Bridge Pattern
기능의 계층과 구현의 계층을 분리한다

01. Bridge 패턴

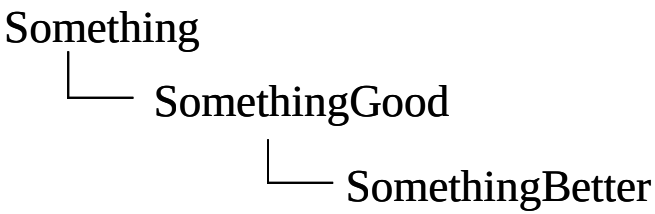
- Bridge (다리)
  - 두 장소를 연결하는 역할
  - '기능의 클래스 계층'과 '구현의 클래스 계층' 사이에 다리를 놓는다.

- 클래스 계층의 두 가지 역할
  - 기능의 클래스 계층
  - 구현의 클래스 계층

01. Bridge 패턴

- 새로운 '기능'을 추가하고 싶을 때
  - Something 클래스에 새로운 기능을 추가하려고 할 때
    - Something 클래스의 하위 클래스를 새롭게 만든다

    - 소규모의 클래스 계층이 발생함 => '기능의 클래스 계층"
  - SomethingGood 클래스에 다시 새로운 기능을 추가하려면...



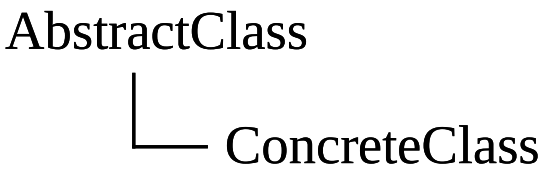
  - 새로운 기능을 추가하고 싶을 때 클래스 계층 안에서 새로 만들려고 하는 클래스와 유사한 클래스를 찾아내, 하위 클래스를 만들어 기능을 추가한다.

01. Bridge 패턴

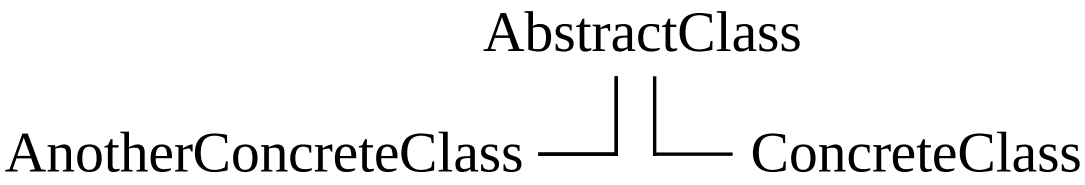
- 새로운 '구현'을 추가하고 싶을 때
  - 추상 클래스는, 일련의 메소드들을 추상 메소드로 선언하고, 인터페이스(API)를 규정한다
  - 하위 클래스 쪽에서 그 추상 메소드를 실제로 구현한다.
  - 상위 클래스는 추상 메소드로, 인터페이스를 규정하는 역할을 한다.
  - 상위 클래스와 하위 클래스의 역할 분담에 의해 부품으로서의 가치(교환 가능성)가 높은 클래스를 만들 수 있다.
    - 이유: 상위 클래스를 구현한 하위 클래스들끼리의 API는 통일되므로

01. Bridge 패턴

- 새로운 '구현'을 추가하고 싶을 때
  - 이 때에도 클래스 계층이 등장한다.



  - 이를 '구현의 클래스 계층'이라고 한다.
  - AbstracClass의 다른 구현을 만들고 싶으면, 다른 하위 클래스를 만들면 된다.



01. Bridge 패턴

- 클래스 계층의 혼재와 클래스 계층의 분리
  - 클래스 계층 구조 하나에, '기능의 클래스 계층'과 '구현의 클래스 계층'이 혼재해 있으면, 새로운 하위 클래스를 만들 때 어려움이 있다.
  - '기능의 클래스 계층'과 '구현의 클래스 계층'을 분리하고, 이들 사이에 '다리'를 놓자.
    - => Bridge 패턴

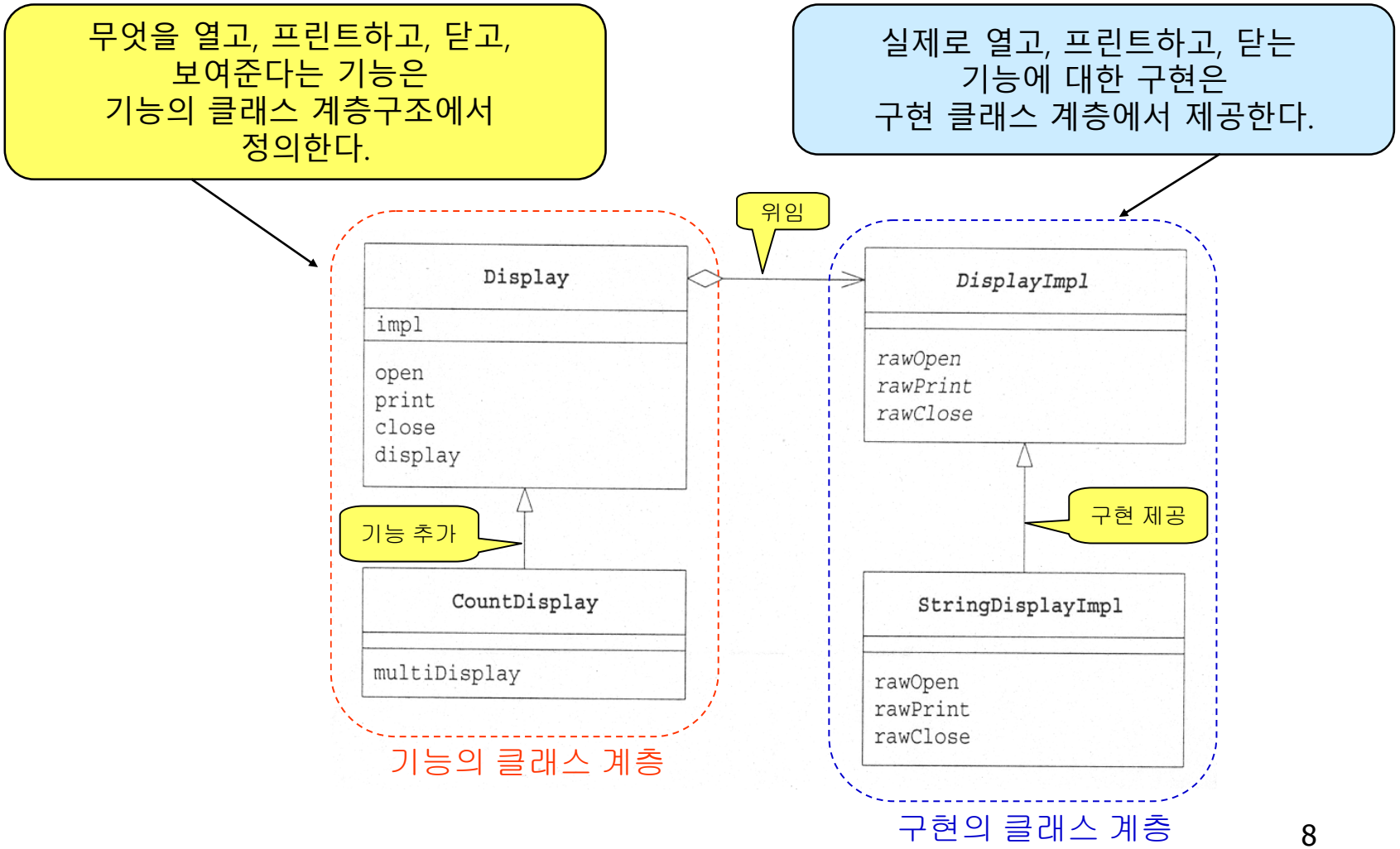
02. 예제 프로그램

- '무언가를 표시하기' 위한 프로그램

다리의 어느 쪽?	이름	해설
기능의 클래스 계층	Display	표시하는 클래스
기능의 클래스 계층	CountDisplay	지정 횟수만 표시하는 기능을 추가한 클래스
구현의 클래스 계층	DisplayImpl	표시하는 클래스
구현의 클래스 계층	StringDisplayImpl	문자열을 사용해서 표시하는 클래스
	Main	동작 테스트용 클래스

02. 예제 프로그램

- 클래스 다이어그램



02. 예제 프로그램

- 기능의 클래스 계층: Display 클래스
  - 추상적인 '무언가를 표시하기 위한 것'으로 '기능의 클래스 계층'의 최상위에 존재
  - impl 필드: Display 클래스의 '구현'을 나타내는 인스턴스
  - 생성자
    - 구현을 나타내는 클래스(DisplayImpl)의 인스턴스를 인자로 넘겨 받는다.
  - open, print, close 메소드: 모두 DisplayImpl의 API를 호출한다.
    - open( ): 표시의 전 처리
    - print( ): 표시하는 작업
    - close( ): 표시 후의 처리

    => Display 인터페이스인 open, print, close가 DisplayImpl의 인터페이스인 rawOpen, rawPrint, rawClose 로 바뀌어 있음을 알 수 있다.
  - display()
    - open, print, close 를 차례로 호출한다.

02. 예제 프로그램

- 기능의 클래스 계층: CountDisplay 클래스
  - Display 클래스에 기능을 추가함
    - Display 클래스에는 '표시한다'는 기능 밖에 없음
    - CountDisplay에는 '지정횟수만큼 표시한다'라는 기능이 추가됨
      - multiDisplay( )

02. 예제 프로그램

- 구현의 클래스 계층: DisplayImpl 클래스
  - '구현의 클래스 계층'의 최상위에 위치함
  - 추상 클래스이며, rawOpen, rawPrint, rawClose 메소드를 가짐

02. 예제 프로그램

- 구현의 클래스 계층: StringDisplayImpl 클래스
  - 문자열을 표시하는 클래스
  - rawOpen, rawPrint, rawClose 메소드 구현
    - printLine( )를 이용해서 문자열을 표시함

02. 예제 프로그램

- Main 클래스
  - Display나 CountDisplay 생성 시, 구현을 제공하는 클래스인 StringDisplayImpl의 객체를 인자로 넘겨준다.

03. 등장 역할

- Abstraction의 역할
  - '기능의 클래스 계층'의 최상위에 있는 클래스
  - Implementor 역할의 메소드를 사용해서 기본적인 기능만을 제공하는 클래스
  - 예제에서 Display 클래스가 해당됨

- RefinedAbstraction의 역할
  - Abstraction 역할에 기능을 추가한 역할
  - 예제에서 CountDisplay 클래스가 해당됨

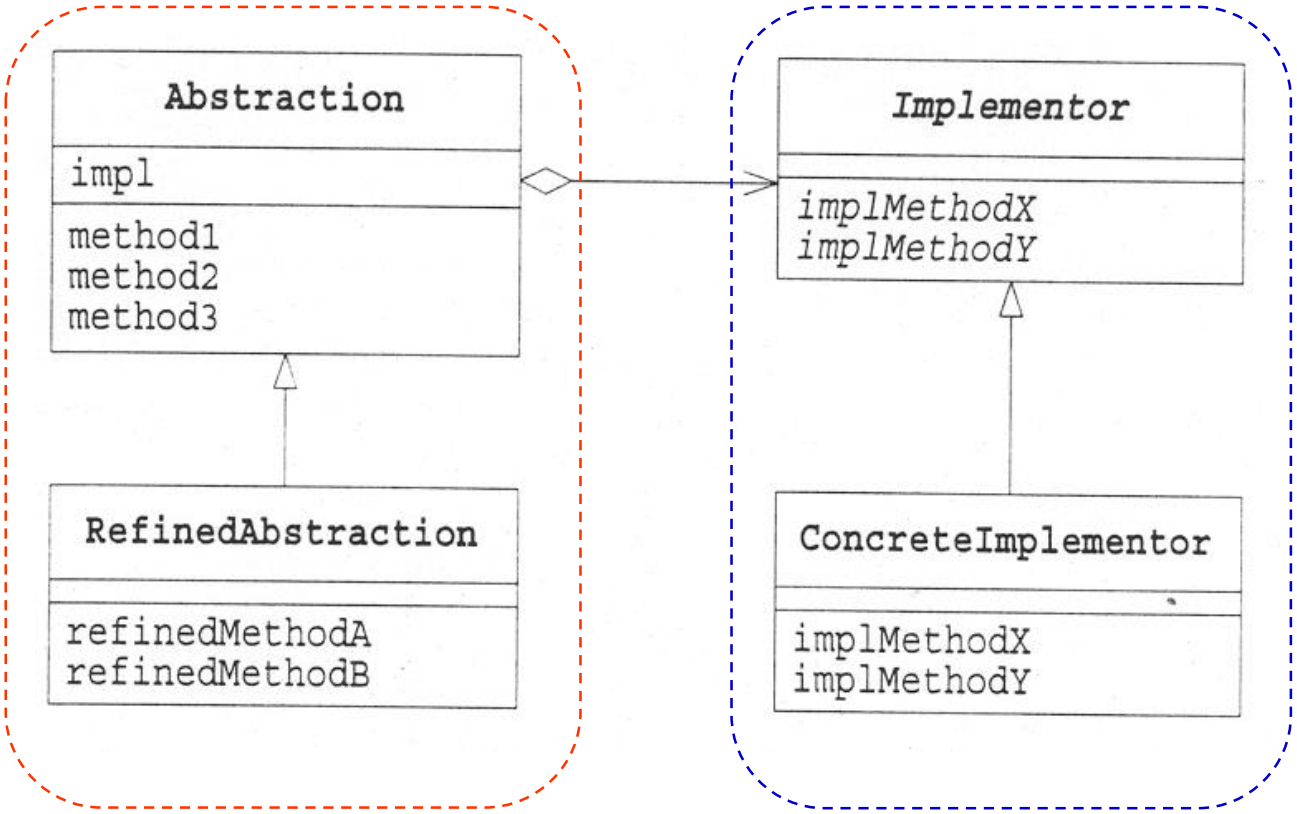
03. 등장 역할

- Implementor의 역할
  - '구현의 클래스 계층'의 최상위에 있는 클래스
  - Abstraction 역할의 인터페이스를 구현하기 위한 메소드를 규정하는 역할
  - 예제에서 DisplayImpl 클래스가 해당됨

- ConcreteImplementor의 역할
  - Implementor 역할의 인터페이스를 구체적으로 구현하는 역할
  - 예제에서, StringDisplayImpl 클래스가 해당됨

03. 등장 역할

- 클래스 다이어그램

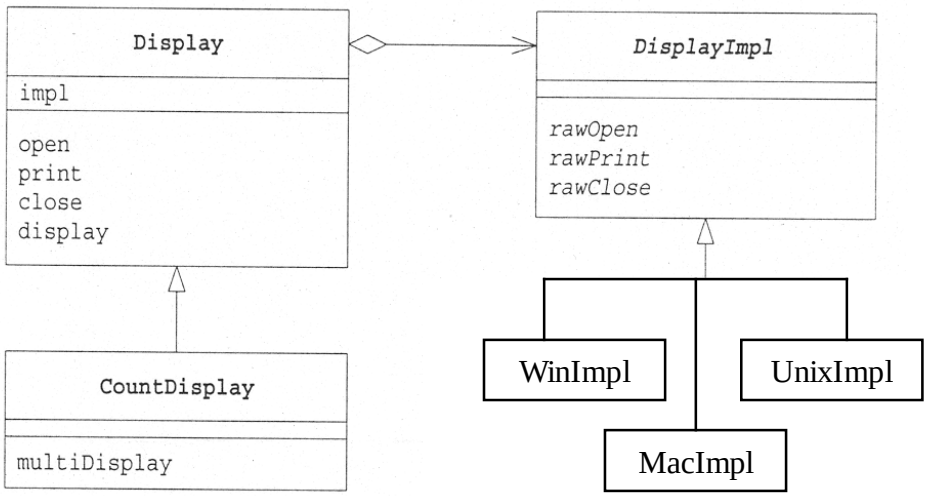


04. 독자의 사고를 넓혀주는 힌트

- 분리해 두면 확장이 편해진다.
  - 두 개로 클래스 계층을 나누어두면, 각각의 클래스 계층을 독립적으로 확장할 수 있다 (예는 연습문제에서)
    - 1) 기능을 추가하고 싶으면, 기능의 클래스 계층에 추가한다.
      - 이 때 구현 클래스 계층은 전혀 수정할 필요가 없다.

04. 독자의 사고를 넓혀주는 힌트

- 분리해 두면 확장이 편해진다.
  - 2) 구현을 추가하고 싶으면 구현 클래스 계층을 확장한다.
    - 예: 어떤 프로그램에 OS(운영체제) 의존 부분이 있어서, Window 판, Macintosh 판, Unix 판으로 구분된다고 하자 => 이 부분을 "구현의 클래스 계층"으로 표현한다.
      - Display 생성 시, 생성자에게 WinImpl, MacImpl, UnixImpl 중 적당한 것 하나를 넘겨주면 된다.



04. 독자의 사고를 넓혀주는 힌트

- 상속은 견고한 연결, 위임은 느슨한 연결
  - 상속은 소스 코드를 바꿀 수 없는 매우 견고한 연결임
    - 클래스간의 관계를 바꾸고 싶을 때는 상속을 사용해서는 안 됨.
  - Display 클래스 내에서 위임이 사용된다.
    - 예: open을 실행할 때, impl.rawOpen( )을 호출하여 '떠넘기기'를 한다.
      - 이 때, impl 이 참조하고 있는 객체는, Display 생성 시 클라이언트(Main 클래스)가 넘겨준 구체적인 StringDisplayImpl 클래스의 인스턴스이다.
      - StringDisplayImpl 이외의 다른 클래스의 객체를 넘겨주면, 구현이 교체되는 효과를 가져온다.

05. 관련 패턴

- Template Method 패턴
  - Template Method 패턴에서는 구현의 클래스 계층을 이용함
    - 상위 클래스에서는 추상 메소드를 사용해서 프로그래밍하고 하위 클래스에서는 그 추상 메소드를 구현함
- Abstract factory 패턴
  - Bridge 패턴에 등장하는 ConcreteImplementor 역할을 환경에 맞춰서 적절히 구축하기 위해 Abstract Factory 패턴이 이용되는 경우가 있음
- Adapter 패턴
  - Bridge 패턴은 기능의 클래스 계층과 구현의 클래스 계층을 확실히 분리한 다음에 연결시키는 패턴임
  - Adapter 패턴도 기능은 비슷하지만 인터페이스(API)가 다른 클래스를 결합시키는 패턴임

06. 요약

- 두 종류의 클래스 계층 사이의 다리를 놓는 Bridge 패턴

- 구현의 클래스 계층에서 제공하는 메소드들을 이용하고 조합하여, 기능 클래스 계층에서 부가적인 기능을 제공한다.