목 차

Section 1.

Warmingup

생성자 호출의 원리

배우게 되는 내용

① 상속에서의 생성자/소멸자 호출 순서

② 생성자를 protected로 만드는 이유.

1. 상속관계에서의 생성자/소멸자 호출의 원리

생성자/소멸자의 호출원리

클래스가 기반 클래스를 가지고 있다면 컴파일러는 기반 클래스의 생성자를 호출하는 코드를 자동 생성해 주게 됩니다. 따라서, 파생 클래스의 객체를 생성하면 기반 클래스의 디폴트 생성자가 먼저 수행되고, 자신의 생성자를 호출하게 됩니다.

사용자가 만든 코드	컴파일러가 생성하는 코드

`class Animal`
`{`
`public:`
`  Animal()  {}`
`  Animal(int a) {}`
`  ~Animal()  {}`
`};`
`class Dog : public Animal`
`{`
`public:`
`  Dog()  { }`
`  Dog(int a) { }`
`  ~Dog()  { }`
`};`
`int main()`
`{`
`  Dog d1;`
`  Dog d2(1);`
`}`
	`class Animal`
	`{`
	`public:`
	`  Animal()  {}`
	`  Animal(int a) {}`
	`  ~Animal()  {}`
	`};`
	`class Dog : public Animal`
	`{`
	`public:`
	`  Dog()  : Animal() { }`
	`  Dog(int a) : Animal() { }`
	`  ~Dog()  { Animal::~Animal(); }`
	`};`
	`int main()`
	`{`
	`  Dog d1;`
	`  Dog d2(1);`
	`}`

따라서, main 함수에서 d1처럼 인자가 없건, d2 처럼 인자를 하나가 있건, Animal의 default 생성자가 호출되게 됩니다.

기반 클래스의 생성자를 명시적으로 호출하는 방법.

기반 클래스의 생성자를 디폴트 생성자가 아닌 다른 생성자가 호출 되게 하려면 파생 클래스의 초기화 리스트 목록에서 기반 클래스의 생성자를 명시적으로 호출해 주어야 합니다.

```
class Animal
{
public:
    Animal(int a)
    {
    }
};
class Dog : public Animal
{
public:
    // 파생 클래스에서 기반 클래스의 생성자를 명시적으로 호출하는 코드
    Dog() : Animal(0) {}
    {
    }
    Dog(int a) : Animal(a) {}
    {
    }
};
int main()
{
    Dog d1;
    Dog d2(1);
}
```

protected constructor

다양한 오픈 소스에 보면 최상위 클래스의 생성자가 protected에 있는 것을 볼 수 있습니다. 안드로이드 Framework 의 최상의 기반 클래스인 RefBase, MFC의 최상위 기반 클래스인 CObject 등의 생성자가 protected 에 있습니다.

이와 같은 protected 생성자의 의미는 다음과 같습니다.

"자신의 객체는 생성할 수 없지만(추상적인 존재)

파생 클래스의 객체(구체적인 존재)는 생성 할 수 있도록 한다."

는 의도가 있습니다.

아래 코드를 참고 해 보세요.

```
class Animal
{
protected:
    Animal() {} // protected constructor
};
class Dog : public Animal
{
};
int main()
{
    Animal a; // error. protected 멤버를 외부에서 호출 할 수 없습니다.

    Dog d;  // ok. Dog 이 생성자가 먼저 호출되고,
            // Dog 생성자안에서 Animal의 생성자가 호출됩니다.
            // 파생 클래스에서는 기반 클래스의 생성자를 호출할 수 있습니다.
}
```

현실세계에서는 "동물"은 추상적인(abstract) 개념이므로 객체가 존재하지 않지만, Dog 는 추상적이지 않은 구체적(concrete)존재 이므로 객체가 존재 합니다. 따라서, 현실 세계의 추상적 개념을 모델링 할 때 protected 생성자를 사용하는 경우가 가끔 있습니다.

물론, Animal 을 설계할 때 C++ 추상 클래스를 사용하는 것이 좋지만 protected 생성자를 사용하는 경우도 종종 볼 수 있습니다.

upcasting

배우게 되는 내용

① upcasting

② 동종을 처리하는 함수

③ 동종을 저장하는 컨테이너

1. upcasting

핵심 개념

C++, java, C#등의 대부분의 객체지향 언어에서는 기반 클래스의 포인터(참조)로 파생 클래스의 객체를 가리킬 수 있습니다.

```
class Animal { };
class Dog : public Animal { };

int main()
{
    Animal* p = new Dog;
}
```

활용 및 장점

Upcasting을 사용하면 동종(동일 기반 클래스를 가지는 클래스들)을 처리하는 함수나 동종을 저장하는 컨테이너를 만들 수 있습니다.

```
class Animal {};
class Dog : public Animal {};
class Cat : public Animal {};

// 1. 동종을 처리하는 함수
void TakeCareOf(Animal* p) // 모든 Animal의 파생 클래스를 인자로 전달 받을수 있습니다.
{
}
int main()
{
    Dog d;
    Cat c;
    TakeCareOf(&d);
    TakeCareOf(&c);
    // 2. 동종을 저장하는 클래스
    vector<Animal*> house; // 모든 Animal의 파생 클래스를 담을 수 있습니다.
}
```

주의 할 점

기반 클래스의 포인터로 파생 클래스를 가리킬 수는 있지만 이때 기반 클래스의 포인터로 파생클래스의 고유한 멤버에 접근할 수는 없습니다. 접근하기 위해서는 명시적 캐스팅을 해야 합니다.

```
class Animal
{
public:
    virtual ~Animal() {}
};
class Dog : public Animal
{
public:
    void wagTail() {}
};
int main()
{
    Animal* p = new Dog;

    p->wagTail(); // error

    // p가 Dog 가 확실한 경우. static_cast
    static_cast<Dog*>(p)->wagTail(); // ok

    // p가 Dog 여부가 명확하지 않을 때는 dynamic_cast를 사용합니다.
    Dog* pDog = dynamic_cast<Dog*>(p);

    if (pDog != 0)
        pDog->wagTail();
}
```

interface 와 coupling

배우게 되는 내용

① 추상 클래스

② interface

③ tightly coupling vs loosely coupling

1. abstract class

▌ 순수 가상함수와 추상 클래스

C++ 에서 가상함수 중에서 구현이 없고 “=0”으로 끝나는 가상함수를 순수 가상 함수 라고 합니다. 또한, 순수 가상함수를 한 개 이상 가지고 있는 클래스를 추상 클래스라고 합니다.

추상 클래스는 다음과 같은 특징이 있습니다.

① 추상 클래스는 객체를 생성할 수 없지만, 포인터변수는 만들 수 있습니다.

② 추상 클래스로부터 파생된 클래스가 순수 가상함수의 구현부를 제공하지 않으면 파생클래스 역시 추상 클래스 입니다.

```
class Animal
{
public:
    virtual void Cry() = 0;
    virtual ~Animal() {}
};
class Dog : public Animal
{
public:
};
int main()
{
    Animal a;   // error
    Dog    d;   // error. 기반 클래스가 가진 Cry() 순수 가상 함수의 구현을 제공하지
                // 않으면 Dog도 역시 추상클래스 입니다.
    Animal* p = 0; // ok
}
```

▌ 추상 클래스의 의도와 인터페이스 (interface)

추상 클래스의 의도는 파생 클래스에게 특정 함수를 만들라고 지시하는 용도로 사용됩니다. 이전의 코드에서 Animal 클래스의 의도는 결국 모든 동물 클래스는 반드시 Cry() 함수를 만들라는 지시 입니다.

특히, 추상 클래스 중에 모든 멤버함수가 순수 가상함수 인 것을 흔히 “interface(인터페이스)”라고 하는데, 이 경우는 파생 클래스가 가져야 하는 규칙을 제공하기 위해 만든 것입니다.

▌ 강한 결합 (tightly coupling)

하나의 클래스가 다른 클래스를 사용할 때 클래스 이름을 직접 사용하는 것을 “강한 결합(tightly coupling)” 이라고 하는데, 강한 결합은 유연성이 부족하고 확장이 불가능한 경직된 구조를 가지게 됩니다.

아래 코드를 생각해 봅시다.

```
class Camera
{
public:
    void startRecording() { cout << "Camera startRecording" << endl; }
    void stopRecording()  { cout << "Camera stopRecording" << endl; }
};
class People
{
public:
    void useCamera(Camera* p)   // tightly coupling
    {
        p->startRecording();
        p->stopRecording();
    }
};

int main()
{
    People p;
    Camera c1;
    p.useCamera(&c1);
}
```

위 코드는 People 클래스가 Camera 클래스를 사용하고 있는데, useCamera() 함수에서 Camera 클래스의 이름을 직접 사용하고 있습니다. 이 경우 문제점은, HDCamera 라는 새로운 종류의 카

메라 클래스가 추가될 경우 People 클래스에서 HDCamera를 사용할수 있게 하려면 People 클래스 자체의 코드가 수정(새로운 멤버 함수 추가)이 되어야 합니다.

하지만, Camera의 인터페이스를 만들어 사용하면 People 클래스의 코드 수정없이 다양한 종류의 Camera 클래스를 추가 할 수 있게 됩니다.

▌ 인터페이스(interface)와 약한 결합 (loosely coupling)

Camera 클래스를 만들 때 모든 카메라가 지켜야 하는 규칙을 추상 클래스(인터페이스)로 먼저 설계하고, People 이 인터페이스를 통해서 Camera를 사용하면 확장성 있는 설계를 할 수 있습니다.

아래 코드를 생각해 봅시다.

```
// 모든 카메라가 지켜야 하는 규칙을 먼저 설계합니다.
// 규칙 : "모든 카메라는 ICamera 인터페이스를 구현해야 한다." 입니다.
struct ICamera
{
    virtual void startRecording() = 0;
    virtual void stopRecording() = 0;
    virtual ~ICamera() {}
};
// People은 카메라 인터페이스인 ICamera를 통해서 Camera를 사용합니다.
class People
{
public:
    void useCamera(ICamera* p)  // loosely coupling
    {
        p->startRecording();
        p->stopRecording();
    }
};
// 모든 카메라는 ICamera 인터페이스를 구현해야 합니다.
class Camera : public ICamera
{
public:
    void startRecording() { cout << "Camera startRecording" << endl; }
    void stopRecording() { cout << "Camera stopRecording" << endl; }
};
class HDCamera : public ICamera
{
public:
```

```
    void startRecording() { cout << "HDCamera startRecording" << endl; }
    void stopRecording() { cout << "HDCamera stopRecording" << endl; }
};
int main()
{
    People p;
    Camera c1;
    p.useCamera(&c1);

    HDCamera c2;
    p.useCamera(&c2);
}
```

위 코드에서 People 클래스는 useCamera() 함수에서 Camera 클래스 이름을 바로 사용하지 않고 인터페이스(추상 클래스) 이름인 ICamera 를 사용하고 있습니다. 이처럼, 다른 클래스를 사용할 때 인터페이스를 통해서 접근하는 것을 약한 결합(loosely coupling) 이라고 합니다.

장점은, 어떠한 새로운 카메라도 ICamera 인터페이스를 구현 한다면(ICamera 로부터 파생 된다면) People에서 사용할 수 있습니다. 이처럼, 기존 코드의 수정없이 기능을 확장할 수 있게 하는 것을 흔히 "OCP" 라고 합니다.

SOLID 원칙과 OCP

"SOLID 원칙" 이란 객체 지향 설계시에 자주 언급되는 "SRP", "OCP", "LSP", "ISP", "DIP" 등의 5가지 원칙의 약자입니다.

이중 "OCP"는 "기능 확장에는 열려 있고(Open), 코드 수정에는 닫혀 있어야(Close)한다는 원칙(Principle)" 으로, 흔히 "개방폐쇄의 법칙"으로 부르기도 합니다.

기존 클래스의 코드 수정없이 기능 확장(새로운 요소의 추가)를 할 수 있어야 한다는 원칙입니다.

객체지향 디자인 맛보기

배우게 되는 내용

① 도형편집기로 배우는 객체지향 디자인

1. 도형 편집기 예제

파워 포인트 같은 프로그램을 보면 다양한 도형을 편집 할 수 있습니다. 이와 같은 프로그램을 만들어 가면서 객체지향 프로그래밍의 원리를 살펴 보겠습니다. 객체 지향 디자인을 위한 다양한 개념이 나오는데, 각 개념에 대한 자세한 내용은 이후에 이어지는 장에서 자세히 다루게 되므로 이번 항목에서는 개념 정도만 다루도록 하겠습니다.

Step1. 각 도형을 타입화 한다.

도형 편집기 같은 프로그램서는 “사각형, 원, 삼각형”등 의 다양한 도형을 다루게 됩니다. 사각형을 표현하기 위해 int 타입 4개를 사용하는 것 보다는 Rect 라는 타입이 있으면 편리하므로 각 도형을 타입으로 만들어 보겠습니다.

```
#include <iostream>
#include <vector>
using namespace std;

class Rect
{
public:
    void Draw() { cout << "Rect Draw" << endl; }
};
class Circle
{
public:
    void Draw() { cout << "Circle Draw" << endl; }
};
int main()
{
    vector<Rect*>   v1; // Rect 만   저장할 수 있습니다.
    vector<Circle*> v2; // Circle 만 저장할 수 있습니다.
}
```

위 코드는 main 함수에서 만들어진 도형을 보관하기 위해 vector를 사용하는데, 문제는 vector<Rect*> 는 Rect만 보관할수 있고, vector<Circle*>은 Circle 만 보관 할 수 있다는 점입니다. 도형을 따로 보관 할 경우, 어떤 도형을 먼저 만들었는지 등을 관리하기가 어렵습니다. 모든 종류의 도형을 하나의 컨테이너에 보관 할 수 있다면 좋지 않을 까요 ?

Step2. 기반 클래스 "Shape" 도입

각 도형의 동일한 기반 클래스를 도입하면 하나의 컨테이너에 모든 도형을 보관할 수 있습니다. 이전의 코드에 Shape 클래스를 도입하고, Rect 와 Circle 클래스의 기반 클래스로 사용합니다. 그리고, main 함수에서 vector<Shape*> 를 보관하면 모든 도형을 보관 할 수 있게 됩니다.

```
#include <iostream>
#include <vector>
using namespace std;

class Shape
{
};
class Rect   : public Shape { // …… };
class Circle : public Shape { // …… };
int main()
{
    vector<Shape*>   v;
    // v에는 종류에 상관없이 모든 도형을 보관할 수 있습니다.
    v.push_back(new Rect);
    v.push_back(new Circle);
}
```

보통 상속의 장점은

"기존 클래스의 멤버를 물려받아 새로운 특징을 추가할 수 있다는 재사용성"

입니다.

또한, 위처럼

"연관된 클래스들을 묶어서 관리할 수 있다"

는 특징도 있습니다.

이제 사용자에게 입력을 받아서 다양한 도형을 만들수 있도록 코드를 추가해 보겠습니다. 사용자가 1을 입력 하면 Rect을 만들고, 2를 입력하면 Circle을 만들고, 9을 입력하면 지금까지 만든 모든 도형의 Draw함수를 호출하도록 만들겠습니다. main 함수를 다음 처럼 변경합니다.

```
int main()
{
    vector<Shape*>   v;

    int cmd;
```

```
    while (1)
    {
        cin >> cmd;
        if      (cmd == 1) v.push_back(new Rect);
        else if (cmd == 2) v.push_back(new Circle);
        else if (cmd == 9)
        {
            for (auto p : v)
                p->Draw();    // compile time error
        }
    }
}
```

어렵지 않은 코드인데, 문제는 아래 코드에서 compile error가 발생합니다.

```
    for (auto p : v)
        p->Draw();    // compile time error
```

왜 에러 일까요 ?

▌ Step3. 파생 클래스의 공통의 특징은 기반 클래스에 있어야 한다.

기반 클래스의 포인터로 파생 클래스를 주소를 담을 수 있지만 파생클래스의 고유한 멤버에 접근할 수는 없습니다.

```
class Shape { };
class Rect : public Shape
{
public:
    void Draw() {  }
};
int main()
{
    Shape* p = new Rect;
    p->Draw();  // error. 기반 클래스가 파생클래스를 가리킬 때
                // 파생 클래스의 고유한 멤버에 접근할 수는 없습니다.
}
```

이전의 코드에서는 사용자의 입력에 따라 Rect또는 Circle 을 만들지만 모든 도형을 같은 컨테이너에 보관하기 위해 vector안에는 Shape*를 보관했습니다. 따라서, vector에서 값을 하나 꺼냈을 때 얻게 되는 포인터는 Shape* 입니다. 그런데 Shape* 안에는 Draw 함수가 없으므로 에러가 나오게 됩니다.

```
int main()
{
    vector<Shape*>   v;

    // ……
    for (auto p : v)
        p->Draw();  // p는 실제로는 Rect 혹은 Circle을 가리키지만
                    // p의 타입은 Shape*입니다
}
```

해결책은 다음의 2가지중에 하나입니다.

① p를 Rect 또는 Circle 로 캐스팅해서 사용하는 방법

② Shape 안에 Draw를 만드는 방법.

그런데, 어떤 도형을 만들었는지는 실행시간에 사용자 입력에 따라 달라지므로 코드를 작성하는 시점에서는 Rect인지 Circle 인지 알 수 없습니다. 결국 해결책은 Shape 클래스 안에도 Draw 함수를 제공해야 합니다.

여기서 핵심은

"모든 도형의 공통적인 특징은 반드시 기반 클래스인 Shape에도 있어야 합니다."

그래야만, 기반 클래스의 Shape* 로 도형을 관리할 때 해당 특징을 사용할 수 있게 됩니다.

[참고] 실제로, Shape* 타입의 p가 실제로 어떤 도형을 가리 키는 지를 조사할 수 있습니다. RTTI 기술을 사용하면 가능합니다. 책의 뒤장에서 다루게 됩니다.

Step4. 파생 클래스가 override 하게 되는 함수는 반드시 가상함수로 만들어야 한다

결국 모든 도형(Rect, Circle)에 Draw 함수가 있다면 Shape 함수에도 반드시 Draw 함수를 제공해야 합니다. 즉, Shape 클래스를 설계 할 때부터 모든 도형의 공통의 특징이 무엇인가를 생각해서 Shape 안에 멤버로 제공해야 합니다.

```
#include <iostream>
#include <vector>
using namespace std;

class Shape
{
public:
    void Draw() { cout << "Shape Draw" << endl; }
};
class Rect : public Shape
{
public:
    void Draw() { cout << "Rect Draw" << endl; }
};
int main()
{
    Shape* p = new Rect;
    p->Draw();  // ok. Shape 안에도 Draw 함수가 있으므로 error가 발생하지 않습니다.
                // 하지만, Rect의 Draw가 아닌 Shape의 Draw가 호출됩니다.
}
```

이제는, Shape*타입의 포인터 p를 사용해서 Draw를 호출 할 수 있습니다. 그런데, 마지막 문제는 실제 객체는 Rect이지만 포인터 p가 Shape* 타입이므로 Shape의 Draw를 호출하게 됩니다. 해결책은 Draw 함수를 가상함수로 만들어야 합니다.

일반적으로 기반 클래스를 설계할 때

"파생 클래스에서 override 하게 되는 멤버함수는 반드시 가상함수로 만들어야 한다."

는 규칙이 있습니다.

결국, 완성된 코드는 다음과 같습니다.

```
#include <iostream>
#include <vector>
using namespace std;
```

```
class Shape
{
public:
    virtual void Draw() { cout << "Shape Draw" << endl; }
    virtual ~Shape() {}
};
class Rect : public Shape
{
public:
    virtual void Draw() { cout << "Rect Draw" << endl; }
};
class Circle : public Shape
{
public:
    virtual void Draw() { cout << "Circle Draw" << endl; }
};

int main()
{
    vector<Shape*>   v;

    int cmd;
    while (1)
    {
        cin >> cmd;
        if      (cmd == 1) v.push_back(new Rect);
        else if (cmd == 2) v.push_back(new Circle);
        else if (cmd == 9)
        {
            for (auto p : v)
                p->Draw();
        }
    }
}
```

위 코드에서 중요한 점은 다음의 코드입니다.

```
for (auto p : v)
    p->Draw();  // 이 순간 p가 어떤 객체를 가리키는 지에 따라 다른 함수가 호출됩니다.
```

"p->Draw()" 와 같이 동일한 함수를 호출하는 것 같은 표현이 상황(p가 가리키는 객체가 실제로 어떤 타입인가 ?)에 따라 다른 함수를 호출하게 되는 것을 **"다형성(Polymorphism)"** 이라고 합니다.

어떤 장점이 있을까요 ? 앞장에서 만든 도형 편집기는 두개의 도형(Rect, Circle)만을 다룰 수 있습니다. 그런데, 시간이 지나서 나중에 새로운 도형인 "Triangle"을 추가 했다고 생각해 봅시다.

```
class Triangle : public Shape
{
public:
    virtual void Draw() { cout << "Triangle Draw" << endl; }
};
```

이제 새롭게 추가된 Triangle 때문에 아래 한줄이 수정될 필요가 있을까요 ?

```
for (auto p : v)
    p->Draw();  // Triangle 클래스가 추가 되어도 이 한 줄은 수정될 필요가 없습니다.
```

즉, 미래에 어떤 도형이 추가되어도

"Shape 클래스로부터 파생되고, Draw()가상 함수를 재정의 했다면"

기존에 있는 "p->Draw()" 코드는 수정될 필요가 없습니다. 아주 좋은 장점 입니다.

이런 특징은 "OCP" 라고 합니다.

객체지향 S/W 설계의 가장 중요한 원칙 중의 하나는

"기능 확장에 열려있고(Open, 미래에 새로운 코드가 추가되어도)"

"코드 수정에 닫혀있어야(Close, 기존 코드는 수정되면 안된다)는 원칙(Princple) 입니다."

대부분의 객체지향 디자인은 "OCP" 라는 원칙을 지키기 위한 코딩 기법입니다. OCP외에도 객체지향 설계원칙은 총 5가지가 있습니다. 흔히, "SOLID"라고 이야기 합니다.

"SOLID(SRP, OCP, LSP, ISP, DIP)"

SOLID에 대한 자세한 설명은 뒷장에서 자세히 다루게 됩니다.

Step5. 공통성과 가변성의 분리

이번에는 Draw() 함수에 동기화 관련 코드를 추가한다고 생각해 봅시다.

```
class Rect
{
public:
    void Draw()
    {
        mutex.lock();
        cout << "Rect Draw" << endl;
        mutex.unlock();
    }
};
```

그런데, 문제는 Draw() 함수에 동기화가 필요 하다면, Rect 뿐 아니라 Circle, Triangle 에도 있어야 되지 않을까요 ? 그런데, Circle 과 Triangle 에도 Rect와 같은 코드를 추가 한다면 코드의 중복이 발생합니다.

아래 코드를 생각해 봅시다.

```
void Draw()
{
    mutex.lock();                   // 모든 도형에 공통적인 코드입니다.
    cout << "Rect Draw" << endl;    // 각 도형마다 달라져야 하는 코드입니다.
    mutex.unlock();                 // 모든 도형에 공통적인 코드입니다.
}
```

결국, Draw() 함수 안에는 모든 도형의 공통적인 코드와 각 도형 마다 달라져야 하는 코드가 같이 존재 합니다.

SW의 설계 원칙 중 하나는

"변하지 않은 코드와 변해야 하는 분리되어야 한다."

는 원칙이 있습니다. 그렇다면, Draw() 안에서 변해야 하는 코드를 다른 가상함수로 분리해 내면 어떨까요 ?

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

```
    {
        mutex.lock();   // 모든 도형에 공통적인 코드입니다.
        DrawImp();      // 변해야 하는 코드는 다른 가상함수로 분리합니다.
        mutex.unlock(); // 모든 도형에 공통적인 코드입니다.
    }
    virtual void DrawImp() = 0;
};
```

결국, Rect, Circle, Triangle 등의 클래스를 만들 때는 Draw()가 아닌 DrawImp() 가상함수를 재정의 하면 됩니다.

이처럼 기반 클래스에는 모든 도형의 공통적인 전체적인 알고리즘을 넣고 변해야 하는 각 단계의 세부 코드를 파생 클래스에서 만드는 것을 "template method" 패턴이라고 부릅니다.

```
#include <iostream>
#include <vector>
using namespace std;

class Shape
{
public:
    void Draw()
    {
        cout << "mutex.lock" << endl;
        DrawImp();
        cout << "mutex.unlock" << endl;
    }
    virtual ~Shape() {}
protected:
    virtual void DrawImp() = 0;
};

class Rect : public Shape
{
public:
    virtual void DrawImp() { cout << "Rect Draw" << endl; }
};
class Circle : public Shape
{
public:
    virtual void DrawImp() { cout << "Circle Draw" << endl; }
};
```

```
int main()
{
    vector<Shape*>   v;

    int cmd;
    while (1)
    {
        cin >> cmd;
        if (cmd == 1) v.push_back(new Rect);
        else if (cmd == 2) v.push_back(new Circle);
        else if (cmd == 9)
        {
            for (auto p : v)
                p->Draw();
        }
    }
}
```

또한, 위 코드에서 주의 깊게 볼 점은

각 도형 클래스의 사용자는 DrawImp() 가 아닌 Draw() 함수를 호출 해야 하므로 Draw()는 public 영역에 DrawImp()는 protected 영역에 만들게 됩니다.

또한, 각 도형은 Draw()를 재정의 하는 것이 아니라 DrawImp()를 재정의 하기 되므로 DrawImp() 가상함수로, Draw()는 비 가상함수로 만들게 됩니다.

이 패턴의 정확한 이름은 "template method" 이지만, C++ 진영에서는 가상함수가 아닌 일반 함수인 Draw()를 사용하게 된다는 의미로 **"NVI(Non Virtual Interface)"** 라고도 부릅니다.

Section 2.

공통성과 가변성의 분리

변하는 것과 변하지 않은 것은 분리

배우게 되는 내용

① 변하는 것과 변하지 않은 것을 분리하는 2가지 방식

② Template method

③ Strategy

template method

카테고리

행위 패턴(Behavioral Pattern)

의도

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.

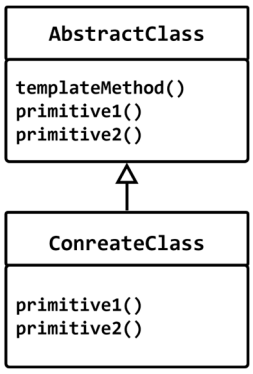
다른 이름

NVI (Non Virtual Interface)

활용성

- 행위 알고리즘의 변하지 않은 부분을 한 번 정의하고 다양해 질 수 있는 부분을 서브 클래스로 정의할 수 있도록 구현하고자 할 때
- 서브 클래스 사이의 공통적인 행위를 추출하여 하나의 공통 클래스로 정의할 때
- 서브 클래스의 확장을 제어할 수 있다.

클래스 다이어그램 (class diagram)



결과

- 코드 재사용을 위한 기술
- 라이브러리를 설계할 때 공통 부분을 분리하는 중요한 디자인 기술.

도형편집기의 draw 구현

코드의 중복

아래 코드는 Rect, Circle 의 draw() 함수에 중복이 많이 나타나는 코드 입니다.

```
class Rect : public Shape
{
public:
    void draw() override
    {
        PainterPath path;
        path.begin();

        // path 멤버 함수로 그림을 그린다.
        path.draw_rect();

        path.end();

        Painter surface;
        surface.draw_path(path);
    }
};
class Circle : public Shape
{
public:
    void draw() override
    {
        PainterPath path;
        path.begin();

        // path 멤버 함수로 그림을 그린다.
        path.draw_circle();

        path.end();

        Painter surface;
        surface.draw_path(path);
    }
};
```

template method 의 적용

```
class Shape
{
public:
    virtual ~Shape() {}

    void draw()
    {
        PainterPath path;
        path.begin();

        // path 멤버 함수로 그림을 그린다.
        draw_imp(path);

        path.end();

        Painter surface;
        surface.draw_path(path);
    }

protected:
    virtual void draw_imp(PainterPath& path) = 0;
};


class Rect : public Shape
{
protected:
    void draw_imp(PainterPath& path) override
    {
        path.draw_rect();
    }
};
class Circle : public Shape
{
protected:
    void draw_imp(PainterPath& path) override
    {
        path.draw_circle();
    }
};
```

strategy

카테고리

행위 패턴(Behavioral Pattern)

의도

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

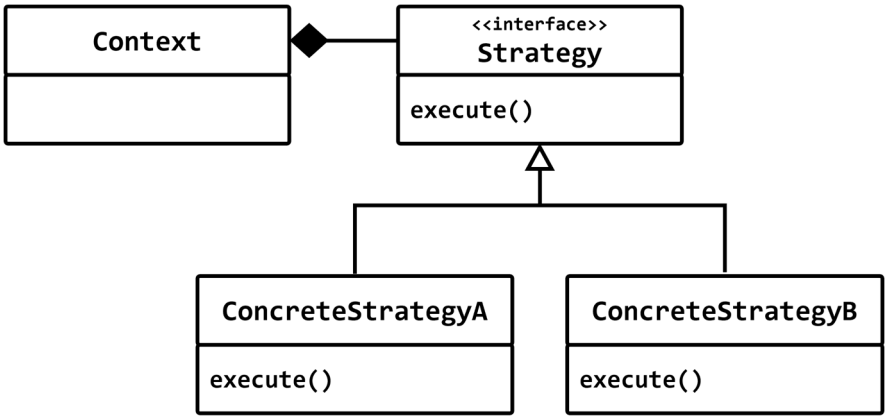
다른 이름

Policy

활용성

- 행위들이 조금씩 다를 뿐 개념적으로 관련된 많은 클래스들이 존재 하는 경우, 각각의 서로 다른 행위 별로 클래스를 작성한다.
- 알고리즘의 변형이 필요한 경우 사용할 수 있다.
- 사용자가 모르고 있는 데이터를 사용해야 하는 알고리즘이 있을 때도 필요 하다.
- 많은 행위를 정의하기 위해 클래스 안에 복잡한 다중 조건문을 사용해야 하는 경우 이런 선택문 보다는 Strategy 클래스로 만드는 것이 바람직하다.

클래스 다이어그램 (class diagram)



결과

- 관련 알고리즘 군을 형성한다.
- 서브 클래싱을 사용하지 않는 다른 방법이다.
- 조건문을 없앨 수 있다.
- 구현의 선택이 가능하다
- 사용자는 서로 다른 전략을 알고 있어야 한다.
- Strategy 와 Context 클래스 사이에 과다한 메시지가 전송된다.
- 객체 수가 증가한다.

1. EditBox의 설계

이번 장에서는 GUI widget 중의 하나인 EditBox와 같은 클래스를 만들면서 객체지향 디자인의 원리를 살펴 보도록 하겠습니다.

EditBox

EditBox는 사용자에게 입력을 받기 위한 GUI 도구 입니다. 우리는 간단하게 콘솔 기반으로 흉내를 내도록 하겠습니다.

```
#include <iostream>
#include <string>
using namespace std;

class EditBox
{
    string data;
public:
    string getData()
    {
        getline(cin, data);
        return data;
    }
};
int main()
{
    EditBox edit;
    string s = edit.getData();  // EditBox 에서 입력된 값을 꺼내 옵니다.
    cout << s << endl;
}
```

위 코드에서는 간단하게 cin 을 사용해서 입력 받은 후에 getData() 멤버 함수를 통해서 입력된 값을 꺼낼 수 있습니다. 그런데, 만약 사용자가 EditBox를 통해서 나이를 입력 받는 생각해 봅시다.

그렇다면, 나이는 숫자로만 구성되므로 문자를 입력 되지 않도록 하는게 좋지 않을까요 ? 즉, EditBox 에서 입력을 받을 때, 입력값의 유효성을 조사할수 있으면 좋지 않을까요 ?

입력값의 validation

이번에는 한 글자씩 입력 받아서 숫자만 입력 가능하도록 변경해 봅시다.

```
#include <iostream>
#include <string>
#include <conio.h>
using namespace std;

class EditBox
{
    string data;
public:
    string getData()
    {
        data.clear();

        while (1)
        {
            char c = getch();
            if (c == 13)    // enter
                break;

            if (isdigit(c)) // 입력 값의 유효성(숫자)을 조사합니다.
            {
                data.push_back(c);
                cout << c;
            }
        }
        cout << endl;
        return data;
    }
};
int main()
{
    EditBox edit;
    string s = edit.getData();
    cout << s << endl;
}
```

위 코드는 사용자로부터 한자씩 입력 받아서 입력 값이 숫자가 맞는지 조사한후 data에 추가합니다. 입력값이 숫자가 아닌 경우 입력값을 무시하고 새롭게 입력 받는 코드입니다.

그런데, 이와 같은 validation 정책은 변경이 가능해야 되지 않을 까요 ? 즉, 나이가 아닌 주소를

입력 받으려면 숫자 뿐 아니라 모든 값이 입력 가능해야 하지 않을까요 ?

Validation 정책의 변경

EditBox의 Validation 정책을 변경하려면 getData()의 if 문을 변경해야 합니다. 그런데, EditBox와 같은 클래스는 라이브러리의 내부 코드이므로 사용자가 라이브러리 코드를 직접 변경하는 것은 바람직하지 않습니다.

즉, EditBox의 코드를 수정하지 않고, Validation 정책을 변경할 수 있도록 설계 해야 합니다. EditBox의 getData()함수를 다시 생각해 봅시다. Validation 정책을 변경하기 위해서 아래 코드에서 변해야 하는 부분을 생각해 봅시다.

```
string getData()
{
    data.clear();

    while (1)
    {
        char c = getch();
        if (c == 13)
            break;

        if ( isdigit(c) ) // validation 정책을 변경하려면 이부분만 변경되면 됩니다.
        {
            data.push_back(c);
            cout << c;
        }
    }
    cout << endl;
    return data;
}
```

위 코드에서 validation 정책을 변경하려면 if 문 안의 isdigit() 부분만 변경되면 되고, 나머지 코드는 변경될 필요가 없습니다. S/W 설계의 가장 중요한 원칙중의 하나는

"변하지 않는 코드와 변하는 코드는 분리 되어야 한다"

는 점입니다.

또한, 변하는 코드는 분리하는 방법은 크게 2가지가 있습니다.

① **변하는 부분을 가상함수로 분리하는 방법.**

② **변하는 부분을 다른 클래스로 분리하는 방법.**

변하는 부분을 가상함수로 분리 – template method

EditBox의 getData() 안에서 validation 코드를 가상함수로 분리하면 EditBox 자체의 코드를 변경하지 않고도 validation 정책을 변경할 수 있습니다.

```
class EditBox
{
    string data;
public:
    // validation 정책을 담은 가상함수.
    virtual bool validate(char c)
    {
        return isdigit(c);
    }
    string getData()
    {
        data.clear();
        while (1)
        {
            char c = getch();
            if (c == 13) break;
            if ( validate(c) )  // validation 정책을 확인하기 위해 가상함수를
            {                   // 호출합니다.
                data.push_back(c);
                cout << c;
            }
        }
        cout << endl;
        return data;
    }
};
```

이제, validation 정책을 변경하려면 EditBox 코드 자체를 변경할 필요 없이, EditBox의 파생 클래스를 만들어서 validate() 가상함수를 재정의 하면 됩니다.

다음은 완성된 코드입니다.

변하는 것을 가상함수로 분리한 경우.

```
#include <iostream>
#include <string>
```

```
#include <conio.h>
using namespace std;

class EditBox
{
    string data;
public:
    virtual bool validate(char c) { return isdigit(c); }
    string getData()
    {
        data.clear();
        while (1)
        {
            char c = getch();
            if (c == 13) break;
            if ( validate(c) )
            {
                data.push_back(c);
                cout << c;
            }
        }
        cout << endl;
        return data;
    }
};
class AddressEdit : public EditBox
{
public:
    virtual bool validate(char c)
    {
        return true;
    }
};
int main()
{
    AddressEdit edit;
    string s = edit.getData();
    cout << s << endl;
}
```

이제 새로운 정책의 EditBox 가 필요하다면 EditBox 를 상속 받아서 정책을 결정하는 가상함수인 validate() 를 오버라이딩 하면 됩니다. 더 이상 기존 EditBox에 대한 수정은 필요하지 않습니다.
이렇게 변하지 않는 전체 알고리즘은 부모 클래스가 비 가상함수를 통해 제공하고, 변하는 부분은 가

상 함수를 통해 분리하는 설계 기법을 GoF의 디자인 패턴에서는 템플릿 메서드(Template Method) 패턴이라 부릅니다.
GoF의 디자인 패턴에서 템플릿 메서드의 의도는 다음과 같습니다.

“객체의 연산에는 알고리즘의 뼈대만을 정의하고 각 단계에서 수행할 구체적 처리는 서브 클래스쪽으로 미루는 패턴입니다. 알고리즘 구조 자체는 그대로 놔둔 채 알고리즘 각 단계의 처리를 서브 클래스에서 재정의할 수 있게 합니다.”

템플릿 메서드를 통해 재사용 가능한 EditBox를 만들었지만, 가상 함수를 통해 정책을 재정의할 경우 다음과 같은 한계가 있습니다

① **실행 시간에 EditBox의 정책을 변경하는 것이 불가능합니다. 가상함수를 통해 정책을 변경하는 것은 결국 객체에 대한 변경이 아닌, 클래스 자체에 대한 변경입니다. 변경된 정책을 사용하기 위해서는 결국 새로운 클래스를 정의해야 합니다.**

② **정책을 재사용할 수 없습니다. 만약 텍스트 처리에 관한 정책이 다른 종류의 클래스에서 필요로 한다면, 재사용 될 수 없습니다.**

정책에 대한 재사용을 위해서는 다른 방법을 사용해야 합니다.

변하는 부분을 다른 클래스로 분리 – strategy 패턴

이번에는 변하는 코드를 가상함수가 아닌 다른 클래스로 분리해 보도록 하겠습니다. 변해야 하므로 교체 가능해야 합니다. 즉, 인터페이스 기반으로 설계 되어야 합니다. 먼저, validation 정책을 담은 인터페이스를 설계 한 후 EditBox에서는 인터페이스 포인터를 가지고 validation을 확인 합니다.

변하는 것을 다른 클래스로 분리한 경우.

```
#include <iostream>
#include <string>
#include <conio.h>
using namespace std;

struct IValidator
{
    virtual bool validate(string s, char c) = 0;
    virtual ~IValidator() {}
};

class EditBox
{
    string data;
    IValidator* pVal = 0;
public:
    void setValidator(IValidator* p) { pVal = p; }

    string getData()
    {
        data.clear();
        while (1)
        {
            char c = getch();
            if (c == 13) break;
            if (pVal == 0 || pVal->validate(data, c))
            {
                data.push_back(c);
                cout << c;
            }
        }
        cout << endl;
        return data;
    }
};
```

```
// validation 정책을 담은 정책 클래스를 설계 합니다.
class LimitDigitValidator : public IValidator
{
    int value;
public:
    LimitDigitValidator(int v) : value(v) {}

    virtual bool validate(string s, char c)
    {
        if (s.size() < value && isdigit(c))
            return true;

        return false;
    }
};

int main()
{
    // EditBox를 만들고 validator를 연결합니다.
    EditBox edit;
    LimitDigitValidator v(5);
    edit.setValidator(&v);

    string s = edit.getData();
    cout << s << endl;
}
```

인터페이스 기반의 클래스로 정책을 분리하게 되면 더이상 클래스에 대한 정책의 변경이 아닌 객체에 대한 정책의 변경이 가능합니다. 실행 시간에 EditBox 의 정책을 변경하는 것이 가능하고, 그 뿐만 아니라 IValidator의 인터페이스를 통해 정책을 사용하는 모든 클래스에서 정책을 재사용 가능합니다. 인터페이스 기반의 클래스로 정책의 변경을 가능하게 하는 설계 기법을 GoF의 디자인 패턴에서는 전략 패턴이라고 부릅니다.

GoF의 디자인 패턴에서 전략 패턴의 의도는 다음과 같습니다.

“동일 계열의 알고리즘군을 정의하고, 각각의 알고리즘을 캡슐화 하며, 이들을 상호 교환이 가능하도록 만드는 패턴입니다. 알고리즘을 사용하는 사용자와 상관없이 독립적으로 알고리즘을 다양하게 변경할 수 있게 합니다.”

변하지 않은 것과 변하는 것의 분리 2

배우게 되는 내용

① Policy Base Design

1. 단위 전략 디자인 (Policy Base Design)

List 클래스

List◇ 라는 컨테이너 클래스를 설계한다고 생각해봅시다. 하지만 List가 만약 전역적으로 사용되어야 한다면 push_front()의 연산을 비롯한 List를 조작하는 연산이 스레드 안전하게 동작하도록 동기화의 구문을 작성해주어야 합니다

```
template <typename T>
class List
{
public:
    void push_front(const T& a)
    {
        mutex.lock();
        //...
        mutex.unlock();
    }
};
```

하지만 위처럼 동기화 코드를 작성하게 되면, 단일 스레드에서 List◇를 사용하는 사용자 입장에서는 동기화에 대한 코드로 인한 성능 저하가 있습니다. 결국 동기화에 대한 정책은 분리되어야 합니다.

동기화 정책을 전략 패턴으로 구현하기

단일 스레드 및 다중 스레드의 전략은 List 뿐 아니라 다른 컨테이너 클래스에서 사용 가능해야 하므로 인터페이스 기반 클래스로 분리하는 전략 패턴을 적용하겠습니다.

List 클래스의 동기화 정책을 인터페이스 기반의 전략 패턴으로 분리한 경우

```
// 동기화 정책을 담은 인터페이스
struct ISync
{
    virtual void lock() = 0;
    virtual void unlock() = 0;
    virtual ~ISync() {}
};
// 다양한 정책 클래스를 설계 합니다.
```

```
class NoLock : public ISync
{
public:
    virtual void lock() {}
    virtual void unlock() {}
};
// List는 동기화 여부를 정책 클래스에 의존 합니다.
template <typename T>
class List
{
    ISync* pSync = 0;
public:
    void setSync(ISync* p) { pSync = p; }

    void push_front(const T& a)
    {
        if ( pSync ) pSync->lock();
        //...
        if ( pSync ) pSync->unlock();
    }
};
int main()
{
    List<int> st;
    NoLock lock;

    // list에 동기화 정책을 연결합니다.
    st.setSync(&lock);
    st.push_front(10);
}
```

결국 위 코드에서 동기화 정책을 변경하려면 ISync 의 파생 클래스를 만들어서 다양한 동기화 정책 클래스를 설계한후 List의 setSync() 함수를 사용해서 연결하면 됩니다.

하지만, 이 방식의 설계에는 다음과 같은 문제가 있습니다.

동기화의 정책을 인터페이스 기반의 클래스를 통해 결정하는 것은 결국 인터페이스의 가상 함수 호출을 통해 결정되기 때문에 성능의 저하가 있습니다. push_front() 처럼 빈번하게 호출되는 함수라면 분명 함수 호출에 대한 오버헤드를 무시할 수 없습니다. 또한 List 같은 컨테이너의 정책은 굳이 실행시간에 변경되는 것도 유용하지 않습니다. 만약 실행 시간에 정책이 변경될 필요가 없고, 함수 호출 에 따른 성능 저하가 걱정된다면, C++ 진영의 다른 설계를 적용하면 됩니다.

C++ 언어에서는 실행시간에 정책 변경이 필요 없을 경우 정책 클래스를 인터페이스 기반이 아닌 템플릿 기반으로 교체 할 수 있습니다. 이 경우 정책에 사용되는 함수가 가상함수가 아닌 인라인

함수 기반으로 설계 될 수 있기 때문에 성능 향상을 볼 수 있습니다.

List 클래스의 동기화 정책을 template 인자로 분리한 경우

```
// 먼저 정책 클래스를 설계 합니다. lock/unlock 함수는 가상함수가 아닌
// inline 함수로 만들수 있습니다.
class NoLock
{
public:
    inline void lock() {}
    inline void unlock() {}
};
class MutexLock
{
    mutex m;
public:
    inline void lock()   { m.lock(); }
    inline void unlock() { m.unlock(); }
};
template <typename T, typename SyncModel = NoLock>
class List
{
    SyncModel sm;
public:
    void push_front(const T& a) {
        sm.lock();
        //......
        sm.unlock();
    }
};
List<int, NoLock> st;
List<int, MutexLock> mt;
```

이제 List의 동기화 정책은 SyncModel 이라는 템플릿 인자에 의해 컴파일 타임에 결정됩니다. 이와 같은 설계의 단점은

"실행 시간에 정책을 교체하는 것이 불가능합니다."

하지만, List의 동기화 정책을 실행시간에 변경할 일은 거의 없으므로 이경우는 문제가 되지 않습니다.

하지만, 장점은

"인터페이스 기반의 클래스가 아니기 때문에 lock과 unlock의 함수가 가상 함수가 아닌 인라인 함수라는 점입니다. lock과 unlock의 함수 호출에 따른 성능의 저하가 전혀 없습니다."

List처럼 템플릿 인자로 정책을 결정하는 설계 기법을 단위 전략(Policy base) 라고 부릅니다. GoF의 디자인 패턴에서는 **"매개변수화 타입(parameterized type)을 이용한 재사용 기법"**으로 언급되어 있습니다.

상태(state) 패턴

배우게 되는 내용

① State 패턴

state

카테고리

행위 패턴(Behavioral Pattern)

의도

객체 자신의 내부 상태에 따라 행위를 변경하도록 한다. 객체는 마치 클래스를 바꾸는 것처럼 보인다.

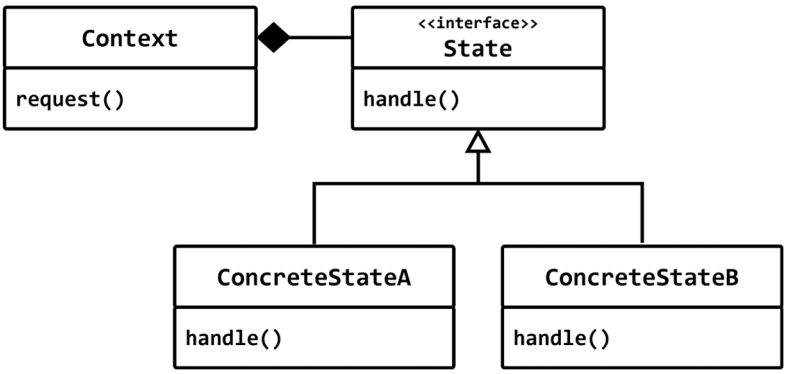
다른 이름

Object for State

활용성

- 객체의 행위는 상태에 따라 달라 질 수 있기 때문에, 객체의 상태에 따라서 런타임시 행위가 바뀌어야 한다.
- 객체에 상태에 따라 수많은 조건 문장을 갖도록 오퍼레이션을 구현 할 수 있다. 이때 객체의 상태를 별도의 객체로 정의 함으로써 행위는 다른 객체와 상관없이 다양화 될 수 있다.

클래스 다이어그램 (class diagram)



결과

- 상태에 따른 행위를 국지화 하여 서로 다른 상태에 대한 행위를 별도의 객체로 관리 한다.
- 상태 전이 규칙을 명확하게 만든다.
- 상태 객체는 공유 될 수 있다.

1. state 패턴

게임에서 사용되는 캐릭터 클래스를 생각해 봅시다.

```
class WonderBoy
{
    int gold;
    int itemState;
public:
    void run()    { cout << "run" << endl; }
    void attack() { cout << "attack" << endl; }
};
int main()
{
    WonderBoy c;
    c.run();
}
```

그런데, 이와 같은 게임 캐릭터 들은 아이템을 획득 할 때 마다 동작이 달라 지게 됩니다. 이런 상황을 코드로 만들어 봅시다.

조건 분기문 사용

첫번째 방법은 if-else 나 switch-case 문을 사용한 조건 분기문을 사용하는 것 입니다.

조건 분기문을 사용해서 item 상태에 따라 게임 캐릭터의 동작을 변경한 코드

```
class WonderBoy
{
    int gold;
    int itemState;
public:
    void run()
    {
        if (itemState == 1)
            cout << "run" << endl;
        else if (itemState == 2)
            cout << "fast run" << endl;
    }
```

```
    void attack() { cout << "attack" << endl; }
};

int main()
{
    WonderBoy c;
    c.run();
}
```

하지만 이러한 코드는 아이템이 추가됨에 따라 코드가 수정 되어야만 합니다. OCP를 만족할 수 없습니다. 이러한 문제를 해결하기 위한 방법은 앞에서 배운 변하는 것을 분리하는 방식을 사용해야 합니다. 2가지 있습니다.

① 변하는 것을 가상함수로 분리

② 변하는 것을 다른 클래스로 분리

변하는 것을 가상함수로 분리

캐릭터의 다양한 동작인 run(), attack() 은 item의 획득 상태에 따라 변하게 됩니다. 이번에는 item 의 상태에 변하는 모든 동작을 가상함수화 해서 파생 클래스에 재정의 하는 방법을 생각해 보겠습니다.

item 상태에 따라 변하는 동작을 가상함수로 만든 코드

```
class WonderBoy
{
    int gold;
    int itemState;
public:
    virtual void run() { cout << "run" << endl; }
    virtual void attack() { cout << "attack" << endl; }
};

class WonderBoyWithFast : public WonderBoy
{
public:
    virtual void run() { cout << "fast run" << endl; }
};

int main()
{
```

```
    WonderBoy *p;
    WonderBoy c;
    WonderBoyWithFast f;
    p = &c;
    p->run();

    p = &f; // 아이템 획득
    p->run();
}
```

하지만 위코드는 문제점은 객체의 동작이 변경되는 것이 아니고 실제로는 객체가 변경되는 것입니다.

변하는 것을 다른 클래스로 분리

이번에는 item의 획득에 따라 변해야 하는 run(), attack() 함수를 다른 클래스로 분리하는 방법을 생각해 보겠습니다. 변할 수 있어야 하므로 인터페이스 기반으로 설계 되어야 합니다.

item 상태에 따라 변하는 동작을 다른 클래스로 분리하는 코드

```
#include <iostream>
using namespace std;

struct IState
{
    virtual void run() = 0;
    virtual void attack() = 0;
    virtual ~IState() {}
};
// item 종류에 따른 다양한 동작을 담은 상태 객체를 제공합니다.
class NoItemState : public IState
{
public:
    virtual void run()    { cout << "run" << endl; }
    virtual void attack() { cout << "attack" << endl; }
};
class FastItemState : public IState
{
public:
    virtual void run()    { cout << "fast run" << endl; }
    virtual void attack() { cout << "power attack" << endl; }
};
```

```
class WonderBoy
{
    int gold;
    IState *state;
public:
    void setState(IState *p) { state = p; }
    void run() { state->run(); }
    void attack() { state->attack(); }
};
int main()
{
    NoItemState nis;
    FastItemState fis;

    WonderBoy w;
    w.setState(&nis);  // 아이템이 없는 상태 입니다.
    w.run();

    w.setState(&fis); // 아이템을 획득한 상태 입니다.
    w.run();
}
```

위 코드는 결국 객체의 상태(Item)에 따라 동작을 바꾸므로 마치 다른 클래스를 사용하는 것과 비슷한 효과를 낼 수 있습니다. 이와 같은 패턴을

"상태(state) 패턴"

이라고 합니다.

클래스 다이어그램이 전략 패턴과 유사한 구조를 나타내게 됩니다. 하지만,

"전략 패턴은 객체가 사용하는 알고리즘을 실행시간에 교체"

할 수 있게 하는 패턴이고,

"상태 패턴은 객체의 상태에 따라 동작을 변경할 때 사용하는 패턴"

입니다.

즉, 구조가 동일해도 의도에 따라 다른 패턴으로 분류됩니다.

Section 3.

재귀적 포함

Menu

배우게 되는 내용

① Menu 시스템 만들기

② Composite Pattern

composite

카테고리

구조 패턴(Structural Pattern)

의도

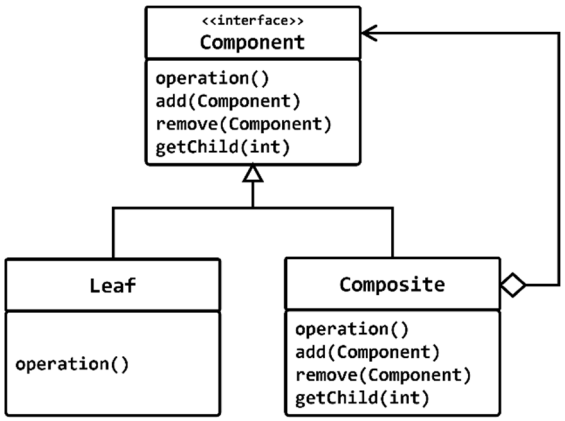
부분과 전체의 계층을 표현하기 위해 복합 객체를 트리 구조로 만든다. Composite 패턴은 클라이언트로 하려면 개별 객체와 복합 객체를 모두 동일하게 다룰 수 있도록 한다.

다른 이름

활용성

부분-집합 관계의 계층도를 표현해야 하고, 복합 관계와 단일 객체 간의 사용 방법에 차이를 두고 싶지 않을 때 사용한다.

클래스 다이어그램 (class diagram)



결과

- 기본 객체와 복합 객체로 합성된 클래스로 하나의 일관된 계층도를 정의 한다.
- 프레임워크 사용자의 코드를 간단히 할 수 있다.
- 새로운 요소들을 쉽게 추가할 수 있다
- 범용성 있는 설계로 만들 수 있다.

1. MenuItem

이번 장에서는 메뉴 시스템을 만들어 보도록 하겠습니다.

▌ Menu 의 종류

메뉴는 다음의 2가지 타입으로 추상화 할수 있습니다.

① **MenuItem : 제목을 가지고 있고, 메뉴 이벤트가 발생하였을 때, 어떤 메뉴 서 이벤트가 발생했는지를 구분할 수 있도록 id가 필요합니다.**

② **PopupMenu : 다른 메뉴를 담는 메뉴 로서 선택시 하위 메뉴를 보여주는 메뉴 입니다.**

이 때 PopupMenu 에는 MenuItem 뿐 아니라 PopupMenu 자체도 담을 수 있습니다.

PopupMenu 가 MenuItem 뿐 아니라 PopupMenu 자체도 담을 수 있으려면 PopupMenu와 MenuItem 은 공통의 기반 클래스가 있어야 합니다.

▌ BaseMenu

BaseMenu는 PopupMenu와 MenuItem 의 공통의 기반 클래스로서, 모든 메뉴의 공통의 특징을 제공하는 역할을 합니다.

```
// 모든 메뉴의 부모 클래스
class BaseMenu
{
    string title;
public:
    BaseMenu(string s) : title(s) {}

    string getTitle() { return title; }

    virtual void command() = 0;
};
```

MenuItem

MenuItem은 하나의 메뉴를 나타내며 선택시 미리 정의된 일을 수행하게 되어 있습니다.

```
class MenuItem : public BaseMenu
{
    int id;
public:
    MenuItem(string s, int n) : BaseMenu(s), id(n) {}

    virtual void command()
    {
        cout << getTitle() << " 메뉴 선택됨" << endl;
        getchar();
    }
};
```

PopupMenu

다음은 이번 항목의 핵심인 PopupMenu 입니다.

```
#define clrscr()  system("cls")   // 화면 지우기..

class PopupMenu : public BaseMenu
{
    vector<BaseMenu*> v; // 핵심!!
public:
    PopupMenu(string s) : BaseMenu(s) {}

    void addMenu( BaseMenu* p) { v.push_back(p); }

    virtual void command()
    {
        while (1)
        {
            clrscr(); // 화면을 지우고
            int sz = v.size();
            for (int i = 0; i < sz; i++)
                cout << i + 1 << ". " << v[i]->getTitle() << endl;

            cout << sz + 1 << ". 상위 메뉴로" << endl;
```

```
            cout << "메뉴를 선택하세요 >> ";
            int cmd;
            cin >> cmd;

            if (cmd < 1 || cmd > sz + 1) // 잘못된 입력
                continue ;

            if (cmd == sz + 1) // 상위 메뉴로 선택..
                break; // 또는 return .

            // 선택된 메뉴를 실행한다. 어떤 메뉴일지 조사할필요가 없다.!!핵심!!
            v[cmd - 1]->command(); // 다형성!!
        }
    }
};
```

Main 함수

이번 디자인의 장점은 main 함수를 보면 확인 할 수 있습니다. 하나의 메뉴가 객체로 구성 되므로 메뉴의 추가/삭제/변경을 아주 간단하게 만들 수 있습니다.

```
int main()
{
    PopupMenu* menubar = new PopupMenu("MenuBar");

    PopupMenu* p1 = new PopupMenu("화면 설정");
    PopupMenu* p2 = new PopupMenu("소리 설정");

    menubar->addMenu(p1);
    menubar->addMenu(p2);

    p1->addMenu(new MenuItem("해상도 변경", 11));
    p1->addMenu(new MenuItem("색상   변경", 12));
    p1->addMenu(new MenuItem("명도   변경", 13));
    p2->addMenu(new MenuItem("크기   변경", 21));
    p2->addMenu(new MenuItem("음색   변경", 22));

    menubar->command();
}
```

Composite 패턴

Composite 패턴은 객체의 포함관계를 나타내는 패턴입니다.

복합객체(PopupMenu)는 개별 객체(MenuItem) 뿐 아니라 복합객체 자신도 보관합니다. 재귀적 포함으로 복합객체를 만드는 기법입니다.
핵심 개념은 다음과 같습니다.

① **복합객체와 개별객체는 공통의 기반 클래스가 있어야 합니다.**
② **복합객체와 개별객체의 사용법이 동일시 된다. 동일시 하는 함수(command) 는 기반 클래스에 있어야 합니다.**

Decorator

배우게 되는 내용

① Decorator 패턴

decorator

카테고리

구조 패턴(Structural Pattern)

의도

객체에 동적으로 새로운 서비스를 추가 할 있게 한다. Decorator 패턴은 기능의 추가를 위해서 서브클래스를 생성하는 것보다 융통성 있는 방법을 제공한다.

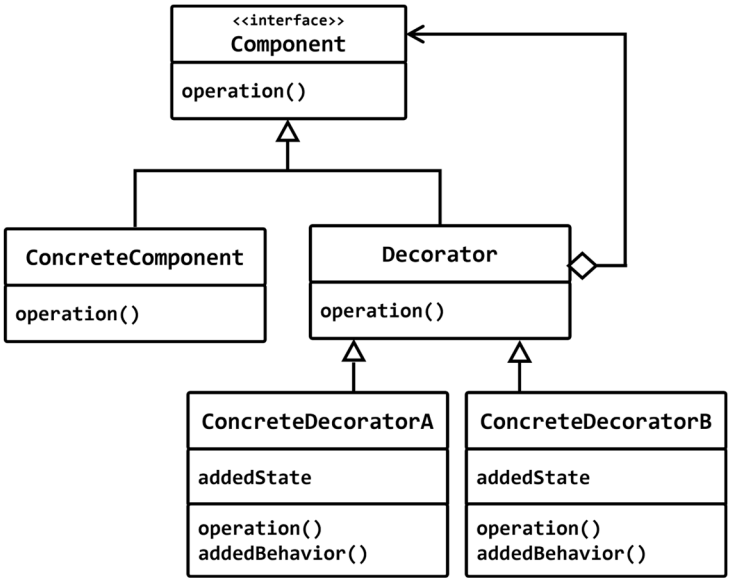
다른 이름

Wrapper

활용성

- 동적으로 투명하게(transparent) 객체에 새로운 서비스를 추가하는 방법이며, 다른 객체에 어떠한 영향도 주지 않게 된다.
- 제거될 수 있는 서비스 일 때
- 실제 상속에 의해 서브클래스를 계속 만드는 방법이 실질적이지 못할 때 유용하다.

클래스 다이어그램 (class diagram)



결과

- 단순한 상속보다 설계의 융통성을 증대 시킬 수 있다.
- 지금 예상하지 못한 특성들을 한꺼번에 다 개발하기 위해 고민하고 노력하기 보다는 지속적인 Decorator 객체를 통해서 발견하지 못하고 누락된 서비스들을 추가할 수 있다.
- Decorator 와 Component 가 동일한 것은 아니다.
- Decorator 를 사용함으로써 작은 규모의 객체들을 많이 생성한다.

1. Decorator 패턴

Decorator 패턴은 객체에 동적으로 기능을 추가할 때 사용되는 패턴 입니다. Composite 패턴과 유사한 구조를 가지는 패턴입니다. 이번 항목에서 간단한 예제를 통해서 Decorator 패턴에 대해서 살펴 보도록 하겠습니다.

Picture 클래스

그림 하나를 관리하면서 화면에 그림을 그리는 클래스를 생각해 봅시다.

```
#include <iostream>
#include <string>
using namespace std;

// 사진을 관리하면서 화면에 그림을 그리는 클래스

class Picture
{
    string imgsrc;
public:
    void Draw() { cout << imgsrc << " Draw" << endl; }
};

int main()
{
    Picture pic;
    pic.Draw();
}
```

이제 Picture 클래스가 그린 그림에 액자 그림을 추가하는 것을 생각해 봅시다. 일반적인 객체지향 프로그래밍에서는 기존 클래스에 기능을 추가 하기 위해서는 상속 문법을 주로 사용합니다.

상속을 통한 기능의 추가

Picture 클래스로부터 상속 받아서 액자를 그리는 기능을 추가하는 클래스를 생각해 봅시다.

```
class Picture
{
    string name;
public:
    void Draw() { cout << "Picture Draw" << endl; }
};
class DrawFrame : public Picture
{
public:
    void Draw()
    {
        Picture::Draw();              // 원래 기능(사람 그림)을 수행하고
        cout << "Frame Draw" << endl; // 추가 기능(액자 그림)을 수행
    }
};
int main()
{
    Picture pic;
    pic.Draw();
    DrawFrame df;
    df.Draw();
}
```

위 코드의 문제점은 pic 객체에 기능이 추가된 것이 아니라 Picture 클래스에 기능이 추가 되었다는 점입니다. 일반 적으로 상속은 기존 클래스에 기능을 추가하는 것으로 정적으로 기능의 추가 됩니다. 객체가 아닌 클래스에 기능이 추가됩니다.

그렇다면, 클래스가 아닌 객체에 기능을 추가하려면 어떻게 해야 할까요 ?

포함을 사용한 기능의 추가

포함을 사용하면 클래스가 아닌 객체에, 정적인 기능 추가가 아닌 동적인 기능을 추가할 수 있습니다.

```
class Picture
{
    string name;
public:
    void Draw() { cout << "Picture Draw" << endl; }
};
class FrameDecorator
{
    Picture* pic;
public:
    FrameDecorator(Picture* p) : pic(p) {}
    void Draw()
    {
        pic->Draw();                    // 원래 기능을 수행하고
        cout << "Draw Frame" << endl; // 추가 기능 수행
    }
};
int main()
{
    Picture pic;
    pic.Draw();

    FrameDecorator fd(&pic); // 객체 pic에 기능을 추가합니다.
    fd.Draw();
}
```

만약 FrameDecorator와 같은 기능 추가 객체가 하나가 아니라 여러 개가 있고, 객체에 기능추가 후 다시 기능을 추가하려면 어떻게 하면 될까요 ?

객체와 기능 추가 객체는 동일 기반 클래스가 있어야 한다.

객체에 기능을 추가하고, 다시 추가로 기능을 추가하기 위해서는 객체와 기능 추가 객체는 동일의 기반 클래스가 있어야 합니다.

결국 Composite 패턴과 유사한 형태의 모양을 가지게 됩니다.

```
#include <iostream>
#include <string>
using namespace std;

// 객체에 기능을 추가하고 다시 기능을 추가 할 수 있어야 합니다.
// 객체와 기능 추가 객체(Decorator)는 동일 부모가 있어야 합니다.
struct Item
{
    virtual void Draw() = 0;
    virtual ~Item() {}
};
// 객체
class Picture : public Item
{
    string name;
public:
    void Draw() { cout << "Picture Draw" << endl; }
};
// 객체에 기능을 추가하는 객체(Decorator)
class FrameDecorator : public Item
{
    Item* pic;
public:
    FrameDecorator(Item* p) : pic(p) {}
    void Draw()
    {
        pic->Draw();
        cout << "Draw Frame" << endl;
    }
};
class FlowerDecorator : public Item
{
    Item* pic;
public:
    FlowerDecorator(Item* p) : pic(p) {}
    void Draw()
    {
```

```
        pic->Draw();
        cout << "Flower Frame" << endl;
    }
};
int main()
{
    Picture pic;
    FrameDecorator fd(&pic);
    FlowerDecorator fd2(&fd);
    fd2.Draw();
}
```

기능 추가 객체의 공통의 기반 클래스

기능을 추가하는 객체인 FrameDecorator와 FlowerDecorator 는 공통의 특징을 가지고 있습니다. 공통의 특징을 나타내는 기반 클래스를 제공하면 편리합니다.

```
struct Item
{
    virtual void Draw() = 0;
    virtual ~Item() {}
};
class Picture : public Item
{
    string name;
public:
    void Draw() { cout << "Picture Draw" << endl; }
};
// 기능 추가 객체의 기반 클래스
class IDecorator : public Item
{
    Item* pic;
public:
    IDecorator(Item* p) : pic(p) {}
    void Draw() { pic->Draw(); }
};
class FrameDecorator : public IDecorator
{
public:
    FrameDecorator(Item* p) : IDecorator(p) {}
    void Draw()
```

```
    {
        IDecorator::Draw();
        cout << "Draw Frame" << endl;
    }
};
class FlowerDecorator : public IDecorator
{
public:
    FlowerDecorator(Item* p) : IDecorator(p) {}
    void Draw()
    {
        IDecorator::Draw();
        cout << "Flower Frame" << endl;
    }
};
int main()
{
    Picture pic;
    FrameDecorator fd(&pic);
    FlowerDecorator fd2(&fd);
    fd2.Draw();
}
```

Stream Class 와 Decorator

C#, java 등의 객체지향 언어에서는 Stream 클래스에 기능을 추가하기 위해 Decorator 패턴을 사용합니다.

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string>

struct Stream
{
    virtual void Write(const std::string& s) = 0;
    virtual ~Stream() {}
};

class FileStream : public Stream
{
    FILE* file;
public:
    FileStream(const char* s, const char* mode = "wt")
    {
        file = fopen(s, mode);
    }
    ~FileStream() { fclose(file); }

    void Write(const std::string& buff)
    {
        std::cout << buff << " 쓰기" << std::endl;
    }
};

// 1. Stream 클래스들과 기능추가 클래스는 동일 기반 클래스 이어야한다
class ZipDecorator : public Stream
{
    Stream* stream;
public:
    ZipDecorator(Stream* s) : stream(s) {}

    void Write(const std::string& s)
    {
        std::string s2 = s + " 압축";  // 추가할 기능(데이타 압축)
        stream->Write(s2); // 원래의 기능수행
                           // (파일, 네트워크, 파이프)에 쓰기
    }
```

```
};

int main()
{
    FileStream fs("a.txt");
    fs.Write("Hello");

    ZipDecorator zd(&fs);
    zd.Write("Hello");
}
```

Part 4.

간접층의 원리

Adapter

배우게 되는 내용

① 상속을 사용한 어답터

② 포함을 사용한 어답터

adapter

카테고리

구조 패턴(Structural Pattern)

의도

클래스의 인터페이스를 클라이언트가 기대하는 형태의 인터페이스로 변환 한다. Adapter 패턴은 서로 일치 하지 않은 인터페이스를 갖는 클래스들을 함께 동작 시킨다.

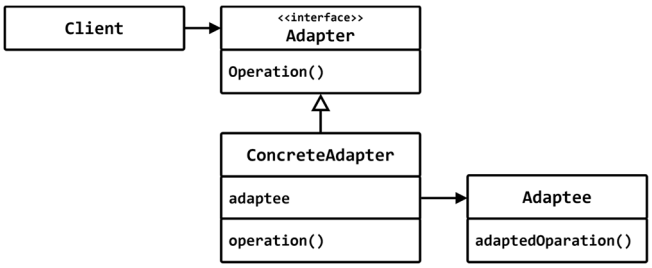
다른 이름

Wrapper

활용성

- 기존의 클래스를 사용해야 하나 인터페이스가 수정되어야 하는 경우
- 이미 만들어진 것을 재사용하고자 하나 이 재사용 가능한 라이브러리를 수정할 수 없는 경우.
- 객체 생성의 책임을 서브클래스에 위임 시키고 서브 클래스에 대한 정보를 은닉하고자 하는 경우

클래스 다이어그램 (class diagram)



결과

- Adapter 클래스가 Adaptee 클래스에 맞게 인터페이스를 Target 클래스로 변형한다. 또한 Adapter 클래스는 Adaptee 클래스의 행위를 재정의할 수 있다.

1. Adapter 패턴

TextView 클래스

화면에 문자를 출력하는 TextView 라는 클래스를 생각해 봅시다.

```
class TextView
{
    string data;
public:
    TextView(string s) : data(s) {}

    void Show() { cout << data << endl; }
};
```

그런데, 이 클래스를 앞에서 만든 도형편집기와 연동하고 싶다고 생각해 봅시다. 그런데, 도형편집기와 연동이 되려면 모든 도형 클래스는 Shape 클래스로부터 파생 되어야 하고 Draw() 함수가 있어야 합니다.

결국 TextView의 Show() 함수의 이름을 Draw()로 변경해야 합니다.

이처럼 기존 클래스의 인터페이스(함수이름)을 변경해서 시스템이 요구하는 클래스를 만드는 디자인 기법을

"어답터(Adapter) 패턴"

이라고 합니다.

▌ 상속을 통한 Adapter – 클래스 어답터

상속을 사용해서 기존의 Show() 함수를 Draw()함수로 변경할 수 있습니다.

```
class Shape
{
public:
    virtual void Draw() = 0;
    virtual ~Shape() {}
};
class Text : public TextView, // 이 클래스의 기능을
             public Shape     // 이 인터페이스의 요구사항으로 변경한다
{
public:
    Text(string s) :TextView(s) {}

    void Draw() { TextView::Show(); }
};
int main()
{
    TextView tv("hello");
    Shape* p = &tv;
}
```

위와 같은 상속을 통한 어답터는 객체가 아닌 클래스의 인터페이스를 변경하게 됩니다. 그래서 흔히

“클래스 어답터”

라고 합니다.

포함을 사용한 어답터 – 객체 어답터

포함을 사용하면 클래스가 아닌 객체의 인터페이스를 변경할 수 있습니다.

```
class Shape
{
public:
    virtual void Draw() = 0;
    virtual ~Shape() {}
};
class Text : public Shape
{
    TextView* txtView;
public:
    Text(TextView* p) : txtView(p) {}

    void Draw() { p->Show(); }
};

int main()
{
    TextView tv("hello");
    Text t(&tv);   // tv객체의 인터페이스를 변경해서
    Shape* p = &t; // 도형편집기에서 사용합니다.
}
```

STL Container Adapter

C++ 표준 라이브러리인 STL에는 Sequence Container(vector, list, deque) 의 인터페이스를 변경해서 stack, queue, priority_queue 로 만들어주는 adapter 패턴을 사용하고 있습니다.

```
template <typename T, typename C = deque<T>> class stack
{
    C st;
public:
    inline void push(const T& a) { st.push_back(a); }
    inline void pop()          { st.pop_back(); }
    inline T& top()               { return st.back(); }
};
```

STL iterator Adapter

std::reverse_iterator 를 사용하면 기존 반복자의 동작과 반대로 동작하는 반복자를 만들수 있습니다.

```
int main()
{
    std::vector<int> s = {1,2,3,4,3,2,1};

//  auto first = s.begin();
//  auto last  = s.end();

//  std::reverse_iterator< std::vector<int>::iterator  > first(s.end());
    std::reverse_iterator first(s.end());
    std::reverse_iterator last(s.begin());

    auto ret = std::find(first, last, 3);
}
```

분산 프로그래밍 - Proxy Stub 구조

배우게 되는 내용

① Proxy

proxy

카테고리

구조 패턴(Structural Pattern)

의도

다른 객체에 접근하기 위해 중간 대리 역할을 하는 객체를 둔다.

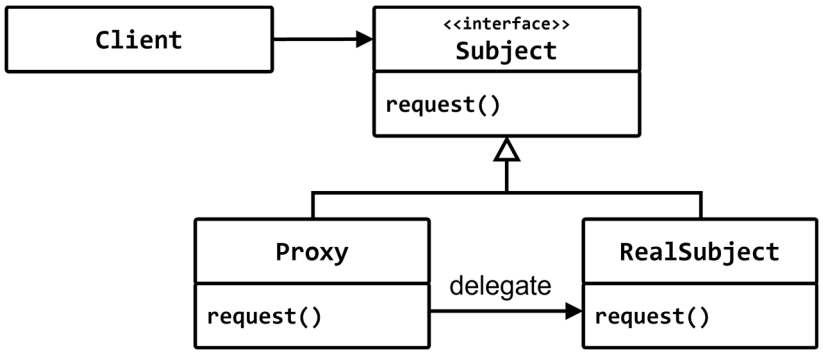
다른 이름

Surrogate

활용성

- 원격지(Remote) Proxy 는 서로 다른 주소 공간에 존재하는 객체에 대한 지역적 표현으로 사용된다. (Proxy – Stub)
- 가상(Virtual) Proxy 는 요청이 있을 때만 필요한 복잡한 객체를 생성한다.
- 보호용(Protect) Proxy 는 원래 객체에 대한 실제 접근을 제어 한다.
- 스마트 참조는(Smart reference) 객체로 접근이 일어날 때 추가적인 행동을 수행하는 노출된 포인터를 대신한다.

클래스 다이어그램 (class diagram)



1. Proxy

이번에는 분산 프로그래밍에서 널리 사용되는 Proxy-Stub 구조에 대해서 살펴 보도록 하겠습니다.

Inter Process Communication (IPC)

먼저 간단한 IPC서버를 만들어 보겠습니다. 시스템 프로그래밍의 세부 내용을 감추기 위해 "ecourse_dp.hpp" 헤더 파일에 IPC관련 기능을 제공하고 있습니다. 먼저 서버입니다.

계산기 서버.

```
#include <iostream>
#include "ecourse_dp.hpp"
using namespace std;
using namespace ecourse;

class Calc
{
public:
    int Add(int a, int b) { return a + b; }
    int Sub(int a, int b) { return a - b; }
};
Calc calc;


int dispatch( int code, int x, int y )
{
    printf("[DEBUG] %d, %d, %d\n", code, x, y );

    switch( code )
    {
    case 1: return calc.Add( x, y);
    case 2: return calc.Sub( x, y);
    }
    return -1;
}

int main()
{
    ec_start_server("CalcService", dispatch);
}
```

계산기 서버는 Client 가 1을 전달하면 덧셈을 2를 전달하면 뺄셈을 수행해서 원격지로 전달해주는 서버 입니다.

계산기 클라이언트

이번에는 계산기 클라이언트 입니다. 계산기 클라이언트는 서버의 번호를 구한후에 서버에게 1, 2를 전달해서 덧셈, 뺄셈의 결과를 얻어오고 있습니다.

계산기 클라이언트

```
#include <iostream>
#include "ecourse_dp.hpp"
using namespace std;
using namespace ecourse;

int main()
{
    int server = ec_find_server("CalcService");

    cout << "server : " << server << endl;

    int ret = ec_send_server(server, 1, 10, 20);

    cout << ret << endl; // 30
}
```

이 코드의 문제점은 클라이언트 개발자가 덧셈을 하려면 1, 뺄셈을 하려면 2 라는 명령 코드를 기억하고 있어야 합니다. 지금은 2개 밖에 없지만 실전에서는 수십~수백개의 명령코드가 있을 수 있습니다.

Proxy

계산기 서버에 접속을 쉽게 할 수 있도록 서버 제작자가 Calc 라는 클래스를 제공하면 어떨까요 ?

그렇다면 클라이언트 제작자는 1,2 라는 숫자 대신 Add(), Sub() 함수를 통해서 서버와 통신 할 수 있지 않을 까요 ?

Proxy를 사용한 계산기 클라이언트

```
#include <iostream>
#include "ecourse_dp.hpp"
using namespace std;
using namespace ecourse;

// Proxy..
class Calc
{
    int server;
public:
    Calc() { server = ec_find_server("CalcService");    }

    int Add(int a, int b) { return ec_send_server(server, 1, a, b);}
    int Sub(int a, int b) { return ec_send_server(server, 2, a, b);}
};

int main()
{
    Calc* pCalc = new Calc;

    cout << pCalc->Add(1, 2) << endl;
    cout << pCalc->Sub(10, 8) << endl;
}
```

인터페이스

결국 Proxy는 서버의 모든 기능을 클라이언트에게 제공하기 위한 클래스 입니다. 그렇다면 Proxy 가 서버의 모든 기능을 동일한 이름으로 제공하는 것을 보장하면 좋지 않을까요 ?

인터페이스를 사용하면 서버의 함수 이름과 Proxy의 함수 이름이 동일하다는 것을 보장할 수 있습니다.

ICalc.h

```
struct ICalc
{
    virtual int Add(int a, int b) = 0;
    virtual int Sub(int a, int b) = 0;

    virtual ~ICalc() {} // 부모의 소멸자는 가상이어야 한다.
};
```

그리고, Proxy를 만들 때 ICalc 인터페이스를 구현합니다.

Client3.cpp

```
#include <iostream>
#include "ecourse_dp.hpp"
#include "ICalc.h"
using namespace std;
using namespace ecourse;

class Calc : public ICalc
{
    int server;
public:
    Calc() { server = ec_find_server("CalcService");    }

    int Add(int a, int b) { return ec_send_server(server, 1, a, b);}
    int Sub(int a, int b) { return ec_send_server(server, 2, a, b);}
};

int main()
{
    Calc* pCalc = new Calc;

    cout << pCalc->Add(1, 2) << endl;
    cout << pCalc->Sub(10, 8) << endl;
}
```

Proxy를 동적 모듈로 제공

Proxy 는 결국 서버 제작자가 만들어서 Client 제작자에게 배포 하게 됩니다.

일반적으로 서버 제작자는 동적모듈등의 방식으로 Proxy를 배포 하게 됩니다.

CalcProxy.cpp

```
#include "ecourse_dp.hpp"
#include "ICalc.h"
using namespace ecourse;

class Calc : public ICalc
{
    int server;
    int count = 0;
public:
    Calc()  { server = ec_find_server("CalcService");    }
    ~Calc()  { cout << "~Calc" << endl; }

    void AddRef()  { ++count;}
    void Release() { if ( --count == 0 ) delete this; }

    int Add(int a, int b) { return ec_send_server(server, 1, a, b);}
    int Sub(int a, int b) { return ec_send_server(server, 2, a, b);}
};

extern "C" __declspec(dllexport)
ICalc* CreateCalc()
{
    return new Calc;
}
```

빌드 하는 법

g++ -shared CalcProxy.cpp -o CalcProxy.dll

또는

cl CalcProxy.cpp /LD

동적 모듈로 제공된 Proxy를 사용하는 클라이언트

Client 제작자는 서버 제작자로부터 Proxy 용 인터페이스 헤더와 동적 모듈을 받아서 사용하게 됩니다.

Client4.cpp

```
#include <iostream>
#include "ecourse_dp.hpp"
#include "ICalc.h"
using namespace std;
using namespace ecourse;

typedef ICalc* (*F)();

int main()
{
    // 동적 모듈 load
    void* addr = ec_load_module("CalcProxy.dll");

    F f = (F)ec_get_function_address(addr, "CreateCalc");

    ICalc* pCalc = f(); // CreateCalc()
    //---------------------------------------

    cout << pCalc->Add(1, 2) << endl;
    cout << pCalc->Sub(10, 8) << endl;
}
```

Façade 패턴

배우게 되는 내용

① facade

facade

카테고리

구조 패턴(Structural Pattern)

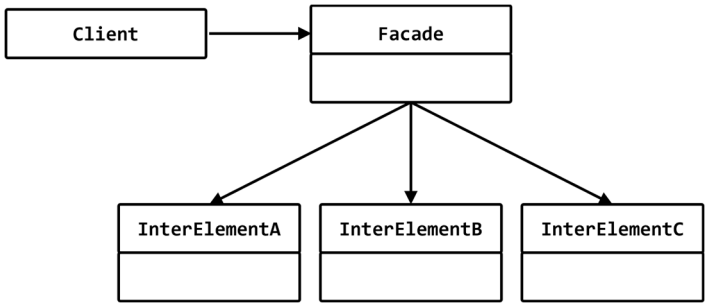
의도

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

활용성

- 복잡한 서브 시스템에 대한 단순한 인터페이스 제공이 필요할 때 유용하다.
- 클라이언트와 구현 클래스 간에 너무 많은 종속성이 존재 할 때 Facade 의 사용으로 클라이언트와 다른 서브 시스템간 의 결합도를 줄일 수 있다.
- 서브 시스템들의 계층화를 이루고자 할 때, Facade 는 각 서브시스템의 계층별 접근점을 제공한다.

클래스 다이어그램 (class diagram)



결과

- 서브 시스템의 구성 요소를 보호 할 수 있다. 이로써 클라이언트가 다루어야 할 객체의 수가 줄어든다.
- 서브 시스템과 클라이언트 코드 간의 결합도 를 줄일 수 있다.
- 서브 시스템 클래스를 사용하는 것을 완전히 막지는 않는다.

1. façade

이번에는 간단한 TCP 서버를 생각해 봅시다.

C언어로 만든 TCP 서버

먼저 C언어로 만든 TCP 서버입니다.

```
#include <iostream>
#include <WinSock2.h>
using namespace std;
#pragma comment( lib, "ws2_32.lib")

int main()
{
    WSADATA w;
    WSAStartup(0x202, &w); // 네트워크 라이브러리 초기화(Linux 에서는 필요없습니다.)

    // 1. 소켓 생성
    int sock = socket(PF_INET, SOCK_STREAM, 0); // TCP 소켓

    // 2. 소켓에 주소 지정
    SOCKADDR_IN addr = { 0 };
    addr.sin_family = AF_INET;
    addr.sin_port = htons(4000);
    addr.sin_addr.s_addr = inet_addr("127.0.0.1");

    bind(sock, (SOCKADDR*)&addr, sizeof(addr));

    // 3. 소켓을 대기 상태로변경
    listen(sock, 5);

    // 4. 클라이언트 대기
    SOCKADDR_IN addr2 = { 0 };
    int sz = sizeof(addr2);

    accept(sock, (SOCKADDR*)&addr2, &sz);// Client가 접속할 때 까지 대기한다.
    WSACleanup();
}
```

C언어를 사용해서 네트워크 프로그램을 하려면 많은 함수와 많은 구조체를 사용해서 코드를 작성해야 합니다.

C++에서는 클래스로 래핑하면 훨씬 간단하고 N/W 프로그램을 작성할 수 있습니다.

SRP (Single Responsibility Principle)

C++의 객체지향 설계의 원칙에는 단일 책임의 원칙이 있습니다. 하나의 클래스는 하나의 기능만을 가지는 것이 좋다는 원칙입니다.

SRP 원칙에 따라, IP 주소를 관리하는 클래스, N/W 라이브러리를 초기화하는 클래스, Socket 클래스를 각각 만들어 보도록 하겠습니다.

```
#include <iostream>
#include <WinSock2.h>
using namespace std;
#pragma comment( lib, "ws2_32.lib")

class NetworkInit
{
public:
    NetworkInit()
    {
        WSADATA w;
        WSAStartup(0x202, &w); // 네트워크 라이브러리 초기화
    }
    ~NetworkInit()
    {
        WSACleanup();
    }
};
class IPAddress
{
    SOCKADDR_IN addr;
public:
    IPAddress(const char* ip, short port)
    {
        addr.sin_family = AF_INET;
        addr.sin_port = htons(port);
        addr.sin_addr.s_addr = inet_addr(ip);
    }
    SOCKADDR* getRawAddress() { return (SOCKADDR*)&addr; }
};
```

```
class Socket
{
    int sock;
public:
    Socket(int type) { sock = socket(PF_INET, type, 0); }

    void Bind(IPAddress* ip)
    {
        // C함수를 다시 호출해서 bind
        ::bind(sock, ip->getRawAddress(), sizeof(SOCKADDR_IN));
    }
    void Listen() { ::listen(sock, 5); }
    void Accept()
    {
        SOCKADDR_IN addr2 = { 0 };
        int sz = sizeof(addr2);
        accept(sock, (SOCKADDR*)&addr2, &sz);
    }
};
int main()
{
    NetworkInit init;
    IPAddress ip("127.0.0.1", 4000);
    Socket sock(SOCK_STREAM); // TCP
    sock.Bind(&ip);
    sock.Listen();
    sock.Accept();
}
```

이제, main 함수는 훨씬 간결해 졌습니다. 그런데, 문제는 비록 main 함수가 간결해 졌지만 이 코드를 작성하고 이해하려면 TCP의 절차를 명확히 알아야 한다는 점입니다.

만약, TCP 서버의 절차를 래핑 하는 상위 클래스를 제공하면 어떨까요 ?

TCPServer Facade

하위 클래스의 복잡함을 단순화 하는 상위 클래스를 제공하는 façade 패턴이라고 합니다. 잘 만들어진 façade 클래스는 라이브러리 사용자를 쉽게 만들어 줍니다.

```
class TCPServer
{
    NetworkInit init;
    Socket sock;
public:
    TCPServer() : sock(SOCK_STREAM) {}
    void Start(const char* sip, short port)
    {
        IPAddress ip(sip, port);
        sock.Bind(&ip);
        sock.Listen();
        sock.Accept();
    }
};

int main()
{
    TCPServer server;
    server.Start("127.0.0.1", 4000);
}
```

Bridge 패턴

배우게 되는 내용

① Bridge 패턴

bridge

카테고리

구조 패턴(Structural Pattern)

의도

구현과 추상화 개념을 분리하여 각각을 독립적으로 변형 할 수 있게 한다

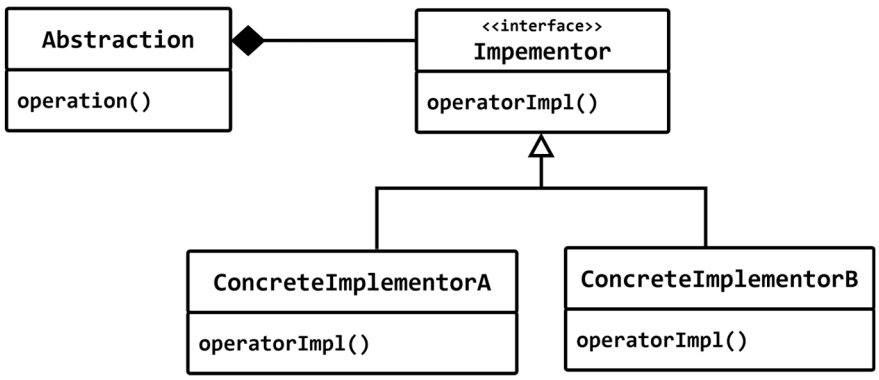
다른 이름

Handle/body

활용성

- 추상화 개념과 이에 대한 구현간의 종속성을 없애고 싶을 때
- 추상화 개념과 구현 모두가 독립적으로 상속에 의해 확장 가능할 경우.
- 추상화 개념에 대한 구현 내용을 변경하는 것은 다른 관련 프로그램에 아무런 영향을 주지 않는다.
- C++ 사용자들은 클라이언트로부터 구현을 완벽하게 은닉하길 원한다.
- 클래스 계층도 에서 클래스의 수가 급증하는 것을 방지 하고자 할 때
- 여러 객체들에 걸쳐 구현을 공유하고자 할 때 또 이런 사실을 다른 코드에 공개 하고 싶지 않을 때

클래스 다이어그램 (class diagram)



결과

- 인터페이스와 구현의 결합도 약화
- 확장성 재고. Abstraction 과 Implementor 를 독립적으로 확장할 수 있다.
- 구현 사항을 은닉할 수 있다.

1. Bridge

인터페이스 기반 설계

인터페이스 기반으로 클래스를 이용하면, 다른 클래스를 코드 수정 없이 이용하는 것이 가능합니다.

```
struct IMP3
{
    virtual void Play() = 0;
    virtual void Stop() = 0;
    virtual ~IMP3() {}
};
class IPod : public IMP3
{
public:
    void Play() { cout << "Play With IPod" << endl; }
    void Stop() { cout << "Stop With IPod" << endl; }
};

class People
{
public:
    void useMP3(IMP3* p) { p->Play(); }
};
```

만약 사용자가 새로운 기능을 요구한다면, 인터페이스가 변경되어야 합니다. 인터페이스가 변경되면 해당하는 인터페이스를 구현하는 모든 클래스는 다시 설계되어야 합니다.

구현부와 인터페이스의 분리

사용자가 구현부를 직접 사용하게 하지 말고, 중간층을 도입하면 변화에 쉽게 대응할 수 있습니다.

```
class MP3
{
    IMP3* engine;
public:
    MP3(IMP3* e) : engine(e)
    {
        if (engine == 0) engine = new IPod;
    }
    // 모든 함수는 결국 engine 을 사용하게 된다
    void Play() { engine->Play(); }
    void Stop() { engine->Stop(); }

    // IMP3를 변경하지 않아도 새로운 서비스를 할수 있다.
    void PlayOneMinute()
    {
        engine->Play();
        //Sleep(1000);
        engine->Stop();
    }
};
class People
{
public:
    void useMP3(MP3* p) { p->Play(); }
};
```

이처럼 구현부와 인터페이스를 분리해서 상호 독립적인 업데이트를 가능하게 하는 패턴을 브릿지 패턴이라고 합니다. 브릿지 패턴을 도입하면 구현부의 변화없이 인터페이스를 변경하는 것이 가능하고,인터페이스의 변화 없이 구현부를 변경하는 것이 가능합니다.

Section 5.

통보, 열거, 방문

Observer

배우게 되는 내용

① Observer Pattern

observer

카테고리

행위 패턴(Behavioral Pattern)

의도

객체 사이의 1:N 의 종속성을 정의 하고 한 객체의 상태가 변하면 종속된 다른 객체에 통보가 가고 자동으로 수정이 일어 나게 한다.

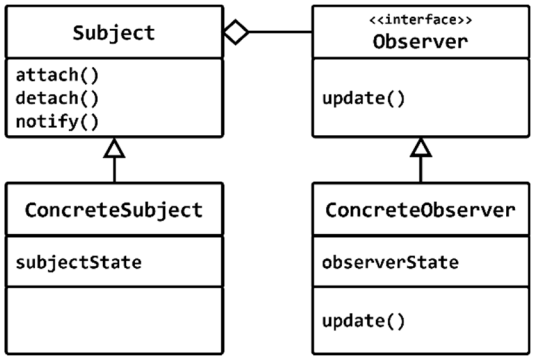
다른 이름

Dependent, Publish – Subscribe

활용성

- 추상화 개념이 두 가지의 측면을 갖고 하나가 다른 하나에 종속적일 때, 이런 종속 관계를 하나의 객체로 분리시켜 이들 각각을 재 사용할 수 있다.
- 한 객체에 가해진 변경으로 다른 객체를 변경해야 할 때 프로그래머들은 얼마나 많은 객체들이 변경되어야 하는지를 몰라도 된다.
- 객체는 다른 객체에 변화를 통보 할 수 있는데, 변화에 관심 있어 하는 객체들이 누구 인지에 대한 가정 없이도 이루어져야 할 때

클래스 다이어그램 (class diagram)



결과

- Subject 와 Observer 클래스 간에는 추상화된 결합도 만이 존재 한다.
- Broadcast 방식의 교류를 가능하게 한다.
- 예측하지 못한 수정

1. Observer

관찰자 패턴 개념

이번 장에서는 GoF의 관찰자 패턴에 대해서 정리해 봅시다. GoF의 관찰자 패턴은 다음과 같은 의도를 가지고 있습니다. “객체들 사이에 일 대 다의 의존 관계를 정의해 두어, 어떤 객체의 상태가 변할 때 그 객체에 의존성 을 가진 다른 객체들이 그 변화를 통지 받고 자동으로 갱신 될 수 있게 만드는 패턴입니다.”
엑셀에서 테이블이 변할 때 테이블과 연결된 모든 그래프가 자동으로 update 되는 것이 관찰자 패턴의 대표적인 예입니다.

기본 구현

관찰자 패턴은 간단하게 구현할 수 있습니다. 테이블에는 다양한 그래프가 연결 될 수 있으므로 그래프의 인터페이스가 필요합니다.

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

// 그래프는 확장이 가능해야 한다. 인터페이스를 먼저 설계한다.
struct IGraph
{
    virtual void OnUpdate(int* data) = 0;
    virtual ~IGraph() {}
};
// 관찰의 대상(테이블, Subject 라고 부른다.)
class Table
{
    vector<IGraph*> v;
    int data[5];
public:
    Table() { memset(data, 0, sizeof(int) * 5); }

    void attach(IGraph* p) { v.push_back(p); }
    void detach(IGraph* p) {}
    void notify(int* data)
    {
        for (int i = 0; i < v.size(); i++)
```

```
            v[i]->OnUpdate(data);
    }
    //----------------------
    void edit()
    {
        while (1)
        {
            int index;
            cout << "index >> "; cin >> index;
            cout << "data  >> "; cin >> data[index];

            // 모든 그래프(관찰자)에게 알려 준다
            notify(data);
        }
    }
};

// 다양한 모양의 그래프(관찰자) 클래스
class PieGraph : public IGraph
{
public:
    virtual void OnUpdate(int* p)
    {
        cout << "***** Pie Graph *****" << endl;
        for (int i = 0; i < 5; i++)
            cout << i << " : " << p[i] << endl;
    }
};

class BarGraph : public IGraph
{
public:
    virtual void OnUpdate(int* p)
    {
        cout << "***** Bar Graph *****" << endl;
        for (int i = 0; i < 5; i++)
            cout << i << " : " << p[i] << endl;
    }
};

int main()
{
    Table t;
    // table에 관찰자(그래프) 등록
```

```
    PieGraph pg; t.attach(&pg);
    BarGraph bg; t.attach(&bg);
    t.edit();
}
```

기능의 분리

위 코드의 문제점은 Table 클래스 안에는 관찰자 패턴을 위한 기본 코드와 Table의 데이터를 조작하는 코드가 섞여 있다는 점입니다. 관찰자 로직에 관련된 코드를 기반 클래스로 뽑으면 재사용성을 높일수 있습니다.

```
// 관찰자 패턴의 공통의 로직은 data 구조에 상관없이 공통되어 있습니다.
// 기반 클래스로 설계되는 것이 좋습니다.
// 변하지 않은것 => 부모에
// 변하는 것    => 자식이 변경할수 있게(가상함수)
class Subject
{
    vector<IGraph*> v;
public:
    void attach(IGraph* p) { v.push_back(p); }
    void detach(IGraph* p) {}
    void notify(int* data)
    {
        for (int i = 0; i < v.size(); i++)
            v[i]->OnUpdate(data);
    }
};
class Table : public Subject
{
    int data[5];
public:
    Table() { memset(data, 0, sizeof(int) * 5); }

    void edit()
    {
        while (1)
        {
            int index;
            cout << "index >> "; cin >> index;
            cout << "data  >> "; cin >> data[index];
```

```
            notify(data);
        }
    }
};
```

Iterator

배우게 되는 내용

① Interface 기반 iterator

② Template 기반 iterator

iterator

카테고리

행위 패턴(Behavioral Pattern)

의도

복합 객체 요소들의 내부 표현 방식을 공개하지 않고도 순차적으로 접근할 수 있는 방법을 제공한다.

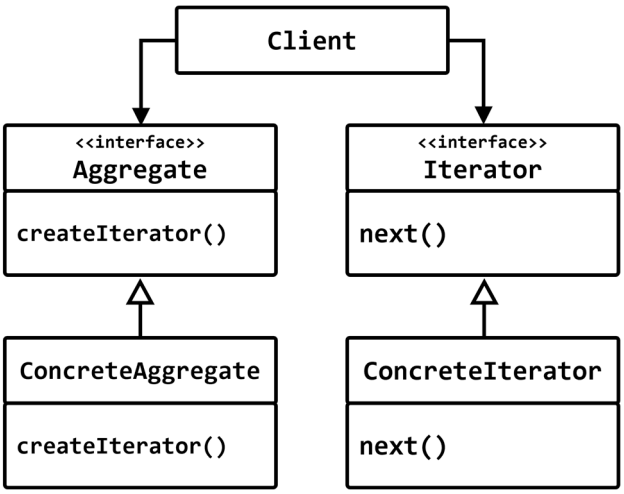
다른 이름

Cursor

활용성

- 객체 내부의 표현 방식을 모르고도 집합 객체의 각 요소들을 순회 할 수 있다.
- 집합 객체를 순회 하는 다양한 방법을 제시 할 수 있다.
- 서로 다른 집합 객체 구조에 대해서도 동일한 방법으로 순회 할 수 있다.

클래스 다이어그램 (class diagram)



결과

- 집합 객체에 대한 다양한 순회 방법을 제공한다.
- Iterator 는 Aggregate 클래스의 인터페이스를 단순화 할 수 있다.
- 집합 객체에 대해 하나 이상의 순회 방법을 정의 할 수 있다.

1. iterator

▌ 반복자(iterator) 개념

반복자는 복합객체의 내부 구조에 상관 없이 일관된 방식으로 요소를 열거할 때 사용합니다.

먼저, list와 vector 2개의 컨테이너를 생각해 봅시다.

```
#include <iostream>
using namespace std;

template<typename T> struct Node
{
    T   data;
    Node* next;
    Node(T a, Node* n) : data(a), next(n) {}
};

template<typename T> class slist
{
    Node<T>* head;
public:
    slist() : head(0) {}
    void push_front(const T& a) { head = new Node<T>(a, head); }
    T&  front() { return head->data; }
};
//----------------------------
template<typename T> class Vector
{
    T* buff;
public:
    Vector(int sz) { buff = new T[sz]; }

    T& operator[](int index) { return buff[index]; }
};
```

인터페이스 기반 반복자(iterator)

```
template<typename T> struct IEnumerator
{
    virtual bool MoveNext() = 0;
    virtual T&   GetObject() = 0;
    virtual ~IEnumerator() {}
};
template<typename T> struct Node
{
    T   data;
    Node* next;
    Node(T a, Node* n) : data(a), next(n) {}
};
// 싱글리스트의 반복자
template<typename T> class slist_enumerator : public IEnumerator < T >
{

    Node<T>* current;
public:
    slist_enumerator(Node<T>* p = 0) : current(p) {}

    // 반복자는 2개 함수가 반드시 있어야 한다.
    virtual bool MoveNext()  { current = current->next; return current; }
    virtual T&   GetObject() { return current->data; }
};
// 모든 컨테이너에서는 반복자를 꺼낼수 있어야 한다.
// 컨테이너의 인터페이스
template<typename T> struct IEnumerable
{
    virtual IEnumerator<T>* GetEnumeratorN() = 0;

    virtual ~IEnumerable<T>() {}
};
template<typename T> class slist : public IEnumerable<T>
{
    Node<T>* head;
public:
    slist() : head(0) {}

    virtual IEnumerator<T>* GetEnumeratorN()
    {
```

```
        return new slist_enumerator<T>(head);
    }
    void push_front(const T& a) { head = new Node<T>(a, head); }
    T&  front() { return head->data; }
};
// vector 반복자
template<typename T> class Vector_enumerator : public IEnumerator < T >
{
    T* current;
    T* last;
public:
    Vector_enumerator(T* p1, T* p2 ) : current(p1), last(p2) {}

    // 반복자는 2개 함수가 반드시 있어야 한다.
    virtual bool MoveNext()  { ++curent; return current != last; }
    virtual T&   GetObject() { return *current; }
};
template<typename T> class Vector : public IEnumerable<T>
{
    T* buff;
public:
    Vector(int sz) { buff = new T[sz]; }

    T& operator[](int index) { return buff[index]; }

    virtual IEnumerator<T>* GetEnumeratorN()
    {
        return new Vector_enumerator<T>(head);
    }
};
// 인자로 전달된 컨테이너의 모든 요소를 출력하는 함수.
template<typename T> void show( IEnumerator<T>* p)
{
    do
    {
        cout << p->GetObject() << endl; // 첫번째 요소 출력

    } while (p->MoveNext());
}
int main()
{
    slist<int> s;
    s.push_front(10);
    s.push_front(20);
```

```
    s.push_front(30);

    show(s.GetEnumeratorN());
}
```

Template 기반 반복자

STL에서는 템플릿 기반 반복자를 사용하고 있습니다.

```
template<typename T> struct Node
{
    T   data;
    Node* next;
    Node(T a, Node* n) : data(a), next(n) {}
};

template<typename T> class slist_iterator
{
    Node<T>* current;
public:
    slist_iterator(Node<T>* p = 0) : current(p) {}

    inline slist_iterator& operator++()
    {
        current = current->next;
        return *this;
    }
    inline T&   operator*() { return current->data; }
};

template<typename T> class slist
{
    Node<T>* head;
public:
    slist() : head(0) {}

    inline slist_iterator<T> begin()
    {
        return slist_iterator<T>(head);
    }

    void push_front(const T& a) { head = new Node<T>(a, head); }
    T&  front() { return head->data; }
};

template<typename T> void show(T p)
{
    cout << *p << endl;
    ++p;
```

```
    cout << *p << endl;
}

int main()
{
    slist<int> s;
    s.push_front(10);
    s.push_front(20);
    s.push_front(30);

    slist_iterator<int> p = s.begin();
    show(p);

}
```

일반적으로 template 기반 컨테이너는 타입의 안정성이 뛰어나고, 객체 뿐 아니라 기본 타입도 저장 할 수 있고, 꺼낼 때 캐스팅을 할 필요도 없습니다. 하지만, 너무 많은 타입으로 사용되면 코드가 커지는 문제점이 있습니다.

Visitor

배우게 되는 내용

① Visitor 패턴

visitor

카테고리

행위 패턴(Behavioral Pattern)

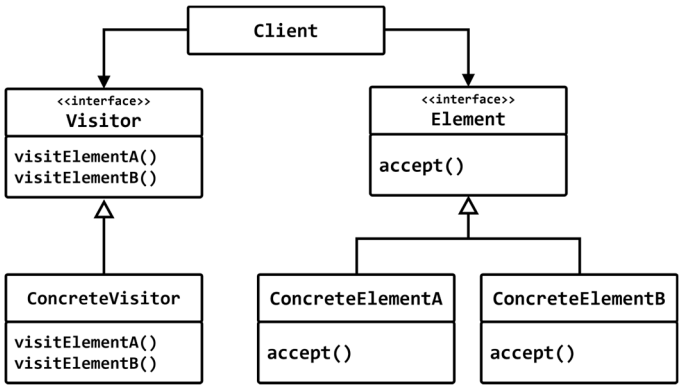
의도

객체 구조에 속한 요소에 수행될 오퍼레이션을 정의 하는 객체. Visitor 패턴은 처리되어야 하는 요소에 대한 클래스를 변경하지 않고 새로운 오퍼레이션을 정의할 수 있게 한다.

활용성

- 객체 구조가 다른 인터페이스를 가진 클래스들을 포함하고 있어서 구체적 클래스에 따라 이들 오퍼레이션을 수행하고자 할 때
- 구별되고 관리되지 않은 많은 오퍼레이션이 객체에 수행될 필요가 있지만, 오퍼레이션으로 인해 클래스들을 복잡하게 하고 싶지 않을 때.
- 객체 구조를 정의한 클래스는 거의 변하지 않지만 전체 구조에 걸쳐 새로운 오퍼레이션을 추가하고 싶을 때.

클래스 다이어그램 (class diagram)



결과

- 새로운 오퍼레이션을 쉽게 추가 한다.
- 방문자를 통해 관련된 오퍼레이션을 하나로 모아 관련되지 않은 오퍼레이션과 분리
- 새로운 ConcreteElement 를 추가하기 어렵다.
- 상태를 누적할 수 있다.
- 캡슐화 전략을 위배할 수 있다.

1. Visitor

▌ 방문자(Visitor) 패턴

복합 객체의 내부 구조에 상관없이 복합 객체의 내부요소에 연산을 수행하는 객체.

```
template<typename T> struct Node
{
    T   data;
    Node* next;
    Node(T a, Node* n) : data(a), next(n) {}
};
// 방문자의 인터페이스
template<typename T> struct IVisitor
{
    virtual void visit(T& a) = 0;
    virtual ~IVisitor() {}
};
// 모든 컨테이너는 방문자를 받아 들여야 한다. accept가 있어야 한다.
// 복합객체(컨테이너)의 인터페이스
template<typename T> struct IAcceptor
{
    virtual void accept(IVisitor<T>* p) = 0;
    virtual ~IAcceptor() {}
};
template<typename T> class slist : public IAcceptor<T>
{
    Node<T>* head;
public:
    // 아래 함수가 방문자의 핵심입니다.  잘 생각해 보세요
    virtual void accept(IVisitor<T>* p)
    {
        Node<T>* current = head;
        while (current != 0 )
        {
            p->visit(current->data); // 방문자에게 요소 전달
            current = current->next;
        }
    }
    slist() : head(0) {}
    void push_front(const T& a) { head = new Node<T>(a, head); }
    T&  front() { return head->data; }
```

```
};
// 이제 다양한 방문자를 만들면 됩니다.
template<typename T> class TwiceVisitor : public IVisitor<T>
{
public:
    virtual void visit(T& a) { a = a * 2; }
};
template<typename T> class ShowVisitor : public IVisitor<T>
{
public:
    virtual void visit(T& a) { cout << a << endl; }
};
template<typename T> class ZeroVisitor : public IVisitor<T>
{
public:
    virtual void visit(T& a) { a = 0; }
};
int main()
{
    slist<int> s;
    s.push_front(10);
    s.push_front(20);
    s.push_front(30);

    TwiceVisitor<int> tv;
    s.accept(&tv); // 모든 요소가 2배가 된다

    ZeroVisitor<int> zv;
    s.accept(&zv); // 모든 요소를 0으로 된다

    ShowVisitor<int> sv;
    s.accept(&sv); // 모든 요소 출력
}
```

메뉴 방문자

```
class PopupMenu;
class MenuItem;

struct IMenuVisitor
{
    virtual void visit(PopupMenu* p) = 0;
    virtual void visit(MenuItem * p) = 0;
    virtual ~IMenuVisitor() {}
};
struct IAcceptor
{
    virtual void accept(IMenuVisitor* p) = 0;
    virtual ~IAcceptor() {}
};
class BaseMenu : public IAcceptor
{
    string title;
public:
    BaseMenu(string s) : title(s) {}
    string getTitle() { return title; }
    void setTitle(string s) { title = s; }
    virtual void command() = 0;
};
class MenuItem : public BaseMenu
{
    int id;
public:
    void accept(IMenuVisitor* p)
    {
        p->visit(this);
    }
    MenuItem(string s, int n) : BaseMenu(s), id(n) {}

    virtual void command()
    {
        cout << getTitle() << " 메뉴 선택됨" << endl;
        getch();
    }
};
class PopupMenu : public BaseMenu
{
    vector<BaseMenu*> v;
```

```
public:
    PopupMenu(string s) : BaseMenu(s) {}
    void accept(IMenuVisitor* p)
    {
        p->visit(this);

        for (int i = 0; i < v.size(); i++)
            v[i]->accept(p);
    }
    void addMenu(BaseMenu* p) { v.push_back(p); }

    virtual void command()
    {
        while (1)
        {
            clrscr(); // 화면을 지우고
            int sz = v.size();
            for (int i = 0; i < sz; i++)
                cout << i + 1 << ". " << v[i]->getTitle() << endl;

            cout << sz + 1 << ". 상위 메뉴로" << endl;
            cout << "메뉴를 선택하세요 >> ";
            int cmd;
            cin >> cmd;
            if (cmd < 1 || cmd > sz + 1)
                continue;
            if (cmd == sz + 1)
                break;
            v[cmd - 1]->command();
        }
    }
};
//메뉴의 타이틀을 수정하는 방문자
class MenuTitleDecorator : public IMenuVisitor
{
public:
    virtual void visit(MenuItem* p) {} // MenuItem은 타이틀을 변경안함
    virtual void visit(PopupMenu* p)
    {
        string s = p->getTitle();
        s = s + " >> ";
        p->setTitle(s); // 타이틀을 수정한다.
    }
};
```

```
int main()
{
    PopupMenu* menubar = new PopupMenu("MenuBar");
    PopupMenu* p1 = new PopupMenu("화면 설정");
    PopupMenu* p2 = new PopupMenu("소리 설정");

    menubar->addMenu(p1);
    menubar->addMenu(p2);

    p1->addMenu(new MenuItem("해상도 변경", 11));
    p1->addMenu(new MenuItem("색상   변경", 12));
    p1->addMenu(new MenuItem("명도   변경", 13));
    p2->addMenu(new MenuItem("크기   변경", 21));
    p2->addMenu(new MenuItem("음색   변경", 22));


    // 방문자가 방문하면 멤버 data가 수정되는 경우가 많다.
    // 캡슐화 관점에서는 좋지 않은 디자인 이다
    // setTitle()등의 함수가 필요하게 된다.
    MenuTitleDecorator mtd;
    menubar->accept(&mtd);


    // 이제 시작하려면 ??
    menubar->command();
}
```

flyweight

카테고리

구조 패턴(Structural Pattern)

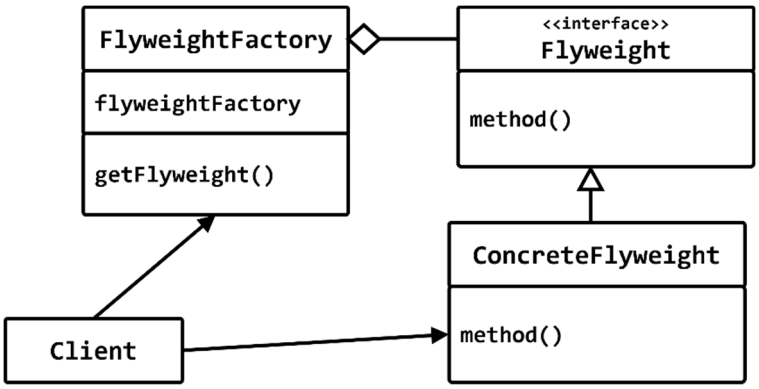
의도

작은 크기의 객체들이 여러 개 있는 경우, 객체를 효과적으로 사용하는 방법으로 객체를 공유하게 한다.

활용성

- 어플리케이션이 대량의 객체를 사용해야 할 때
- 객체의 수가 너무 많아져 저장 비용이 너무 높아 질 때
- 대부분의 객체 상태를 부가적인 것으로 만들 수 있을 때
- 부가적인 속성들을 제거한 후 객체들을 조사해 보니, 객체의 많은 묶음이 비교적 적은 수의 공유된 객체로 대체될 수 있을 때
- 어플리케이션이 객체 식별자에 비중을 두지 않은 경우

클래스 다이어그램 (class diagram)



결과

- 공유해야 하는 인스턴스의 전체 수를 줄일 수 있다.
- 객체별 본질적 상태의 양을 줄일 수 있다.
- 부가적인 상태는 연산 되거나 저장될 수 있다.

Image 클래스

아래 코드는 동일 Image 를 2번 load 하게 되는 코드 입니다.

```
class Image
{
    std::string image_url;
public:
    Image(std::string url) : image_url(url)
    {
        std::cout << url << " Downloading..." << std::endl;
    }
    void Draw() { std::cout << "Draw " << image_url << std::endl; }
};

int main()
{
    Image* img1 = new Image("www.naver.com/a.png");
    img1->Draw();

    Image* img2 = new Image("www.naver.com/a.png");
    img2->Draw();

}
```

using static member function

아래 코드는 static member function 을 사용해서 동일한 그림의 경우 공유하게 하는 코드입니다.

```
class Image
{
    std::string image_url;

    Image(std::string url) : image_url(url)
    {
        std::cout << url << " Downloading..." << std::endl;
    }

public:
    void Draw() { std::cout << "Draw " << image_url << std::endl; }

    // Image 객체를 만드는 static 멤버 함수
    // => 이제 객체는 아래 함수를 통해서만 만들수 있습니다.
    // => 객체를 "한곳"에서만 만든다는 것은 생성된 모든 객체를 "관리" 할수 있는
    //    기회가 생기게 됩니다.

    static std::map<std::string, Image*> image_map;

    static Image* Create(const std::string& url)
    {
        if (image_map[url] == nullptr)
        {
            image_map[url] = new Image(url);
        }
        return image_map[url];
    }
};
std::map<std::string, Image*> Image::image_map;

int main()
{
    Image* img1 = Image::Create("www.naver.com/a.png");
    img1->Draw();
    Image* img2 = Image::Create("www.naver.com/a.png");
    img2->Draw();

    std::cout << img1 << std::endl;
    std::cout << img2 << std::endl;
}
```

using factory

아래 코드는 factory class 을 사용해서 동일한 그림의 경우 공유하게 하는 코드입니다.

```
class Image
{
    std::string image_url;
    Image(std::string url) : image_url(url)
    {
        std::cout << url << " Downloading..." << std::endl;
    }
public:
    void Draw() { std::cout << "Draw " << image_url << std::endl; }

    friend class ImageFactory;
};
class ImageFactory
{
    MAKE_SINGLETON(ImageFactory)
    std::map<std::string, Image*> image_map;
public:
    Image* Create(const std::string& url)
    {
        Image* img = nullptr;

        auto ret = image_map.find(url);

        if (ret == image_map.end())
        {
            img = new Image(url);
            image_map[url] = img;
        }
        else
            img = ret->second;  ..
        return img;
    }
};
int main()
{
    ImageFactory& factory = ImageFactory::getInstance();

    Image* img1 = factory.Create("www.naver.com/a.png");
    img1->Draw();

    Image* img2 = factory.Create("www.naver.com/a.png");
    img2->Draw();

    std::cout << img1 << std::endl;
    std::cout << img2 << std::endl;
}
```

Section 6.

객체 생성

객체를 생성하는 방법.

배우게 되는 내용

① new

② static member function

③ Factory class

④ Clone

1. 객체를 생성하는 방법

C++에서 객체를 생성하는 방법

C++에서는 다양한 방법으로 객체를 생성할 수 있습니다.

- new 사용
- static member function 사용
- factory class 사용
- prototype

new 사용

객체를 만드는 가장 자유로운 방법 입니다.

```
class Shape
{
public:
    virtual ~IShape() {}
};
class Rect : public Shape
{
};
class Circle : public Shape
{
};
int main()
{
    Shape* p = new Rect;
    delete p;
}
```

new 는 객체를 만드는 가장 자유로운 방법입니다. 하지만 생성되는 객체의 개수를 5 개 까지로 제한하고 싶다면 어떻게 해야 할까요 ? 객체의 생성 과정에 어떤 제한을 두고 싶다면 객체를 자유롭게 만들기 보다는 한 곳에서 만드는 것이 좋을 때가 있습니다.

▌ Static 멤버 함수를 사용한 객체 생성.

Static 멤버 함수를 사용해서 객체를 생성하면 객체의 생성을 한곳에 집중할 수 있습니다.

```
class Rect : public Shape
{
public:
    Shape * Create() { return new Rect; }
};

int main()
{
    Shape* p = Rect::Create();
}
```

이 경우 객체를 한곳에서만 생성하므로 개수를 제한 하는 등의 작업을 수행할 수 있습니다.

또한, 다른 함수의 인자로 객체의 생성 정보를 전달할 수 있습니다.

```
void foo(Shape(*f)())
{
    // 함수 인자로 전달된 생성 함수를 통해서
    // 객체가 생성됩니다.
    f();
}

int main()
{
    foo(&Rect::Create);
    foo(&Circle::Create);
}
```

Factory 클래스를 사용한 객체 생성.

객체를 생성하는 전용 클래스(Factory)를 제공하므로서 객체의 본연의 기능과 객체 생성의 기능을 분리 할 수 있습니다.

```
class ShapeFactory
{
public:
    Shape * CreateShape(int type)
    {
        Shape* p = 0;
        switch (type)
        {
        case 1: p = new Rect; break;
        case 2: p = new Circle; break;
        }
        return p;
    }
};
int main()
{
    // 도형을 만드는 공장을 생성합니다.
    ShapeFactory factory;
    Shape* p = factory.CreateShape(1);
}
```

이와 같은 factory class 의 장점은 뒷장에서 다루게 됩니다.

복사에 의한 객체 생성.

기존에 존재하는 객체의 복사본을 통해서도 객체를 생성 할 수 있습니다. “Prototype” 이라고 불리는 기법입니다.

```
class Shape
{
public:
    virtual ~IShape() {}
    virtual Shape* Clone() = 0;
};
```

```
class Rect : public Shape
{
public:
    virtual Shape* Clone() { return Rect(*this); }
};

class Circle : public Shape
{
public:
    virtual Shape* Clone() { return Circle(*this); }
};

int main()
{
    Rect* blueRect = new Rect;

    Shape* p = blueRect->Clone();
}
```

Singleton

배우게 되는 내용

① Mayer Singleton

② Android Singleton

singleton

카테고리

생성 패턴(Creational Pattern)

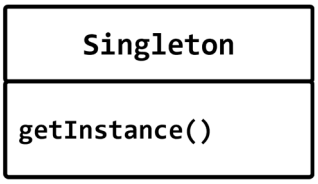
의도

클래스의 인스턴스는 오직 하나임을 보장하며 이에 대한 접근은 어디 에서 든지 하나로만 통일하여 제공한다.

활용성

- 클래스의 인스턴스가 오직 하나여야 함을 보장하고, 잘 정의된 접근 방식에 의해 모든 클라이언트가 접근할 수 있도록 해야 할 때.
- 유일하게 존재하는 인스턴스가 상속에 의해 확장 되어야 할 때, 클라이언트는 코드의 수정 없이 확장된 서브 클래스의 인스턴스를 사용 할 수 있어야 할 때

클래스 다이어그램 (class diagram)



결과

- 유일하게 존재하는 인스턴스로의 접근을 통제할 수 있다.
- 변수 영역을 줄인다
- 오퍼레이션의 정제를 가능하게 한다.
- 인스턴스의 개수를 변경하기가 자유롭다
- 클래스 오퍼레이션을 사용하는 것보다 훨씬 유연한 방법이다. (Mono-State Pattern)
  - C++의 정적 멤버 함수는 가상 함수 일 수 없다.

1. Singleton

Mayer Singleton

유일한 하나의 객체를 static 지역변수로 만드는 싱글톤

```
class Cursor
{
private:
    Cursor() {}

public:
    static Cursor& getInstance()
    {
        static Cursor instance;
        return instance;
    }
private:
    Cursor(const Cursor& c) = delete;
    void operator=(const Cursor&c) = delete;
};

int main()
{
    Cursor& c1 = Cursor::getInstance();
    Cursor& c2 = Cursor::getInstance();

    cout << &c1 << ", " << &c2 << endl;
}
```

싱글톤 매크로 코드

싱글톤의 재사용을 위한 기법

```
#define MAKE_SINGLETON(classname)                   \
    private:                                        \
        classname() {}                              \
        classname(const classname&)      = delete;  \
        void operator=(const classname&) = delete;  \
```

```
    public:                                             \
        static classname&   getInstance()               \
        {                                               \
            static classname instance;                  \
            return instance;                            \
        }

class Test
{
    MAKE_SINGLETON(Test)
};
int main()
{
    Test& t = Test::getInstance();
}
```

Android Framework 싱글톤

CRTP 로 구현된 힙에 만들어진 싱글톤

```
class Mutex
{
public:
    void Lock()   { cout << "Mutex Lock" << endl; }
    void Unlock() { cout << "Mutex Unlock" << endl; }

    class Autolock
    {
        Mutex& mLock;
    public:
        inline Autolock(Mutex& m) : mLock(m) { mLock.Lock(); }
        inline ~Autolock()                   { mLock.Unlock(); }
    };
};

template<typename TYPE > class Singleton
{
protected:
    Singleton() {}

private:
```

```
    Singleton(const Singleton& c);
    void operator=(const Singleton&c);

    static TYPE* sInstance;
    static Mutex sLock;
public:
    static TYPE& getInstance()
    {
        Mutex::Autolock a(sLock);
        if (sInstance == 0)
            sInstance = new TYPE;
        return *sInstance;
    }
};
template<typename TYPE> TYPE* Singleton<TYPE>::sInstance = 0;
template<typename TYPE> Mutex Singleton<TYPE>::sLock;


class Mouse : public Singleton<Mouse>
{
};

int main()
{
    Mouse& m = Mouse::getInstance();
}
```

도형편집기와 Factory

배우게 되는 내용

① 도형을 만드는 Factory

② Create method 과 factory

③ Prototype Factory

1. 도형 편집기와 Factory

Factory class 도입

- Factory 클래스를 통해서 도형을 생성합니다.
- 새로운 도형 추가시 코드 변화를 최소화 할 수 있습니다.
- 대부분의 Factory는 singleton 패턴을 사용하는 경우가 많습니다.

```
class Shape {};
class Rect   : public Shape {};
class Circle : public Shape {};

// 도형을 만드는 공장을 만든다.
class ShapeFactory
{
    MAKE_SINGLETON(classname)
public:
    Shape* CreateShape(int type)
    {
        Shape* p = 0;
        switch (type)
        {
        case 1: p = new Rect;   break;
        case 2: p = new Circle; break;
        }
        return p;
    }
};

int main()
{
    ShapeFactory factory = ShapeFactory::getInstance();

    vector<Shape*> v;
    while (1)
    {
        int cmd;
        cin >> cmd;

        if (cmd > 0 && cmd < 5)
```

```
        {
            Shape* p = factory.CreateShape(cmd);
            if (p)
                v.push_back(p);
        }
    }
}
```

Create 정적 함수와 Factory class

- Factory class를 생성하는 정적 함수를 등록합니다.
- Factory class 와 각 도형 클래스 사이에 결합도가 느슨해 집니다.

```
#include <map>

class Shape {};
class Rect   : public Shape
{
public:
    static Shape* Create() { return new Rect; }
};
class Circle : public Shape
{
public:
    static Shape* Create() { return new Circle; }
};


class ShapeFactory
{
    typedef Shape*(*CREATOR)();

    map<int, CREATOR> createmap;
public:
    void Register(int type, CREATOR c)
    {
        createmap[type] = c;
    }
    Shape * CreateShape(int type)
    {
```

```
        Shape* p = 0;

        if (createmap[type] != 0)  // createmap.find()로 찾는 것이 좋습니다
            p = createmap[type]();

        return p;
    }
};
int main()
{
    ShapeFactory factory;

    // 공장에 제품을 등록 한다.
    factory.Register(1, &Rect::Create);
    factory.Register(2, &Circle::Create);


    vector<Shape*> v;
    while (1)
    {
        int cmd;
        cin >> cmd;

        if (cmd > 0 && cmd < 5)
        {
            Shape* p = factory.CreateShape(cmd);
            if (p)
                v.push_back(p);
        }
    }
}
```

▌공장에 제품을 자동 등록

- 간단한 테크닉과 매크로 기술을 사용하면 도형을 공장에 등록하는 과정을 자동화 할 수 있습니다.

```
// 공장에 제품을 등록해주는 클래스
struct RegisterShape
{
    RegisterShape( int type, Shape*(*f)() )
```

```
    {
        ShapeFactory& factory = ShapeFactory::getInstance();

        factory.Register(type, f);
    }
};

// 모든 도형이 지켜야 하는 규칙을 담은 매크로를 제공한다.

#define DECLARE_SHAPE(classname)                                    \
            static Shape* Create() { return new classname;}         \
            static RegisterShape rs;

#define IMPLEMENT_SHAPE(key, classname)                             \
        RegisterShape classname::rs(key, &classname::Create);

class Rect : public Shape
{
public:
    DECLARE_SHAPE(Rect)
};
IMPLEMENT_SHAPE(1, Rect)

class Circle : public Shape
{
public:
    DECLARE_SHAPE(Circle)
};
IMPLEMENT_SHAPE(2, Circle)
```

prototype

카테고리

생성 패턴(Creational Pattern)

의도

견본적(prototypical) 인스턴스를 사용하여 생성할 객체의 종류를 명시하고 이렇게 만들어진 견본을 복사하여 새로운 객체를 생성한다.

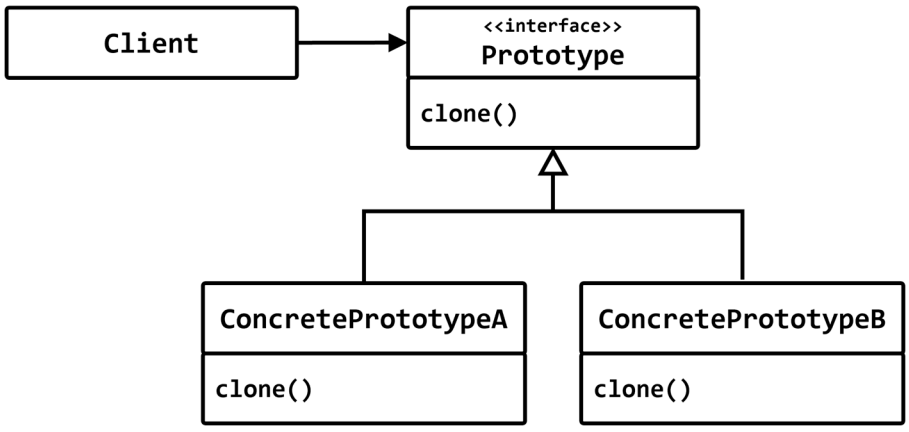
다른 이름

가상 생성자 (Virtual Constructor)

활용성

- 런 타임시 만들 인스턴스의 클래스를 명세할 수 있을 때
- 클래스 계층도와 병렬성을 갖는 팩토리 클래스의 계층을 피해야 할 경우
- 클래스의 인스턴스들 이 서로 다른 상태 조합중에 어느 하나를 가질 때 사용한다.

클래스 다이어그램 (class diagram)



결과

- 실행시간에 새로운 제품을 삽입하고 삭제 할 수 있다.
- 값들을 다양화함으로써 새로운 객체를 명세 한다.
- 구조를 다양화함으로써 새로운 객체를 정의 할 수 있다.

- 서브 클래스의 수를 줄인다.
- 동적으로 생성된 클래스에 따라 애플리케이션을 형성할 수 있다.

Prototype을 사용한 Factory

- 공장에 생성 함수를 등록하지 말고, 견본품을 등록합니다.
- 사각형이라도 “빨간색 크기가 5인 사각형”, “파란색 사각형” 등 자주 사용하는 객체를 등록해 등록해두 필요할 때 복사에 의해서 객체를 생성할 수 있습니다.

```
class Shape
{
public:
    virtual Shape* Clone() = 0;
};
class Rect : public Shape
{
public:
    static Shape* Create() { return new Rect; }
};
class Circle : public Shape
{
public:
    virtual Shape* Clone() { return new Circle(*this); }
};

class ShapeFactory
{
    map<int, Shape*> prototype_map;
public:
    void Register(int type, Shape* c)
    {
        prototype_map[type] = c;
    }
    Shape* CreateShape(int type)
    {
        Shape* p = 0;

        if (prototype_map[type] != 0)
            p = prototype_map[type]->Clone();

        return p;
    }
};

int main()
{
```

```
    ShapeFactory factory;

    // 자주 사용하는 도형이 견본을 공장에 등록한다.
    //-------------------------------------
    Rect* r1 = new Rect; // 파란색 크기 5
    Rect* r2 = new Rect; // 파란색 크기 15
    Circle* r3 = new Circle; // 파란색 크기 5

    factory.Register(1, r1);
    factory.Register(2, r2);
    factory.Register(3, r3);
    //-------------------------------------
    vector<Shape*> v;
    while (1)
    {
        int cmd;
        cin >> cmd;

        if (cmd > 0 && cmd < 5)
        {
            Shape* p = factory.CreateShape(cmd);
            if (p)
                v.push_back(p);
        }
    }
}
```

Abstract Factory

배우게 되는 내용

① Factory의 interface

② 제품의 군을 생성하기 위한 인터페이스

abstract factory

카테고리

생성 패턴(Creational Pattern)

의도

상세 화된 서브 클래스를 정의 하지 않고도 서로 관련성이 있거나 독립적인 여러 객체의 군을 생성하기 위한 인터페이스를 제공한다.

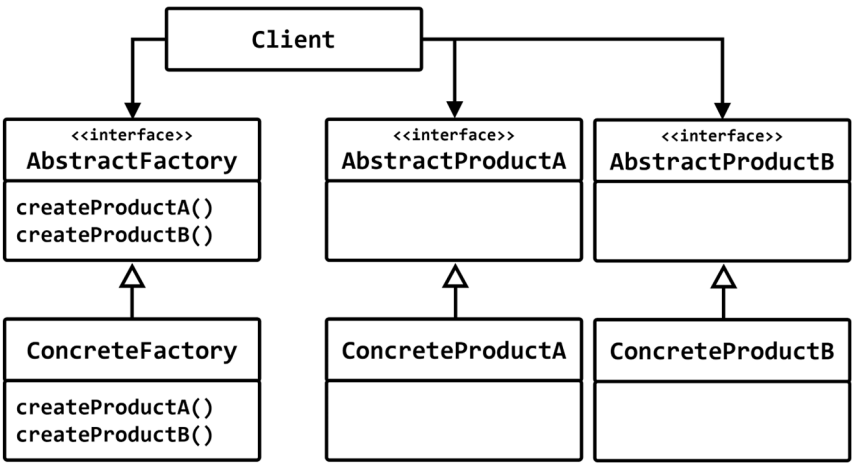
다른 이름

kit

활용성

- 생성되고 구성되고 표현되는 방식과 무관하게 시스템을 독립적으로 만들고자 할 때
- 하나 이상의 제품군들 중 하나를 선택해서 시스템을 설정해야 하고 한번 구성한 제품을 다른 것으로 대체 할 수 있을 때
- 관련된 객체군을 함께 사용해서 시스템을 설계하고, 이 제품이 갖는 제약 사항을 따라야 할 때
- 제품에 대한 클래스 라이브러리를 제공하고, 그들의 구현이 아닌 인터페이스를 표현하고 싶을 때

클래스 다이어그램 (class diagram)



결과

- 구체적인 클래스를 분리 한다.
- 제품군 을 쉽게 대체할 수 있도록 한다
- 제품 간의 일관성을 증진한다.
- 새로운 종류의 제품을 제공하기 어렵다.

1. Abstract Factory 개념

Style 옵션에 따라 서로 다른 모양으로 나타나는 GUI 라이브러리

- 각 스타일에 따른 Widget 클래스가 필요 합니다.
- 각 Widget의 인터페이스가 있어야 합니다.

```
struct IButton
{
    virtual void Draw() = 0;
    virtual ~IButton() {}
};
struct IEdit
{
    virtual void Draw() = 0;
    virtual ~IEdit() {}
};
struct WinButton : public IButton
{
    void Draw() { cout << "WinButton Draw" << endl; }
};
struct WinEdit : public IEdit
{
    void Draw() { cout << "WinEdit Draw" << endl; }
};
//-----------------------------------------------------
struct GTKButton : public IButton
{
    void Draw() { cout << "GTKButton Draw" << endl; }
};
struct GTKEdit : public IEdit
{
    void Draw() { cout << "GTKEdit Draw" << endl; }
};
int main(int argc, char** argv)
{
    IButton* pBtn = 0;

    if (strcmp(argv[1], "GTK") == 0)
        pBtn = new GTKButton;
    else
```

```
        pBtn = new WinButton;

    pBtn->Draw();
}
```

그런데, 위 코드에서 만약 버튼을 여러 번 만들어야 한다면 어떻게 해야 할까요 ?

Widget를 만드는 Factory

- 각 스타일에 따른 Widget 클래스을 만드는 Factory를 제공합니다.

```
// 각 스타일의 컨트롤을 만드는 공장이 필요하다.
struct WinFactory
{
    IButton* CreateButton() { return new WinButton; }
    IEdit*   CreateEdit()   { return new WinEdit; }
};
struct GTKFactory
{
    IButton* CreateButton() { return new GTKButton; }
    IEdit*   CreateEdit()   { return new GTKEdit; }
};
int main(int argc, char** argv)
{
    ?*  factory = 0;

    if (strcmp(argv[1], "GTK") == 0)
        factory = new GTKFactory;
    else
        factory = new WinFactory;
}
```

위 코드에서 ? 에는 어떤 타입을 사용해야 할까요 ?

Factory 의 인터페이스

- Factory 도 인터페이스를 설계 합니다.

```
struct IFactory
{
    virtual IButton* CreateButton() = 0;
    virtual IEdit*   CreateEdit() = 0;
};

// 각 스타일의 컨트롤을 만드는 공장이 필요하다.
struct WinFactory : public IFactory
{
    //……
};
struct GTKFactory : public IFactory
{
    //……
};
int main(int argc, char** argv)
{
    IFactory* factory = 0;

    if (strcmp(argv[1], "GTK") == 0)
        factory = new GTKFactory;
    else
        factory = new WinFactory;

    IButton* pBtn = factory->CreateButton();
    pBtn->Draw();
}
```

Factory Method

배우게 되는 내용

① 객체를 생성하기 위한 인터페이스

factory method

카테고리

생성 패턴(Creational Pattern)

의도

객체를 생성하기 위해 인터페이스를 정의 하지만, 어떤 클래스의 인스턴스를 생성할 지에 대한 결정은 서브 클래스가 한다. Factory Method 패턴에서는 클래스의 인스턴스를 만드는 시점을 서브클래스로 미룬다.

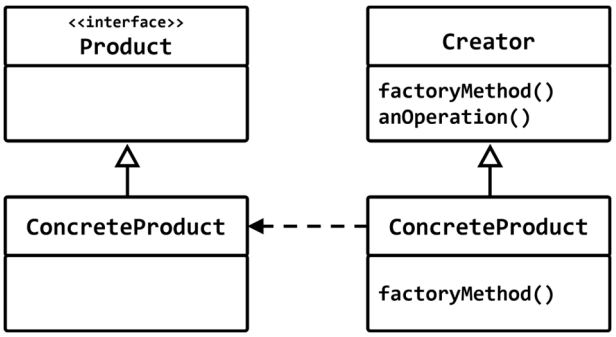
다른 이름

가상 생성자 (Virtual Constructor)

활용성

- 생성할 객체 타입을 예측할 수 없을 때
- 생성할 객체를 기술하는 책임을 서브클래스에 정의하고자 하는 경우
- 객체 생성의 책임을 서브클래스에 위임 시키고 서브 클래스에 대한 정보를 은닉하고자 하는 경우

클래스 다이어그램 (class diagram)



결과

- 서브 클래스에 대한 훅(hook) 메소드를 제공한다.
- 병렬적 클래스 계층도를 연결하는 역할을 담당한다.

1. Factory Method

▌Style 옵션에 관계 없이 항상 특정 스타일로 보이는 Dialog 클래스

- WinDialog 와 GTKDialog

```
class WinDialog
{
public:
    void Init()
    {
        WinButton* btn  = new WinButton;
        WinEdit*   edit = new WinEdit;

        // btn->move(10, 10);   edit->move(20, 20);

        btn->Draw();
        edit->Draw();
    }
};
class GTKDialog
{
public:
    void Init()
    {
        GTKButton* btn  = new GTKButton;
        GTKEdit*   edit = new GTKEdit;

        // btn->move(10, 10);   edit->move(20, 20);

        btn->Draw();
        edit->Draw();
    }
};
```

WinDialog 와 GTKDialog는 유사한 코드가 많이 있습니다. 공통의 기반 클래스를 만들어서 유사한 코드를 기반 클래스에 넣으면 어떨까요 ?

동일한 코드는 기반 클래스에서 제공

- WinDialog 와 GTKDialog의 공통의 특징을 기반 클래스에 놓습니다.
- 변해야 하는 부분을 가상함수로 합니다.

```
class BaseDialog
{
public:
    void Init()
    {
        IButton* btn = CreateButton();
        IEdit*   edit = CreateEdit();
        // btn->move(10, 10);   edit->move(20, 20);
        btn->Draw();
        edit->Draw();
    }
    // 객체를 생성하기 위한 인터페이스
    virtual IButton* CreateButton() = 0;
    virtual IEdit*   CreateEdit() = 0;
};
// 스타일 옵션에 상관없이 항상 XP 모양의 Dialog
class WinDialog : public BaseDialog
{
public:
    virtual IButton* CreateButton() { return new WinButton;}
    virtual IEdit*   CreateEdit()   { return new WinEdit; }
};
class GTKDialog : public BaseDialog
{
public:
    virtual IButton* CreateButton() { return new GTKButton; }
    virtual IEdit*   CreateEdit()   { return new GTKEdit; }
};
```

WinDialog 와 GTKDialog는 유사한 코드가 많이 있습니다. 공통의 기반 클래스를 만들어서 유사한 코드를 기반 클래스에 넣으면 어떨까요 ?

Factory Method

- 객체를 생성하기 위한 인터페이스를 제공하지만 어떤 객체를 생성할지는 파생 클래스에서 결정합니다.
- Factory Method는 객체의 생성을 파생 클래스에서 결정하게 합니다.
- BaseDialog 의 CreateButton() 함수.

Builder

배우게 되는 내용

① builder

builder

카테고리

생성 패턴(Creational Pattern)

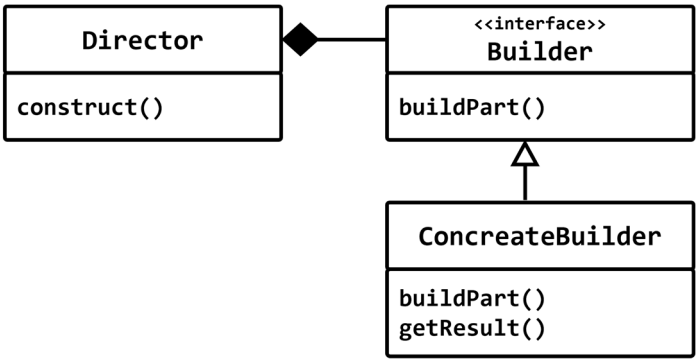
의도

복잡한 객체를 생성하는 방법과 표현하는 방법을 정의하는 클래스를 별도로 분리하여 서로 다른 표현이라도 이를 생성할 수 있는 동일한 구축 공정을 제공할 수 있도록 한다.

활용성

- 복합 객체의 생성 알고리즘이 이를 합성하는 요소 객체들이 무엇인지 이들의 조립 방법에 독립적일 때
- 합성할 객체들의 표현이 서로 다르더라도 구축 공정이 이를 지원해야 할 때

클래스 다이어그램 (class diagram)



결과

- 제품에 대한 내부 표현을 다양하게 변화할 수 있다.
- 생성과 표현에 필요한 코드를 분리 한다.
- 복합 객체를 생성하는 공정을 좀더 세밀하게 나눌 수 있다.

1. Builder

축구 게임을 생각해 봅시다. 축구 게임은 게임 시작 시 나라를 선택 할 수 있습니다. 선택한 나라에 따라 유니폼의 색상이 달라 지게 됩니다. 하지만, 모든 나라의 선수는 유니폼을 입고, 축구화를 신고 있다는 사실은 동일합니다.
결국, 축구게임의 선수 하나하나가 객체라고 할 때 사람을 만들고, 유니폼을 만들고, 신발을 만들어서 사람에게 입히는 과정은 모두 동일합니다. 하지만, 유니폼의 색상, 축구화의 모양은 나라 별로 다르게 됩니다.
이런 경우 에도 역시 전략 패턴이나, 상태 패턴과 같이 변하는 것을 다른 클래스로 분리하면 편리하게 만들수 있습니다.

Builder

빌더 패턴은 동일한 구축 공정으로 객체를 만들지만 각 공정 에 따른 표현이 달라지는 객체를 만들 때 사용하는 설계 방법입니다

캐릭터를 만드는 동일한 공정은 감독관(Director)에 있고, 각 공정에서 따른 다른 표현의 객체는 빌더에서 생성합니다.

```
#include <iostream>
#include <string>
using namespace std;

typedef string Hat;
typedef string Uniform;
typedef string Shoes;
typedef string Character;

// 각 Player를 만들때 사용할 인터페이스
struct IPlayerBuilder
{
    virtual void makeHat() = 0;
    virtual void makeUniform() = 0;
    virtual void makeShoes() = 0;
    virtual Character getResult() = 0;
    virtual ~IPlayerBuilder(){}
};

// 모든 캐릭터는 만드는 과정은  동일합니다.
// 동일한 공정은 담은 클래스(GOF 패턴에서는 Director, 감독관이라고 합니다.)

class Director
{
```

```
    IPlayerBuilder* builder;
public:
    void setBuilder(IPlayerBuilder* p) { builder = p; }

    Character construct()
    {
        // 이곳에서 동일공정으로 객체를 만듭니다.
        builder->makeHat();
        builder->makeUniform();
        builder->makeShoes();

        return builder->getResult();
    }
};

// 각 팀 별로 다른 캐릭터를 만드는 빌더를 제공합니다.

class Korean : public IPlayerBuilder
{
    Character c;
public:
    void makeHat()     { c += "갓\n"; }
    void makeUniform() { c += "한복\n"; }
    void makeShoes()   { c += "짚신\n"; }

    Character getResult() { return c; }
};

class American : public IPlayerBuilder
{
    Character c;
public:
    void makeHat()     { c += "야구모자\n"; }
    void makeUniform() { c += "양복\n"; }
    void makeShoes()   { c += "구두\n"; }
    Character getResult() { return c; }
};

int main()
{
    American a;
    Korean   k;

    Director d;
```

```
    d.setBuilder(&k); // 팀 선택 화살표를 누를 때 마다 이 함수 호출!!

    Character c = d.construct();
    cout << c << endl;
};
```

Section 7.

MISC

command

카테고리

행위 패턴(Behavioral Pattern)

의도

요청을 객체로 캡슐화 함으로써 서로 다른 요청으로 클라이언트를 파라미터화 하고, 요청을 저장하거나 기록을 남겨서 오퍼레이션의 취소도 가능하게 한다.

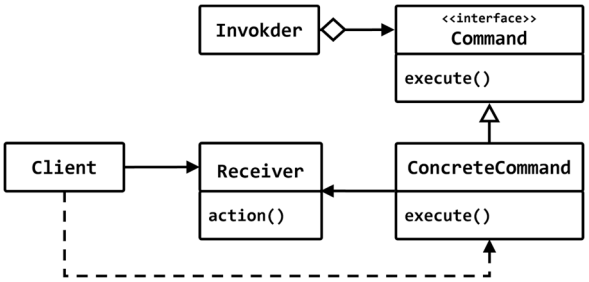
다른 이름

Action, Transaction

활용성

- 수행할 행위 자체를 객체로 파라미터화 하고자 할 때. 절차 지향 프로그램의 Callback 함수 개념
- 서로 다른 시간에 요청을 명시하고, 저장하고, 수행할 수 있다.
- 명령어 객체 자체에 실행 취소에 필요한 상태를 저장할 수 있다.
- 변경 과정에 대한 로그를 남겨 두면 시스템이 고장 났을 때 원상태로 복구가 가능하다.
- 기본적인 오퍼레이션의 조합으로 좀더 고난위도의 오퍼레이션을 구성할 수 있다

클래스 다이어그램 (class diagram)



결과

- Command 는 오퍼레이션을 호출하는 객체와 오퍼레이션을 수행방법을 구현하는 객체를 분리한다.
- Command 자체도 클래스로서 다른 객체와 같은 방식으로 조작되고 확장할 수 있다.
- 명령어를 조합해서 다른 명령어를 만들 수 있다.
- 새로운 명령을 추가하기 쉽다.

1. Command

도형 편집기 예제에 Undo/Redo 기능을 추가하려면 메뉴 선택시 수행하는 모든 기능을 저장했다가 복구 가능해야 합니다.
이 와 같은 "요청을 캡슐화해서 저장/복구"가 가능하게 하는 패턴을 Command 패턴이라고 합니다.

예제

도형편집기 예제에 Undo 기능 추가

```
// 프로그램에서 사용하는 모든 명령을 캡슐화 한다.
struct ICommand
{
    virtual void Execute() = 0;
    virtual void Undo() {}
    virtual bool CanUndo() { return false; }
    virtual ~ICommand() {}
};

class AddCommand : public ICommand
{
    std::vector<Shape*>& v;
public:
    AddCommand(std::vector<Shape*>& v) : v(v) {}

    void Execute() override { v.push_back(CreateShape()); }
    bool CanUndo() override { return true; }
    void Undo() override
    {
        Shape* p = v.back();
        v.pop_back();
        delete p;
    }

    virtual Shape* CreateShape() = 0;
};
class AddRectCommand : public AddCommand
{
public:
    using AddCommand::AddCommand;
    Shape* CreateShape() override { return new Rect; }
```

```
};

class AddCircleCommand : public AddCommand
{
public:
    using AddCommand::AddCommand;
    Shape* CreateShape() override { return new Circle; }
};

class DrawCommand : public ICommand
{
    std::vector<Shape*>& v;
public:
    DrawCommand(std::vector<Shape*>& v) : v(v) {}

    void Execute() override
    {
        for (auto p : v) p->Draw();
    }

    bool CanUndo() override { return true; }
    void Undo() override
    {
        system("cls");
    }
};
```

Composite 패턴을 적용한 MacroCommand 만들기

도형편집기 예제에 Undo 기능 추가

```
class MacroCommand : public ICommand // Composite!!
{
    std::vector<ICommand*> v;
public:
    void addCommand(ICommand* p) { v.push_back(p); }
    void Execute()
    {
        for (auto p : v)
            p->Execute();
```

```
    }
};
```

memento

카테고리

행위 패턴(Behavioral Pattern)

의도

캡슐화를 위해 하지 않고 객체 내부 상태를 캡슐화 하여, 나중에 객체가 이 상태로 복구 가능하게 한다.

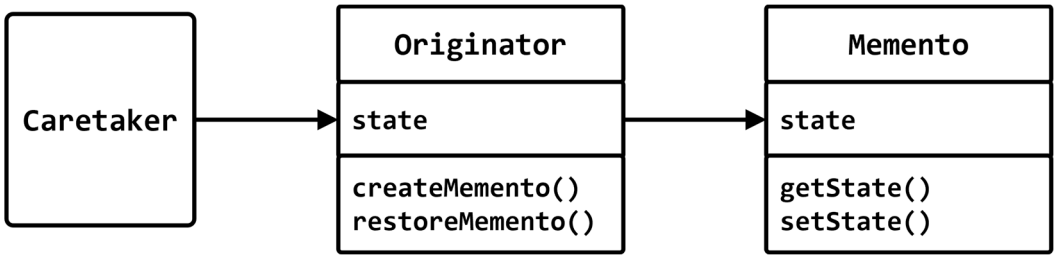
다른 이름

Token

활용성

- 각 시점에서의 객체 상태를 저장한 후 나중에 이 상태로 복구해야 할 때
- 상태를 얻는데 필요한 직접 인터페이스는 객체의 자세한 구현 내용을 드러나게 하고 객체의 캡슐화를 위배하는 것이므로, 이를 해결할 때.

클래스 다이어그램 (class diagram)



결과

- 캡슐화된 경계를 유지 할 수 있다
- Originator 클래스를 단순화 할 수 있다.
- 메멘토의 사용으로 더 많은 비용을 들여야 할 수도 있다.
- 좁은 범위의 인터페이스와 넓은 범위의 인터페이스를 정의 해야 한다.
- 메멘토를 다루기 위해 감추어진 비용이 존재 한다.

2. Memento

객체의 상태를 저장/복구 할수 있게 하는 패턴을 “Memento” 패턴이라고 합니다.

예제

상태 저장이 가능한 Graphics 클래스

```
#include <iostream>
#include <map>
#include <vector>
using namespace std;

// 객체의 캡슐화를 위배 하지 않으면서
// 객체의 상태를 저장했다가 나중에 복구 할수 있도록 한다.
class Graphics
{
    int penWidth = 1;
    int penColor = 0;
    int temporary_data;

    struct Memento
    {
        int penWidth;
        int penColor;
        Memento(int w, int c) : penWidth(w), penColor(c) {}
    };

    std::map<int, Memento*> state_map;
public:
    int Save()
    {
        static int key = 0;
        ++key;
        Memento* p = new Memento(penWidth, penColor);
        state_map[key] = p;

        return key;
    }
    void Restore(int token)
    {
        penColor = state_map[token]->penColor;
```

```
        penWidth = state_map[token]->penWidth;
    }

    void DrawLine(int x1, int y1, int x2, int y2)
    {
    }
    void SetStrokeColor(int c) { penColor = c; }
    void SetStrokeWidth(int w) { penWidth = w; }
};

int main()
{
    Graphics g;

    g.SetStrokeColor(0);
    g.SetStrokeWidth(10);
    g.DrawLine(0, 0, 100, 100);
    g.DrawLine(0, 0, 200, 200);
    int token = g.Save(); // 객체의 상태를 저장했다가, 나중에 복구할수 있게 하자.
                          // "memento"

    g.SetStrokeColor(1);
    g.SetStrokeWidth(20);
    g.DrawLine(0, 0, 300, 300);
    g.DrawLine(0, 0, 400, 400);

    // 처음에 그렸던 선과 동일하게 그리고 싶다.
    g.Restore(token);
}
```

chain of responsibility

카테고리

행위 패턴(Behavioral Pattern)

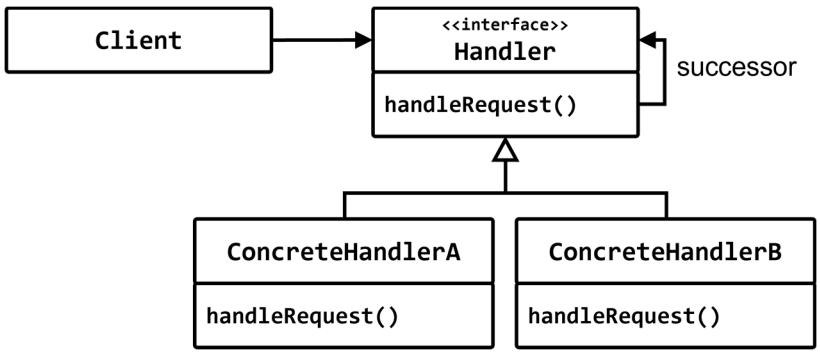
의도

요청을 처리할 수 있는 기회를 하나 이상의 객체에 부여함으로써 요청하는 객체와 처리하는 객체 사이의 결합도를 없애려는 것이다. 요청을 해결할 객체를 만날 때 까지 고리를 따라서 요청을 전달한다.

활용성

- 하나 이상의 객체가 요청을 처리해야 하는 경우, 핸들러 가 누가 선행자 인지를 모를 때, 다음 핸들러는 자동으로 결정된다.
- 메시지를 받을 객체를 명시하지 않은 채 여러 객체 중 하나에게 처리를 요청하고 싶을 때
- 요청을 처리할 객체 집합을 동적으로 정의하고자 할 때

클래스 다이어그램 (class diagram)



결과

- 객체들 간의 행위적 결합도가 적어 진다.
- 객체에게 책임성을 할당하는데 있어 응용력을 높일 수 있다.
- 메시지 수신을 보장할 수는 없다.

3. Chain Of Responsibility

요청의 처리를 한 개의 객체가 아닌 여러 개의 객체에서 처리할수 있게 하는 디자인 패턴을 “Chain Of Responsibility” 라고 합니다.

예제 1

```
#include <iostream>

struct Handler
{
    Handler* next = 0;

    virtual bool HandleRequest(int problem) = 0;

    void Handle(int problem)
    {
        // 1. 자신이 요청을 처리
        if (HandleRequest(problem) == true)
            return;

        // 2. 처리되지 않은 경우 다음 객체가 있다면 전달
        if (next != 0)
            next->Handle(problem);  // HandleRequest가 아닌 Handle이 핵심
    }
};
//-----------------------
struct OddHandler : public Handler
{
    bool HandleRequest(int problem) override
    {
        std::cout << "arrive at OddHAndler" << std::endl;
        return problem % 2 == 1;
    }
};
struct TenHandler : public Handler
{
    bool HandleRequest(int problem) override
    {
        std::cout << "arrive at TenHandler" << std::endl;
```

```
        return problem == 10;
    }
};
struct EvenHandler : public Handler
{
    bool HandleRequest(int problem) override
    {
        std::cout << "arrive at TenHandler" << std::endl;
        return problem % 2 == 0;
    }
};
int main()
{
    OddHandler oh;
    TenHandler th;
    EvenHandler eh;
    oh.next = &th;
    th.next = &eh;
    // oh => th => eh
    oh.Handle(5);
}
```

예제 2

자식 윈도우에서 발생된 요청(event) 를 부모 윈도우로 전달하는 예제

```
#define USING_GUI
#include "cppmaster.h"
#include <map>
#include <vector>

class CWnd;
map<int, CWnd*> this_map;

class CWnd
{
    int mHandle;

    CWnd* parent; // 부모 윈도우는 한개이다.
    std::vector<CWnd*> child_vec; // 자식윈도우는 여러개 이다.
public:

    void AddChild(CWnd* child)
```

```
    {
        child_vec.push_back(child);
        child->parent = this;
        ec_add_child(this->mHandle, child->mHandle);
    }

    void Create()
    {
        mHandle = ec_make_window(foo, "A");
        this_map[mHandle] = this;
    }

    static int foo(int hwnd, int msg, int a, int b)
    {
        CWnd* const pThis = this_map[hwnd];

        switch (msg)
        {
        case WM_LBUTTONDOWN: pThis->FireLButtonDown(); break;
        case WM_KEYDOWN:     pThis->KeyDown(); break;
        }
        return 0;
    }

    void FireLButtonDown()
    {
        // 1. 자신이 처리를 시도
        if (LButtonDown() == true)
            return;

        //2. 부모윈도우가 있다면 전달
        if (parent != 0)
            parent->FireLButtonDown();
    }

    virtual bool LButtonDown() { return false; }
    virtual bool KeyDown() { return false; }
};


class MyWindow : public CWnd
{
public:
    bool LButtonDown() { cout << "LBUTTON" << endl; return true; }
```

```
};
class ImageView : public CWnd
{
public:
    bool LButtonDown() { cout << "ImageView LBUTTON" << endl; return true; }
};

int main()
{
    MyWindow w;
    w.Create();

    ImageView view;
    view.Create();
    w.AddChild(&view); // view 를 w의 자식윈도우로 붙인다.

    ec_process_message();
}
```

mediator

카테고리

행위 패턴(Behavioral Pattern)

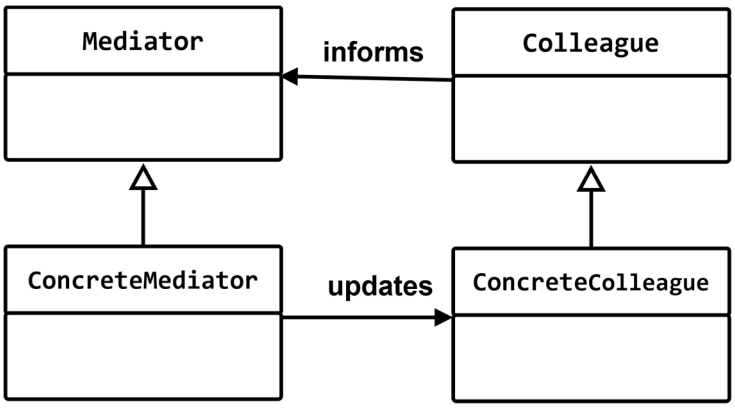
의도

객체들 간의 상호 작용을 객체로 캡슐화 한다. 객체들 간의 참조 관계를 객체에서 분리 함으로써 상호작용만을 독립적으로 다양하게 확대 해석할 수 있다.

활용성

- 여러 객체가 잘 정의된 형태이기는 하지만 복잡한 상호 관계를 가지는 경우, 객체들 간의 의존성을 잘 이해하기 어려울 때
- 객체의 재사용이 다른 객체와의 연결 관계의 복잡함으로 인해 방해를 받을 때
- 여러 클래스에 분산된 행위들이 상속 없이 수정되어야 할 때

클래스 다이어그램 (class diagram)



결과

- 서브 클래스의 수를 제한 한다.
- Colleague 들 사이의 종속성을 줄인다.
- 객체의 프로토콜을 단순화 하는 장점이 있다.
- 객체들 간의 협력 방법을 하나의 클래스로 추상화 한다.
- 통제의 집중화가 이루어 진다.

4. Mediator

객체간의 관계가 복잡한 경우 중재자를 도입하면 편리합니다.

▌ 중재자를 도입하기 전의 코드

```
// 체크 박스 2개와 라디오박스 2개가 있다고 가정할때.
// 단점 : 아래 처럼 만들면 4개의 객체가 서로를 알고 있어야 합니다.
//        하나의 객체의 상태 변화시 나머지 객체의 상태를 조사하느 코드가
//        4개의 객체 모두에 있어야 합니다.
//        객체의 관계가 너무 복잡해 집니다. - "중간층(중재자)"가 있으면
//                                            편리합니다.
class MyCheck : public CheckBox
{
public:
    void ChangeState()
    {
        // 체크박스 2개가 모두 체크 되고 라디오1이 체크 되어야만 버튼이
        // enable 된다고 가정합니다.
        if ( GetCheck() == true  &&
             check2.GetCheck() == true &&
             radio1.GetCheck() == true &&
             radio2.GetCheck() == false)
        {
            cout << "Button Enable" << endl;
        }
        else
        {
            cout << "Button Disable" << endl;
        }
    }
};
```

▌ 중재자를 도입후의 코드

```
중재자는 모든 협력자를 알고 있어야 합니다.
```

```
// 중재자 클래스
```

```
// 중재자는 모든 협력자(Collegue)를 알고 잇어야 합니다.
class LogInMediator : public IMediator
{
    CheckBox* c1;
    CheckBox* c2;
    RadioBox* r1;
    RadioBox* r2;
public:
    LogInMediator(CheckBox* a, CheckBox* b, RadioBox* c, RadioBox* d)
        : c1(a), c2(b), r1(c), r2(d)
    {
        c1->SetMediator(this);
        c2->SetMediator(this);
        r1->SetMediator(this);
        r2->SetMediator(this);
    }
    void ChangeState()
    {
        // 이제 모든 협력자의 관계설정은 이곳에 집중됩니다.
        if (c1->GetCheck() && c2->GetCheck() &&
            r1->GetCheck() && r2->GetCheck())
        {
            cout << "버튼 Enable" << endl;
        }
        else
            cout << "버튼 disable" << endl;
    }
};

int main()
{
    CheckBox c1, c2;
    RadioBox r1, r2;
    LogInMediator m(&c1, &c2, &r1, &r2);
    _getch(); c1.SetCheck(true);
    _getch(); c2.SetCheck(true);
    _getch(); r1.SetCheck(true);
    _getch(); r2.SetCheck(true);
    _getch(); c1.SetCheck(false);
}
```

예제

통보 센터

```
#include <iostream>
#include <functional>
#include <string>
#include <map>
#include <vector>
using namespace std::placeholders;

class NotificationCenter
{
    typedef std::function<void(void*)> HANDLER;

    std::map<std::string, std::vector<HANDLER> > notif_map;
public:
    void RegisterObserver(const std::string& key, HANDLER f)
    {
        notif_map[key].push_back(f);
    }
    void PostNotificationWithName(const std::string& key, void* hint)
    {
        for (auto f : notif_map[key])
            f(hint);
    }
    // global 통보센터라는 개념도 만들어 봅시다.
    static NotificationCenter& defaultCenter()
    {
        static NotificationCenter instance;
        return instance;
    }
};
//-----------------------
void foo(void* p) { std::cout << "foo" << std::endl; }
void goo(void* p) { std::cout << "goo" << std::endl; }
int main()
{
    // IOS 개발의 핵심의 "통보센터" 입니다.
//  NotificationCenter nc;

    // 공동의 작업은 아래처럼 사용하자 라고 약속
    NotificationCenter& nc = NotificationCenter::defaultCenter();
```

```
    nc.RegisterObserver("LOWBATTERY", foo);
    nc.RegisterObserver("LOWBATTERY", goo);

    // 배터리 모듈을 책임지는 사람은
    nc.PostNotificationWithName("LOWBATTERY", (void*)30);
}
```

interpreter

카테고리

행위 패턴(Behavioral Pattern)

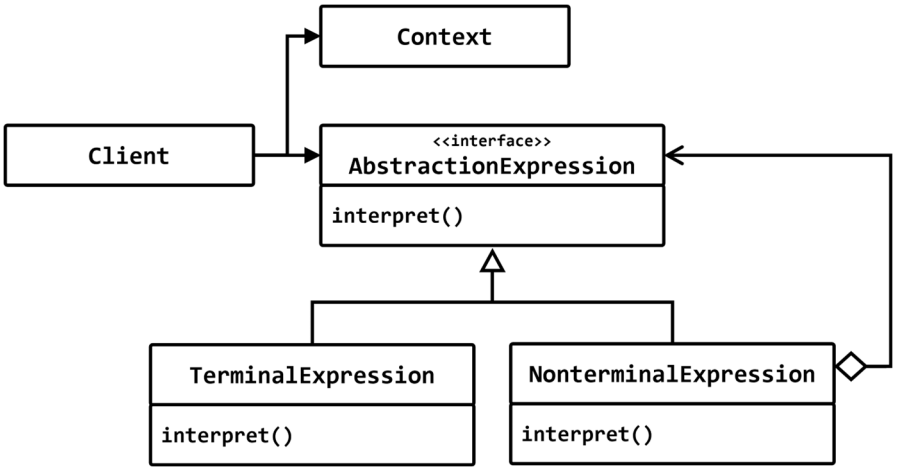
의도

언어에 따라서 문법에 대한 표현을 정의 한다. 또 언어의 문장을 해석하기 위해 정의에 표현에 기반하여 분석기를 정의 한다.

활용성

- 정의할 언어의 문법이 간단한 경우
- 효율성은 별로 고려 사항이 되지 않는다.

클래스 다이어그램 (class diagram)



결과

- 문법의 변경과 확장이 쉽다
- 문법의 구현이 용이하다.
- 복잡한 문법은 관리하기 어렵다.
- 표현식을 해석하는 새로운 방법을 추가할 수 있다.

■ 디자인 패턴 요약

생성패턴	Abstract Factory	상세 화된 서브 클래스를 정의 하지 않고도 서로 관련성이 있거나 독립적인 여러 객체의 군을 생성하기 위한 인터페이스를 제공한다.
	Factory Method	객체를 생성하기 위해 인터페이스를 정의 하지만, 어떤 클래스의 인스턴스를 생성할 지에 대한 결정은 서브 클래스가 한다. Factory Method 패턴에서는 클래스의 인스턴스를 만드는 시점을 서브클래스로 미룬다.
	Prototype	견본적(prototypical) 인스턴스를 사용하여 생성할 객체의 종류를 명시하고 이렇게 만들어진 견본을 복사하여 새로운 객체를 생성한다.
	Builder	복잡한 객체를 생성하는 방법과 표현하는 방법을 정의하는 클래스를 별도로 분리하여 서로 다른 표현이라도 이를 생성할 수 있는 동일한 구축 공정을 제공할 수 있도록 한다.
	Singleton	클래스의 인스턴스는 오직 하나임을 보장하며 이에 대한 접근은 어디에서든지 하나로만 통일하여 제공한다.
구조패턴	Adapter	클래스의 인터페이스를 클라이언트가 기대하는 형태의 인터페이스로 변환 한다. Adapter 패턴은 서로 일치 하지 않은 인터페이스를 갖는 클래스들을 함께 동작시킨다.
	Bridge	구현과 추상화 개념을 분리하여 각각을 독립적으로 변형 할 수 있게 한다.
	Composite	부분과 전체의 계층을 표현하기 위해 복합 객체를 트리 구조로 만든다. Composite 패턴은 클라이언트로 하려면 개별 객체와 복합 객체를 모두 동일하게 다룰 수 있도록 한다.
	Decorator	객체에 동적으로 새로운 서비스를 추가 할 있게 한다. Decorator 패턴은 기능의 추가를 위해서 서브 클래스를 생성하는 것보다 융통성 있는 방법을 제공한다.
	Façade	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.
	Flyweight	작은 크기의 객체들이 여러 개 있는 경우, 객체를 효과적으로 사용하는 방법으로 객체를 공유하게 한다.
	Proxy	다른 객체에 접근하기 위해 중간 대리 역할을 하는 객체를 둔다.
행위패턴	Chain Of Responsibility	요청을 처리할 수 있는 기회를 하나 이상의 객체에 부여함으로써 요청하는 객체와 처리하는 객체 사이의 결합도를 없애려는 것이다. 요청을 해결할 객체를 만날 때까지 고리를 따라서 요청을 전달한다.
	Command	요청을 객체로 캡슐화 함으로써 서로 다른 요청으로 클라이언트를 파라미터화 하고, 요청을 저장하거나 기록을 남겨서 오퍼레이션의 취소도 가능하게 한다.
	Iterator	복합 객체 요소들의 내부 표현 방식을 공개하지 않고도 순차적으로 접근할 수 있는 방법을 제공한다.
	Mediator	객체들 간의 상호 작용을 객체로 캡슐화 한다. 객체들간의 참조 관계를 객체에서 분리 함으로써 상호작용만을 독립적으로 다양하게 확대 해석할 수 있다.

	Memento	캡슐화를 위해 하지 않고 객체 내부 상태를 캡슐화 하여, 나중에 객체가 이 상태로 복구 가능하게 한다.
	Observer	객체 사이의 1:N 의 종속성을 정의 하고 한 객체의 상태가 변하면 종속된 다른 객체에 통보가 가고 자동으로 수정이 일어 나게 한다.
	State	객체 자신의 내부 상태에 따라 행위를 변경하도록 한다. 객체는 마치 클래스를 바꾸는 것처럼 보인다.
	Strategy	다양한 알고리즘이 존재 하면 이들 각각을 하나의 클래스로 캡슐화하여 알고리즘의 대체가 가능하도록 한다. Strategy 패턴을 이용하면 클라이언트와 독립적인 다양한 알고리즘으로 변형할 수 있다. 알고리즘을 바꾸더라도 클라이언트는 아무런 변경을 할 필요가 없다.
	Template Method	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.
	Visitor	객체 구조에 속한 요소에 수행될 오퍼레이션을 정의 하는 객체. Visitor 패턴은 처리되어야 하는 요소에 대한 클래스를 변경하지 않고 새로운 오퍼레이션을 정의할 수 있게 한다.
	Interpreter	언어에 따라서 문법에 대한 표현을 정의 한다. 또 언어의 문장을 해석하기 위해 정의에 표현에 기반하여 분석기를 정의 한다.