브릿지 패턴

**정의**

브릿지 패턴은 GoF 의 23개 디자인 패턴 중 하나이고,

소프트웨어들을 더욱 쉽게 구현, 변경, 테스트, 재사용 가능하게 만들어준다.

구현부에서 추상층을 분리하여 각자 독립적으로 변형할 수 있게 하는 패턴이다.

**브릿지 패턴으로 해결할 수 있는 문제**

런타임에서 선택될 수 있게 컴파일 타임 바인딩과 구현을 피해야 하는 문제.

상속과 구현이 각각 따로 정의되고 확장되어야 하는 문제.

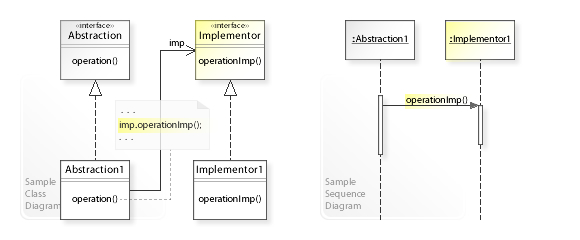
**브릿지 패턴이 제시하는 해결책**

Implementor 오브젝트에 Abstraction 을 델리게이팅 하여 구현한다. (오른쪽 이미지)

다른 클레스 하이라키에 배치하여 상속을 구현과 분리한다. (왼쪽 이미지)

이 패턴으로 Implementor 오브젝트를 Abstraction 으로 런타임에 구성할 수 있게 된다.

**구조**

[](https://en.wikipedia.org/wiki/File:W3sDesign_Bridge_Design_Pattern_UML.jpg)

**예시 코드**

// Helps in providing truly decoupled architecture

public interface IBridge

{

void Function1();

void Function2();

}

public class Bridge1 : IBridge

{

public void Function1()

{

Console.WriteLine("Bridge1.Function1");

}

public void Function2()

{

Console.WriteLine("Bridge1.Function2");

}

}

public class Bridge2 : IBridge

{

public void Function1()

{

Console.WriteLine("Bridge2.Function1");

}

public void Function2()

{

Console.WriteLine("Bridge2.Function2");

}

}

public interface IAbstractBridge

{

void CallMethod1();

void CallMethod2();

}

public class AbstractBridge : IAbstractBridge

{

public IBridge bridge;

public AbstractBridge(IBridge bridge)

{

this.bridge = bridge;

}

public void CallMethod1()

{

this.bridge.Function1();

}

public void CallMethod2()

{

this.bridge.Function2();

}

}

**장점**

클래스를 구현과 추상 부분으로 분리할 수 있고, 이는 독립적인 확장이 가능하다.

컴파일 타임 바인딩 대신 런타임에서 바인딩 할 수 있다.

구현에 따라 상세한 기능을 외부로부터 숨길 수 있는 은닉 효과도 얻을 수 있다.

**단점**

추상화를 통해 코드를 분리할 경우 코드 디자인 설계가 복잡해진다.

도움을 받은 링크들

https://hirlawldo.tistory.com/169

https://en.wikipedia.org/wiki/Bridge\_pattern