

## 과정 소개

- 객체지향 프로그래밍 언어를 사용해서 S/W 개발시 발생하는 다양한 문제를 해결하기 위한 디자인 기법을 배우는 것이 목표.
- C++ 언어를 기반으로 진행 ( 다른 객체지향 언어 에도 적용될 수 있음 )
- 전통적인 디자인 기법부터 C++기반의 오픈 소스가 다양한 기술 까지 설명

## 디자인 패턴이란 ?

- **구구단 프로그램을 작성하려면 ?**

⇒ 중첩된 반복문을 사용하면 된다.

- **프로그램에서 `undo/redo` 하는 기능을 추가 하고 싶다면 ?**

⇒ 초급 개발자는 코드를 어떻게 작성해야 할지 쉽게 생각나지 않는다.

⇒ 하지만, 이미 수많은 프로그램에 `undo/redo` 기능이 제공되고 있다.

⇒ 많은 좋은 프로그램에서 사용하는 검증된 기법을 먼저 배우면 좋지 않을까 ?

## 디자인 패턴이란 ?

- 프로그램에는 전형적인 코딩 패턴이 있다.

- ⇒ 디자인 패턴이란 ? 전형적인 코딩 스타일에 “이름 을 붙인 것”
- ⇒ 1995년 발간된 책의 이름 ( 4명의 저자, Gangs Of Four)
- ⇒ 저자들이 코딩 스타일 자체를 만든 것이 아닌 당시 유행하는 기법에 “이름” 을 부여한 것
- ⇒ undo/redo 를 위한 기술은 **command 패턴** 이라고 알려진 기술

## 디자인 패턴이란 ?

- 전통적인 디자인 패턴에서는 23개의 패턴으로 분류.
  - ⇒ 중요한 것과 현재는 잘 사용되지 않은 것도 있음.
  - ⇒ 과정에서는 중요한 것 위주로 설명
- 강의에서는
  - ⇒ 도입 부분에서는 간단한 C++ 문법 정리와
  - ⇒ 인터페이스 등의 객체지향 프로그래밍의 핵심 개념 설명
  - ⇒ 이후에, 각각의 패턴을 설명

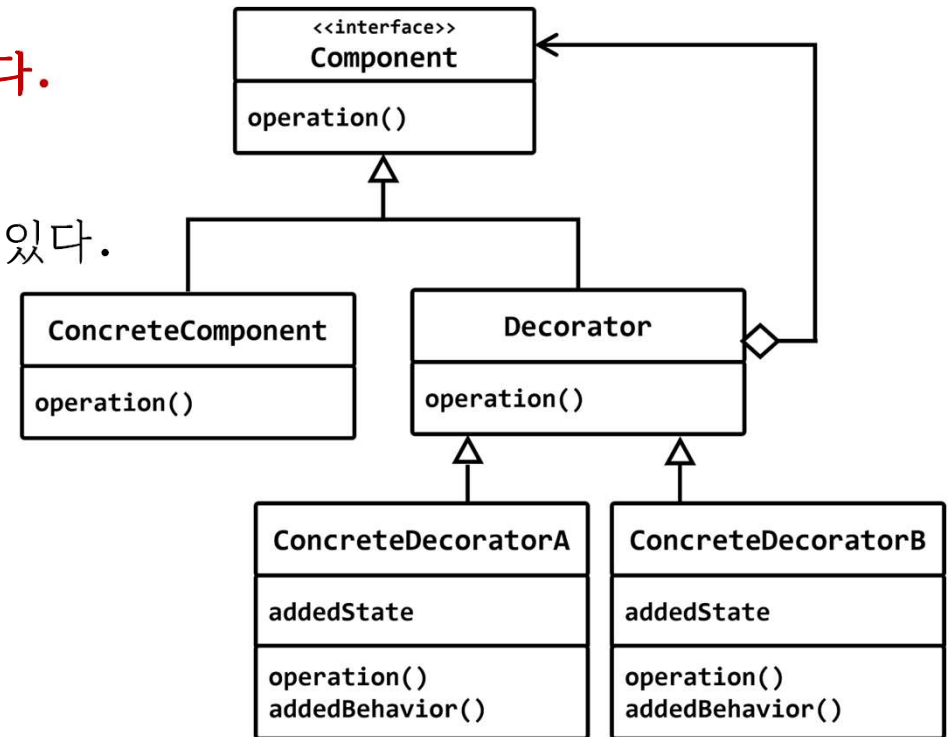
# Decorator

- 구조 패턴 ( Structural Pattern )

- 의도 ( intent )

⇒ 객체에 동적으로 서비스를 추가할 수 있게 한다.

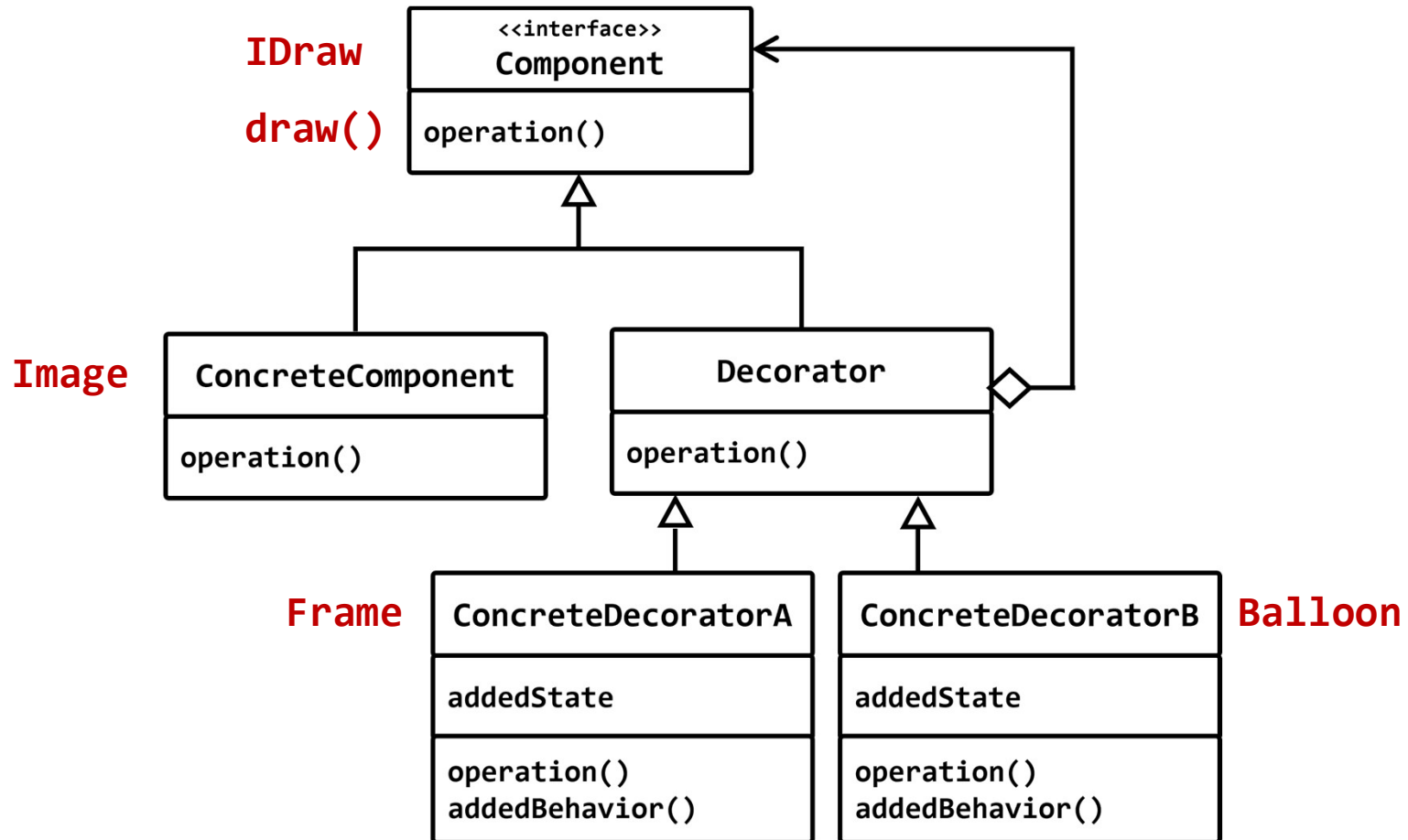
⇒ 상속을 사용해서 서비스를 추가 하는 것보다  
유연한 방법으로 새로운 서비스를 추가할 수 있다.



# Decorator

- 구조 패턴 ( Structural Pattern )
- 의도 ( intent )
  - ⇒ 객체에 동적으로 서비스를 추가할 수 있게 한다.
  - ⇒ 여러 개의 기능을 중복해서 추가할 수 있다.
  - ⇒ 상속을 사용해서 서비스를 추가(클래스에 기능 추가) 하는 것보다 유연한 방법으로 새로운 서비스를 추가할 수 있다.

# Decorator



# Adapter

- 구조 패턴 ( Structural Pattern )

- 의도 ( intent )

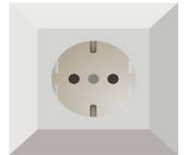
⇒ 인터페이스를 “클라이언트가 기대하는 형태의 인터페이스로 변환”.



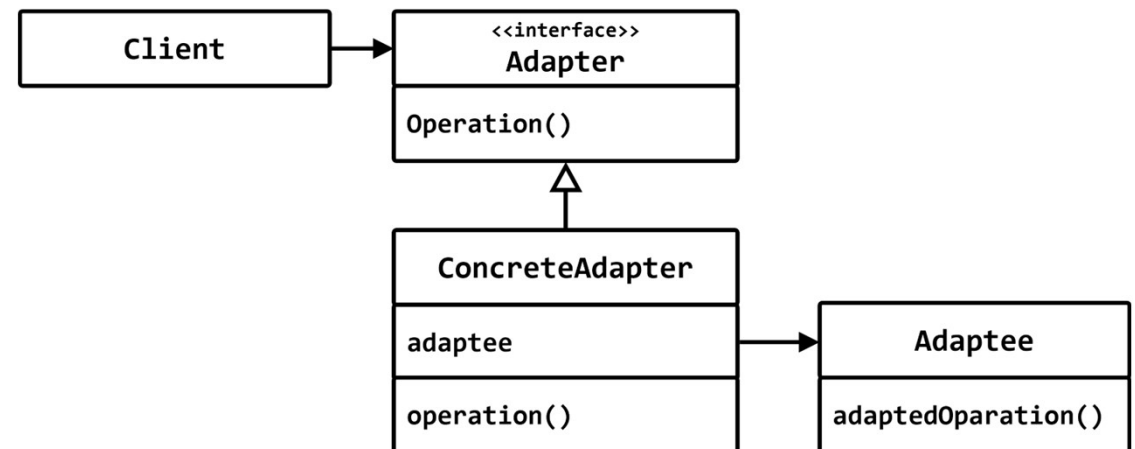
12V



**Adapter**



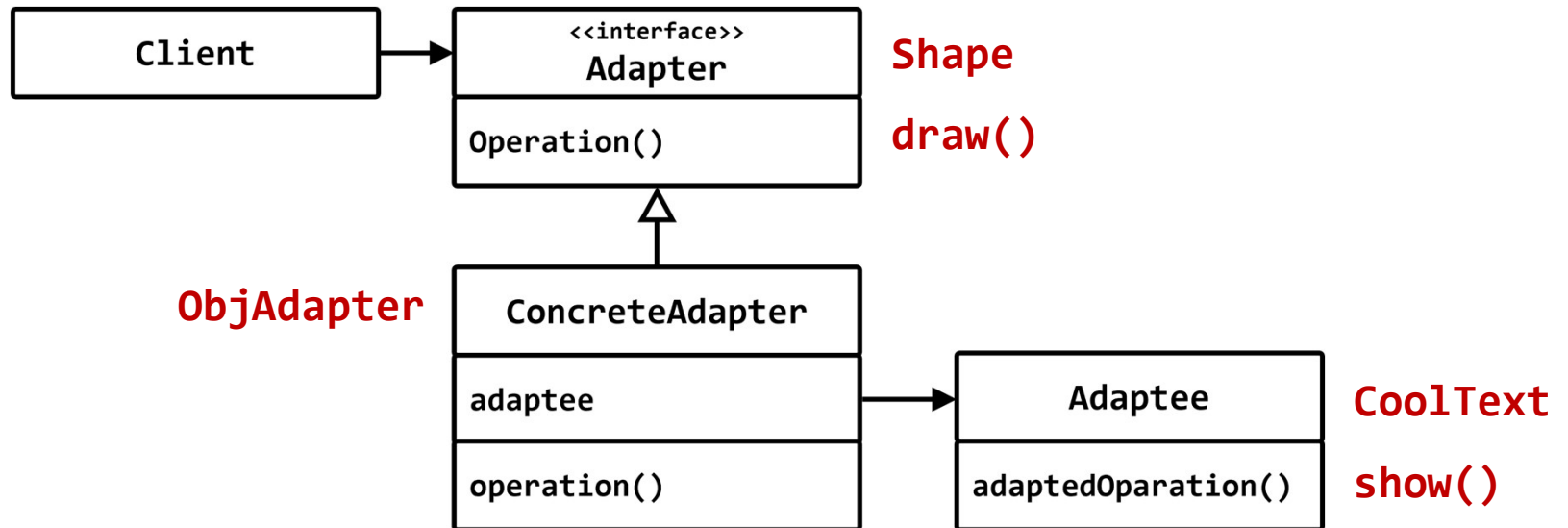
220V





# Adapter

- 구조 패턴 ( Structural Pattern )
- 의도 ( intent )
  - ⇒ 인터페이스를 “클라이언트가 기대하는 형태의 인터페이스로 변환”.



# Adapter

```
class ClsAdapter : public CoolText, public Shape
{
public:
    ClsAdapter(const std::string& text) : CoolText(text) {}

    void draw() override { CoolText::show();}
};
```

클래스에 대한  
인터페이스 변경(상속)

```
class ObjAdapter : public Shape
{
    CoolText* ct;
public:
    ObjAdapter(CoolText* ct) : ct(ct) {}

    void draw() override { ct->show();}
};
```

객체에 대한  
인터페이스 변경(포함)

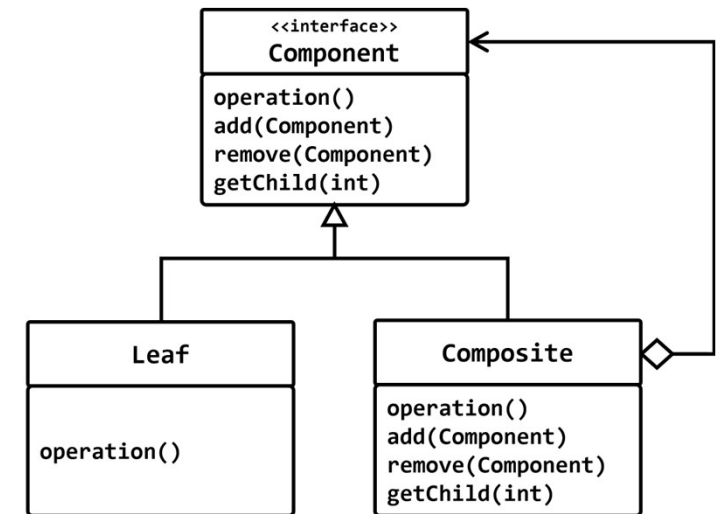
# Composite

- 구조 패턴 ( Structural Pattern )

- 의도 ( intent )

⇒ 부분과 전체의 계층을 표현하기 위해 **복합 객체를 트리 구조**로 만든다.

Composite 패턴은 클라이언트로 하려면 **개별 객체와 복합 객체를 모두 동일하게 다룰 수 있도록** 한다.



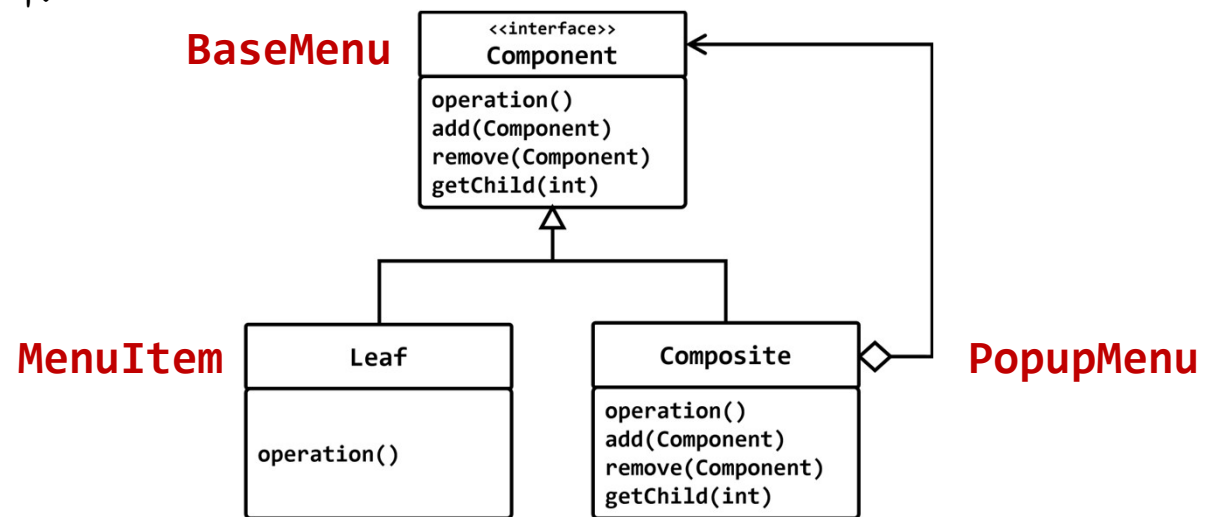
# Composite

- 구조 패턴 ( Structural Pattern )

- 의도 ( intent )

⇒ 부분과 전체의 계층을 표현하기 위해 **복합 객체를 트리 구조**로 만든다.

Composite 패턴은 클라이언트로 하려면 **개별 객체와 복합 객체를 모두 동일하게 다룰 수 있도록** 한다.

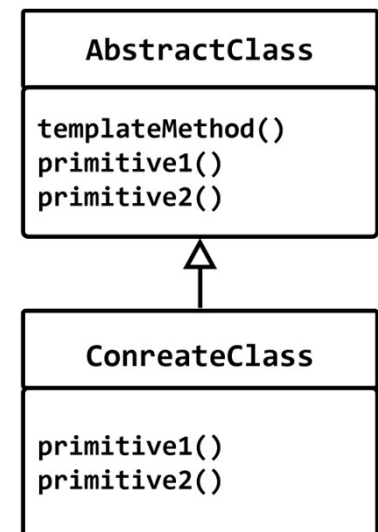


# template method

- 행위 패턴 ( Behavior Pattern )

- 의도 ( intent )

⇒ 오퍼레이션에는 알고리즘의 처리 과정만을 정의 하고 각 단계에서 수행할 구체적인 처리는 서브클래스 에서 정의 한다. Template Method 패턴은 “**알고리즘의 처리과정은 변경하지 않고 알고리즘 각 단계의 처리를 서브클래스에서 재정의**” 할 수 있게 한다.



# template method

```
class Shape
{
public:
    void draw()
    {
        _____
        _____
        _____
        _____
        _____
        _____
    }
    virtual method1() = 0;
    virtual method2() = 0;
};
```

```
method1() override {}
method2() override {}
```

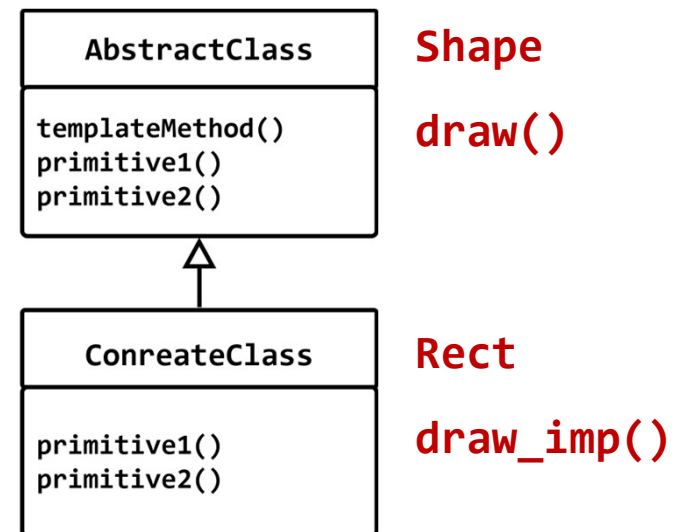
알고리즘의 처리과정은 변경되지 않는다.

파생 클래스에 따라서 변화가 필요한 부분  
(변경할 기회를 주고 싶은 코드)  
가상함수로 분리

# template method

- 행위 패턴 ( Behavior Pattern )
- 의도 ( intent )

⇒ 오퍼레이션에는 알고리즘의 처리 과정만을 정의 하고 각 단계에서 수행할 구체적인 처리는 서브클래스 에서 정의 한다. Template Method 패턴은 “**알고리즘의 처리과정은 변경하지 않고 알고리즘 각 단계의 처리를 서브클래스에서 재정의**” 할 수 있게 한다.

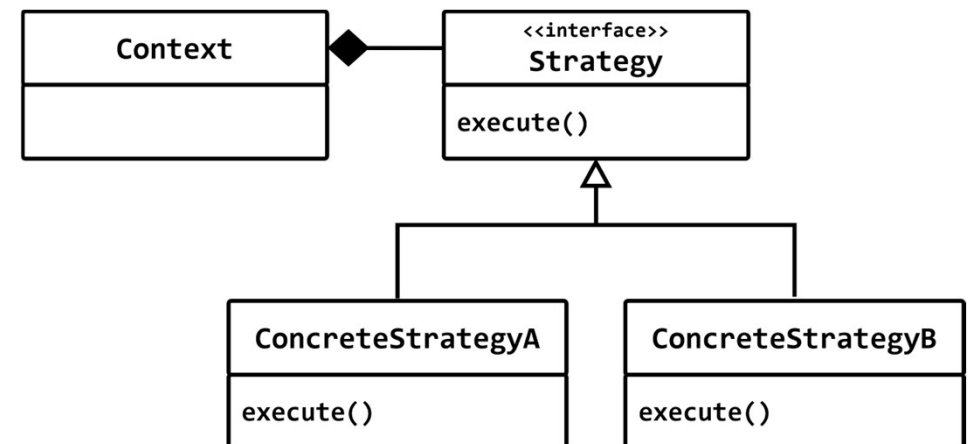


# Strategy(전략패턴)

- 행위 패턴 ( Behavior Pattern )

- 의도 ( intent )

⇒ 다양한 알고리즘이 존재 하면 이들 각각을 하나의 “클래스로 캡슐화 하여 알고리즘의 대체가 가능”하도록 한다. Strategy 패턴을 이용하면 클라이언트와 독립적인 다양한 알고리즘으로 변형할 수 있다. 알고리즘을 바꾸더라도 클라이언트는 아무런 변경을 할 필요가 없다.

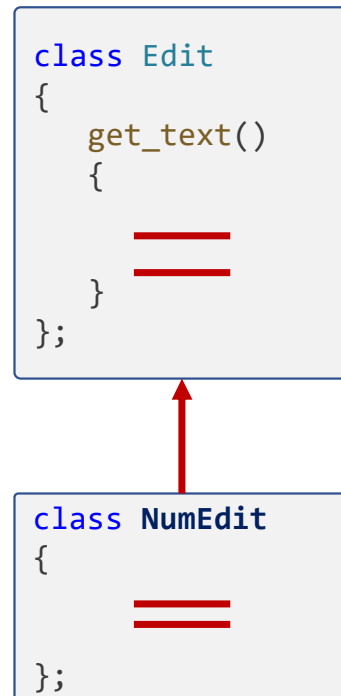




# Strategy(전략패턴)

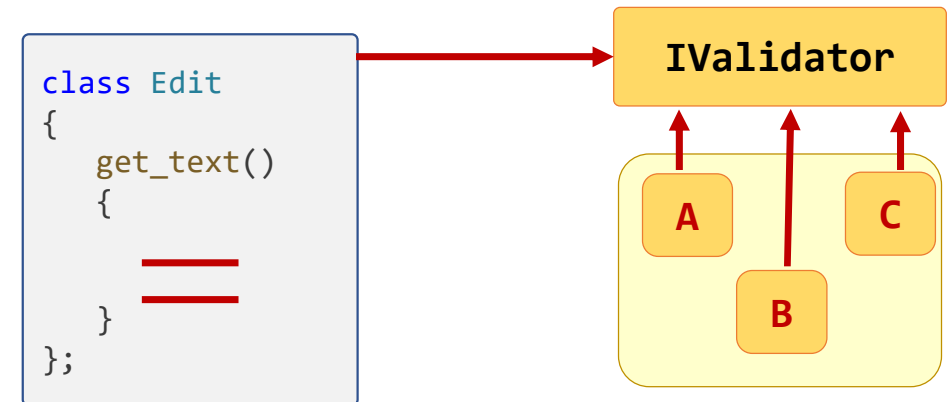
- Validation 정책을 **가상함수로 분리**

- NumEdit 가 **Validation** 정책을 소유
- ⇒ 다른 클래스에서 **Validation** 정책을 사용할 수 없다.
- ⇒ 실행시간에 **Validation** 정책을 교체 할 수 없다.



template method

- Validation 정책을 **다른 클래스로 분리**



- Edit와 **Validation** 정책이 **서로 다른 클래스로 분리**
- ⇒ 다른 클래스에서 **Validation** 정책을 사용할 수 **있다**.
- ⇒ 실행시간에 **Validation** 정책을 교체 할 수 **있다**.

strategy

## Strategy(전략패턴)


- Edit 예제의 경우는 strategy 가 더 적합하지만
- template method 자체가 나쁜 것은 아님.
- **사각형을 그리는 방법은**
  - ⇒ 다른 클래스에서 사용해야 될 일이 없고,
  - ⇒ 실행시간에 교체할 이유도 없음.
  - ⇒ 가상함수로 구현되면 멤버 함수 이므로  
멤버 데이터 접근도 편리함.

```
class Shape
{
public:
    void draw()
    {

    }

    virtual draw_imp() = 0;
};
```

```
class Rect : public Shape
{
    draw_imp()
};
```

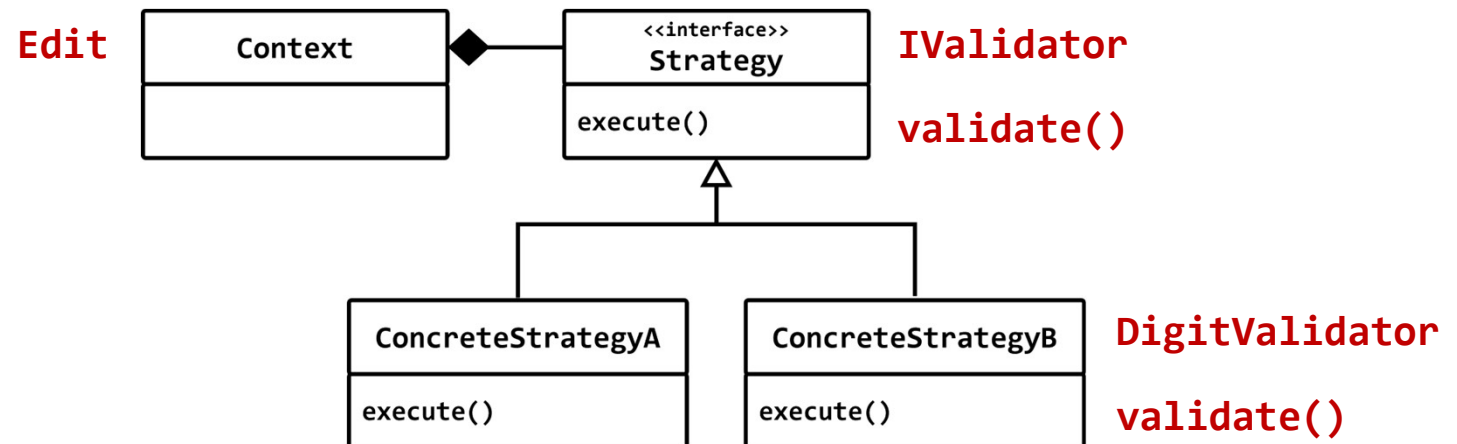


# Strategy(전략패턴)

- 행위 패턴 ( Behavior Pattern )

- 의도 ( intent )

⇒ 다양한 알고리즘이 존재 하면 이들 각각을 하나의 “클래스로 캡슐화 하여 알고리즘의 대체가 가능”하도록 한다. Strategy 패턴을 이용하면 클라이언트와 독립적인 다양한 알고리즘으로 변형할 수 있다. 알고리즘을 바꾸더라도 클라이언트는 아무런 변경을 할 필요가 없다.



# facade

facade

TCPServer

서브 시스템을 사용하기 쉽게 하는  
포괄적 개념의 인터페이스

2차 API

객체지향, 클래스, 1차 API 의 각각의 기능에 1:1로 대응하는 클래스

1차 API

C언어 기반, 각각의 분야(네트워크, 음악, 영상 등)의 함수와 데이터

OS

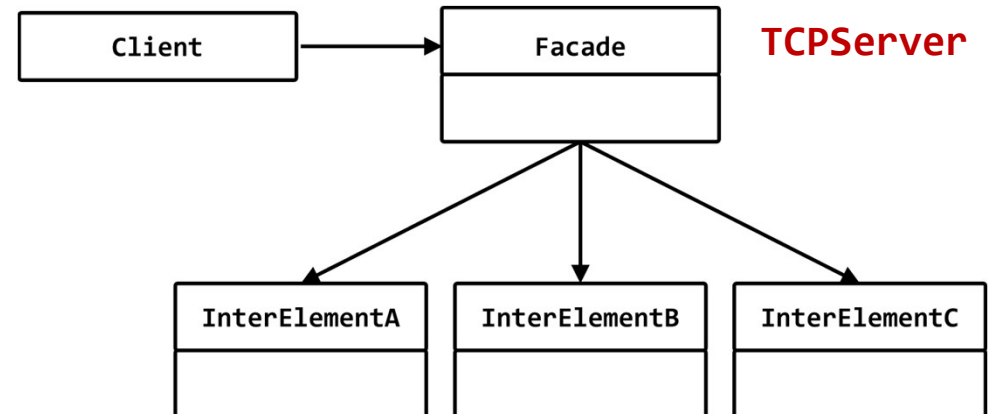
Windows, Linux, OSX 등.. 대부분 C 언어로 작성된 OS

# facade

- 구조 패턴 ( Structural Pattern )

- 의도 ( intent )

⇒ 서브 시스템을 합성하는 다수의 객체들의 인터페이스 집합에 대해 “**일관된 하나의 인터페이스를 제공**” 할 수 있게 할 수 있게 한다. Facade 는 서브시스템을 “**사용하기 쉽게 하기 위한 포괄적 개념의 인터페이스를 정의**” 한다

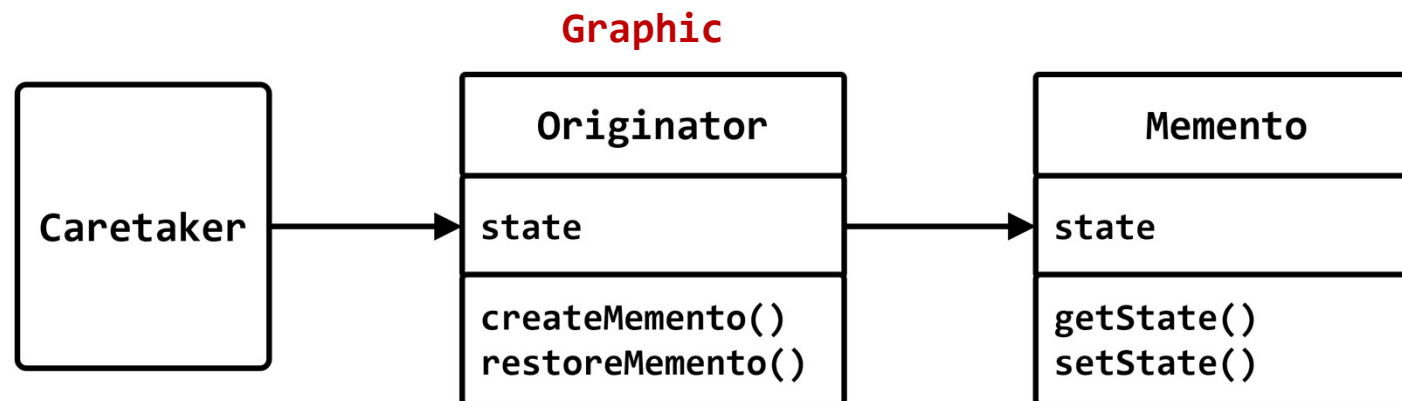


# memento

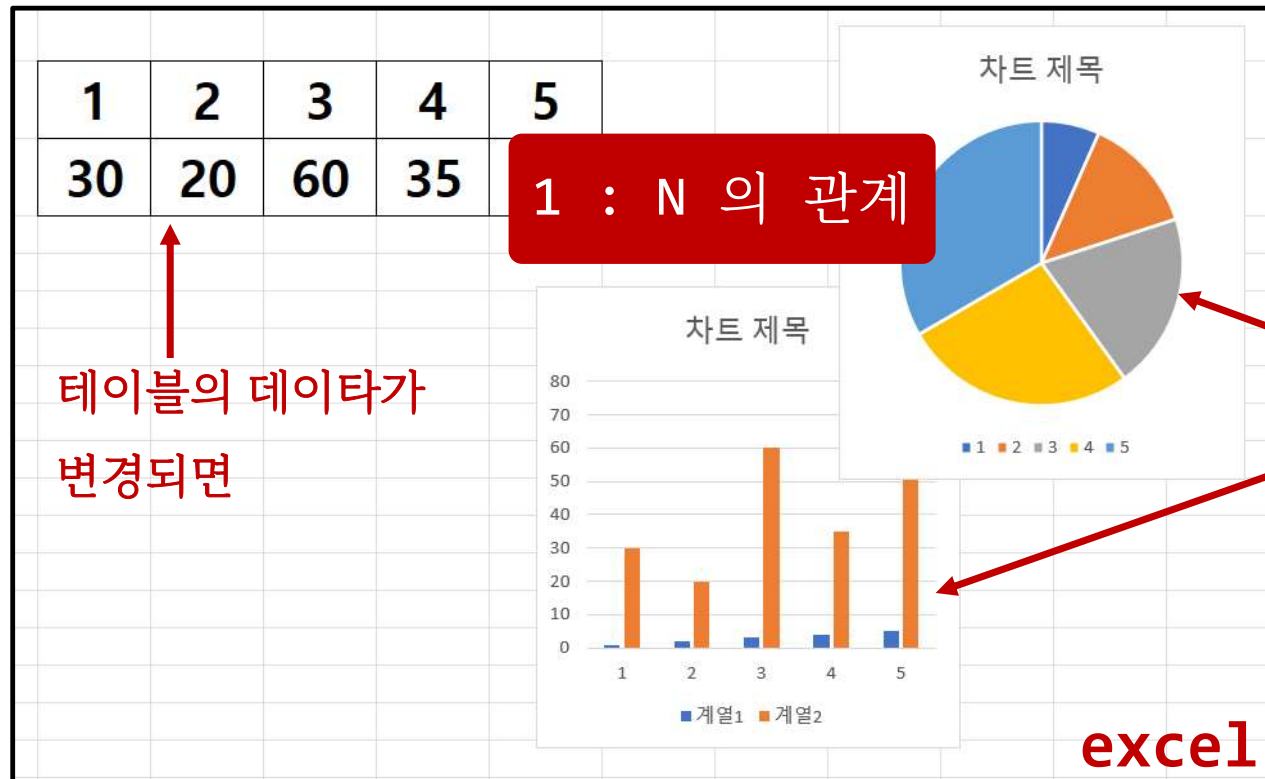
- 행위 패턴 ( Behavior Pattern )

- 의도 ( intent )

⇒ 캡슐화를 위배 하지 않고 “객체 내부 상태를 캡슐화 해서 저장하고, 나중에 객체가 이 상태로 복구” 가능하게 한다.



observer



테이블의 데이터가  
변경되면

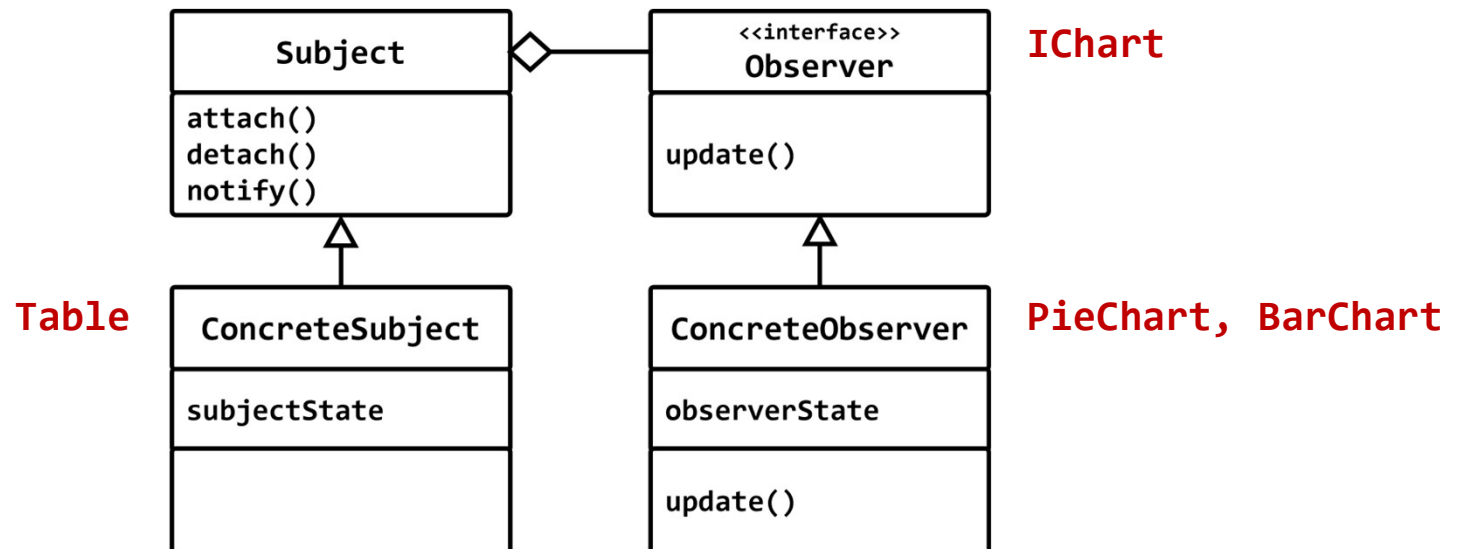
테이블과 연결된  
차트가 자동으로  
업데이트 된다.

# observer

- 행위 패턴 ( Behavior Pattern )

- 의도 ( intent )

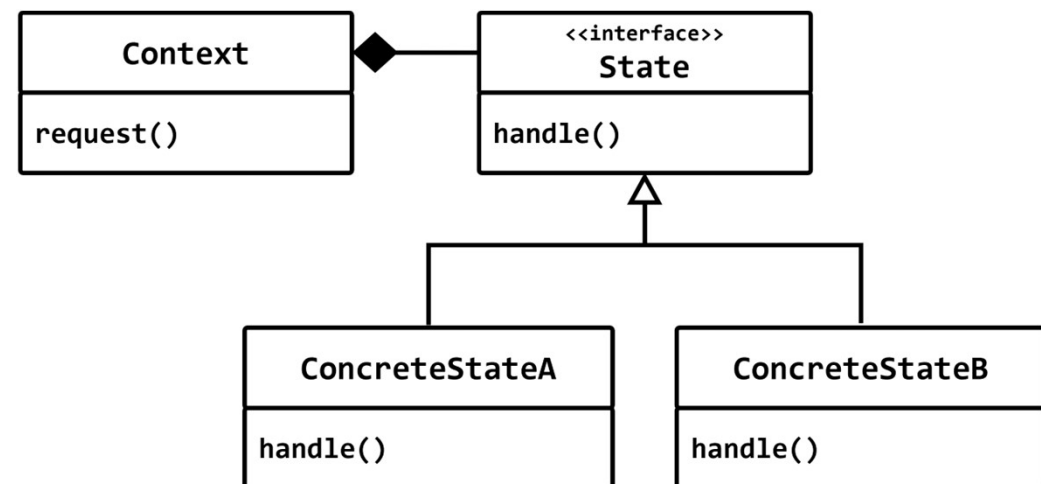
⇒ 객체 사이의 “1:N 의 종속성을 정의” 하고 “한 객체의 상태가 변하면 종속된 다른 객체들에 통보가 가고 자동으로 수정”이 일어 나게 한다





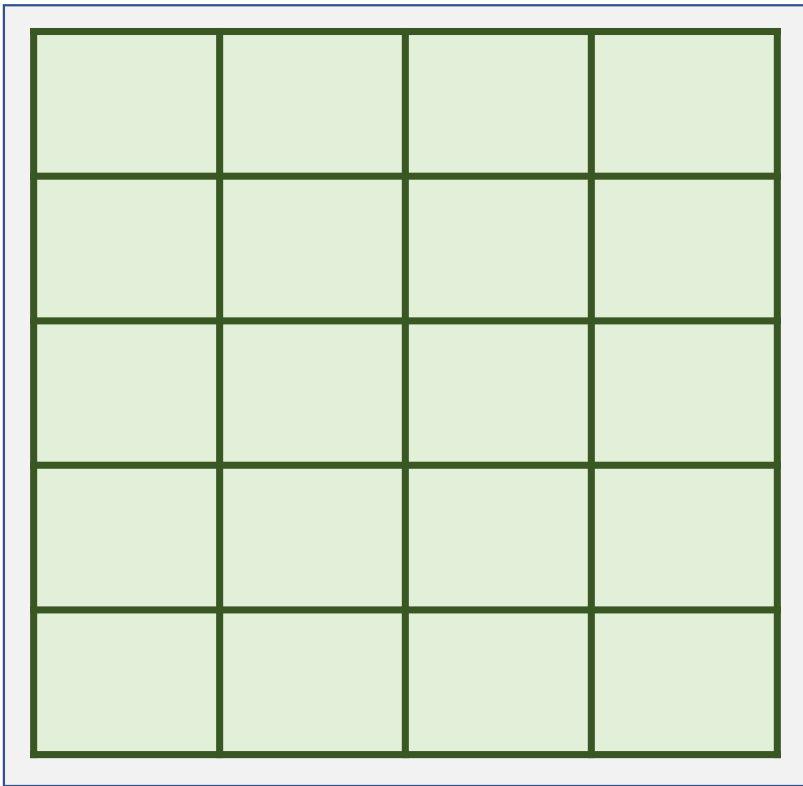
# state

- 행위 패턴 ( Behavior Pattern )
- 의도 ( intent )
  - ⇒ 객체 자신의 “내부 상태에 따라 행위를 변경” 하도록 한다.
  - 객체는 마치 클래스를 바꾸는 것처럼 보인다



# visitor

## APT



방문자 패턴

“요소(각각의 세대)에 대한 연산을 정의”하는 객체를 만들어서

객체(APT)의 모든 요소에 연산을 수행하는 패턴

→ “어떻게 모든 요소를 방문하게 만들 것인가”가 핵심



가스 검침원  
모든 세대를 방문하면서  
가스를 검침

방문자(Visitor)

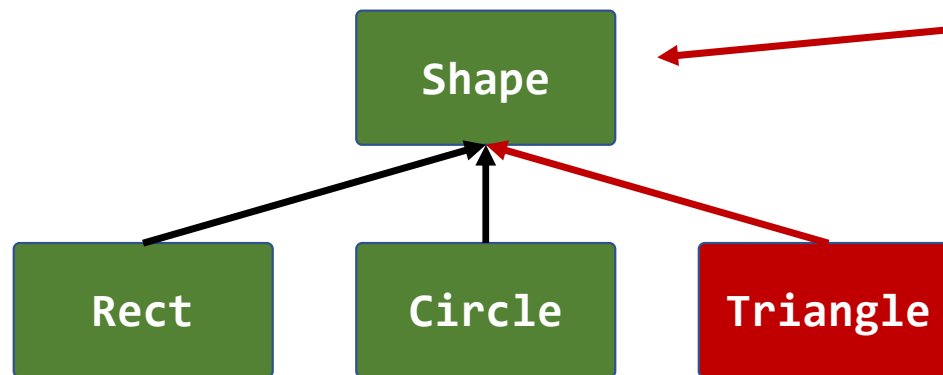
# visitor

- 2개의 예제
- 1번째 예제
  - ⇒ 방문자의 의도를 나타내는 좋은 예제는 아님.
  - ⇒ 하지만 간단하고, 이해 하기 쉬움.
- 2번째 예제
  - ⇒ 약간 복잡함.
  - ⇒ 좋은 예제

## visitor의 의미

- 객체지향 디자인의 특징

- ⇒ 새로운 요소(타입)을 추가 : 쉽다.
- ⇒ 새로운 오퍼레이션(가상함수)의 추가 : 어렵다.



move() 라는 가상함수를 추가하면  
모든 도형을 변경해야 한다.

Triangle 을 추가하는 것은 쉽다  
(다형성을 잘 활용했다면)

## visitor의 의미

- 가상함수를 추가하지 말고 가상함수가 할 일을 방문자

```
ShapeMoveVisitor smv;
shapeContainer.accept(&smv);
```

- 객체지향 디자인의 특징
  - ⇒ 새로운 요소(타입)을 추가
  - ⇒ 새로운 오퍼레이션(가상함수)의 추가

- Visitor 패턴을 사용하면

- ⇒ 새로운 요소(타입)을 추가 : 어렵다
- ⇒ 새로운 오퍼레이션(방문자로 작성)의 추가 : 쉽다

새로운 요소(메뉴)가 추가되면  
방문자 인터페이스가 수정된다.  
모든 방문자 클래스를 수정해야 한다.

```
struct IMenuVisitor
{
    virtual void visit(MenuItem* mi) = 0;
    virtual void visit(PopupMenu* pm) = 0;
    virtual void visit(SpecialMenu* pm) = 0;
    virtual ~IMenuVisitor() {}
};
```

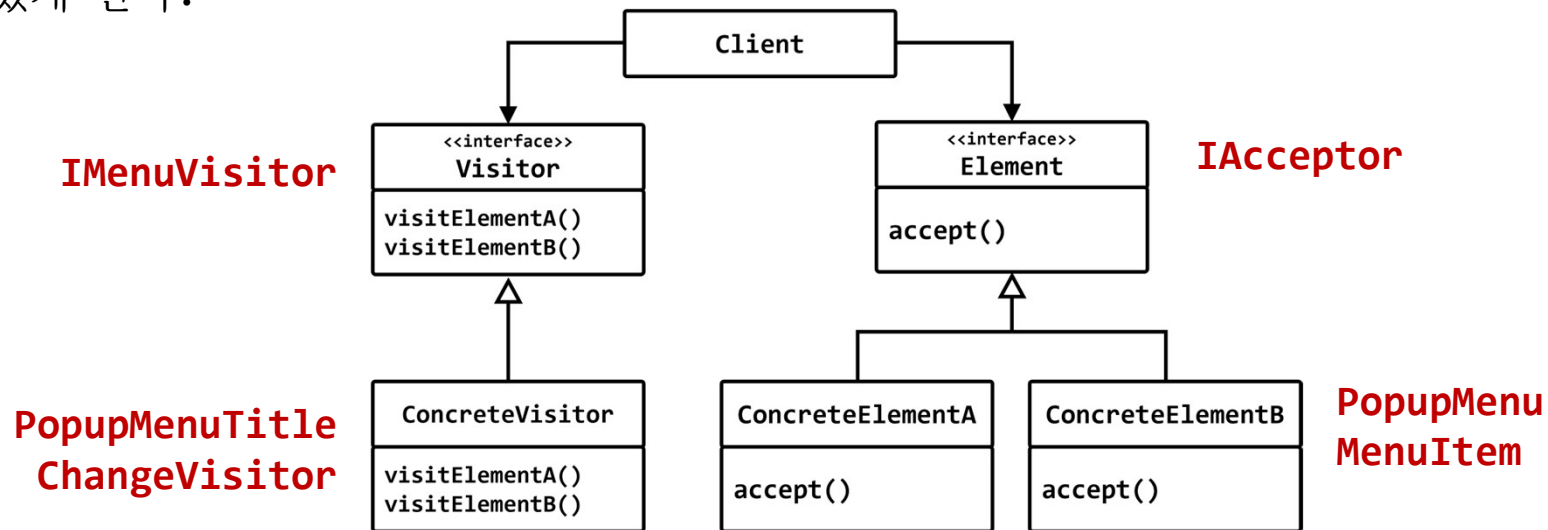
# visitor

- 행위 패턴 ( Behavior Pattern )

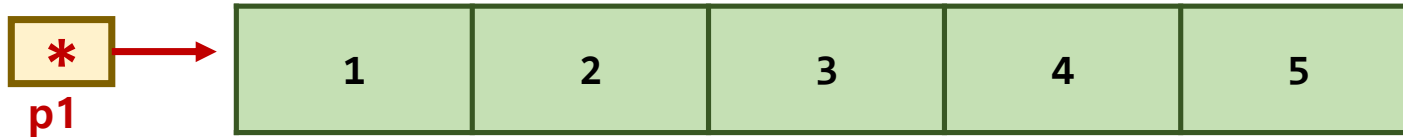
- 의도 ( intent )

⇒ 객체 구조에 속한 **요소에 수행될 오퍼레이션을 정의** 하는 객체.

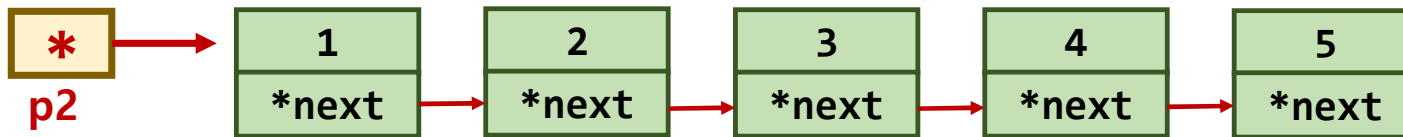
Visitor 패턴은 처리되어야 하는 요소에 **대한 클래스를 변경하지 않고 새로운 오퍼레이션을 정의**할 수 있게 한다.



# iterator(반복자)



배열(또는 vector)  
연속된 메모리 사용



Linked List  
연속되지 않은 메모리

- C 언어를 사용해서 모든 요소를 순차적으로 접근하려면
  - ⇒ 컨테이너의 내부를 노출(버퍼 시작 주소)
  - ⇒ 방법이 다르다.( ++p1, p2 = p2->next )
  - ⇒ 사용자가 컨테이너의 구조를 알고 있어야 한다.

# iterator(반복자)

- C++ 언어(STL) 을 사용하면

⇒ 컨테이너의 내부 구조를 노출하지 않고도, 동일한 방법으로 모든 요소에 순차적으로 접근할 수 있다.

```
std::list<int> s = {1,2,3,4,5};  
std::vector<int> v = {1,2,3,4,5};
```

← 컨테이너의 내부 구조는 다르지만

```
auto p1 = s.begin();  
auto p2 = v.begin();
```

← iterator 라는 객체가 있기 때문에 가능

```
++p1;  
++p2;
```

← 동일한 방법으로 요소에 접근

```
int n1 = *p1;  
int n2 = *p2;
```

← 컨테이너의 내부 구조를 알 필요가 없다

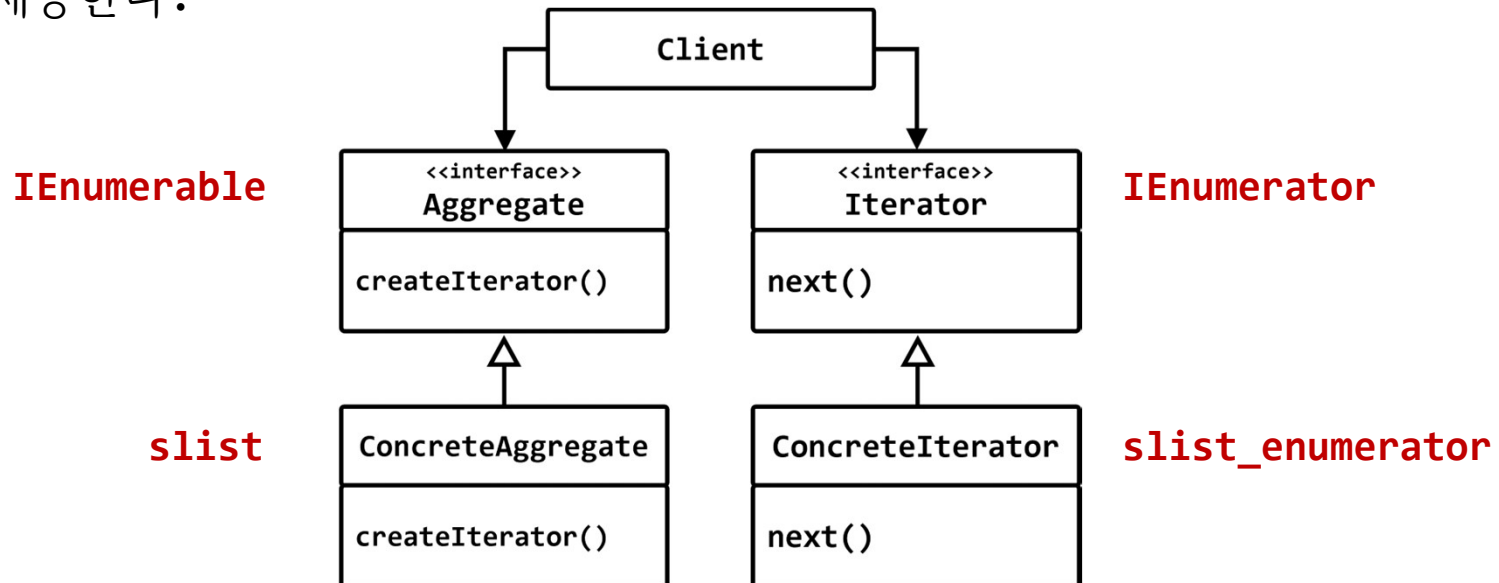


# iterator(반복자)

- 행위 패턴 ( Behavior Pattern )

- 의도 ( intent )

⇒ 복합 객체 요소들의 “내부 표현 방식을 공개하지 않고도 순차적으로 접근” 할 수 있는 방법을 제공한다.



## iterator(반복자)

- 2가지 예제

### ① **interface** 기반의 설계

- ⇒ 전통적인 디자인 패턴에서 설명되는 방식
- ⇒ Java, C#, swift, python 등이 사용하는 방식

### ② **C++ STL** 라이브러리가 사용하는 설계 방식

- ⇒ 성능 향상을 위한 기법.

- linked list 코드 필요

# singleton

- 생성 패턴 ( Creational Pattern )



- 의도 ( intent )

⇒ 클래스의 “인스턴스는 오직 하나임을 보장”하며 어디에서든지 “동일한 방법으로 접근”하는 방법을 제공한다.

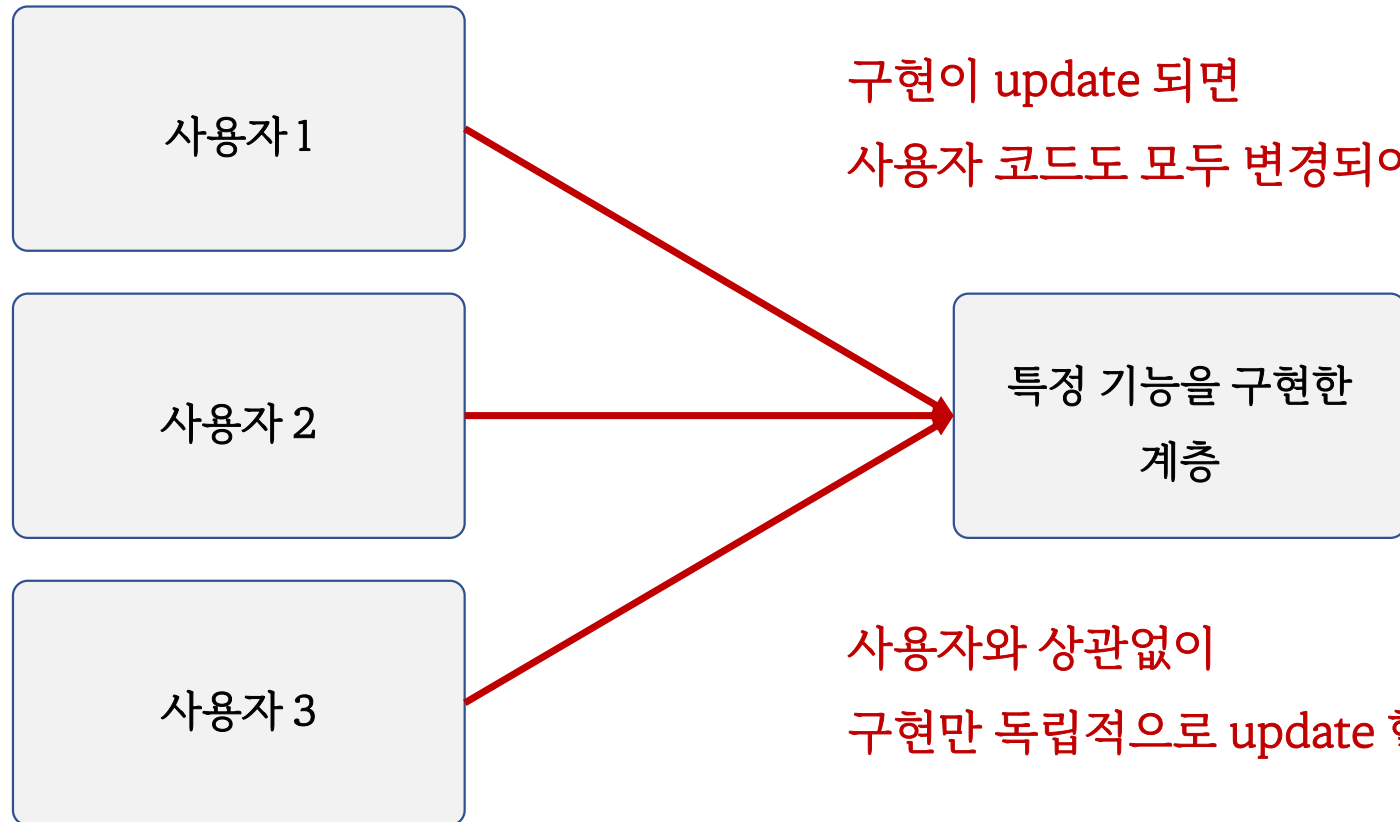
- 단점과 비판

⇒ 결국은 전역변수와 유사.  
⇒ 멀티 스레드간의 접근 문제  
⇒ 객체(함수)간의 결합도 증가. 재사용성 감소

Singleton

getInstance()

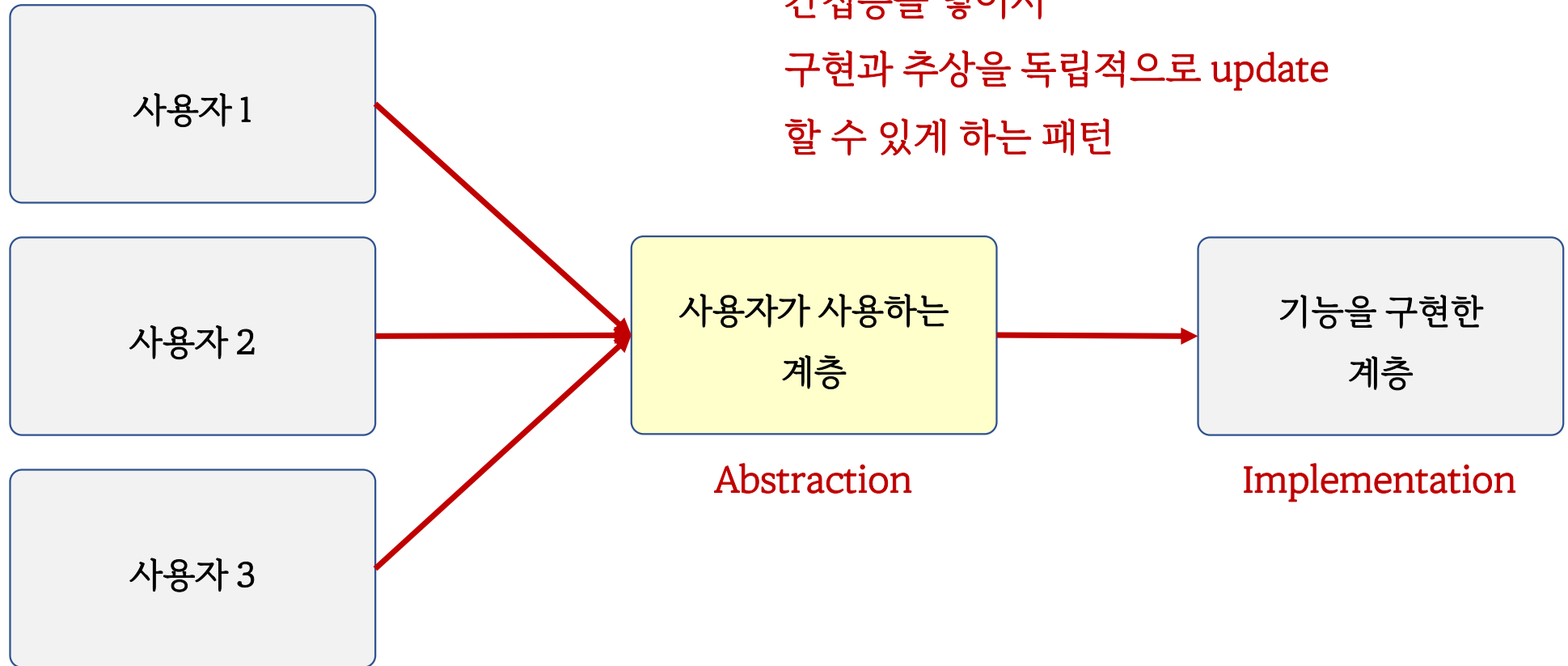
# Bridge



구현이 update 되면  
사용자 코드도 모두 변경되어야 한다.

사용자와 상관없이  
구현만 독립적으로 update 할 수 있을까 ?

# Bridge

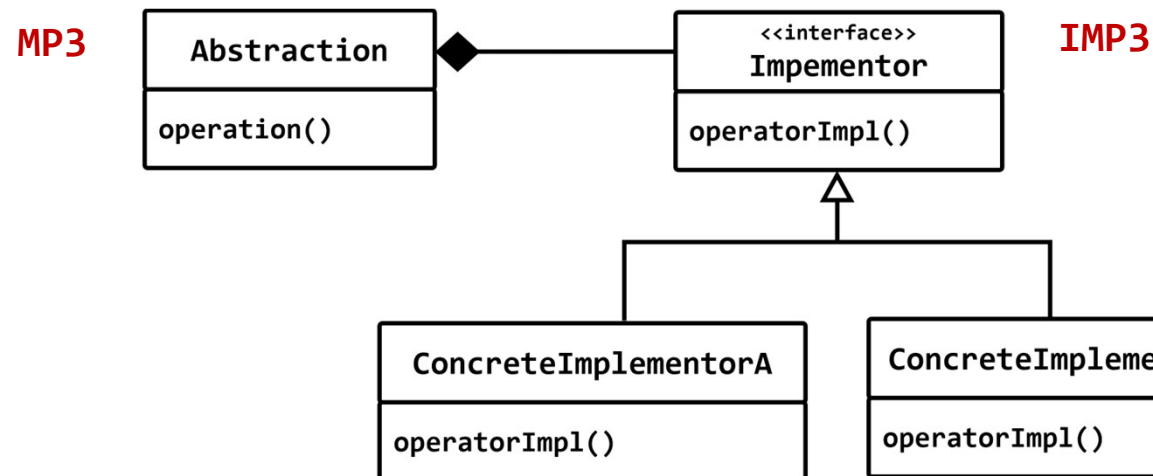


# Bridge

- 행위 패턴 ( Behavior Pattern )

- 의도 ( intent )

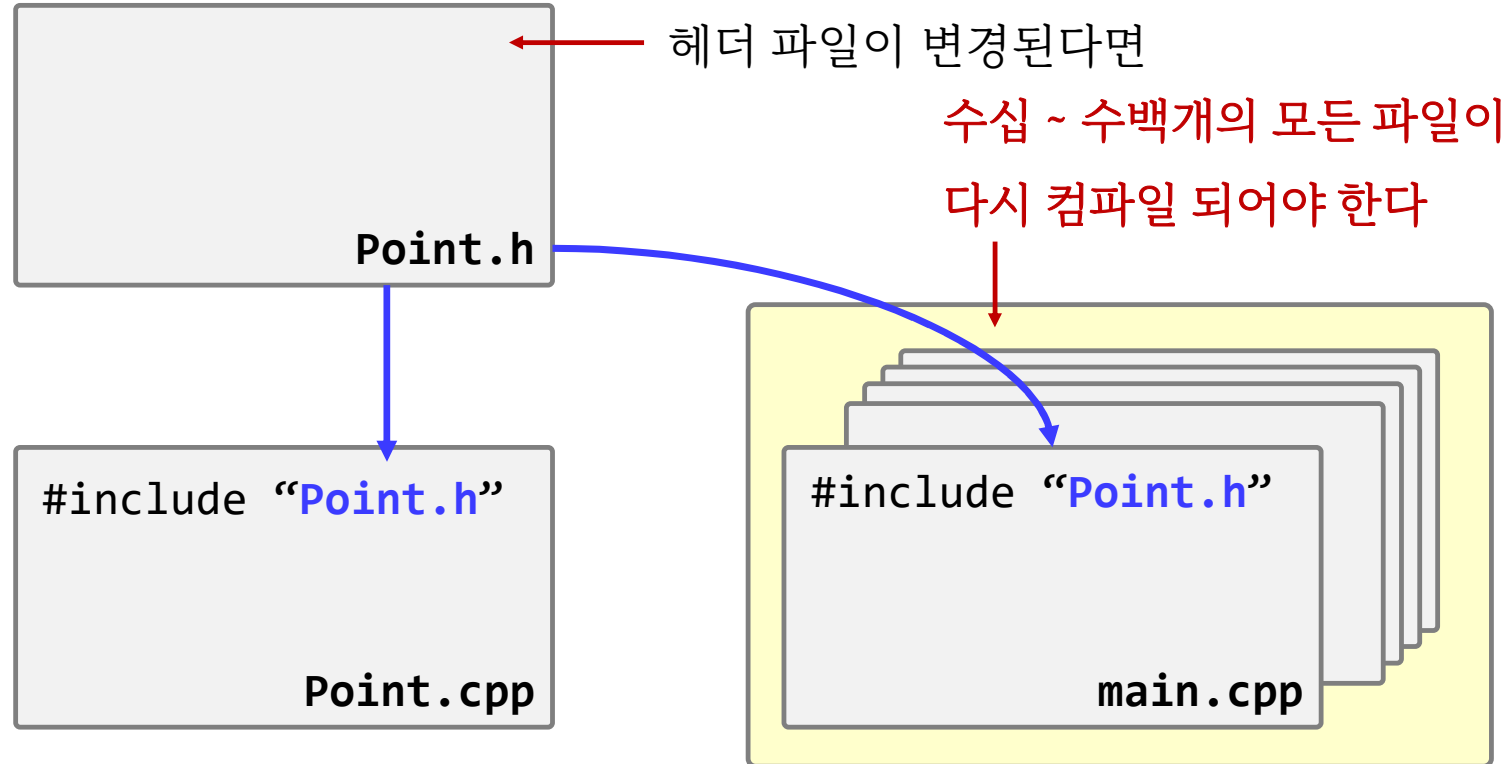
⇒ “구현과 추상화 개념을 분리하여 각각을 독립적으로 변형” 할 수 있게 한다



# PIMPL

- C++ 진영에서 Bridge 패턴을 부르는 또 다른 용어
- Pointer to **IMPL**ementation
- C++ 진영에서 널리 사용되는 예제

# PIMPL



상호 독립적인 update 가 되지 않는 경우



flyweight

ABCDEFG**G**HI←

msword

각각의 글자의 다양한 속성을 변경할 수 있다.

폰트 관련 속성은  
공유 되는 경우가 많다.

```
class Char
{
    char value;
    int font_size;
    int font_weight;
    COLOR font_color;
    std::string font_family;
    // 폰트와 관련된 다양한 속성들...
}
```

# flyweight

ABCDEF**G**HIL←

msword

Flyweight 패턴 적용

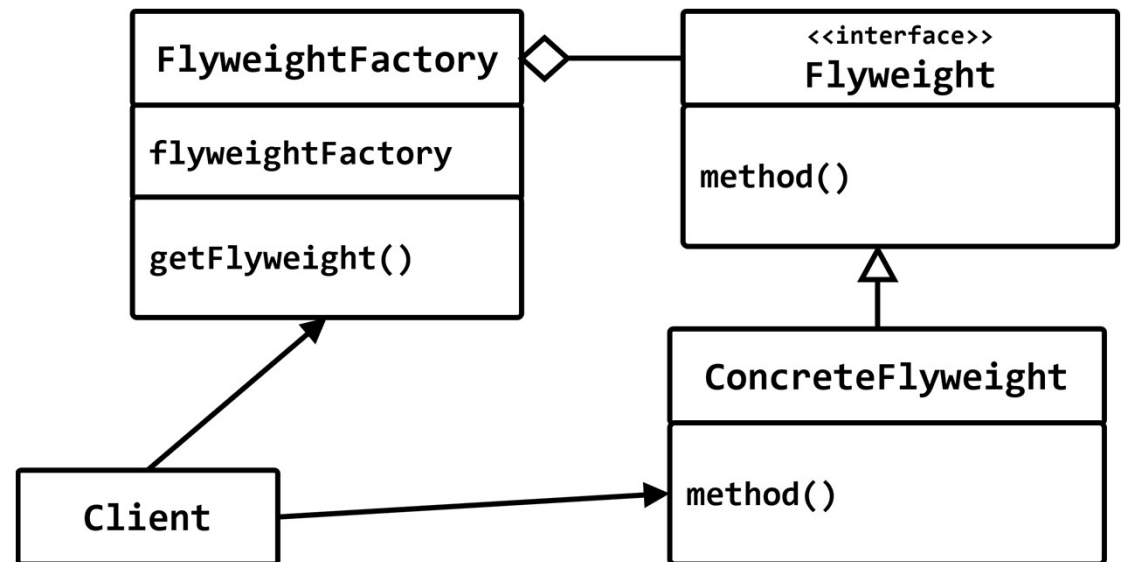
```
class Char
{
    char value;
    Font* font;
};

class Font
{
    int font_size;
    int font_weight;
    COLOR font_color;
    std::string font_family;
    // 이외에 폰트와 관련된 다양한 속성들...
};

class FontFactory
{
};
```

# flyweight

- 구조 패턴 ( Structural Pattern )
- 의도 ( intent )
  - ⇒ 속성이 동일한 “객체를 공유” 하게 한다.



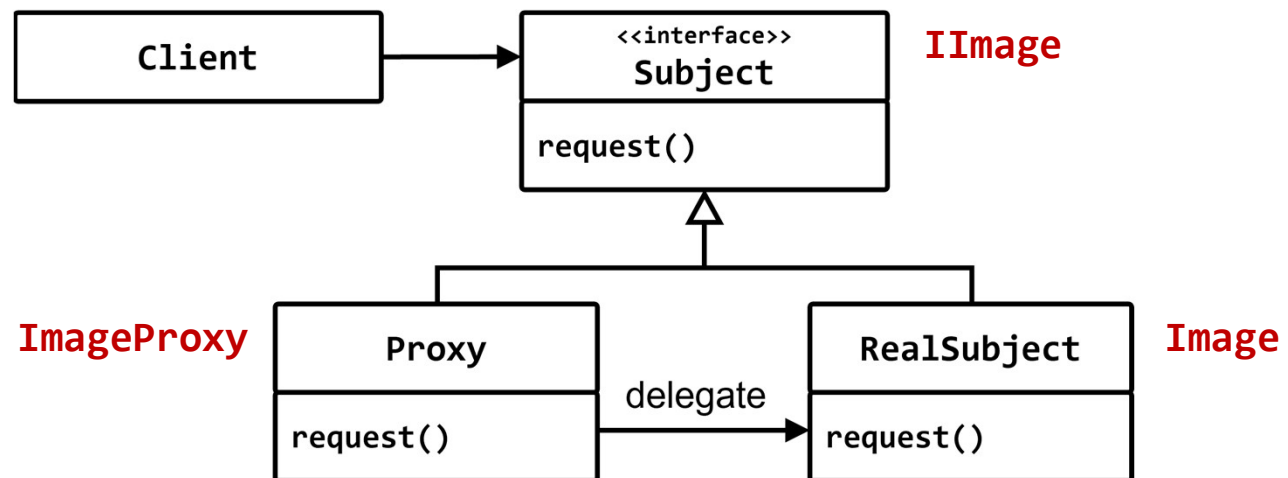
# proxy

- 구조 패턴 ( Structural Pattern )

- 의도 ( intent )

⇒ 다른 객체에 접근하기 위해 “중간 대리 역할을 하는 객체”를 둔다.

⇒ 다양한 문제를 해결하기 위한 간접층



# factory

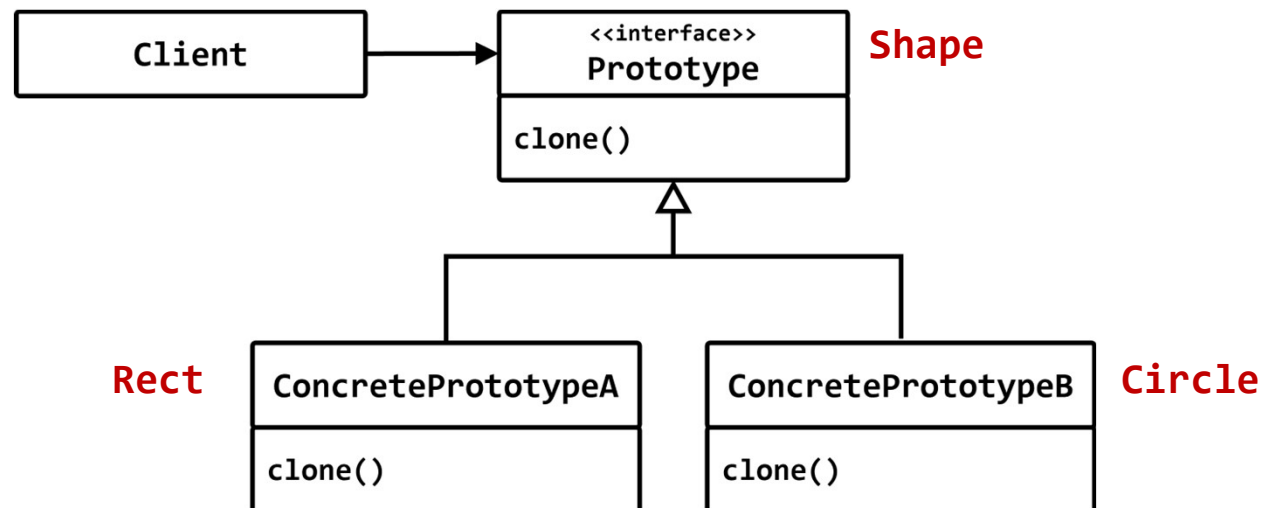
- 전통적인 디자인 패턴의 분류에는 “**factory**” 라는 패턴은 없고,  
“**abstract factory**” 가 존재
  - ⇒ 하지만, 관례적으로 “**factory**” 라고 불리는 기술이 많이 사용되고,
  - ⇒ “**abstract factory**” 를 이해 하려면, 먼저 “**factory 자체를 이해**”.

# prototype

- 생성 패턴 ( Creational Pattern )

- 의도 ( intent )

⇒ 견본적(prototypical) 인스턴스를 사용하여 생성할 객체의 종류를 명시하고 이렇게 만들어진 “**견본을 복사하여 새로운 객체를 생성**”한다



## 객체를 생성하는 방법

**Rect r;**

객체의 수명이 정해져 있다.  
사용자가 원할때 파괴 할수 없다.

**r = new Rect;**

가장 자유로운 방법. 자유롭게 생성하고 자유롭게 파괴

**r = Rect::create();**

**객체의 생성을 한 곳에서. 다양한 제약을 사용할 수 있다.**  
오직 한 개 만 만들 수 있게 ➔ **singleton**  
속성이 동일하면 공유 ➔ **flyweight**  
생성함수 주소를 자료구조에 보관도 가능.

**r = factory.create();**

공장을 통한 객체의 생성

**r = sample.clone()**

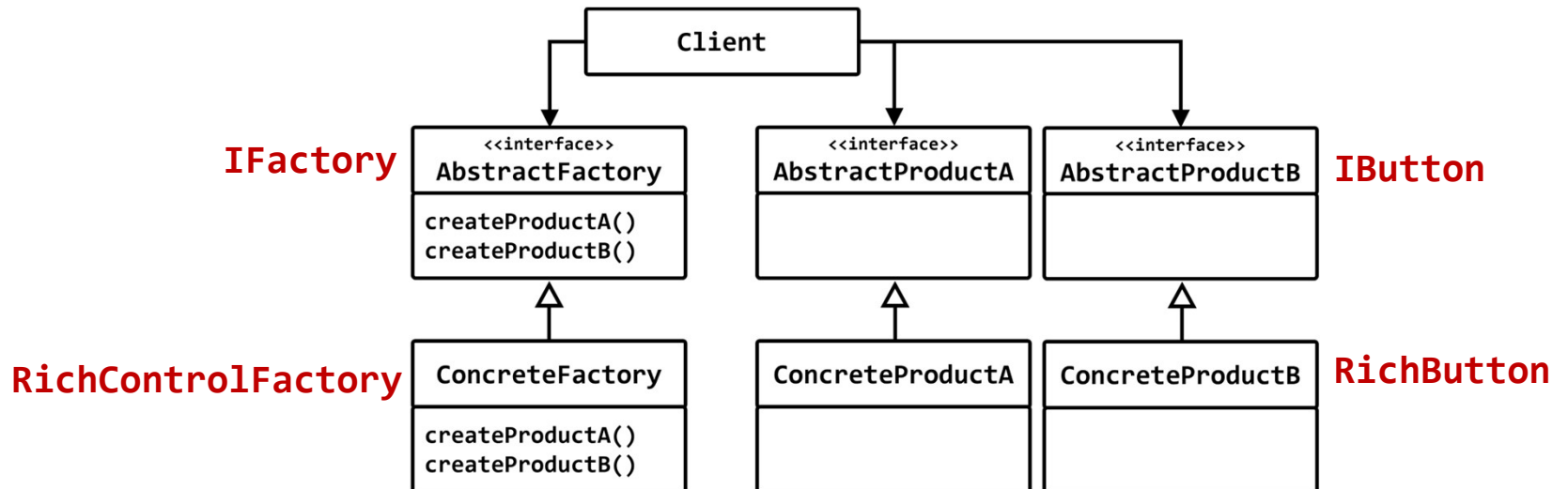
기존 객체에 복사를 통한 새로운 객체 생성 ➔ **prototype**

# abstract factory

- 생성 패턴 ( Creational Pattern )

- 의도 ( intent )

⇒ 상세화된 서브 클래스를 정의 하지 않고도 서로 관련성이 있거나 독립적인 “여러 객체의 군을 생성하기 위한 인터페이스를 제공” 한다.





# factory method

- 생성 패턴 ( Creational Pattern )

- 의도 ( intent )

⇒ 상세화된 서브 클래스를 정의 하지 않고도 서로 관련성이 있거나 독립적인 “여러 객체의 군을 생성하기 위한 인터페이스를 제공” 한다.

