# 어댑터 패턴

어댑터 패턴은 클래스의 소스를 변경하지 않고, 어댑터를 제작하고 추가함으로써 기존 기능을 새로운 기능에 매핑하는 패턴이다.

아래와 같이 마린 클래스가 있다고 하자.

class Marine

{

public Marine()

{

Speed = 1;

}

public int Speed { get; private set; }

public void RunSteamPack()

{

Speed = 2;

}

}

이동 속도를 의미하는 Speed 속성과 속도를 증가시키는 RunSteamPack 메서드가 있다. 그리고 질럿 클래스는 아래와 같이 구현되어 있다고 하자.

class Zealot

{

public Zealot()

{

Speed = 1;

}

public int Speed { get; set; }

public void Charge()

{

Speed = 3;

}

}

마린과 마찬가지로 이동 속도를 의미하는 Speed 속성이 있으며, 역시 속도를 증가시키는 메서드인 Charge가 정의되어 있다. 그렇다면 마린과 질럿을 하나씩 생성하고 각각 이동 속도를 증가시키는 코드는 이렇게 쓸 수 있다.

static void Main(string[] args)

{

Marine marine = new Marine();

MakeItFast(marine);

Console.WriteLine(marine.Speed);

Zealot zealot = new Zealot();

MakeItFast(zealot);

Console.WriteLine(zealot.Speed);

}

static void MakeItFast(object unit)

{

if (unit is Marine)

((Marine)unit).RunSteamPack();

else if (unit is Zealot)

((Zealot)unit).Charge();

}

마린과 질럿의 속도를 증가시키는 메서드 MakeItFast를 바깥으로 뺐는데, 속도를 올리기 위해 마린과 질럿이 각자 호출해야 하는 메서드가 다르기 때문에, 1)매개변수를 object로 받아야 하고, 2)is 연산자를 사용하여 각 타입을 검사한 후 캐스팅을 하는 코드가 포함되었다.

만일 마린과 질럿이 공통된 부모 클래스, 예를 들어 Unit이 있고 거기에 속도를 증가시키는 메서드가 있으면 아래와 같이 쓸 수 있을 것이다.

static void MakeItFast(Unit unit)

{

unit.SpeedUp();

}

하지만 Unit에는 SpeedUp 같은 메서드가 없으며, 마린과 질럿의 소스를 우리가 고칠 수 없는 상황이라면, 위와 같은 코드는 작성할 수가 없다.

바로 이런 상황에서 Adapter 패턴을 유용하게 사용할 수 있다. 즉 MakeItFast 메서드가 매개변수로 마린이나 질럿을 받는 게 아니라, SpeedUp이 가능한 인터페이스(ISpeedAdapter)를 받도록 하고, 마린과 질럿에 대해 각각 ISpeedAdapter를 구현하는 어댑터를 만들어서 MakeItFast 메서드로 넘기도록 할 것이다. 이를 코드로 구현하면 아래와 같다.

interface ISpeedAdapter

{

void SpeedUp();

}

static void MakeItFast(**ISpeedAdapter** adapter)

{

adapter.SpeedUp();

}

이제 마린이 ISpeedAdapter 자리에 들어갈 수 있도록 마린을 위한 아답터를 만들어야 한다.

class MarineAdapter : ISpeedAdapter

{

public MarineAdapter(Marine marine)

{

\_marine = marine;

}

public Marine \_marine;

public void SpeedUp()

{

\_marine.RunSteamPack();

}

}

마린 어댑터는 ISpeedAdapter를 구현하니까 SpeedUp 메서드를 구현하여야 하는데, 생성자의 매개변수로 전달받은 마린의 RunSteamPack 메서드를 레퍼하여 구현하였다. 질럿의 경우도 유사하다.

class ZealotAdapter : ISpeedAdapter

{

public ZealotAdapter(Zealot zealot)

{

\_zealot = zealot;

}

public Zealot \_zealot;

public void SpeedUp()

{

\_zealot.Charge();

}

}

이번에는 질럿의 Charge 메서드를 SpeedUp 메서드가 래퍼하고 있다.

static void Main(string[] args)

{

Marine marine = new Marine();

**MakeItFast(new MarineAdapter(marine));**

Console.WriteLine(marine.Speed);

Zealot zealot = new Zealot();

**MakeItFast(new ZealotAdapter(zealot));**

Console.WriteLine(zealot.Speed);

}

static void MakeItFast(ISpeedAdapter adapter)

{

adapter.SpeedUp();

}

이렇게 마린과 질럿의 소스를 수정하는 대신에, 각각 적절한 어댑터를 장착함으로써 MakeItFast 메서드의 중복을 피하고 간결하게 작성할 수 있게 되었다.

# 메멘토 패턴

메멘토 패턴을 사용하면 클래스의 상태 변화가 있을 때 마다 새로운 상태로 변경되기 전에 기존의 상태를 별도로 저장해두어, 필요할 때면 언제든 과거의 특정 상태로 돌아갈 수 있다. 먼저 마린 클래스를 살펴보자. 마린 클래스는 공격력과 방어력을 가지고 있다.

public class Marine

{

public Marine()

{

Weapon = 1;

Armor = 1;

}

public int Weapon { get; private set; }

public int Armor { get; private set;}

public void UpgradeWeapon()

{

Weapon++;

}

public void UpgradeArmor()

{

Armor++;

}

public void ShowUpgradeInfo()

{

Console.WriteLine($"w:{Weapon}, a:{Armor}");

}

}

Weapon과 Armor 속성의 경우 set의 접근지정자가 private이기 때문에 클래스 외부에서는 값을 변경할 수 없다. 이 때 마린을 생성하고 공격력 1차례, 방어력 1 차례, 다시 공격력을 1차례 업그레이드 하는 코드는 아래와 같다.

­­­­private static void Main(string[] args)

{

Marine marine = new Marine(); // 공격력 1, 방어력 1

marine.UpgradeWeapon(); // 공격력 2, 방어력 1

marine.UpgradeArmor(); // 공격력 1, 방어력 2

marine.UpgradeWeapon(); // 공격력 2, 방어력 2

marine.ShowUpgradeInfo();

}

w:3, a:2

여기서 어떤 이유로 마지막 두 번의 업그레이드를 취소하여하 한다고 가정해보자.즉 마린의 공격력이 2, 방어력이 1인 상황으로 돌아가야 하는 것이다.

private static void Main(string[] args)

{

Marine marine = new Marine(); // 공격력 1, 방어력 1

marine.UpgradeWeapon(); // 공격력 2, 방어력 1

marine.UpgradeArmor(); // 공격력 1, 방어력 2

marine.UpgradeWeapon(); // 공격력 2, 방어력 2

**marine.DowngradeArmor();**

**marine.DowngradeWeapon();**

marine.ShowUpgradeInfo();

}

이런 식으로 공격력과 방어력을 다운그레이드 하는 메서드를 만드는 방법도 있겠지만, 이 경우 앞서 어떤 작업을 수행했는지를 정확하게 알고 있어야 하고, 그걸 취소하는 메서드를 순서에 맞게 호출해야 한다는 단점이 있다. 즉 위 경우에는 마지막 작업인 공격력 업그레이드를 먼저 취소한 후 방어력 업그레이드를 취소하여야 한다. 그래서 마지막 두 번의 업그레이드를 취소하는 코드는 이렇게 작성하는 것이 더욱 좋을 것 같다.

private static void Main(string[] args)

{

Marine marine = new Marine(); // 공격력 1, 방어력 1

marine.UpgradeWeapon(); // 공격력 2, 방어력 1

marine.UpgradeArmor(); // 공격력 1, 방어력 2

marine.UpgradeWeapon(); // 공격력 2, 방어력 2

**marine.Revert();**

**marine.Revert();**

marine.ShowUpgradeInfo();

}

어떤 작업을 어떤 순서로 실행했는지 굳이 기억할 필요없이, 단지 마지막으로 수행한 두 개의 작업을 취소하라는 코드이다. 이렇게 코드를 작성하기 위해서는 먼저 특정 시점에서의 마린의 상태(여기서는 공격력과 방어력)를 표현하기 위한 클래스가 필요하다.

public class UpgradeInfo

{

public UpgradeInfo(int weapon, int armor)

{

Weapon = weapon;

Armor = armor;

}

public int Weapon { get; set; }

public int Armor { get; set; }

}

마린은 공격력과 방어력을 업그레이드 하기 전에 매번 현재의 상태를 저장하도록 수정한다. 마치 사진을 찍어두는 것처럼 생각하면 되겠다.

public class Marine

{

public Marine()

{

Weapon = 1;

Armor = 1;

}

public int Weapon { get; private set; }

public int Armor { get; private set; }

**private readonly Stack<UpgradeInfo> \_histories = new Stack<UpgradeInfo>();**

public void IncreaseWeapon()

{

**\_histories.Push(new UpgradeInfo(Weapon, Armor));**

Weapon++;

}

public void IncreaseArmor()

{

**\_histories.Push(new UpgradeInfo(Weapon, Armor));**

Armor++;

}

public void ShowUpgradeInfo()

{

Console.WriteLine("w:{0}, a:{1}", Weapon, Armor);

}

**public void Revert()**

**{**

**UpgradeInfo upgradeInfo = \_histories.Pop();**

**Weapon = upgradeInfo.Weapon;**

**Armor = upgradeInfo.Armor;**

**}**

}

UpgradeInfo를 차곡차곡 쌓아두기 위해 스택을 준비해 놓고, 공격력이나 방어력을 증가시키기 전에 먼저 현재 상태를 사진을 찍듯이 찍어 스택에 집어넣는다. 마지막으로 한 작업을 취소하기 위해서는 스택에 있는 UpgradeInfo를 꺼내고, 거기에 있는 값으로 공격력과 방어력을 변경한다.

메멘토 패턴을 사용하는 가장 흔한 예는 텍스트 편집기 등의 되돌리기 기능일 것이다. 중요한 점은 UpgradeInfo처럼 어떤 객체의 한 시점에서의 값을 표현할 수 있는 클래스를 만드는 것인데, 보통 커맨트 패턴과 함께 사용하는 것이 일반적이다.

# 커맨드 패턴

커맨드 패턴은 서로 관련이 없는 작업들을 커맨드로 추상화시켜 동일한 방법으로 조작하는 패턴이다. 예를 들어 마린을 어떤 좌표로 이동시키는 메서드와 마린의 공격력을 업그레이드하는 메서드는 아무 관련이 없지만, 이 둘을 모두 커맨드로 간주하여 다루는 식이다.

예를 들어보자. 마린 클래스는 위치와 공격력, 방어력을 가지고 있고, 특정 좌표로 이동하고 공격력과 방어력을 업그레이드하는 메서드도 가지고 있다.

public class Marine

{

public Marine()

{

X = 1;

Y = 1;

Weapon = 1;

Armor = 1;

}

public int X { get; set; }

public int Y { get; set; }

public int Weapon { get; set; }

public int Armor { get; set; }

public void Move(int x, int y)

{

X = x;

Y = y;

}

public void UpgradeWeapon()

{

Weapon++;

}

public void UpgradeArmor()

{

Armor++;

}

}

그렇다면 다음과 같은 작업을 수행하는 코드는 이렇게 쓸 수 있다.

1) 마린의 공격력을 업그레이드한다.

2) 마린을 (50,100)으로 이동시킨다.

3) 마린의 방어력을 업그레이드한다.

static void Main(string[] args)

{

Marine marine = new Marine();

marine.UpgradeWeapon();

marine.Move(50,100);

marine.UpgradeArmor();

}

물론 이렇게 코드를 작성해도 문제가 없지만,