매우 적합하므로, 거의 20년 동안 기업 차원에서 소프트웨어 디자인 분야의 핵심 원칙으로 자리잡았습니다.

SOLID 원칙을 따르다 보면 미리 처리해야 할 추가 작업이 발생하기도 합니다. 기능의 일부를 추상화 또는 인터페이스로 리팩터링해야 할 수도 있습니다. 하지만 길게 보면 결국은 도움이 되는 경우가 많습니다.

자신의 프로젝트에 SOLID 원칙을 얼마나 엄밀하게 적용할지 스스로 결정하세요. 절대적인 규칙은 없습니다. 본 가이드에서는 다루지 않지만, 각 원칙을 따르는 과정에도 미묘한 차이가 있는 수많은 방식이 존재합니다. 특정 구문보다는 원칙의 바탕이 되는 개념이 더 중요하다는 사실을 기억하시기 바랍니다.

어떻게 활용할지 확신이 들지 않으면 KISS 원칙을 다시 참고하세요. 단순하게 유지하되, 원칙론만 내세워 스크립트에 억지로 원칙을 적용하려 하면 안 됩니다. 필요에 따라 유기적으로 적용하세요.

자세한 정보는 유나이트 오스틴 행사에서 진행한 Unity SOLID 프레젠테이션에서 확인하세요.

게임 개발을 위한 디자인 패턴

SOLID 원칙을 이해하고 나면 디자인 패턴을 더 심도 있게 알아보고 싶어질 수 있습니다.

디자인 패턴을 활용하면 일상적인 소프트웨어 문제 해결에 사용하는 익숙한 솔루션을 용도에 맞게 변경할 수 있습니다. 하지만 패턴은 즉시 사용 가능한 라이브러리나 프레임워크가 아니며, 결과를 달성하기 위한 일련의 구체적인 단계인 알고리즘도 아닙니다.

디자인 패턴은 일종의 청사진에 가깝습니다. 실질적인 구성은 개발자가 해야 하는 일반적인 계획입니다. 두 개의 프로그램이 동일한 패턴을 따르면서도 매우 다른 코드로 구성될 수 있습니다.

실무에서 같은 문제에 직면할 때, 많은 개발자들이 대체로 유사한 솔루션을 떠올립니다. 어떤 솔루션이 여러 번 제기된다면 누군가 패턴을 '발견'하고 공식적인 이름을 지정할 수 있습니다.

GoF

현재 사용되는 많은 소프트웨어 디자인 패턴은 에릭 감마, 리차드 헬름, 랄프 존슨, 존 블리시디스가 공동으로 집필한 GoF의 디자인 패턴: 재사용성을 지닌 객체 지향 소프트웨어의 핵심 요소에 그 뿌리를 두고 있습니다. 이 책은 다양한 일상 애플리케이션에서 식별된 23가지의 디자인 패턴을 설명합니다.

이 책의 원저자 4명을 흔히 GoF(4인방)라 하며, 원본 패턴을 GoF 패턴이라 부르기도 합니다. 책에 인용된 예시는 대부분 C++(및 Smalltalk)으로 작성되었지만, C# 등의 객체 지향 언어에도 해당 아이디어를 적용할 수 있습니다.

GoF의 디자인 패턴이 1994년에 처음 발간된 이래, 개발자들은 다양한 분야에서 수십 가지의 객체 지향 패턴을 발견해 왔습니다. 많은 엔지니어링 전문 분야에는 잘 확립된 패턴이 존재하며, 게임 개발 분야도 마찬가지입니다.

디자인 패턴 학습

디자인 패턴을 배우지 않아도 게임 프로그래머 업무는 가능하지만, 디자인 패턴을 배우면 더 나은 개발자로 거듭나는 데 큰 도움이 됩니다. 디자인 패턴이라는 이름으로 불리는 이유는 잘 알려진 문제에 대처하는 일반적인 솔루션이기 때문입니다.

소프트웨어 엔지니어는 일반적인 개발 과정에서 항상 디자인 패턴을 재발견합니다. 여러분도 무의식적으로 이러한 패턴을 이미 구현했을 수 있습니다.

스스로 패턴을 찾는 훈련을 하세요. 그러면 다음과 같은 부분에 도움을 얻을 수 있습니다.

— **객체 지향 프로그래밍 학습:** 디자인 패턴은 심오한 StackOverflow 게시물에서 찾아야 하는 비밀이 아닙니다. 개발에서 마주하는 일상적인 어려움에 대처하는 흔한 방법일 뿐입니다. 패턴을 보면 얼마나 많은 개발자들이 같은 문제를 접했는지 알 수 있습니다. 지금은 패턴을 사용하지 않는다 해도, 언젠가 필요할 수 있다는 점을 기억하세요.

— **다른 개발자와의 대화:** 팀 작업에서 소통해야 할 때 패턴을 사용하면 편리합니다. '커맨드 패턴'이나 '오브젝트 풀'을 언급하면 숙련된 Unity 개발자는 무엇을 구현하려 하는지 정확하게 이해할 것입니다.

— **새로운 프레임워크 둘러보기:** 에셋 스토어에서 빌트인 패키지와 같은 것을 임포트할 때는 여기에서 언급한 패턴 중 한 가지 이상을 마주치게 됩니다. 디자인 패턴을 인식하면 새로운 프레임워크가 작동하는 방식 및 프레임워크 생성과 관련된 사고 프로세스를 이해하는 데 도움이 됩니다.

물론 모든 디자인 패턴이 모든 종류의 게임 애플리케이션에 적용되지는 않습니다. 매슬로의 망치로 패턴을 찾느라 애쓰지 마세요. 모든 문제가 못은 아니니까요.

다른 모든 툴과 마찬가지로 디자인 패턴의 유용성은 컨텍스트에 달려 있습니다. 각 패턴은 특정 상황에서 장점을 보이지만, 그만큼의 단점도 수반합니다. 소프트웨어 개발에서는 모든 의사결정에서 타협을 거치게 됩니다.

많은 수의 게임 오브젝트를 실시간으로 생성하려는 중이며, 그럴 경우 성능에 영향을 줄까요? 코드를 재구성해 문제를 해결할 수 있을까요?

이러한 디자인 패턴을 알아두고, 적절한 시기가 오면 게임 개발 아이디어 저장소에서 꺼내 당면한 문제를 해결하세요.

더 읽기

GoF의 디자인 패턴: 재사용성을 지닌 객체 지향 소프트웨어의 핵심 요소 외에 주목할 만한 다른 도서로, 로버트 나이스트롬의 게임 프로그래밍 패턴이 있습니다. 이 책은 다양한 소프트웨어 패턴을 자세하고 명료하게 설명합니다. gameprogrammingpatterns.com에서 무료로 웹 버전을 읽어 볼 수 있습니다.

Unity 내부의 패턴

Unity에는 확립된 여러 게임 개발 패턴이 이미 구현되어 있으므로, 개발자가 직접 작성하는 수고를 덜 수 있습니다. 여기에는 다음 패턴이 포함됩니다.

— **게임 루프:** 게임 애플리케이션을 구동하는 하드웨어는 매우 다양하므로, 모든 게임의 핵심에는 클럭 속도와 무관하게 기능을 발휘해야 하는 무한 루프가 있습니다. 속도가 다른 다양한 컴퓨터에 대응하기 위해, 게임 개발자는 고정 타임스텝(지정된 초당 프레임 사용) 및 가변 타임스텝(엔진이 이전 프레임 이후 소요된 시간을 측정)을 사용해야 하는 경우가 많습니다.

이 부분은 Unity에서 담당하므로, 개발자가 직접 구현할 필요가 없습니다. 개발자는 Update, LateUpdate, FixedUpdate 등의 MonoBehaviour 메서드를 사용해 게임플레이만 관리하면 됩니다.

— **업데이트:** 게임 애플리케이션에서 각 오브젝트의 동작을 한 번에 한 프레임씩 업데이트하는 경우가 많습니다. Unity에서 직접 업데이트를 재생성할 수도 있지만, MonoBehaviour 클래스는 이 작업을 자동으로 수행합니다. Update, LateUpdate 또는 FixedUpdate 메서드를 적절히 사용해 게임 오브젝트와 컴포넌트를 게임 클럭의 최소 단위에 맞춰 수정할 수 있습니다.

— **프로토타입:** 원본에 영향을 주지 않고 오브젝트를 복사해야 하는 경우가 자주 있습니다. 이 생성 패턴을 활용하면 비슷해 보이는 다른 오브젝트를 만들기 위해 한 오브젝트를 복사 및 복제해야 하는 문제를 해결할 수 있습니다. 이렇게 하면 게임에서 매번 오브젝트 유형을 생성하느라 클래스를 별도로 정의할 필요가 없습니다.

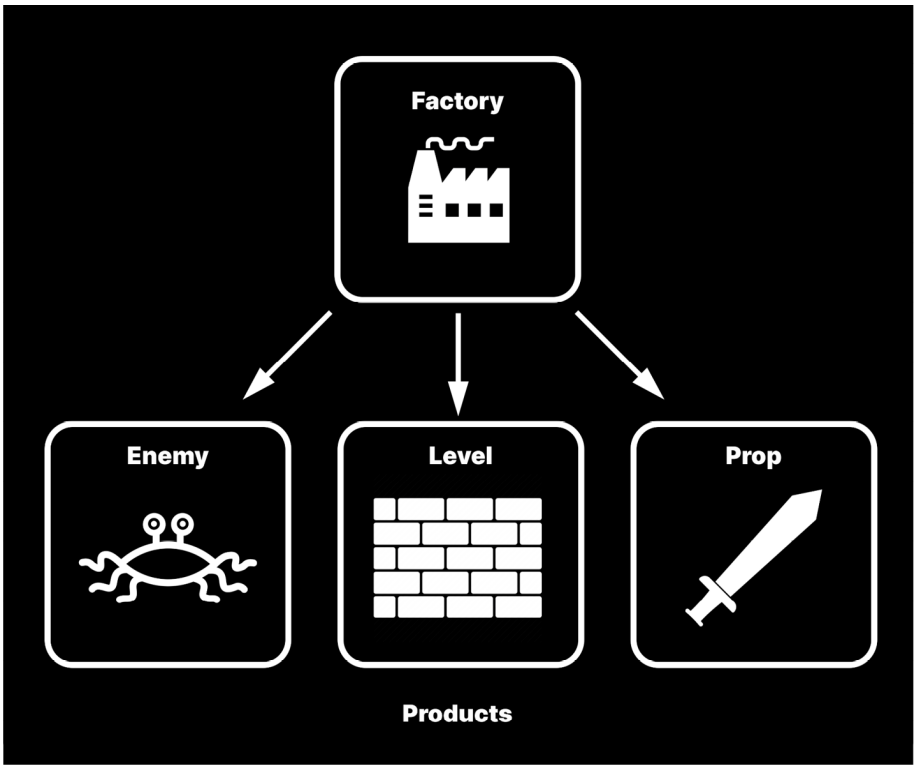
  Unity의 프리팹 시스템은 게임 오브젝트의 프로토타이핑 형식을 구현합니다. 이 경우 컴포넌트까지 갖춘 템플릿 오브젝트를 복제할 수 있습니다. 특정한 프로퍼티를 오버라이드하여 다른 프리팹 안에 프리팹 배리언트 또는 네스티드 프리팹(Nested Prefab)을 만들어서 계층 구조를 생성합니다. 특별한 프리팹 편집 모드를 사용하여 별도로 또는 컨텍스트에 따라 프리팹을 편집하세요.

— **컴포넌트:** 대부분의 Unity 작업자는 이 패턴을 알고 있습니다. 다수의 항목을 책임지는 큰 클래스를 만드는 대신, 하나의 항목만 책임지는 작은 컴포넌트를 만드세요.

  합성을 사용해 컴포넌트를 고르고 선택하면, 이를 조합하여 복잡한 동작을 구현할 수 있습니다. 물리를 사용하려면 Rigidbody 및 Collider 컴포넌트를 추가하고, 3D 지오메트리를 사용하려면 MeshFilter 및 MeshRenderer를 추가하세요. 각 게임 오브젝트의 기능과 고유성은 컴포넌트의 조합에 따라 달라집니다.

Unity가 물론 모든 것을 해결하지는 못합니다. 빌트인 패턴 외에 다른 패턴이 필요한 상황이 분명히 발생할 것입니다. 다음 장에서 몇 가지 예를 살펴보겠습니다.

팩토리 패턴



공장은 하나 이상의 제품을 생산할 수 있습니다.

다른 오브젝트를 만드는 특수 오브젝트가 있으면 도움이 되는 경우가 많습니다. 많은 게임에서 게임플레이를 진행하는 도중 다양한 요소를 생성하며, 실제로 필요하기 전에는 런타임 시 무엇이 필요한지 모르는 경우가 많습니다.

팩토리 패턴에서는 이름만 들어도 알 수 있는 '공장'이라는 특수 오브젝트를 지정합니다. 하나의 레벨에서 공장은 '제품'의 생산과 관련된 많은 세부 사항을 캡슐화합니다. 이렇게 하면 코드를 손쉽게 정리할 수 있습니다.

하지만 각 제품이 공통의 인터페이스나 기본 클래스를 따르는 경우라면, 더 나아가 제품에 고유의 제조 로직을 더 포함하여 공장 자체에서는 보이지 않도록 할 수 있습니다. 따라서 새로운 오브젝트 생성의 확장성이 높아집니다.

또한 공장을 서브 클래스화하여 특정 제품만 제조하는 여러 개의 공장을 만들 수도 있습니다. 이렇게 하면 적이나 장애물을 비롯한 다양한 항목을 런타임에 생성하는 데 도움이 됩니다.

예시: 간단한 공장

게임 레벨에서 아이템을 인스턴스화하는 팩토리 패턴을 만든다고 생각해 보세요. 프리팹을 사용해 게임 오브젝트를 만드는 한편, 인스턴스를 생성할 때마다 커스텀 동작을 실행하고 싶을 수 있습니다.

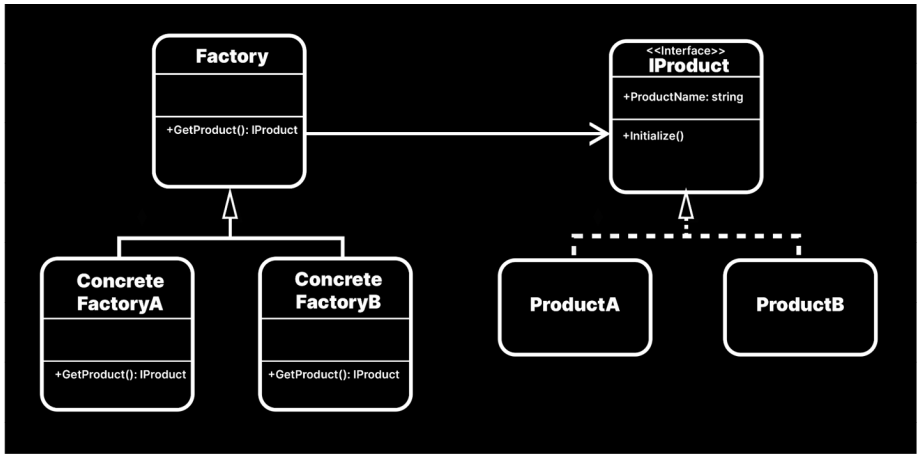
`if` 문 또는 `switch` 문을 사용하여 이 로직을 유지하는 대신, `IProduct`라는 인터페이스를 만들고 `Factory`라는 추상 클래스를 만듭니다.

```
public interface IProduct
{
    public string ProductName { get; set; }

    public void Initialize();
}

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

    // 모든 공장이 공유하는 메서드
    …
}
```

제품은 각 메서드에 맞는 특정 템플릿을 따라야 하지만, 그 외에는 어떤 기능도 공유할 필요가 없습니다. 따라서 `IProduct` 인터페이스를 정의합니다.

공장에는 일반적으로 공유하는 기능이 필요할 수 있으므로 이 샘플은 추상 클래스를 사용합니다. 서브 클래스를 사용할 때는 SOLID 원칙의 리스코프 치환을 고려하세요.

결과는 대체로 다음 구조와 같습니다.



인터페이스를 사용하여 제품 간 공유 프로퍼티 및 로직 정의

`IProduct` 인터페이스는 제품 사이의 공통 요소를 정의합니다. 이 예시에서는 `ProductName` 프로퍼티와 `Initialize`에서 제품이 실행하는 모든 로직이 공통 요소입니다.

이제 `IProduct` 인터페이스만 준수하면 필요한 제품(`ProductA`, `ProductB` 등)을 원하는 수만큼 정의할 수 있습니다.

기본 클래스인 Factory에는 IProduct를 반환하는 GetProduct 메서드가 있습니다. 추상 메서드이므로 Factory 인스턴스를 직접 만들 수는 없습니다. 두어 가지의 구상 서브 클래스(ConcreteFactoryA 및 ConcreteFactoryB)를 파생시키면 실질적으로 다른 제품을 얻게 됩니다.

이 예시에서는 특정한 위치에서 프리팹 게임 오브젝트를 더 쉽게 인스턴스화할 수 있도록 GetProduct가 Vector3 위치를 가집니다. 각 구상 공장의 필드에는 해당하는 템플릿 프리팹도 저장됩니다.

다음은 ProductA 및 ConcreteFactoryA 샘플입니다.

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

    private ParticleSystem particleSystem;

    public void Initialize()
    {
        // 이 제품에 대한 모든 고유 로직
        gameObject.name = productName;
        particleSystem = GetComponentInChildren<ParticleSystem>();
        particleSystem?.Stop();
        particleSystem?.Play();
    }
}

public class ConcreteFactoryA : Factory
{

    [SerializeField] private ProductA productPrefab;

    public override IProduct GetProduct(Vector3 position)
    {
        // 프리팹 인스턴스를 생성하고 제품 컴포넌트 가져오기
        GameObject instance = Instantiate(productPrefab.gameObject,
position, Quaternion.identity);
        ProductA newProduct = instance.GetComponent<ProductA>();

        // 각 제품에 자체 로직 포함
        newProduct.Initialize();

        return newProduct;
    }
}
```

여기서는 `IProduct`를 구현하는 제품 클래스 MonoBehaviour가 공장의 프리팹을 활용하도록 설정했습니다.

각 제품이 어떻게 고유의 `Initialize` 버전을 가질 수 있는지 참고하세요. 예시인 ProductA 프리팹은 ParticleSystem을 포함하며 이는 ConcreteFactoryA가 복사본을 인스턴스화할 때 재생됩니다. 공장 자체는 파티클을 트리거하는 로직을 포함하지 않으며, 모든 제품에 공통적으로 적용되는 `Initialize` 메서드만 호출합니다.

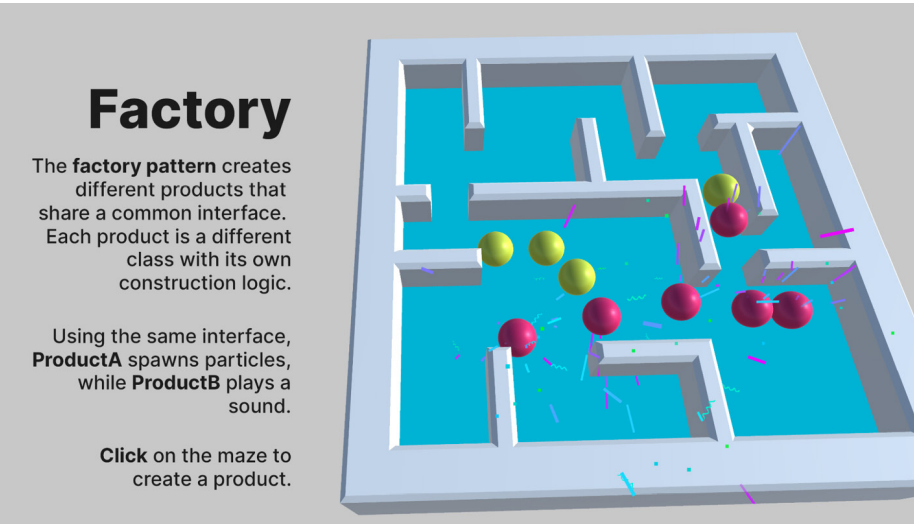
다음 샘플 프로젝트에서 `ClickToCreate` 컴포넌트가 여러 공장 사이를 전환하며 서로 다르게 동작하는 ProductA와 ProductB를 만드는 방법을 자세히 살펴보세요. ProductB는 생성되며 소리를 재생하고, ProductA는 파티클 효과를 발동합니다.

장점과 단점

팩토리 패턴은 많은 제품을 설정할 때 가장 유용합니다. 애플리케이션에서 제품 유형을 새로 정의하더라도 기존 유형이 변경되지 않으며, 이전 코드를 수정할 필요가 없습니다.

각 제품의 내부 로직을 제품 자체 클래스 내부에 두면 공장 코드를 비교적 짧게 유지할 수 있습니다. 각 공장은 제품별로 `Initialize`를 호출하는 방법만 알고 기본적인 세부 사항은 알지 못합니다.

하지만 이때 패턴을 구현하려면 다수의 클래스와 서브 클래스를 만들어야 한다는 단점이 있습니다. 다른 패턴과 마찬가지로 이로 인해 약간의 오버헤드가 발생하며, 이는 매우 다양한 제품이 필요하지 않다면 없어도 되는 오버헤드입니다.



한 제품은 소리를 재생하고 다른 제품은 파티클 효과를 재생합니다. 두 제품 모두 같은 인터페이스를 사용합니다.

개선 방안

여기에서 소개하는 내용과 매우 다른 방식으로 공장을 구현할 수 있습니다. 팩토리 패턴을 직접 만들 때는 다음과 같은 부분을 조정해 보세요.

— **딕셔너리를 사용하여 제품 검색:** 제품을 키-값 페어로 딕셔너리에 저장하는 것이 좋습니다. 고유한 문자열 식별자(이름 또는 ID 등)를 키로 사용하고 유형을 값으로 사용하세요. 그러면 제품 또는 제품의 공장을 더 편리하게 검색할 수 있습니다.

— **공장(또는 공장 관리자)을 정적 클래스로 설정:** 이렇게 하면 사용은 더 쉽지만 추가 설정이 필요합니다. 정적 클래스는 인스펙터(Inspector)에 표시되지 않으므로 제품의 컬렉션도 정적으로 만들어야 합니다.

— **게임 오브젝트 및 MonoBehaviour가 아닌 요소에 적용:** 프리팹 또는 기타 Unity별 컴포넌트로 한정하지 마세요. 팩토리 패턴은 어떤 C# 오브젝트에도 적용할 수 있습니다.

— **오브젝트 풀 패턴과 결합:** 공장이 반드시 새로운 오브젝트를 인스턴스화하거나 만들어야 하는 것은 아닙니다. 계층 구조에 있는 기존의 오브젝트를 검색할 수도 있습니다. 많은 오브젝트를 한 번에 인스턴스화하는 경우(예: 무기에서 발사되는 발사체), 오브젝트 풀 패턴을 활용하여 메모리 관리를 더 최적화하세요.

공장은 필요한 모든 게임플레이 요소를 생성할 수 있습니다. 하지만 공장의 목적이 제품 생성에만 한정되지 않는 경우가 많습니다. 팩토리 패턴을 다른 더 큰 작업(예: 게임 레벨의 일부인 다이얼로그 상자의 UI 요소 설정)의 일부로 사용할 수 있습니다.

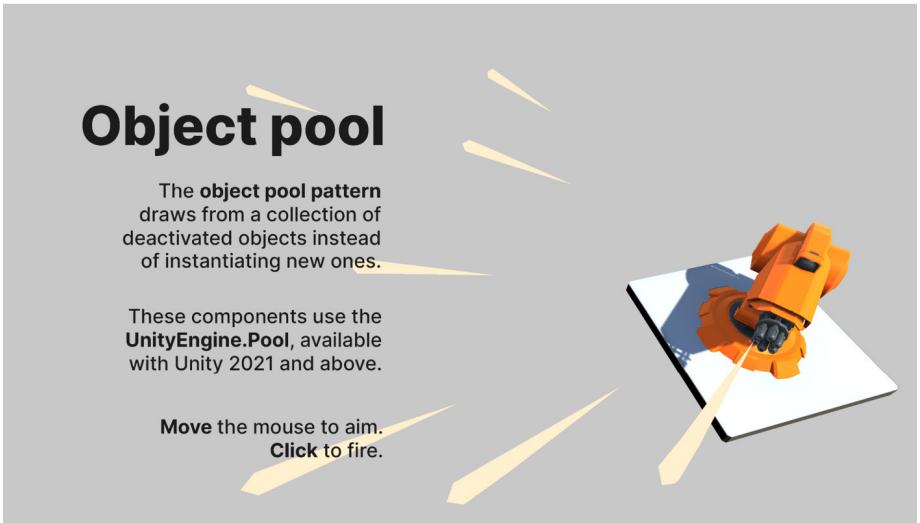
오브젝트 풀

오브젝트 풀링은 많은 수의 게임 오브젝트를 생성하고 파괴할 때 CPU의 부담을 줄이는 최적화 기법입니다.

오브젝트 풀 패턴은 비활성화된 풀에서 준비된 상태로 대기하는 초기화된 오브젝트 세트를 사용합니다. 오브젝트가 필요할 때 애플리케이션은 오브젝트를 인스턴스화할 필요가 없습니다. 대신 풀에서 게임 오브젝트를 요청하고 활성화하면 됩니다.

사용이 끝난 오브젝트는 파괴하는 대신 비활성화하고 풀로 되돌립니다.

오브젝트 풀은 가비지 컬렉션의 스파이크로 불안정한 상황이 발생하는 것을 줄일 수 있습니다. GC 스파이크는 대량의 오브젝트 생성이나 파괴 시 메모리 할당으로 인해 발생하는 경우가 많습니다. 로딩 화면처럼 사용자가 끊김 현상을 알아차리지 않는 적절한 시점에 오브젝트 풀을 미리 인스턴스화할 수 있습니다.



오브젝트 풀은 게임플레이 끊김 현상 없이 총알을 발사하도록 구현하는 데 도움이 됩니다.

예시: 단순한 풀 시스템

2개의 MonoBehaviour가 정의된 단순한 풀링 시스템을 고려하세요.

— 가져올 게임 오브젝트의 컬렉션이 있는 ObjectPool

— 프리팹에 추가된 PooledObject 컴포넌트. 복제된 각 항목이 풀에 대한 레퍼런스를 유지하는 데 도움이 됩니다.

ObjectPool에서 풀의 크기를 설명하는 필드, 저장하려는 PooledObject 프리팹, 풀 자체를 형성할 컬렉션(본 예시의 스택)을 설정합니다.

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

    // 풀링된 오브젝트를 컬렉션에 저장
    private Stack<PooledObject> stack;

    private void Start()
    {
        SetupPool();
    }

    // 풀 생성(지연을 인지할 수 없을 때 호출)
    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 메서드는 오브젝트 풀을 채웁니다. PooledObjects의 스택을 새로 만든 다음 objectToPool의 사본을 인스턴스화하여 initPoolSize 요소로 채웁니다. Start에 SetupPool을 호출하여 게임플레이 중에 한 번 실행되도록 합니다.

풀링된 항목을 검색(GetPooledObject)하고 풀로 해당 항목을 반환(ReturnToPool)할 메서드도 필요합니다.

```
// 풀에서 첫 번째 액티브 게임 오브젝트를 반환합니다.
    public PooledObject GetPooledObject()
    {

        // 풀이 충분히 크지 않으면 새로운 PooledObjects를 인스턴스화합니다.
        if (stack.Count == 0)
        {
            PooledObject newInstance = Instantiate(objectToPool);
            newInstance.Pool = this;
            return newInstance;
        }
        // 그렇지 않으면 목록에서 다음 항목을 가져옵니다.
        PooledObject nextInstance = stack.Pop();
        nextInstance.gameObject.SetActive(true);
        return nextInstance;
    }

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

GetPooledObject는 풀이 비어 있을 때만 새 PooledObject를 만듭니다. 그렇지 않으면 사용 가능한 다음 요소를 반환합니다. 풀이 충분히 큰 경우에는 대체로 기존 게임 오브젝트에 대한 레퍼런스만 가져와야 합니다.

그러면 GetPooledObject를 호출하는 클라이언트는 풀링된 오브젝트를 제자리로 이동/회전시켜야 합니다.

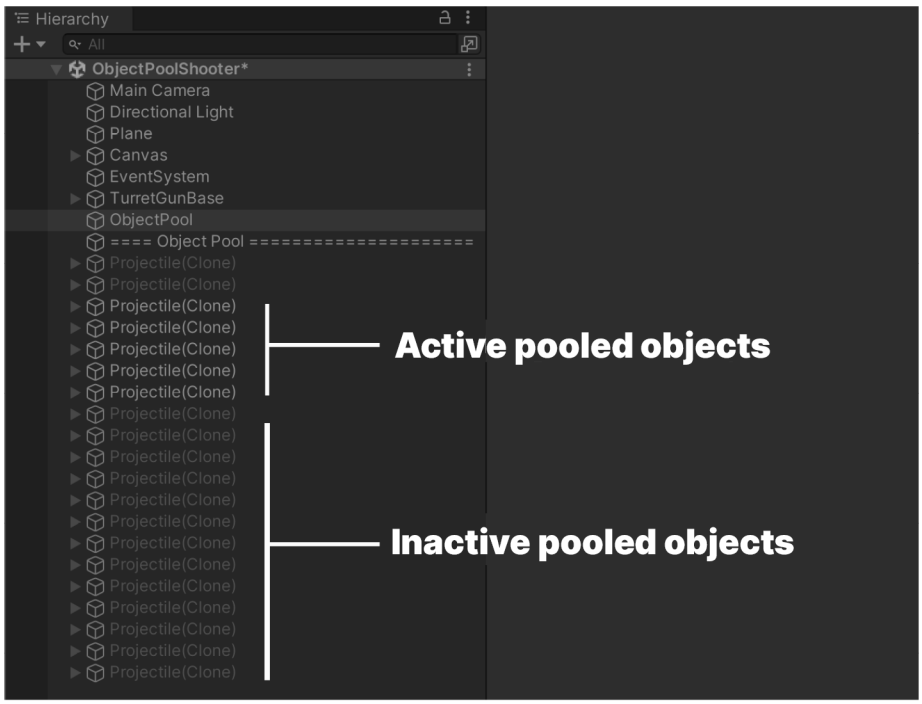
풀링된 각 요소는 단지 ObjectPool을 참조하기 위한 작은 PooledObject 컴포넌트를 가집니다.

```
public class PooledObject : MonoBehaviour
{
    private ObjectPool pool;
    public ObjectPool Pool { get => pool; set => pool = value; }

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

Release를 호출하면 게임 오브젝트가 비활성화되고 풀 대기열로 반환됩니다.

함께 제공되는 프로젝트에서 기본적인 사용 예시를 볼 수 있습니다. 여기서는 게임 오브젝트에 ExampleGun 스크립트가 연결됩니다. 해당 스크립트에는 오브젝트 풀에 대한 레퍼런스가 있습니다. 사용자가 무기를 발사하면 무기 스크립트는 Object.Instantiate를 호출하는 대신 GetPooledObject 메서드를 호출합니다.

발사체 자체에는 ExampleProjectile 스크립트와 PooledObject 스크립트가 있습니다. ExampleProjectile에는 발사된 각 총알 게임 오브젝트를 몇 초 후에 비활성화한 다음 사용 가능한 풀로 반환하는 Deactivate 메서드가 있습니다.



풀링된 오브젝트 비활성화 및 재사용

이렇게 하면 화면 밖으로 수백 발의 총알을 발사하는 것처럼 연출할 수 있지만, 실제로는 그저 총알을 비활성화하고 재사용하는 것입니다. 풀은 반드시 활성화된 오브젝트를 동시에 표시할 수 있을 만큼 커야 합니다.

풀 크기를 초과해야 하는 경우, 풀에서 추가 오브젝트를 인스턴스화할 수 있습니다. 하지만 대체로 풀은 기존의 비활성 오브젝트에서 가져옵니다.

Unity의 ParticleSystem을 사용해 봤다면 오브젝트 풀을 직접 경험한 적이 있는 것입니다. ParticleSystem 컴포넌트에는 최대 파티클 수에 대한 설정이 있습니다. 이 설정을 사용하면 가능한 파티클을 재활용하고, 파티클 효과가 최대 수를 초과하지 않게 방지합니다. 오브젝트 풀도 유사하게 작동하지만 선택하는 모든 게임 오브젝트에 사용할 수 있습니다.

개선 방안

위에서의 예시는 단순한 편이었습니다. 오브젝트 풀을 실제 프로젝트에 배포할 때는 다음과 같이 업그레이드해 보세요.

— **정적 또는 싱글톤으로 설정:** 다양한 소스에서 풀링된 오브젝트를 생성해야 하는 경우에는 오브젝트 풀을 정적으로 설정해 보세요. 그러면 애플리케이션의 어디에서든 액세스할 수 있으나, 인스펙터 사용은 불가능합니다. 또는 오브젝트 풀 패턴을 싱글톤으로 설정하면 어디에서든 액세스하여 간편하게 사용할 수 있습니다.

— **딕셔너리로 다수의 풀 관리:** 많은 수의 다양한 프리팹을 풀링해야 하는 경우, 개별적인 풀에 저장하고 키-값 페어를 저장하면 쿼리할 풀을 알 수 있습니다(프리팹의 InstanceID는 고유 키로 작동할 수 있음).

— **사용되지 않는 게임 오브젝트를 창의적으로 제거:** 오브젝트 풀을 효과적으로 활용하는 방법 중 하나는 사용되지 않는 오브젝트를 숨기고 풀로 반환하는 것입니다. 가능한 기회를 모두 활용해 풀링된 오브젝트(화면 바깥에 있거나 폭발에 의해 숨겨진 오브젝트 등)를 비활성화하세요.

— **오류 확인:** 이미 풀에 있는 오브젝트를 릴리스하지 마세요. 런타임에서 오류가 발생할 수 있습니다.

— **최대 크기/제한 추가:** 풀링된 오브젝트가 많으면 메모리를 소모합니다. 풀이 너무 많은 리소스를 사용하지 않도록 특정 한도를 초과하는 오브젝트를 제거해야 할 수 있습니다.

오브젝트 풀을 사용하는 방식은 애플리케이션에 따라 달라집니다. 이 패턴은 주로 탄막 슈팅 게임처럼 총이나 무기에서 다수의 발사체를 발사해야 하는 경우에 보입니다.

다수의 오브젝트를 인스턴스화할 때마다 가비지 컬렉션의 스파이크로 인해 짧은 끊김 현상이 발생할 위험이 있습니다. 오브젝트 풀은 이러한 문제를 완화하여 게임플레이를 원활하게 유지합니다.

2021 이상 버전의 Unity를 사용 중이라면 빌트인 오브젝트 풀링 시스템이 포함되므로, 위 예시에서처럼 PooledObject 또는 ObjectPool 클래스를 자체적으로 만들 필요가 없습니다.

UnityEngine.Pool

오브젝트 풀 패턴은 이제 Unity 2021에서 자체 UnityEngine.Pool API를 지원할 정도로 널리 사용됩니다. 이 API는 오브젝트 풀 패턴을 가지는 오브젝트를 추적하기 위한 스택 기반 ObjectPool을 제공합니다. 필요에 따라 CollectionPool(List, HashSet, Dictionary 등)도 사용할 수 있습니다.

샘플 프로젝트(씬 참조)에서는 커스텀 풀 컴포넌트가 더이상 필요하지 않습니다. 대신 상단에 using UnityEngine.Pool; 라인을 사용하여 총 스크립트를 업데이트하세요. 그러면 빌트인 ObjectPool로 발사체 풀을 생성할 수 있습니다.

```
using UnityEngine.Pool;

public class RevisedGun : MonoBehaviour
{
    …

    // Unity 2021 이상 버전에서 사용 가능한 스택 기반 ObjectPool
    private IObjectPool<RevisedProjectile> objectPool;

    // 이미 풀에 있는 기존 항목을 반환하려 할 때 예외를 반환
    [SerializeField] private bool collectionCheck = true;
    // 풀의 용량과 최대 크기를 제어하는 추가 옵션
    [SerializeField] private int defaultCapacity = 20;
    [SerializeField] private int maxSize = 100;

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

    // 오브젝트 풀을 채울 항목을 만들 때 호출됨
    private RevisedProjectile CreateProjectile()
    {
        RevisedProjectile projectileInstance =
Instantiate(projectilePrefab);
        projectileInstance.ObjectPool = objectPool;
        return projectileInstance;
    }

    // 오브젝트 풀로 항목을 반환할 때 호출됨
    private void OnReleaseToPool(RevisedProjectile pooledObject)
    {
        pooledObject.gameObject.SetActive(false);
    }

    // 오브젝트 풀에서 다음 항목을 검색할 때 호출됨
    private void OnGetFromPool(RevisedProjectile pooledObject)
    {
        pooledObject.gameObject.SetActive(true);
    }

    // 풀링된 항목의 최대 개수를 초과할 때 호출됨(풀링된 오브젝트 파괴)
    private void OnDestroyPooledObject(RevisedProjectile pooledObject)
    {
        Destroy(pooledObject.gameObject);
    }

    private void FixedUpdate()
    {
        …

    }
}
```

스크립트 내용의 대부분이 원본 ExampleGun 스크립트에서 작동합니다. 하지만 이제 다음과 같은 시점에 로직을 설정하는 유용한 기능이 ObjectPool 생성자에 포함됩니다.

— 풀을 채우기 위해 풀링된 항목을 먼저 생성하기

— 풀에서 항목 가져오기

— 풀로 항목 반환하기

— 풀링된 오브젝트 파괴하기(예: 최대 한도에 도달한 경우)

이제 생성자로 전달할 해당 메서드를 정의해야 합니다.

빌트인 ObjectPool에서 어떤 식으로 기본 풀 크기 및 최대 풀 크기 옵션도 포함하는지 참고하세요. 최대 풀 크기를 초과하는 항목은 자동 파괴 행동을 트리거하여 메모리 사용량을 억제합니다.

발사체 스크립트는 ObjectPool에 레퍼런스를 유지하도록 일부 수정됩니다. 이렇게 하면 오브젝트를 풀로 더 간편하게 릴리스할 수 있습니다.

```
public class RevisedProjectile : MonoBehaviour
{
    …

    private IObjectPool<RevisedProjectile> objectPool;

    // 발사체에 ObjectPool에 대한 레퍼런스를 제공하는 공용 프로퍼티
    public IObjectPool<RevisedProjectile> ObjectPool { set =>
objectPool = value; }

   …
}
```

UnityEngine.Pool API가 오브젝트 풀을 더 빠르게 설정하므로, 이제 패턴을 처음부터 다시 만들 필요가 없어져 그만큼 시간을 단축할 수 있습니다.

싱글톤 패턴

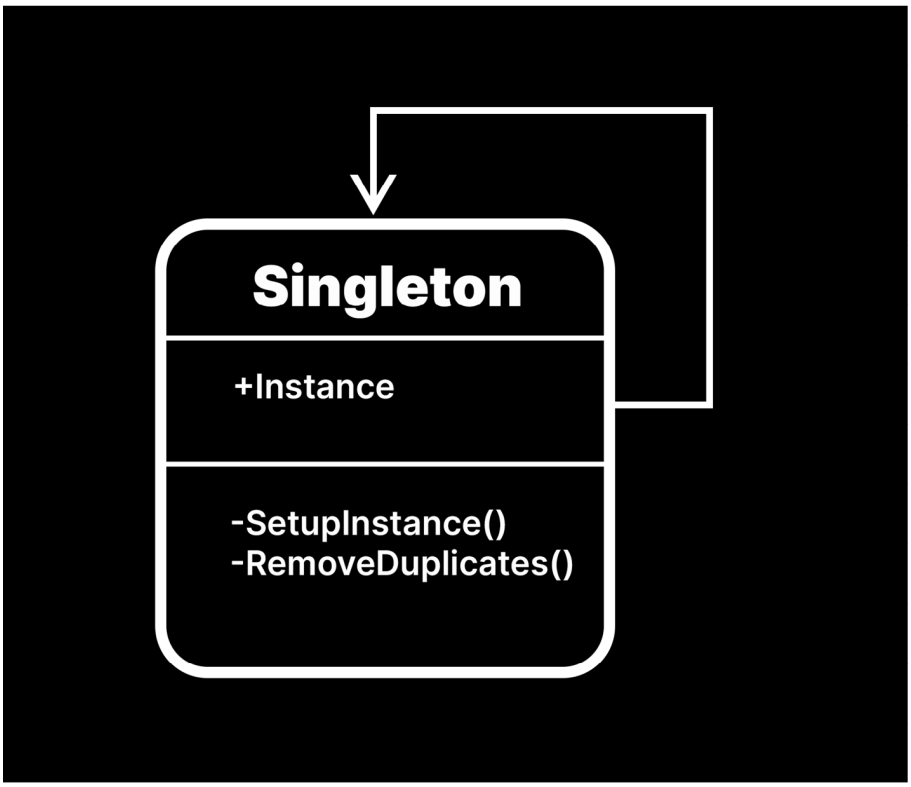
싱글톤은 일종의 누명을 쓰고 있습니다. 싱글톤은 Unity 개발이 처음이라면 실무에서 가장 먼저 알게 되는 패턴 중 하나이자, 가장 과소평가되는 디자인 패턴이기도 합니다.

원조 GoF에 따르면 싱글톤 패턴은 다음을 수행합니다.

— 클래스가 자체의 인스턴스 하나만을 인스턴스화하도록 보장

— 해당하는 하나의 인스턴스에 대한 손쉬운 글로벌 액세스 제공

전체 씬에서 행동을 조정하는 오브젝트가 정확히 하나만 필요할 때 유용합니다. 예를 들면 씬에 메인 게임 루프를 총괄하는 게임 관리자가 딱 하나만 필요할 수 있습니다. 한 번에 하나의 파일 관리자만 파일 시스템에 작성하기를 원할 수도 있습니다. 이러한 관리자 레벨의 오브젝트는 대체로 싱글톤 패턴을 적용하기에 좋은 대상입니다.



SimpleSingleton은 첫 인스턴스 이후의 모든 인스턴스를 파괴합니다.

게임 프로그래밍 패턴에서는 싱글톤을 득보다는 해가 많은 안티 패턴으로 소개합니다. 이렇게 평판이 좋지 않은 이유는 쉽게 사용할 수 있는 만큼 오용되기 쉬운 패턴이기 때문입니다. 개발자는 싱글톤을 부적절한 상황에 적용하는 경향이 있으며, 이로 인해 불필요한 전역 상태나 종속성이 발생하게 됩니다.

Unity에서 싱글톤을 만드는 방법을 알아보고 어떤 장단점이 있는지 살펴보겠습니다. 그런 다음 자신의 애플리케이션에 적용할 가치가 있는지 직접 결정하세요.

예시: 단순한 싱글톤

다음은 가장 단순한 싱글톤의 예시입니다.

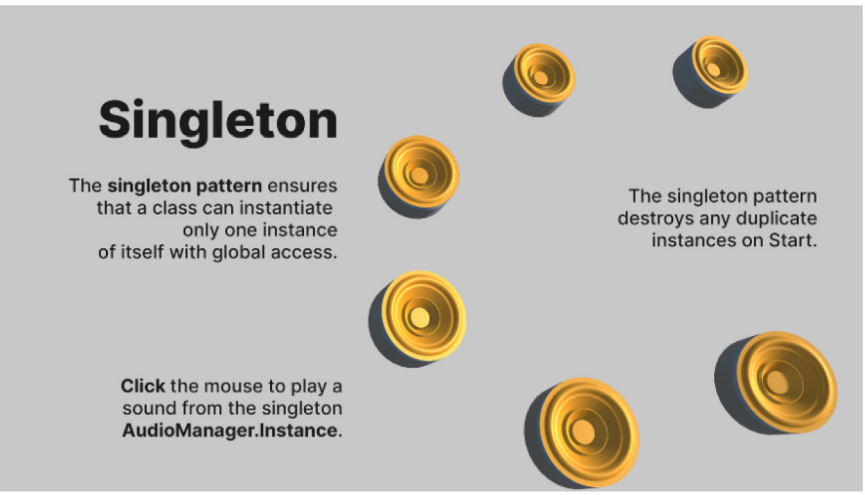
```
using UnityEngine;

public class SimpleSingleton : MonoBehaviour
{
    public static SimpleSingleton Instance;

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

공용 정적 Instance는 씬에서 Singleton의 인스턴스 하나를 갖게 됩니다.

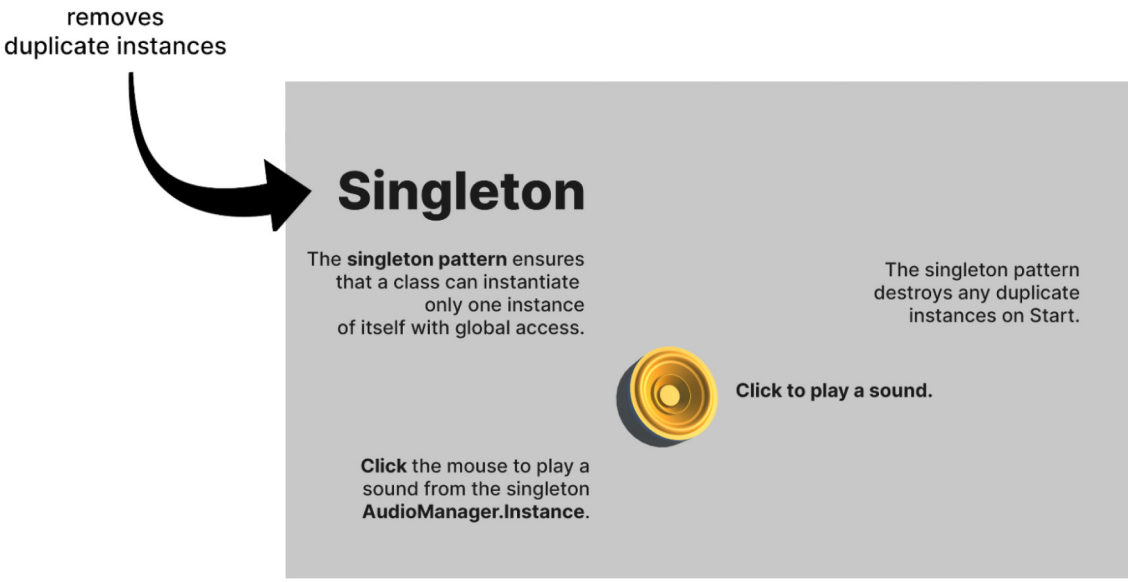
Awake 메서드에서 인스턴스가 이미 설정되어 있는지 확인합니다. Instance가 현재 null이면 인스턴스는 이 특정 오브젝트로 설정됩니다. 이 오브젝트는 씬에서 가장 첫 번째 싱글톤이어야 합니다.

그렇지 않으면 이 인스턴스는 복제본이어야 하며, 싱글톤이 씬에서 그러한 컴포넌트를 오직 하나만 갖게 하도록 Destroy(gameObject)를 호출합니다.

런타임에 계층 구조에서 둘 이상의 게임 오브젝트에 스크립트를 연결하는 경우, Awake의 로직은 첫 오브젝트만 유지하고 나머지는 폐기합니다.



싱글톤 패턴은 오직 하나의 인스턴스만 허용합니다.



Instance 필드는 공용 및 정적 필드입니다. 어떤 컴포넌트라도 씬 어디에서든 하나의 싱글톤에 글로벌 액세스할 수 있습니다.

지속성 및 지연 인스턴스화

SimpleSingleton은 작성된 대로 작동합니다. 하지만 다음과 같은 두 가지 문제가 있습니다.

— 새로운 씬을 로드하면 게임 오브젝트가 파괴됩니다.

— 사용하기 전에 계층 구조에서 싱글톤을 설정해야 합니다.

싱글톤은 보편적인 관리자 스크립트 역할을 하는 경우가 많으므로 DontDestroyOnLoad를 사용하여 지속성을 갖게 해서 이점을 활용할 수 있습니다.

나아가 지연 인스턴스화를 사용하면 싱글톤이 처음으로 필요할 때 자동으로 만들어지도록 할 수 있습니다. 게임 오브젝트를 만들고 적절한 싱글톤 컴포넌트를 추가하기 위한 로직만 있으면 됩니다.

다음은 개선된 싱글톤의 예시입니다.

```
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);
          }
      }
```

Instance는 이제 프라이빗 instance 지원 필드의 공용 프로퍼티입니다. 싱글톤을 처음 참조할 때는 가져오려는 인스턴스가 있는지 확인하세요. 인스턴스가 존재하지 않으면, SetupInstance 메서드가 적절한 컴포넌트를 갖는 게임 오브젝트를 만듭니다.

DontDestroyOnLoad(gameObject)는 씬 로드로 인해 계층 구조에서 싱글톤이 지워지지 않게 합니다. 이제 싱글톤 인스턴스는 지속적이며, 게임에서 씬을 변경해도 액티브 상태를 유지합니다.

제네릭 사용

어떤 버전의 스크립트도 같은 씬 내에서 여러 싱글톤을 만드는 방법을 다루지 않습니다. 예를 들어 AudioManager로서 작동하는 싱글톤과 GameManager로 작동하는 다른 싱글톤이 필요한 경우, 이 두 싱글톤은 현재 공존할 수 없습니다. 관련 코드를 복제하고 로직을 각 클래스로 붙여 넣어야 합니다.

대신 다음과 같이 스크립트의 제네릭 버전을 만듭니다.

```
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);
        }
    }
}
```

이렇게 하면 어떤 클래스나 싱글톤으로 변환할 수 있습니다. 클래스를 선언하면 간단히 제네릭 싱글톤에서 상속합니다. 예를 들어 GameManager라는 MonoBehaviour를 다음과 같이 선언하여 싱글톤으로 만들 수 있습니다.

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

이제 필요할 때마다 언제나 공용 정적 `GameManager.Instance`를 참조할 수 있습니다.

장점과 단점

싱글톤은 여러 측면에서 SOLID 원칙을 위반한다는 점에서 본 가이드의 다른 패턴과 다릅니다. 많은 개발자들이 다음과 같은 여러 이유로 싱글톤에 호의적이지 않습니다.

— **싱글톤에는 글로벌 액세스가 필요합니다.** 싱글톤은 글로벌 인스턴스로 사용하므로 많은 종속성을 숨길 수 있으며, 이에 따라 훨씬 해결하기 어려운 버그를 유발합니다.

— **싱글톤은 테스트를 어렵게 만듭니다.** 유닛 테스트는 반드시 서로 독립적으로 수행해야 합니다. 싱글톤은 씬 전반에서 많은 게임 오브젝트의 상태를 변경할 수 있으므로 테스팅에 방해가 될 수 있습니다.

— **싱글톤은 결합도가 높아지게 합니다.** 본 가이드에서 소개하는 패턴은 대부분 종속성 분리를 시도하지만, 싱글톤은 이와 반대입니다. 결합도가 높으면 리팩터링하기가 어렵습니다. 컴포넌트 하나를 변경하면 연결된 컴포넌트에 영향을 줄 수 있으므로 코드가 지저분해질 수 있습니다.

싱글톤에 대한 반감은 상당한 편입니다. 수년 동안 유지 관리하려는 엔터프라이즈 수준의 게임을 제작할 예정이라면 싱글톤은 적절한 선택이 아닙니다.

하지만 엔터프라이즈 수준의 애플리케이션이 아닌 게임도 많으며, 그러한 게임은 비즈니스 소프트웨어에 준하는 방식으로 계속 확장할 필요가 없습니다.

사실 싱글톤에는 확장성이 필요하지 않은 소규모 게임 개발에 유용할 수 있는 장점이 있습니다.

— **싱글톤은 비교적 빨리 배울 수 있습니다.** 코어 패턴 자체가 다소 직관적입니다.

— **싱글톤은 사용자 친화적입니다.** 다른 컴포넌트에서 싱글톤을 사용하려 할 때, 공용 및 정적 인스턴스만 참조하면 됩니다. 싱글톤 인스턴스는 필요할 때 씬의 어떤 오브젝트에서도 항상 사용할 수 있습니다.

— **싱글톤은 성능을 보장합니다.** 정적 싱글톤 인스턴스에 항상 글로벌 액세스가 가능하므로, 속도가 느린 경향이 있는 GetComponent 또는 Find 연산의 결과를 캐시하지 않아도 됩니다.

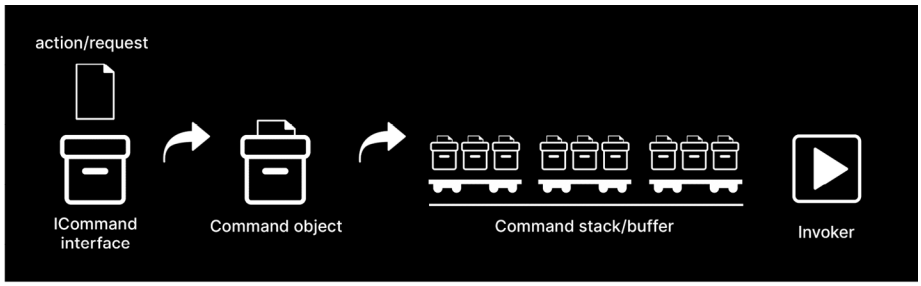
이러한 방식으로 씬의 다른 모든 게임 오브젝트에서 항상 액세스할 수 있는 관리자 오브젝트(게임 플로 관리자 또는 오디오 관리자 등)를 만들 수 있습니다. 또한 오브젝트 풀을 구현한 경우, 풀링 시스템을 싱글톤으로 디자인하면 풀링된 오브젝트를 더 쉽게 가져올 수 있습니다.

프로젝트에서 싱글톤을 사용하기로 했으면 최소한으로 유지하세요. 무분별하게 사용하지 않아야 합니다. 싱글톤은 글로벌 액세스의 장점을 활용할 수 있는 소수의 스크립트에만 사용하세요.

커맨드 패턴

고유의 GoF 패턴 중 하나인 커맨드는 일련의 특정 행동을 추적하려는 경우에 유용합니다. 실행 취소/다시 실행 기능이 사용되거나 입력 내역이 목록으로 유지되는 게임을 플레이해 본 적이 있다면 아마 커맨드 패턴을 본 적이 있을 것입니다. 사용자가 실제로 여러 턴을 실행하기 전에 계획할 수 있는 전략 게임을 생각해 보세요. 그것이 바로 커맨드 패턴입니다.

메서드를 직접 호출하는 대신 커맨드 패턴을 사용하면 '커맨드 오브젝트'라는 하나 이상의 메서드 호출을 캡슐화할 수 있습니다.



커맨드 패턴으로 행동 저장하기

커맨드 오브젝트를 대기열이나 스택 같은 컬렉션에 두면 오브젝트의 실행 타이밍을 제어할 수 있습니다. 이는 작은 버퍼 같은 기능을 합니다. 그런 다음에는 일련의 행동을 나중에 재생할 수 있도록 잠재적으로 지연하거나 실행을 취소할 수 있습니다.

커맨드 패턴을 구현하려면 행동을 포함할 일반 오브젝트가 필요합니다. 이 커맨드 오브젝트에는 로직을 통해 수행할 작업과 해당 작업을 실행 취소하는 방법이 포함됩니다.

커맨드 오브젝트 및 커맨드 호출자

다양한 방법으로 구현할 수 있지만, 인터페이스를 사용하는 버전은 다음과 같습니다.

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

이 예시에서는 모든 게임플레이 행동이 ICommand 인터페이스를 적용합니다(추상 클래스로 구현할 수도 있음).

각 커맨드 오브젝트는 자체 Execute 및 Undo 메서드를 처리합니다. 따라서 게임에 더 많은 커맨드를 추가해도 기존의 커맨드에는 아무런 영향을 주지 않습니다.

커맨드를 실행 및 취소하려면 다른 클래스가 필요합니다. CommandInvoker 클래스를 생성합니다. ExecuteCommand 및 UndoCommand 메서드와 더불어, 커맨드 오브젝트의 시퀀스를 포함하기 위한 undo 스택이 있습니다.

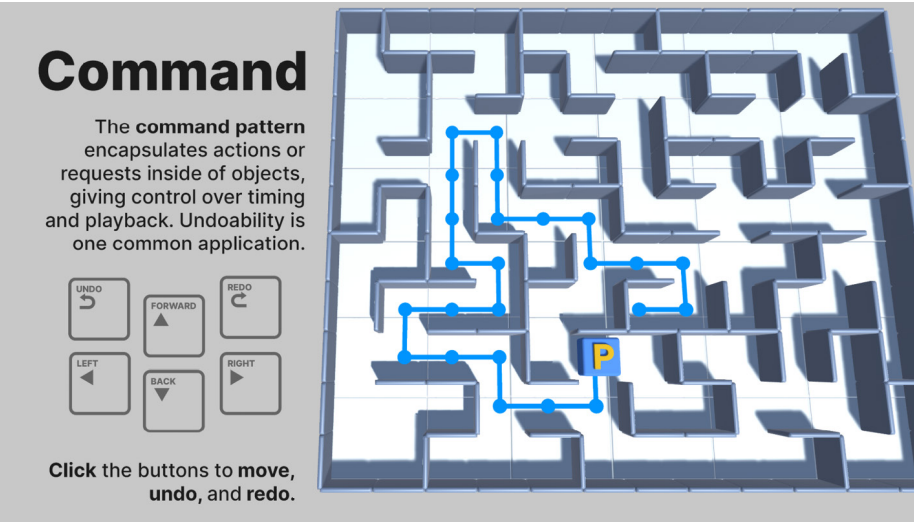
```
public class CommandInvoker
{
    private static Stack<ICommand> undoStack = new Stack<ICommand>();

    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();
        }
    }
}
```

예시: 실행 취소 가능한 이동

플레이어를 애플리케이션의 미로 안에서 이동하도록 구현하는 경우를 예로 들겠습니다. 먼저 플레이어의 위치 이동을 처리하는 PlayerMover를 만들 수 있겠습니다.

```
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);
    }
}
```

플레이어를 동서남북 방향에 따라 안내하기 위해 Move 메서드로 Vector3를 전달합니다. 레이캐스트를 사용해 적절한 LayerMask의 벽을 감지할 수도 있습니다. 물론 커맨드 패턴에 적용하려는 항목의 구현과 패턴 자체는 서로 별개입니다.



샘플의 플레이어 이동

커맨드 패턴을 따르려면 `PlayerMover`의 `Move` 메서드를 오브젝트로 캡처합니다. Move를 직접 호출하는 대신, `ICommand` 인터페이스를 구현하는 새로운 클래스인 `MoveCommand`를 만듭니다.

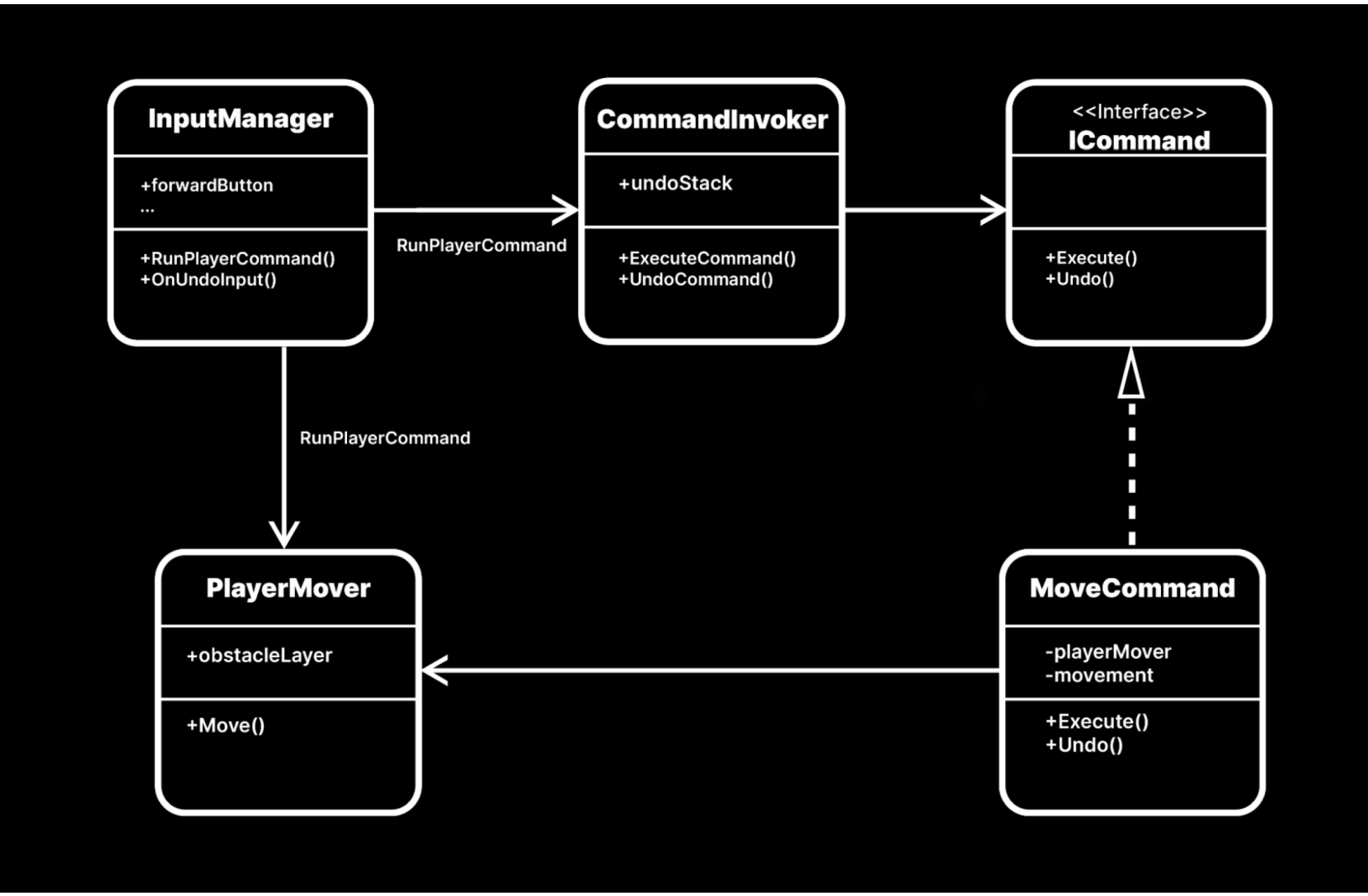
```
public class MoveCommand : ICommand
{
    PlayerMover playerMover;
    Vector3 movement;

    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`에는 구현하려는 로직을 저장할 Execute 메서드가 필요합니다. 구현하려는 로직이 무엇이든 여기에 저장되므로 movement 벡터가 있는 Move를 호출합니다.

`ICommand`에는 씬을 이전 상태로 복원하기 위한 Undo 메서드도 필요합니다. 이 예시에서는 Undo 로직이 이동 벡터를 감산하며, 기본적으로 플레이어를 반대편으로 밀어냅니다.

MoveCommand는 실행해야 하는 모든 파라미터를 저장합니다. 이를 생성자로 설정합니다. 이 예시에서는 적절한 PlayerMover 컴포넌트와 이동 벡터를 저장합니다.

커맨드 오브젝트를 만들고 필요한 파라미터를 저장하면 CommandInvoker의 정적 ExecuteCommand 및 UndoCommand 메서드가 MoveCommand에 전달됩니다. 그러면 MoveCommand의 Execute 또는 Undo가 실행되며 실행 취소 스택에서 커맨드 오브젝트를 추적합니다.



CommandInvoker, ICommand, MoveCommand

InputManager는 PlayerMover의 Move 메서드를 직접 호출하지 않습니다. 대신 RunMoveCommand 메서드를 추가하여 새로운 MoveCommand를 만들고 CommandInvoker로 전송합니다.

```
private void RunPlayerCommand(PlayerMover playerMover, Vector3
movement)
{
    if (playerMover == null)
    {
        return;
    }

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

이제 UI 버튼의 다양한 onClick 이벤트를 설정하여 4개의 이동 벡터를 가진 RunPlayerCommand를 호출합니다.

샘플 프로젝트에서 InputManager를 어떻게 구현했는지 자세히 살펴보거나, 키보드 또는 게임패드를 사용하는 입력을 직접 설정하세요. 이제 플레이어는 미로를 탐색할 수 있습니다. 실행 취소 버튼을 눌러 시작 화면으로 되돌아가세요.

장점과 단점

다시 실행 또는 실행 취소 기능을 여러 커맨드 오브젝트를 생성하듯이 간단하게 구현할 수 있습니다. 커맨드 버퍼를 사용하면 특정한 컨트롤로 행동 시퀀스를 재생할 수도 있습니다.

특정 버튼을 순서에 맞게 클릭하면 콤보 기술이나 공격이 트리거되는 격투 게임을 예로 들겠습니다. 커맨드 패턴으로 플레이어 행동을 저장하면 콤보를 훨씬 간편하게 설정할 수 있습니다.

한편 커맨드 패턴은 다른 디자인 패턴들과 마찬가지로 더 많은 구조를 유발합니다. 추가 클래스와 인터페이스가 애플리케이션에서 커맨드 오브젝트를 배포하는 데 충분한 이점을 제공하는 위치를 파악해 결정해야 합니다.

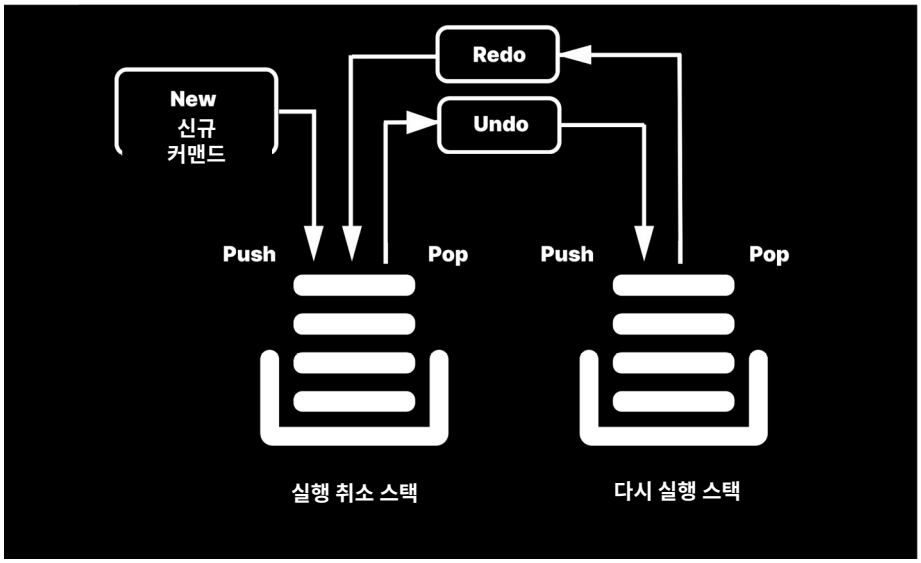
개선 방안

기본을 익힌 후에는 커맨드의 타이밍을 조정하고 컨텍스트에 따라 커맨드를 연속적으로 또는 역순으로 재생할 수 있습니다.

커맨드 패턴을 통합할 때 고려해야 할 부분은 다음과 같습니다.

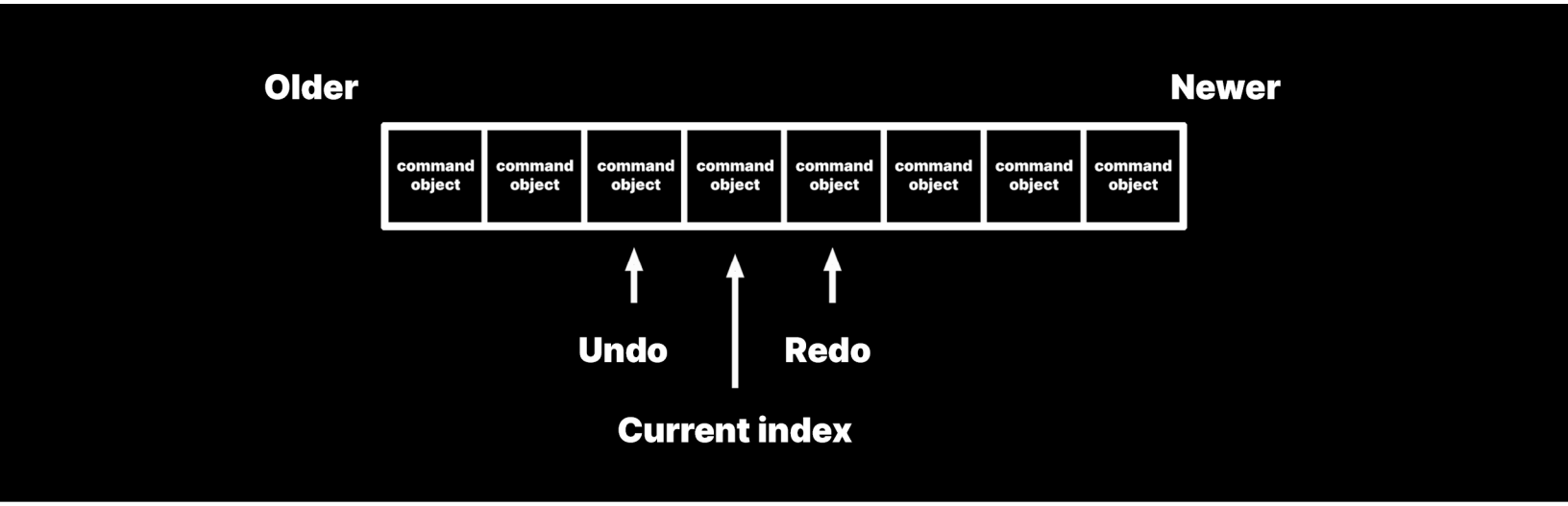
— **커맨드를 더 많이 만듭니다.** 샘플 프로젝트에는 한 가지 유형의 커맨드 오브젝트인 `MoveCommand`만 포함됩니다. `ICommand`를 구현하는 커맨드 오브젝트를 원하는 수만큼 만들고 `CommandInvoker`를 사용하여 추적할 수 있습니다.

— **스택만 하나 더 추가하면 다시 실행 기능을 추가할 수 있습니다.** 커맨드 오브젝트를 취소할 때는 다시 실행 작업을 추적하는 별도의 스택으로 푸시하면 됩니다. 그러면 실행 취소 내역을 빠르게 순환하거나 해당 행동을 다시 실행할 수 있습니다. 사용자가 완전히 새로운 이동을 호출하면 다시 실행 스택을 지웁니다(제공된 샘플 프로젝트에서 구현된 결과를 볼 수 있음).



스택 실행 취소 및 다시 실행

— **커맨드 오브젝트의 버퍼에 다른 컬렉션을 사용합니다.** FIFO(선입선출) 동작이 필요한 경우 대기열을 사용하는 것이 더 편할 수 있습니다. 목록을 사용한다면 현재 액티브 인덱스를 추적하세요. 액티브 인덱스 이전의 커맨드는 실행 취소가 가능합니다. 인덱스 이후의 커맨드는 다시 실행할 수 있습니다.



목록이나 다른 컬렉션은 커맨드 버퍼의 역할을 합니다.

— **스택 크기를 제한합니다.** 실행 취소 및 다시 실행 작업으로 순식간에 통제 불능 상태에 빠질 수도 있습니다. 스택을 커맨드의 최종적인 개수로 제한하세요.

— **필요한 파라미터는 생성자로 전달합니다.** MoveCommand 예시에서처럼 로직을 캡슐화하는 데 도움이 됩니다.

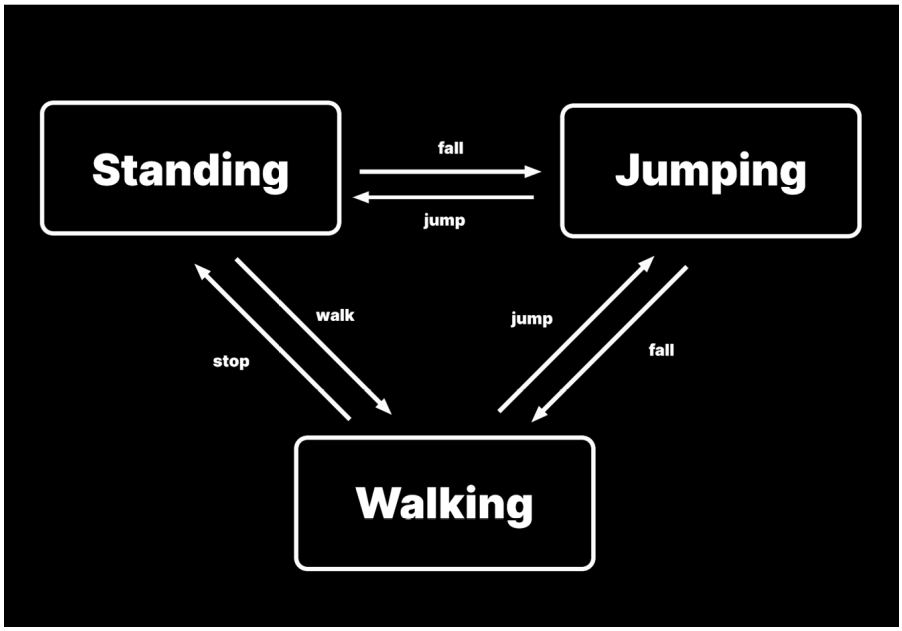
다른 외부 오브젝트처럼 `CommandInvoker`는 커맨드 오브젝트의 내부 작업을 보지 않으며, Execute 또는 Undo만 호출합니다. 생성자를 호출할 때 작업에 필요한 모든 데이터를 커맨드 오브젝트에 제공하세요.

상태 패턴

플레이어블 캐릭터를 구성한다고 생각해 보세요. 특정 시점에서 캐릭터는 땅에 서 있을 것입니다. 컨트롤러를 움직이면 캐릭터는 달리거나 걷는 모습을 나타내고, 점프 버튼을 누르면 공중으로 점프합니다. 몇 프레임이 경과하면 캐릭터는 착지 후 다시 대기 상태로 서 있게 됩니다.

상태 및 상태 머신

게임에서는 상호 작용이 이뤄지며, 런타임에 변경되는 여러 시스템을 추적해야 합니다. 캐릭터의 다양한 상태를 나타내는 다이어그램을 그리면 다음과 같은 모습일 것입니다.



단순한 상태 다이어그램

몇 가지 차이점이 있지만 플로 차트와 유사합니다.

- 이 다이어그램은 여러 가지 상태(대기/서 있기, 걷기, 달리기, 점프 등)로 구성되며, 어떤 시점에도 현재 상태 하나만 액티브 상태입니다.
- 각 상태는 런타임 시 조건에 따라 다른 한 가지 상태로의 전환을 트리거할 수 있습니다.
- 전환이 발생하면 출력 상태는 새로운 액티브 상태로 전환됩니다.

이 다이어그램에서 설명하는 것은 FSM(유한 상태 머신)입니다. 게임 개발에서 전형적인 사용 사례 중 하나는 게임 액터나 프랍의 내부 상태를 추적하는 것입니다.

`enum`과 `switch` 문을 사용하는 단순한 접근 방식을 사용하여 코드의 기본 FSM을 설명할 수 있습니다.

```
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()
    {
        // 걷기 및 점프 컨트롤 처리
    }

    private void Walk()
    {
        // 걷기 로직
    }

    private void Idle()
    {
        // 대기 상태 로직
    }

    private void Jump()
    {
        // 점프 로직
    }
}
```

이렇게 작성하면 작동은 하더라도 PlayerController 스크립트가 금방 복잡해질 수 있습니다. 상태와 복잡도를 추가하려면 PlayerController 스크립트의 내부를 매번 재확인해야 합니다.

예시: 단순한 상태 패턴

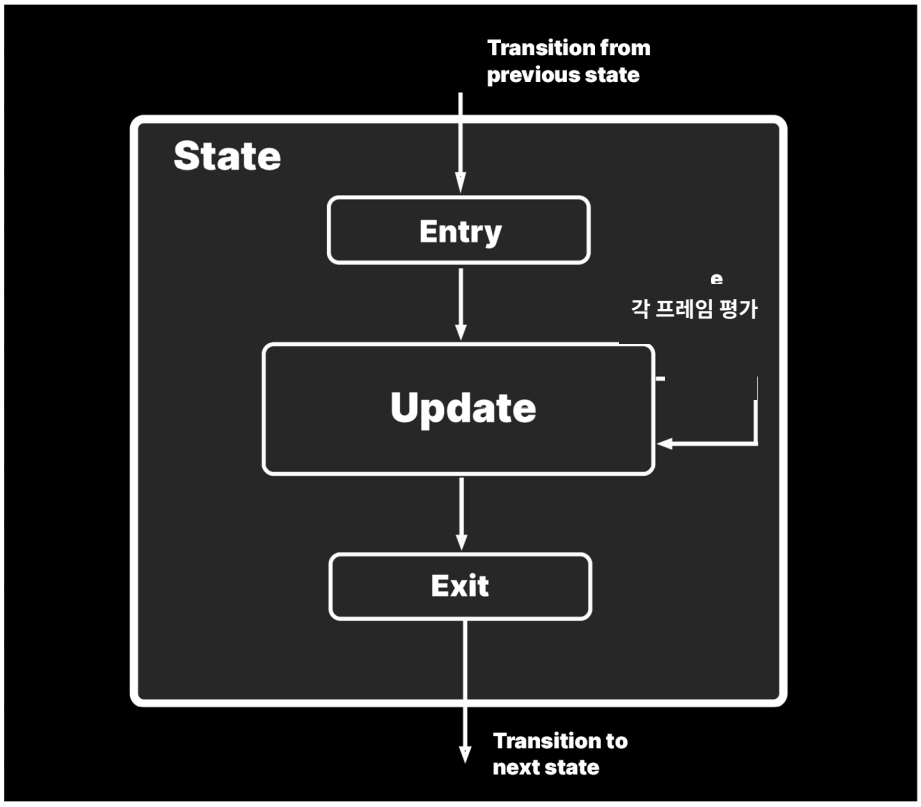
다행히도 상태 패턴은 로직을 정리하는 데 도움이 될 수 있습니다. 원조 GoF에 따르면 상태 패턴은 다음 두 가지 문제를 해결합니다.

— 오브젝트는 내부 상태 변경 시 자신의 동작을 변경해야 합니다.

— 상태별 동작은 독립적으로 정의됩니다. 새로운 상태를 추가해도 기존 상태의 동작에 영향을 주지 않습니다.

위 예시 UnrefactoredPlayerController 클래스는 상태 변경을 추적할 수 있지만, 두 번째 문제는 해결하지 못합니다. 새로운 상태를 추가할 때 기존 상태에 대한 영향을 최소화해야 하지만, 대신 상태를 오브젝트로 캡슐화할 수 있습니다.

각 상태를 다음과 같이 구조화한다고 생각해 보세요.



Entry, Exit, Update로 캡슐화된 상태

여기에서는 상태를 입력하고 조건에 의해 컨트롤 플로가 종료될 때까지 각 프레임을 계속 루프 처리합니다. 이 패턴을 구현하기 위해 IState 인터페이스를 만듭니다.

```
public interface IState
{

    public void Enter()
    {
        // 상태에 처음 진입할 때 실행되는 코드
    }

    public void Update()
    {
        // 프레임당 로직. 새로운 상태로 전환하는 조건 포함
    }

    public void Exit()
    {
        // 상태에서 벗어날 때 실행되는 코드
    }
}
```

게임에서 각 구상 상태는 `IState` 인터페이스를 실행합니다.

— **Entry:** 이 로직은 상태에 처음 진입할 때 실행됩니다.

— **Update:** 이 로직은 매 프레임마다 실행됩니다(Execute 또는 Tick이라고도 함). 물리에 대한 FixedUpdate, LateUpdate 등을 사용하여 MonoBehaviour처럼 Update 메서드를 추가로 세그먼트화할 수 있습니다.

  Update의 모든 기능은 상태 변경을 트리거하는 조건이 감지될 때까지 각 프레임을 실행합니다.

— **Exit:** 이 로직의 코드는 상태에서 벗어나 새로운 상태로 전환되기 전에 실행됩니다.

IState를 구현하는 각 상태에 대한 클래스를 만들어야 합니다. 샘플 프로젝트에서는 `WalkState`, `IdleState`, `JumpState`에 대해 별도의 클래스가 설정되었습니다.

다른 클래스인 StateMachine은 컨트롤 플로가 상태에 진입하고 상태에서 벗어나는 방법을 관리합니다. 세 가지 예시 상태와 관련된 StateMachine의 코드 예시는 다음과 같습니다.

```
[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();
        }
    }
}
```

패턴을 따르기 위해 StateMachine은 관리를 받는 각 상태에 대한 공용 오브젝트를 참조합니다(이 경우에는 walkState, jumpState, idleState). StateMachine은 MonoBehaviour에서 상속받지 않으므로 생성자를 사용하여 각 인스턴스를 설정합니다.

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

생성자에 필요한 모든 파라미터를 전달할 수 있습니다. 샘플 프로젝트에서 PlayerController는 각 상태에서 참조되며, 프레임마다 각 상태를 업데이트하는 데 사용합니다(아래 IdleState 예시 참조).

StateMachine에 대해 알아 두어야 할 내용은 다음과 같습니다.

— Serializable 속성을 사용하여 StateMachine 및 공용 필드를 인스펙터에 표시할 수 있습니다. 그러면 다른 MonoBehaviour(PlayerController 또는 EnemyController 등)가 StateMachine을 필드로 사용할 수 있습니다.

— `CurrentState` 프로퍼티는 읽기 전용입니다. StateMachine 자체는 명시적으로 이 필드를 설정하지 않습니다. 그러면 PlayerController 같은 외부 오브젝트는 `Initialize` 메서드를 호출하여 기본 상태를 설정할 수 있습니다.

— 각 상태 오브젝트는 자체적으로 `TransitionTo` 메서드 호출 조건을 결정하여 현재의 액티브 상태를 변경합니다. StateMachine 인스턴스를 설정할 때 필요한 종속 관계(상태 머신 자체 포함)를 각 상태로 전달할 수 있습니다.

  예시 프로젝트에서는 PlayerController가 이미 StateMachine에 대한 레퍼런스를 포함하므로 하나의 `player` 파라미터만 전달합니다.

각 상태 오브젝트가 자체 내부 로직을 관리하며, 게임 오브젝트나 컴포넌트를 설명하기 위해 필요한 상태를 필요한 수만큼 만들 수 있습니다. 각 오브젝트는 IState를 구현하는 자체 클래스를 취합니다. SOLID 원칙에 따라 상태를 더 추가해도 이전에 생성된 상태에 대한 영향은 최소로 유지됩니다.

다음은 `IdleState`의 예시입니다.

```
public class IdleState : IState
{
    private PlayerController player;

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

    public void Enter()
    {
        // 상태에 처음 진입할 때 실행되는 코드
    }

    public void Update()
    {
        // 조건이 다른 상태로 전환하기 위해
        // 존재하는지 여부를 감지하기 위한 로직 추가
        …
    }

    public void Exit()
    {
        // 상태에서 벗어날 때 실행되는 코드
    }
}
```

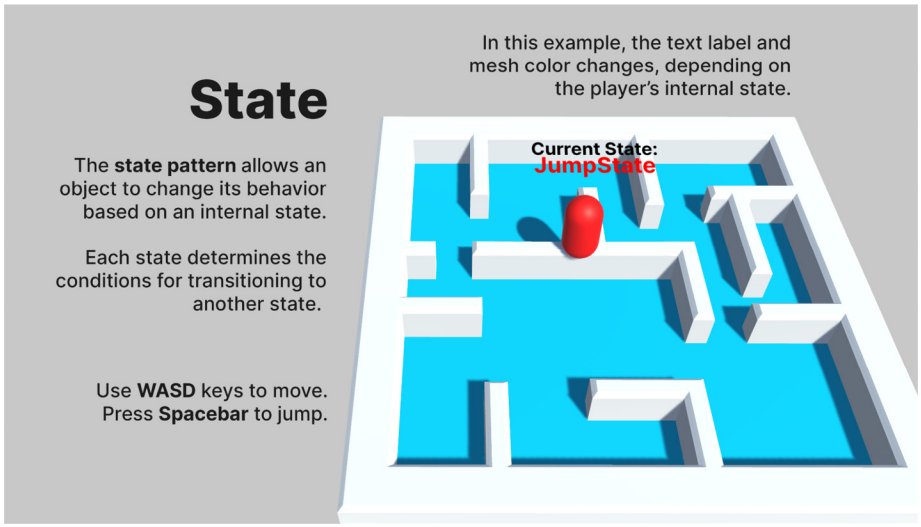
이번에도 생성자를 사용해 PlayerController 오브젝트를 전달합니다. 예시에서 이 `player`에는 StateMachine에 대한 레퍼런스를 비롯해 Update 로직에 필요한 모든 요소가 있습니다. `idleState`는 캐릭터 컨트롤러의 속도 또는 점프 상태를 모니터링하며, 상황에 맞게 StateMachine의 `TransitionTo` 메서드를 호출합니다.

WalkState 및 JumpState 구현에 대한 샘플 프로젝트도 검토하세요. 동작을 전환하는 커다란 클래스 하나를 가지는 대신, 각 상태는 자체 업데이트 로직을 가집니다. 그러면 상태가 서로 독립적으로 작동할 수 있습니다.

장점과 단점

상태 패턴은 오브젝트에 대한 내부 로직을 설정할 때 SOLID 원칙을 준수하는 데 도움이 됩니다. 각 상태는 상대적으로 크기가 작으며 다른 상태로 전환하기 위한 조건만 추적합니다. 개방-폐쇄 원칙에 따라 기존 상태에 영향을 주지 않고 상태를 더 많이 추가할 수 있으며, 번거로운 switch 또는 if 문을 사용하지 않아도 됩니다.

반면, 추적해야 하는 상태가 많지 않다면 추가 구조는 과한 사양일 수 있습니다. 이 패턴은 상태의 복잡도가 특정 수준까지 올라갈 것으로 예상하는 경우에만 유용할 수 있습니다.



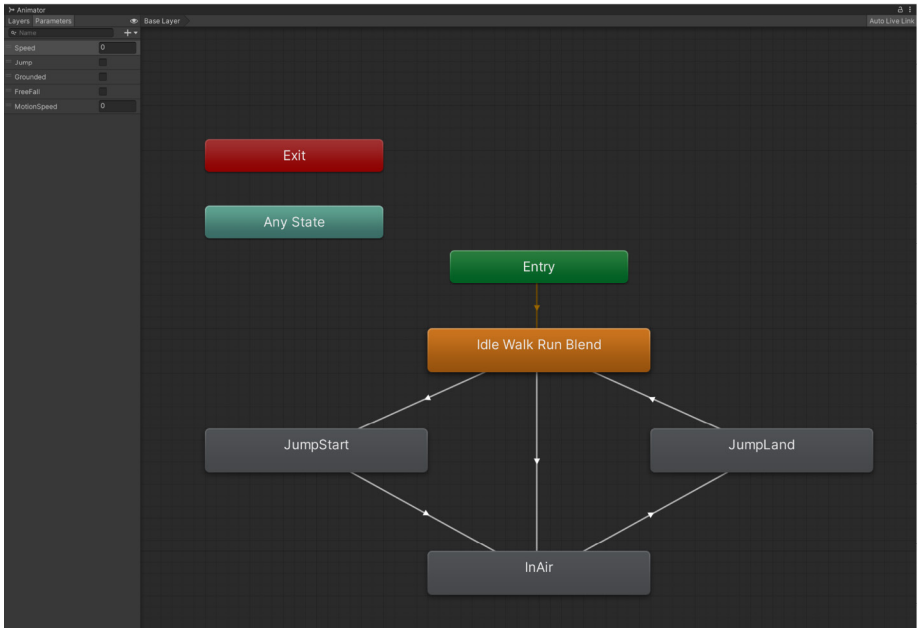
상태 패턴 샘플

개선 방안

샘플 프로젝트 내의 캡슐은 컬러가 변경되며, UI는 플레이어의 내부 상태에 따라 업데이트됩니다. 실제 예시에서는 상태 변경에 따라 훨씬 더 복잡한 효과를 적용할 수 있습니다.

— **상태 패턴과 애니메이션을 결합:** 상태 패턴의 일반적인 적용 분야로 애니메이션이 있습니다. 플레이어나 적 캐릭터는 거시 수준에서 프리미티브(캡슐)로 표현되는 경우가 많습니다. 그러면 내부 상태 변경에 반응하는 애니메이션화된 지오메트리를 사용할 수 있으므로 게임 액터가 달리기, 점프, 수영, 등반 등을 하는 모습을 보일 수 있습니다.

Unity의 Animator 창을 사용해 봤다면 그 워크플로가 상태 패턴과 잘 맞는다는 사실을 알 수 있습니다. 각 애니메이션 클립은 하나의 상태를 점유하며, 한 번에 하나만 액티브 상태가 됩니다.



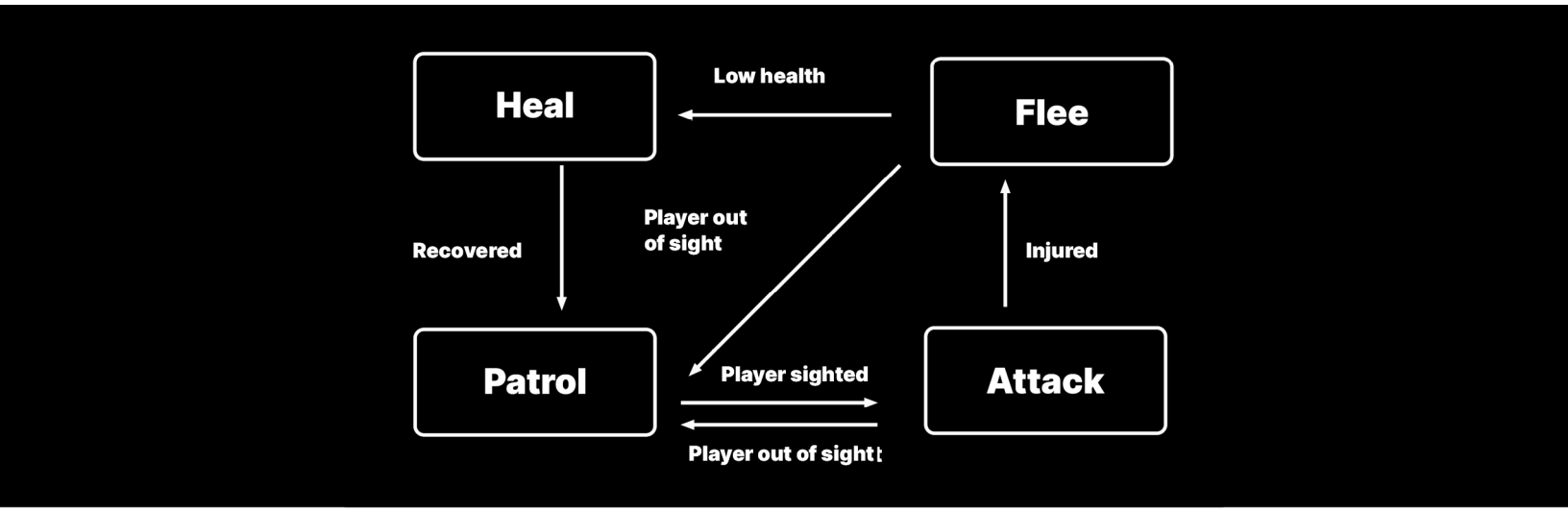
애니메이터 상태 그래프 예시: 구조를 StateMachine과 비교해 보세요.

— **이벤트 추가:** 상태 변경 사항을 외부 오브젝트에 전달하려면 이벤트를 추가하세요 (관찰자 패턴 참조). 상태를 시작하거나 종료하는 이벤트가 있으면 관련된 리스너에 알리고 런타임에 응답하도록 할 수 있습니다.

— **계층 구조 추가:** 상태 패턴을 사용하는 더 복잡한 엔티티를 제시하게 되면, 그때부터는 계층적 상태 머신을 구현해 보세요. 특정 상태는 필연적으로 서로 유사해질 것입니다. 예를 들어 플레이어나 게임 액터는 지면과 닿아 있는 경우에 WalkingState 상태에 있든 RunningState 상태에 있든 피하거나 점프할 수 있습니다.

SuperState를 구현하면 일반적인 동작을 한꺼번에 유지할 수 있습니다. 그러면 상속을 사용하여 하위 상태의 특정 항목을 무엇이든 오버라이드할 수 있습니다. 예를 들면 GroundedState를 먼저 선언할 수 있습니다. 그런 다음 RunningState 또는 WalkingState를 상속할 수 있습니다.

— **단순한 AI 구현:** 유한 상태 머신은 기본적인 적 AI 생성에도 유용할 수 있습니다. 다음은 NPC 브레인 제작에 대한 FSM 접근 방식의 예시입니다.



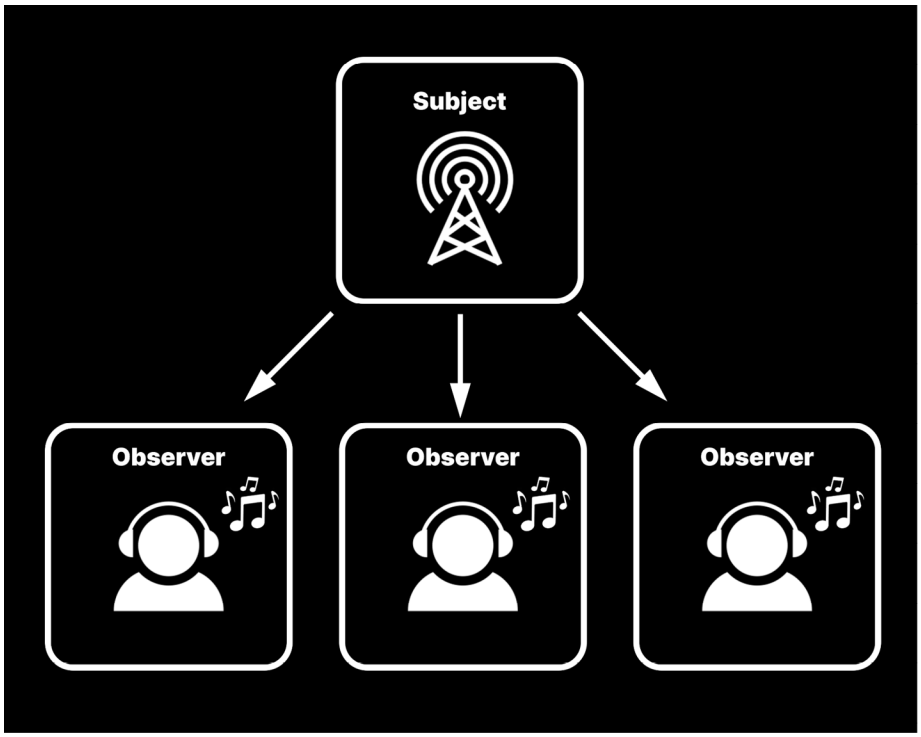
상태 패턴 기반의 단순한 AI

이번에는 완전히 다른 컨텍스트에서 작동하는 상태 패턴입니다. 각 상태는 공격, 도주, 또는 순찰 등의 행동을 나타냅니다. 한 번에 하나만 액티브 상태가 되며, 각 상태는 다음 상태로의 전환을 결정합니다.

관찰자 패턴

게임에서는 런타임에 여러 다양한 상황이 발생할 수 있습니다. 적을 파괴하면 어떻게 될까요? 파워업을 수집하거나 오브젝트를 완성하면 어떻게 될까요? 특정 오브젝트가 직접적인 참조 없이 다른 오브젝트에 알림을 줄 수 있게 하는 메커니즘이 필요한 경우가 많으며, 이에 따라 불필요한 종속 관계가 발생할 수 있습니다.

관찰자 패턴은 일반적으로 이러한 유형의 문제를 해결할 때 사용합니다. '일대다' 종속 관계를 사용해 오브젝트가 통신하되 낮은 결합도를 유지하도록 할 수 있습니다. 한 오브젝트의 상태가 변경되면 종속된 모든 오브젝트가 자동으로 알림을 받습니다. 많은 수의 다양한 청취자에게 방송 신호를 보내는 라디오 송신탑과 유사합니다.



관찰자 패턴은 라디오 송신탑처럼 동작합니다. 주체(subject)가 관찰자(observer)를 대상으로 방송 신호를 보낸다고 생각해 보세요.

방송하는 오브젝트는 주체고, 수신하는 다른 오브젝트는 관찰자입니다.

이 패턴은 주체의 결합도를 낮춥니다. 주체는 관찰자를 알지 못하거나, 신호를 수신한 관찰자가 무엇을 하든 관여하지 않습니다. 관찰자는 주체에 대해 종속 관계를 갖지만, 서로에 대해서는 알지 못합니다.

이벤트

관찰자 패턴은 C# 언어에 빌트인되어 있을 정도로 매우 보편적으로 사용됩니다. 자체적으로 주체-관찰자 클래스를 디자인할 수 있으나, 대체로 그럴 필요는 없습니다. 불필요한 시간 낭비에 대해 언급한 것을 기억하시나요? C#은 이벤트를 사용하여 관찰자 패턴을 실행합니다.

이벤트는 어떤 일이 일어났음을 알리는 단순한 알림이며, 다음과 같은 몇 가지 부분으로 구성됩니다.

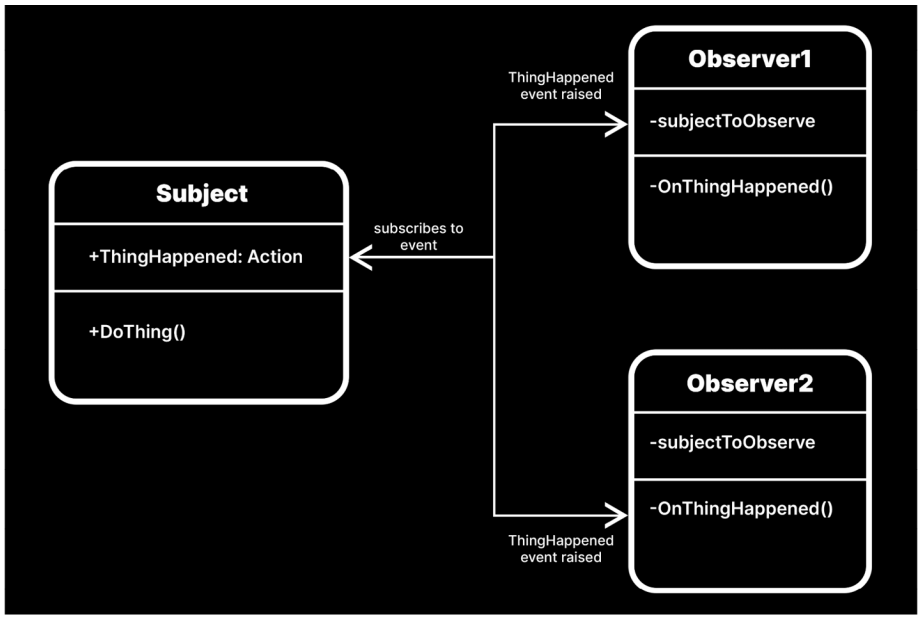
— 퍼블리셔(주체)는 델리게이트 기반의 이벤트를 생성하고, 특정한 함수 서명을 확보합니다. 이벤트는 피격이나 버튼 클릭 등 주체가 런타임에 수행하게 될 행동일 뿐입니다.

— 각 구독자(관찰자)는 이벤트 핸들러라는 메서드를 만들며, 이는 델리게이트의 서명과 일치해야 합니다.

— 각 관찰자의 이벤트 핸들러는 퍼블리셔의 이벤트를 구독합니다. 필요한 수만큼의 관찰자가 구독에 참여하도록 할 수 있습니다. 그러한 관찰자는 모두 이벤트가 트리거되기를 기다립니다.

— 퍼블리셔가 런타임에 이벤트 발생 신호를 보내면, 퍼블리셔가 이벤트를 발생시켰다고 할 수 있습니다. 그러면 차례로 구독자의 이벤트 핸들러가 호출되고, 이에 대한 응답으로 핸들러는 자체 내부 로직을 실행합니다.

이러한 방식으로 주체에서 발생하는 한 이벤트에 많은 컴포넌트가 반응하도록 구현할 수 있습니다. 주체가 버튼 클릭이 이루어졌음을 명시하면 관찰자는 애니메이션 또는 소리를 재생하거나, 컷씬을 트리거하거나, 파일을 저장할 수 있습니다. 어떤 응답이든 가능하므로, 관찰자 패턴은 오브젝트 간에 메시지를 전송하는 데 자주 사용됩니다.



주체는 이벤트를 발생시켜 관찰자에게 알립니다.

예시: 단순한 주체 및 관찰자

기본적인 주체/퍼블리셔를 다음과 같이 정의할 수 있습니다.

```
using UnityEngine;
using System;

public class Subject: MonoBehaviour
{
    public event Action ThingHappened;

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

MonoBehaviour에서 상속하여 더 쉽게 게임 오브젝트에 연결할 수도 있지만, 꼭 그렇게 해야 하는 것은 아닙니다.

자유롭게 자체 커스텀 델리게이트를 정의해도 되지만, System.Action은 대부분의 경우에 작동합니다. 이벤트와 함께 파라미터를 전송해야 하는 경우에는 Action<T> 델리게이트를 사용해 꺾쇠괄호 안에 List<T>로 전달합니다(최대 16개 파라미터).

ThingHappened는 실제 이벤트이며, 주체는 DoThing 메서드에서 호출합니다.

이벤트를 수신 대기할 수 있도록 예시 Observer 클래스를 만들 수 있습니다. 여기서는 편의를 위해 MonoBehaviour를 상속하지만 반드시 그렇게 할 필요는 없습니다.

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

    private void OnThingHappened()
    {
        // 이벤트에 응답하는 모든 로직은 여기로 이동
        Debug.Log("Observer responds");
    }

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

    private void OnDestroy()
    {
        if (subjectToObserve != null)
        {
            subjectToObserve.ThingHappened -= OnThingHappened;
        }
    }
}
```

이 컴포넌트를 게임 오브젝트에 연결하고 인스펙터 순서에 있는 `subjectToObserver`를 참조하여 `ThingHappened` 이벤트를 수신 대기합니다.

`OnThingHappened` 메서드는 관찰자가 이벤트에 대응하여 실행하는 모든 로직을 포함할 수 있습니다. 개발자는 접두사 'On'을 추가하여 이벤트 핸들러를 나타내는 경우가 많습니다 (스타일 가이드의 명명 규칙 사용).

Awake 또는 Start에서 += 연산자를 사용하여 이벤트를 구독할 수 있습니다. 이는 관찰자의 `OnThingHappened` 메서드와 주체의 `ThingHappened`를 결합합니다.

주체의 `DoThing` 메서드가 실행되면 이벤트가 발생합니다. 그러면 관찰자의 `OnThingHappened` 이벤트 핸들러가 자동으로 호출하고 디버그 문을 출력합니다.

참고: `ThingHappened`를 계속 구독 중인 관찰자를 런타임에 삭제하거나 제거하면 해당 이벤트 호출 시 오류가 발생할 수 있습니다. 따라서 -= 연산자를 사용해 MonoBehaviour의 `OnDestroy` 메서드에서 이벤트 구독을 취소해야 합니다.

게임플레이 과정에서 발생하는 거의 모든 상황에 관찰자 패턴을 적용할 수 있습니다. 플레이어가 적을 파괴하거나 아이템을 수집할 때마다 이벤트가 발생할 수 있는 게임을 예로 들겠습니다. 점수나 업적을 추적하는 통계 시스템이 필요한 경우, 관찰자 패턴을 사용하면 원본 게임플레이 코드에 영향을 주지 않고 해당 시스템을 만들 수 있습니다.

많은 Unity 애플리케이션에서 이벤트를 적용하는 부분은 주로 다음과 같습니다.

— 목적 또는 목표

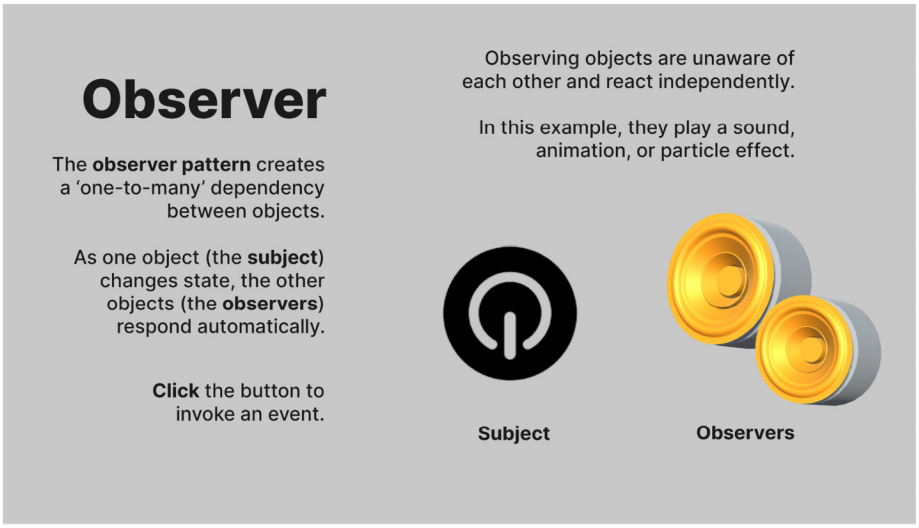
— 승리/패배 조건

— 플레이어 사망, 적 사망, 대미지

— 아이템 획득

— 사용자 인터페이스

주체가 적절한 시점에 이벤트만 발생시키면 됩니다. 그러면 많은 관찰자가 구독할 수 있습니다.



관찰자 샘플 씬

샘플 프로젝트에서 ButtonSubject는 사용자가 마우스 버튼으로 Clicked 이벤트를 호출하도록 합니다. 그러면 AudioObserver 및 ParticleSystemObserver 컴포넌트가 있는 여러 다른 게임 오브젝트가 고유의 방식으로 이벤트에 응답할 수 있습니다.

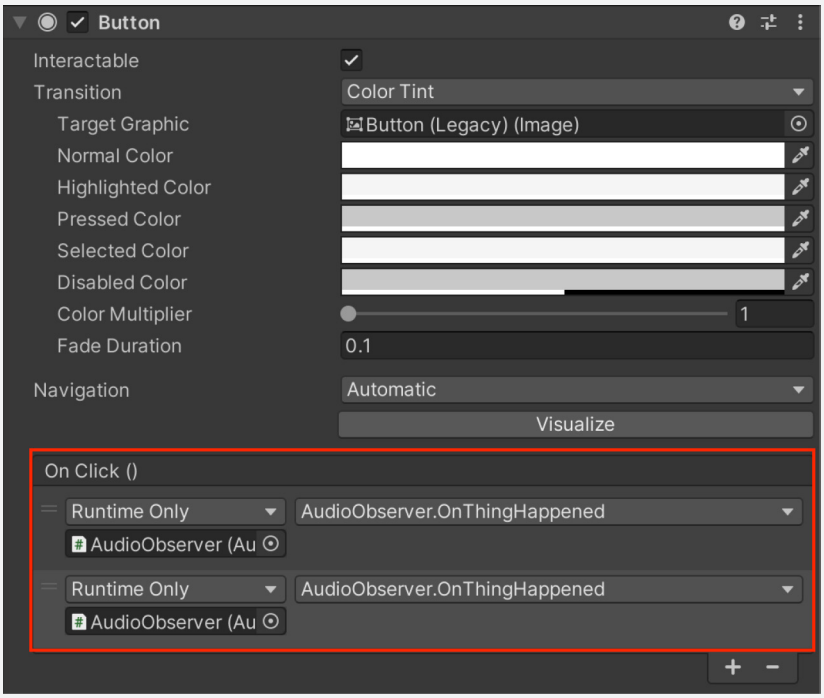
어떤 오브젝트가 '주체' 또는 '관찰자'인지는 오직 용도에 따라 결정됩니다. 무엇이든 이벤트를 발생시키면 주체로 동작하고, 이벤트에 반응하면 관찰자로 동작합니다. 동일한 게임 오브젝트의 다양한 컴포넌트가 주체 또는 관찰자가 될 수 있습니다. 같은 컴포넌트가 한 컨텍스트에서는 주체가 되고 다른 컨텍스트에서는 관찰자가 되기도 합니다.

이를테면 예시의 AnimObserver는 클릭했을 때 버튼에 약간의 이동을 더합니다. ButtonSubject 게임 오브젝트의 일부임에도 관찰자 역할을 수행합니다.

UnityEvent 및 UnityAction

Unity에는 UnityEvent 시스템도 별도로 포함되며, 이 시스템은 **UnityEngine.Events** API에서 UnityAction 델리게이트를 사용합니다.

UnityEvent는 관찰자 패턴을 위한 그래픽 인터페이스를 제공합니다. Unity의 UI 시스템을 사용해 봤다면(예: UI Button의 OnClick 이벤트 생성) 이미 어느 정도 경험이 있는 것입니다.



UnityEvent에는 설정을 위한 그래픽 컴포넌트가 있습니다.

이 예시에서 버튼의 OnClick 이벤트는 AudioObserver의 OnThingHappened 메서드 2개로부터의 응답을 호출하고 트리거합니다. 따라서 코드를 작성하지 않고도 주체의 이벤트를 설정할 수 있습니다.

UnityEvent는 디자이너 또는 프로그래머가 아닌 인원이 게임플레이 이벤트를 생성할 수 있도록 하려는 경우에 유용합니다. 하지만 시스템 네임스페이스의 동급 이벤트나 행동보다 느릴 수 있다는 점에 유의하세요.

UnityEvent 및 UnityAction을 고려할 때는 성능과 사용량을 비교해 판단해야 합니다. Unity Learn의 Create a Simple Messaging System with Events 모듈에서 소개하는 예시를 확인하세요.

장점과 단점

이벤트를 구현하려면 추가 작업이 필요하지만 다음과 같은 장점이 있습니다.

— **관찰자 패턴은 오브젝트의 분리에 도움이 됩니다.** 이벤트 퍼블리셔는 이벤트 구독자에 대해 아무것도 알 필요가 없습니다. 한 클래스와 다른 클래스 사이에 직접적인 종속 관계를 만드는 대신, 주체와 관찰자는 어느 정도 분리된 상태를 유지하며 서로 통신합니다.

— **직접 제작할 필요가 없습니다.** C#에는 이벤트 시스템이 이미 확립되어 있고, 자체 델리게이트를 정의하는 대신 System.Action 델리게이트를 사용할 수 있습니다. 대안으로 Unity의 UnityEvent 및 UnityAction을 사용할 수도 있습니다.

— **각 관찰자는 이벤트 처리 로직을 자체적으로 구현합니다.** 따라서 관찰하는 각 오브젝트는 응답하는 데 필요한 로직을 유지합니다. 따라서 디버그와 유닛 테스트가 더 쉬워집니다.

— **사용자 인터페이스에 매우 적합합니다.** 코어 게임플레이 코드를 UI 로직과 별도로 둘 수 있습니다. 그러면 UI 요소가 특정한 게임 이벤트나 조건을 수신 대기하며 적절히 응답할 수 있습니다. MVP 및 MVC 패턴은 이러한 목적으로 관찰자 패턴을 사용합니다.

관찰자 패턴에서 유의할 부분은 다음과 같습니다.

— **복잡도가 증가합니다.** 다른 패턴과 마찬가지로 이벤트 기반 아키텍처를 생성하려면 미리 더 많은 설정을 해야 합니다. 주체나 관찰자를 삭제할 때에도 주의하세요. OnDestroy에서 관찰자의 등록을 반드시 해제해야 합니다.

— **관찰자에게 이벤트를 정의하는 클래스에 대한 레퍼런스가 필요합니다.** 관찰자는 이벤트를 퍼블리시하는 클래스에 대해 여전히 종속 관계를 가집니다. 모든 이벤트를 처리하는 정적 EventManager(아래)를 사용하면 서로 얽힌 오브젝트를 더 쉽게 정리할 수 있습니다.

— **성능이 문제가 될 수 있습니다.** 이벤트 기반 아키텍처는 더 많은 오버헤드를 유발합니다. 대형 화면과 다수의 게임 오브젝트는 성능 저하를 유발할 수 있습니다.

개선 방안

이 책에서는 관찰자 패턴의 기본 버전만을 다루지만, 모든 게임 애플리케이션에서 필요한 부분을 처리할 수 있도록 패턴을 확장할 수 있습니다.

관찰자 패턴을 설정할 때는 다음과 같은 권장 사항을 고려하세요.

— **ObservableCollection 클래스를 사용합니다.** C#에는 특정 변경 사항을 추적하기 위한 동적 ObservableCollection이 있습니다. 이 클래스는 항목이 추가, 제거되거나 목록이 새로 고침되면 관찰자에게 알릴 수 있습니다.

— **고유 인스턴스 ID를 인수로 전달합니다.** 계층 구조의 각 게임 오브젝트에는 고유의 인스턴스 ID가 있습니다. 둘 이상의 관찰자에게 적용할 수 있는 이벤트를 트리거하는 경우, 고유 ID를 이벤트로 전달하세요(Action<int> 유형 사용). 그런 다음 게임 오브젝트가 고유 ID와 일치하면 이벤트 핸들러의 로직만 실행합니다.

— **정적 EventManager를 만듭니다.** 게임플레이의 많은 부분을 이벤트로 구동할 수 있으므로, 많은 Unity 애플리케이션에 정적 또는 싱글톤 EventManager가 사용됩니다. 관찰자는 이러한 방식을 통해 게임 이벤트의 주요 소스를 주체로 참조하여 더 쉽게 설정할 수 있습니다.

  FPS 마이크로게임에는 커스텀 GameEvent를 실행하고 리스너를 추가하거나 제거하는 정적 헬퍼 메서드를 포함하는 정적 EventManager가 잘 구현되어 있습니다.

  또한 Unity Open Projects에서는 UnityEvent 릴레이를 위해 ScriptableObject를 사용하는 게임 아키텍처를 보여 줍니다. 여기에서는 이벤트를 사용하여 오디오를 재생하거나 새 씬을 로드합니다.

— **이벤트 대기열을 생성합니다.** 씬에 오브젝트가 많은 경우에는 이벤트를 한꺼번에 발생시키지 않는 편이 좋습니다. 이벤트 하나를 호출할 때 천 개의 오브젝트가 재생되며 불협화음을 내는 상황을 떠올려 보세요.

  관찰자 패턴과 커맨드 패턴을 조합해 이벤트를 이벤트 대기열로 캡슐화할 수 있습니다. 그런 다음 커맨드 버퍼를 사용해 이벤트를 한 번에 하나씩 재생할 수 있으며, 필요한 경우 선택적으로 무시할 수도 있습니다(예: 동시에 재생 가능한 오브젝트의 수가 최대치에 도달한 경우).

관찰자 패턴은 MVP(모델 뷰 프리젠터) 아키텍처 패턴에 크게 영향을 미치며, 관련 내용은 다음 장에서 자세히 설명합니다.

MVP(모델 뷰 프리젠터)

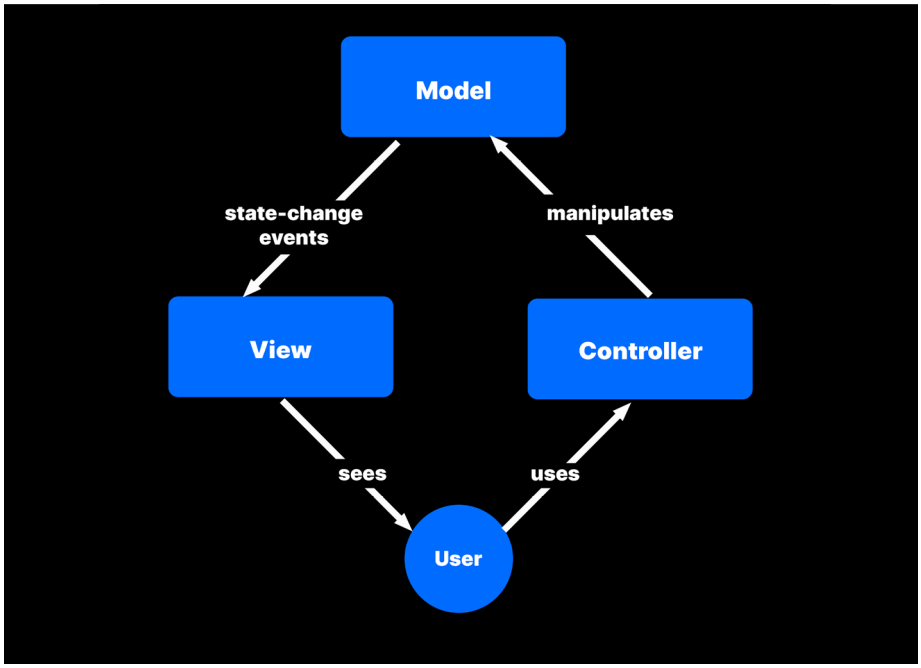
MVC(모델 뷰 컨트롤러)는 사용자 인터페이스를 개발할 때 흔히 사용되는 일련의 디자인 패턴입니다.

MVC의 기본 아이디어는 소프트웨어의 논리적 부분을 데이터 및 프레젠테이션에서 분리하는 것입니다. 그러면 불필요한 종속 관계를 줄이는 데 도움이 되며, 잠재적으로 스파게티 코드도 줄일 수 있습니다.

MVC(모델 뷰 컨트롤러) 디자인 패턴

이름에서 알 수 있듯이 MVC 패턴은 애플리케이션을 다음 세 개의 레이어로 분할합니다.

- **모델은 데이터를 저장합니다.** 모델은 전적으로 값을 저장하는 데이터 컨테이너입니다. 게임플레이 로직이나 계산을 수행하지 않습니다.
- **뷰는 인터페이스입니다.** 뷰는 데이터의 그래픽 표현을 형식화하고 화면에 렌더링합니다.
- **컨트롤러는 로직을 처리합니다.** 두뇌라고 보면 됩니다. 컨트롤러는 게임 데이터를 처리하며 값이 런타임에 어떻게 변경되는지 산출합니다.



모델, 뷰, 컨트롤러

이렇게 분리하면 세 부분이 서로 상호 작용하는 방식에 대한 정의도 이루어집니다. 모델은 애플리케이션 데이터를 관리하며, 뷰는 그러한 데이터를 사용자에게 표시합니다. 컨트롤러는 입력을 처리하고 게임 데이터에 대한 모든 결정이나 계산을 수행합니다. 그런 다음 결과를 모델로 되돌려 보냅니다.

따라서 컨트롤러 자체에는 어떤 게임 데이터도 포함되지 않으며, 뷰에도 포함되지 않습니다. MVC 디자인은 각 레이어가 수행하는 작업을 제한합니다. 한 부분은 데이터를 저장하고, 다른 부분은 데이터를 처리하며, 나머지 부분은 해당 데이터를 사용자에게 표시합니다.

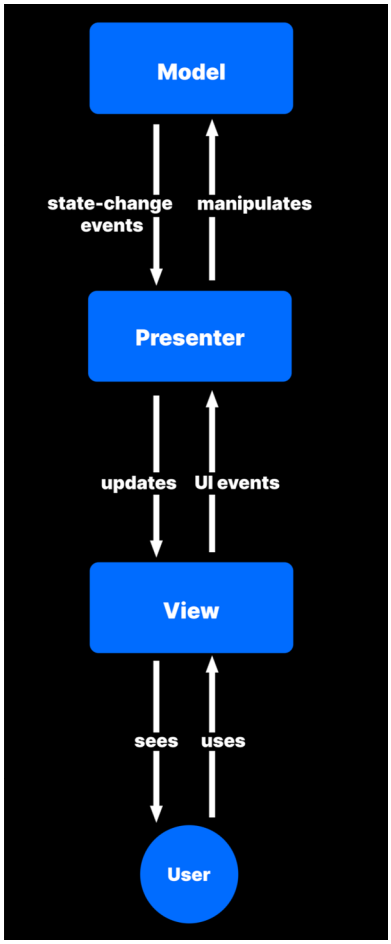
표면적으로는 단일 책임 원칙의 연장선상에 있다고 간주할 수도 있습니다. 각 부분이 하나의 작업을 제대로 수행하며, 이는 MVC 아키텍처가 가진 장점 중 하나입니다.

MVP(모델 뷰 프리젠터) 및 Unity

MVC를 사용해 Unity 프로젝트를 개발할 때, 기존 UI 프레임워크(UI Toolkit 또는 Unity UI)는 자연스럽게 뷰의 기능을 수행합니다. 엔진에서 완전히 구현된 사용자 인터페이스를 제공하므로, 개별적인 UI 컴포넌트를 처음부터 개발할 필요가 없습니다.

하지만 전통적인 MVC 패턴을 따르려면 런타임에 모델 데이터에서 발생하는 변경 사항을 수신 대기하기 위한 뷰별 코드가 필요합니다.

이 방식이 유효하기는 하나, 많은 Unity 개발자는 주로 컨트롤러가 중개자 역할을 하는 MVC의 배리에이션을 사용하는 편입니다. 여기서는 뷰가 직접 모델을 관찰하지 않고, 대신 다음과 같은 작업을 수행합니다.



MVP: MVC의 배리에이션

MVC의 이러한 배리에이션을 MVP(모델 뷰 프리젠터) 디자인이라 합니다. MVP에서도 서로 구분되는 세 가지의 애플리케이션 레이어를 분리합니다. 하지만 각 부분이 맡은 책임은 조금씩 다릅니다.

MVP에서는 프리젠터(MVC에서 컨트롤러)가 다른 레이어의 중개자 역할을 합니다. 모델로부터 데이터를 검색한 다음 뷰에 표시할 수 있도록 형식을 지정합니다. MVP는 입력을 처리할 레이어를 전환합니다. 컨트롤러가 아니라 뷰에서 사용자 입력을 처리합니다.

이벤트와 관찰자 패턴이 이 디자인에 어떻게 적용되는지 참고하세요. 사용자는 Unity UI의 Button, Toggle, Slider 컴포넌트와 상호작용할 수 있습니다. 뷰 레이어는 UI 이벤트를 통해 프리젠터에 이 입력을 되돌려 보내고, 프리젠터는 모델을 조작합니다. 모델의 상태 변경 이벤트는 데이터가 업데이트되었음을 프리젠터에 알립니다. 프리젠터가 수정된 데이터를 뷰로 전달하면 UI가 새로 고침됩니다.

예시: 상태 인터페이스

MVP 예시를 정형화하려면 캐릭터나 아이템의 체력을 표시하는 간단한 시스템을 떠올려 보세요. 데이터와 UI가 혼합된 클래스 하나에 모든 요소를 채울 수도 있겠지만 그렇게 하면 확장성이 떨어질 수 있습니다. 기능을 더 추가하면 확장을 해야 하는 만큼 복잡도가 증가할 것입니다.

대신, Health 컴포넌트를 보다 MVP 중심적인 방식으로 다시 작성할 수 있습니다. 스크립트를 Health 및 HealthPresenter로 나눕니다. Health 컴포넌트의 예시는 다음과 같습니다.

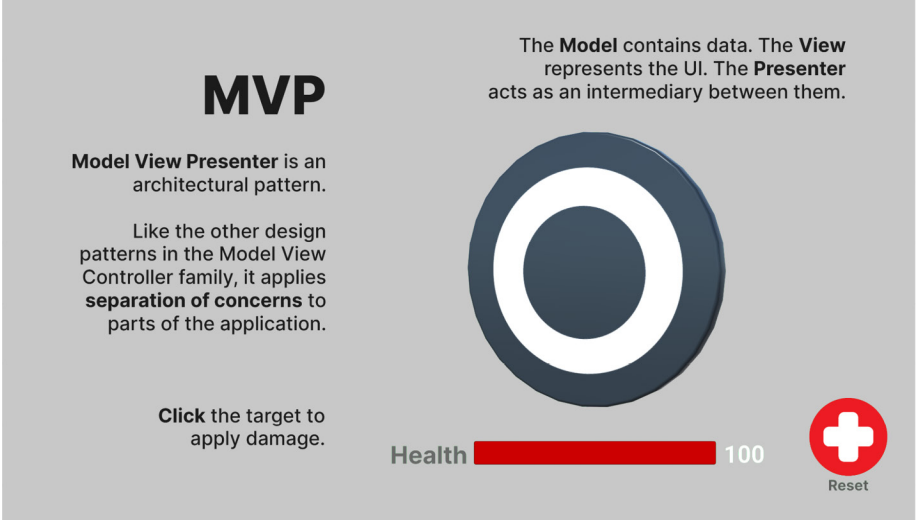
```
public class Health: MonoBehaviour
{
    public event Action HealthChanged;

    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();
    }
}
```

이 버전에서 Health는 모델 역할을 합니다. 값이 변경될 때마다 실제 체력 값을 저장하고 HealthChanged 이벤트를 호출합니다. Health에는 게임플레이 로직이 포함되지 않으며, 데이터를 늘리거나 줄이는 메서드만 있습니다.

하지만 대부분의 오브젝트는 Health 자체를 조작하지 않습니다. 해당 작업은 HealthPresenter를 예약해 처리합니다.

```
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();
    }
}
```

다른 게임 오브젝트는 Damage, Heal, Reset을 사용해 체력 값을 수정하려면 HealthPresenter를 사용해야 합니다. HealthPresenter는 보통 Health가 HealthChanged 이벤트를 발생시킬 때까지 UpdateView로 사용자 인터페이스를 업데이트하기 위해 대기합니다. 이 기능은 모델의 값을 설정하는 데 소요되는 기간이 짧은 경우(예: 값을 디스크에 저장하거나 데이터베이스에 저장)에 유용합니다.



MVP를 사용하는 샘플 상태 인터페이스

샘플 프로젝트에서 사용자는 클릭을 통해 타겟 오브젝트에 대미지를 입히거나 버튼으로 체력을 초기화할 수 있습니다. 이러한 동작은 Health를 직접 변경하지 않고, HealthPresenter (Damage 또는 Reset 호출)에 정보를 제공합니다. UI 텍스트와 UI 슬라이더는 Health가 이벤트를 발생시키고 HealthPresenter에 값이 변경되었음을 알리면 그때 업데이트됩니다.

장점과 단점

MVP 및 MVC는 큰 규모의 애플리케이션에서 장점을 보입니다. 게임 개발에 상당한 규모의 팀이 필요하며 출시 이후 게임을 장기간 유지해야 한다면 다음과 같은 특성이 유리할 수 있습니다.

— **원활한 업무 분배:** 뷰를 프리젠터에서 분리했으므로, 사용자 인터페이스 개발과 업데이트를 나머지 코드 베이스와 거의 독립적으로 수행할 수 있습니다.

  그러면 전문 개발자 사이에서 업무를 분할할 수 있습니다. 팀에 프론트엔드 개발 전문가가 있다면 뷰를 전담하도록 맡기세요. 다른 구성원들과 독립적으로 작업할 수 있습니다.

— **MVP 및 MVC로 간소화된 유닛 테스팅:** 이 디자인 패턴은 게임플레이 로직과 사용자 인터페이스를 분리합니다. 그렇기 때문에 에디터에서 플레이 모드를 사용하지 않아도 오브젝트를 코드와 연동해 시뮬레이션할 수 있으며, 이를 통해 상당한 시간을 단축할 수 있습니다.

— **유지 가능하며 가독성이 높은 코드:** 이 디자인 패턴을 활용하면 비교적 작은 클래스를 만들게 되므로 코드의 가독성이 높아집니다. 일반적으로 종속 관계가 적을수록 소프트웨어 오류가 발생할 수 있는 부분과 잠재적인 버그가 있는 부분이 줄어듭니다.

MVC와 MVP는 웹 개발이나 엔터프라이즈 소프트웨어에 널리 사용되고 있지만, 애플리케이션이 충분히 커지고 복잡해지기 전까지는 이점이 확실하게 드러나지 않는 경우가 많습니다. Unity 프로젝트에서 두 패턴 중 하나를 구현하려면 다음 사항부터 고려해야 합니다.

— **미리 계획해야 합니다.** 본 가이드에서 설명한 다른 패턴들과 달리 MVC와 MVP는 비교적 큰 규모의 아키텍처 패턴입니다. 둘 중 하나를 사용하려면 책임에 따라 클래스를 나눠야 하며, 이 과정에서 약간의 정리 및 사전 작업이 필요합니다.

— **Unity 프로젝트의 모든 요소가 패턴에 적합하지는 않습니다.** '순수한' MVC 또는 MVP 구현에서 화면에 렌더링되는 요소는 모두 실제 뷰의 한 부분입니다. 데이터, 로직, 인터페이스로 쉽게 분리되지 않는 Unity 컴포넌트도 있습니다(예: MeshRenderer). 간단한 스크립트에서는 MVC/MVP가 크게 도움이 되지 않기도 합니다.

  패턴의 이점을 가장 크게 활용할 수 있는 부분이 어디인지 판단할 수 있어야 합니다. 보통은 유닛 테스트 결과가 도움이 됩니다. MVC/MVP로 쉽게 테스트할 수 있다면 애플리케이션의 해당되는 부분에 활용해 볼 수 있습니다. 그렇지 않다면 프로젝트에 억제로 패턴을 적용하려 하지 마세요.

결론

소프트웨어 패턴이 낯선 주제였다면, Unity 개발에서 가장 흔하게 마주치는 여러 패턴을 이해하는 데 본 가이드가 도움이 되었기를 바랍니다.

프리팹 생성을 위한 팩토리 패턴, AI를 위한 상태 패턴 등 어떤 기법이든 필요할 때마다 유용하게 사용해 보세요. 디자인 패턴을 언제 어떻게 적용하는지 파악하면 Unity를 사용하면서 겪을 수 있는 문제를 해결하는 데 도움이 될 것입니다. 특정 패턴을 억지로 맞추느라 고생하지 마세요. 패턴은 사용하는 것만큼 사용하지 않는 것도 중요합니다.

디자인 패턴을 올바르게 적용하면 워크플로의 속도를 높이고 반복적인 문제에 대해 효과적인 해결책을 제시할 수 있습니다. 그러면 플레이어에게 즐겁고 독특한 경험을 선사하는 중요한 작업에 집중할 수 있습니다.

결과적으로, 불필요한 시간 낭비 없이 창의성을 더 확실하게 발휘할 수 있습니다.

기타 디자인 패턴

본 가이드의 내용은 컴퓨팅 및 게임 개발 분야에서 익히 알려진 몇 가지 디자인 패턴의 일부에 불과합니다. 구체적인 내용까지 다루지는 않겠지만, 도움이 될 만한 여러 다른 패턴에 대해 간략하게 소개합니다.

- **어댑터:** 서로 관련이 없는 두 엔티티가 연동할 수 있도록 둘 사이에 인터페이스(래퍼라고도 함)를 제공합니다.
- **플라이웨이트(Flyweight):** 오브젝트의 수가 많으면 기본 클래스에 공통 프로퍼티를 공유하고 리소스를 저장하세요. 예를 들어 숲을 디자인한다면 나무의 모든 보편적인 프로퍼티를 Tree라는 기본 클래스에 저장합니다. 그러면 서브 클래스(PineTree, MapleTree 등)에서 해당 프로퍼티를 반복할 필요가 없습니다.
- **이중 버퍼:** 이 패턴을 사용하면 계산이 완료되는 동안 배열 데이터 세트 두 개를 유지할 수 있습니다. 그러면 한 데이터 세트를 처리하는 동안 다른 데이터 세트를 표시할 수 있으며, 이는 절차적 시뮬레이션(예: 셀룰러 오토마타)이나 항목을 화면에 렌더링하는 데 유용합니다.
- **더티 플래그:** 이 기법을 사용하면 게임에서 변경 사항이 있지만 많은 비용이 드는 작업(예: 디스크에 저장 또는 물리 시뮬레이션 실행)이 아직 실행되지 않은 경우 부울을 설정할 수 있습니다.
- **인터프리터/바이트코드:** 모딩 지원을 추가하거나 프로그래머가 아닌 인원이 게임을 확장하도록 지원하려면, 사용자가 외부 텍스트 파일에서 편집할 수 있는 간소화된 언어를 만들면 됩니다. 그러면 바이트코드 컴포넌트가 해석된 언어를 C# 게임 코드로 변환할 수 있습니다.
- **서브 클래스 샌드박스:** 다양한 동작을 하는 유사한 여러 오브젝트가 있다면, 부모 클래스에서 해당 동작을 보호하도록 정의할 수 있습니다. 그러면 자식 클래스를 적절하게 믹스 앤 매치하여 새로운 조합을 구성할 수 있습니다.
- **유형 오브젝트:** 게임 오브젝트의 종류가 다양하다면 각 오브젝트의 서브클래스를 만드는 대신, 가능한 모든 동작을 하나의 추상 또는 부모 클래스에 정의합니다. 코드를 변경하지 않고도 커스터마이즈 가능한 별도 데이터 파일(ScriptableObject 등)로 개별 오브젝트 고유의 특성을 구분할 수 있습니다. 예를 들어 같은 클래스에서 파생되었으나 서로 다르게 보이는 아이템으로 채운 인벤토리를 만들 수 있습니다. 게임 디자이너는 프로그래머의 도움을 받지 않더라도 데이터 파일을 커스터마이즈하여 각 아이템(예: RPG의 무기)을 특별하게 만들 수 있습니다.
- **데이터 지역성:** 데이터를 최적화해 메모리에 효율적으로 저장되도록 만들면 성능이 향상하는 효과를 누릴 수 있습니다. 클래스를 구조체로 대체하면 데이터를 캐시하기가 더 쉬워집니다. Unity의 ECS 및 DOTS 아키텍처는 이 패턴을 구현합니다.

— **공간 파티셔닝:** 대규모 씬과 게임 월드에서는 특수 구조를 사용하여 게임 오브젝트를 위치별로 구성하세요. Grid, Trie(Quadtree, Octree), Binary search tree 모두 더 효율적으로 분할하고 검색하는 데 도움이 되는 기법입니다.

— **데코레이터:** 기존 구조를 변경하지 않고도 오브젝트에 책임을 부여할 수 있습니다. 데코레이터는 기본 무기 클래스를 변경하지 않고도 무기에 특전을 추가하는 등 특수 능력을 부여하거나 게임 오브젝트를 수정할 수 있습니다.

— **파사드:** 더 복잡한 시스템에 간단한 통합 인터페이스를 제공합니다. 별도의 AI, 애니메이션, 사운드 컴포넌트를 가진 게임 오브젝트가 있는 경우, 해당 컴포넌트에 맞춰 래퍼 클래스를 추가할 수 있습니다(예: PlayerInput, PlayerAudio 등을 관리하는 플레이어 컨트롤러 클래스). 이 파사드는 원본 컴포넌트의 세부 내용을 숨기고 사용을 단순화합니다.

— **템플릿 메서드:** 이 패턴은 알고리즘의 정확한 단계를 서브 클래스로 이전합니다. 예를 들어 추상 클래스에서 알고리즘이나 데이터 구조의 개략적인 골격을 정의하되, 알고리즘 전체 구조의 변경 없이 서브 클래스에서 특정 부분을 오버라이드하도록 할 수 있습니다.

— **전략:** 이 동작 패턴(또는 정책 패턴)은 알고리즘 제품군을 정의하고 각 알고리즘을 클래스 안에 캡슐화하는 데 도움이 됩니다. 이렇게 하면 각 알고리즘(전략)을 런타임에 서로 교환할 수 있습니다. 예를 들어 경로 탐색 시스템을 만들었다면, 전략 패턴을 사용하여 컨텍스트에 따라 게임플레이 중에 서로 교체할 수 있는 여러 알고리즘(A*, Dijkstra의 최단 경로 등)을 정의할 수 있습니다.

— **복합:** 이 구조적 디자인 패턴을 사용하면 오브젝트를 트리 구조로 구성한 다음, 결과로 도출되는 구조를 개별 오브젝트처럼 처리할 수 있습니다. 단순 컴포넌트와 복합 컴포넌트(잎과 컨테이너)로 트리를 구성합니다. 모든 요소가 동일한 인터페이스를 구현하므로 전체 트리에서 동일한 동작을 재귀적으로 실행할 수 있습니다.

Unity 크리에이터를 위한 프로페셔널 교육

Unity 프로페셔널 교육은 Unity에서 더 생산적으로 작업하고 효율적으로 협업할 수 있는 기술과 지식을 제공합니다. 모든 업계 전문가가 기술 수준에 상관없이 참여할 수 있도록 광범위한 교육 카탈로그를 다양한 형태로 제공합니다.

풍부한 경험을 가진 유니티의 교육 콘텐츠 개발자가 유니티 엔지니어링 및 제품 팀과 함께 모든 자료를 제작했습니다. 따라서 항상 최신 Unity 기술에 대한 최신 교육을 받을 수 있습니다.

Unity 프로페셔널 교육으로 팀이 어떻게 발전할 수 있는지 자세히 알아보세요.

unity.com/kr