1. 중재자 패턴(Mediator Pattern)

1) 정의

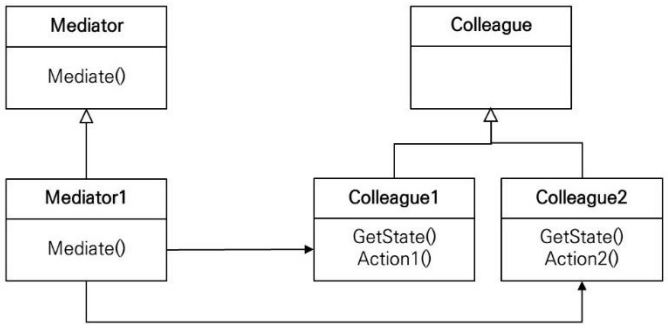
- 객체들의 집합이 상호작용하는지를 함축해 놓은 객체를 정의한 디자인 패턴. 프로그램의 실행 행위를 변경할 수 있기 때문에 행위 패턴으로 간주됨

2) 목적

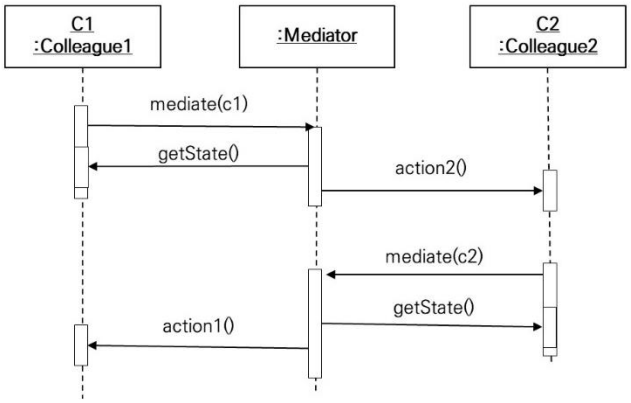
- 중재자 패턴 사용시 객체 간 통신은 중재자 객체 안에 함축됨. 객체들은 더 이상 다른 객체와 서로 직접 통신하지 않으며 중재자를 통해 통신하게 됨. 따라서 객체 간 의존성을 줄일 수 있으므로 객체 간 결합도를 감소시킴

3) 다이어그램

Mediator Pattern_class



Mediator Pattern_Squence



4) 사용 방법

- 상태가 바뀔 때마다 중재자에게 알려주고, 중재자에서 보낸 요청을 수행함

5) 활용 예시

- 서로 연관된 GUI 구성요소들을 관리하기 위한 용도

6) 장점 & 단점

- 장점

  시스템하고 각 객체를 분리시킴으로써 재사용성을 획기적으로 향상시킬 수 있음

  제어 로직을 한 군데 모아 놨기 때문에 관리하기가 수월함

  시스템에 들어있는 객체 사이에서 오가는 메시지의 종류를 확 줄이고 단순화시킬 수 있음

- 단점

  디자인을 잘 하지 못하면 중재자 객체 자체가 너무 복잡해질 수 있음

7) 소스코드

```h
// IMediator.h

#pragma once
#include <string>

class ICollegue;

using std::string;

class IMediator
{
public:
    IMediator() { }
    virtual ~IMediator() { }

    virtual void SetUp() = 0;
    virtual void Notify(ICollegue* sender, string strMsg) const = 0;

};
```

```h
// ICollegue.h

#pragma once
#include <string>

class IMediator;

using std::string;

class ICollegue
{
protected:
    IMediator *m_mediator;

public:
    ICollegue(IMediator *mediator = nullptr) : m_mediator(mediator) { }
    virtual ~ICollegue() { }

    virtual void SetMediator(IMediator* mediator)
    {
        this->m_mediator = mediator;
    }

    virtual void ReciveMessage(string strMsg) = 0;
};
```

```h
// CConcreteMediator.h

#pragma once
#include <vector>
#include <string>
#include "IMediator.h"
#include "ICollegue.h"

using std::vector;
using std::iterator;
using std::string;

class CConcreteMediator : public IMediator
{
private:
    vector<ICollegue*> m_vecCollegue;

public:
    CConcreteMediator();
    ~CConcreteMediator();

    void SetUp();
    void Notify(ICollegue* sender, string strMsg) const;

    void AddCollegue(ICollegue *addColl);
    void RemoveCollegue(ICollegue *removeColl);
};
```

```cpp
// CConcreteMediator.cpp

#include "ConcreteMediator.h"

CConcreteMediator::CConcreteMediator() { }

CConcreteMediator::~CConcreteMediator() { }

void CConcreteMediator::SetUp() { }

void CConcreteMediator::Notify(ICollegue * sender, string strMsg) const
{
    for (int i = 0; i < m_vecCollegue.size(); i++)
    {
        if (m_vecCollegue[i] != sender)
            m_vecCollegue[i]->ReciveMessage(strMsg);
    }
}

void CConcreteMediator::AddCollegue(ICollegue * addColl)
{
    m_vecCollegue.push_back(addColl);
    addColl->SetMediator(this);
}

void CConcreteMediator::RemoveCollegue(ICollegue * removeColl)
{
    vector<ICollegue*>::iterator it;
    for (it = m_vecCollegue.begin(); it < m_vecCollegue.end();)
    {
        if (*it == removeColl)
            it = m_vecCollegue.emplace(it);
        else
            it++;
    }
}
```

```cpp
// CCollegue
#pragma once
#include <iostream>
#include "IMediator.h"
#include "ICollegue.h"

using namespace std;

class CCollegueA : public ICollegue
{
public:
    CCollegueA() { }
    ~CCollegueA() { }

    void DoA()
    {
        cout << "Coll A" << endl;
        this->m_mediator->Notify(this, "A");
    }

    virtual void ReciveMessage(string strMsg)
    {
        cout << "Success\nExit" << endl;
    }
};
```

```cpp
// CCollegue

class CCollegueB : public ICollegue
{
public:
    CCollegueB() { }
    ~CCollegueB() { }

    void DoC()
    {
        cout << "Coll C" << endl;
        this->m_mediator->Notify(this, "C");
    }

    void DoD()
    {
        cout << "Coll D" << endl;
        this->m_mediator->Notify(this, "D");
    }

    virtual void ReciveMessage(string strMsg)
    {
        cout << "ReciveMessage" << endl;
        DoD();
    }
};
```

2. 싱글턴 패턴(Singlton Pattern)

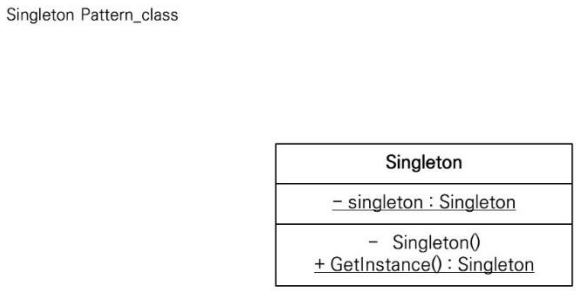
1) 정의

- 생성자가 여러 차례 호출되더라도 실제로 생성되는 객체는 하나이고 최초 생성 이후에 호출된 생성자는 최초의 생성자가 생성한 객체를 리턴하는 디자인 유형

2) 목적

- 해당 클래스의 인스턴스가 하나만 만들어지고, 어디서든지 그 인스턴스에 접근할 수 있도록 전역 접근 제공

3) 다이어그램



4) 사용 방법

- 싱글턴이 적용된 클래스를 생성한 후, 필요한 곳에서 인스턴스를 받아와서 사용

5) 활용 예시

- 상태 객체
- 공통된 객체를 여러 개 생성해서 사용하는 DBCP(Database Connection Pool)와 같은 상황에서 많이 사용

6) 장점 & 단점

- 장점

어디에서든 해당 인스턴스에 쉽게 접근 할 수 있음

- 단점

클래스의 단독 인스턴스가 실제로 필요하지 않은 상황에서 불필요한 제한을 도입

응용 프로그램에 전역 상태 도입

다중 스레드를 사용하는 경우 속도와 자원 문제를 파악한 후 적절하게 구현해야 함

클래스 로더가 여러 개 있을 경우, 싱글턴이 제대로 작동하지 않고 여러 개의 인스턴스가 생길 수 있음

7) 전역변수보다 선호되는 경우

- 불필요한 변수로 전역 네임 스페이스(또는 중첩 네임 스페이스가 있는 언어, 포함 네임 스페이스)를 오염시키지 않음
- 지연 할당 및 초기화를 허용하는 반면 많은 언어의 전역변수는 항상 리소스를 소비함

8) 소스코드

```h
// CSingletonObj.h

#pragma once
#include <iostream>
#include "Singleton.h"

using namespace std;

class CSingletonObj : public CSingleton<CSingletonObj>
{
private:
    int m_iNum;

public:
    void PrintSuccess();

    void SetNumber(int set);
    void PrintNumber();
};
```

```cpp
// CSingletonObj.cpp

#include "SingletonObj.h"

void CSingletonObj::PrintSuccess()
{
    cout << "Success" << endl;
}

void CSingletonObj::SetNumber(int set)
{
    m_iNum = set;
}

void CSingletonObj::PrintNumber()
{
    cout << m_iNum << endl;
}
```

```cpp
// CSingleton.h

#pragma once

template<typename T> class CSingleton
{
private:

protected:
    CSingleton() { }

public:
    virtual ~CSingleton() { }

    static T* GetInstance()
    {
        static T instance;
        return &instance;
    }

};
```