파사드 패턴

```
#include <iostream>

using namespace std;


class PrinterBed//프린팅베드 서브 클래스

{

public:

	void heatUp()

	{

		cout << "Heating up the printer bed." << endl;

	}

	void coolDown()

	{

		cout << "Cooling down the printer bed." << endl;

	}

};


class Extruder//익스투르더 서브 클래스

{

public:

	void heatUp()

	{

		cout << "Heating up the extruder." << endl;

	}

	void coolDown()
```

```
    {
        cout << "Cooling down the extruder." << endl;
    }
};

class Motor//모터 서브 클래스
{
public:
    void TorqueOn()
    {
        cout << "Motor torque on." << endl;
    }
    void TorqueOff()
    {
        cout << "Motor torque off." << endl;
    }
};

class PrinterRun//파사드 서브 클래스
{
private://프라이빗에 서브 클래스
    PrinterBed bed;//서브 클래스에 해당하는 객체 생성
    Extruder extruder;
    Motor motor;
```

```
public:
    void startPrinting()//프린터 동작 함수,메인문에 사용
    {
        cout << "Preparing the 3D printer" << endl;
        bed.heatUp();          // 베드 온도 상승
        extruder.heatUp(); // 익스트루더 온도 상승
        motor.TorqueOn(); // 모터 토크 활성화
        cout << "3D printer is ready for printing." << endl;
    }

    void stopPrinting()
    {
        cout << "Shutting down the 3D printer" << endl;
        bed.coolDown();       // 베드 온도 하강
        extruder.coolDown(); // 익스트루더 온도 하강
        motor.TorqueOff(); // 모터 토크 비활성화
        cout << "3D printer is shut down." << endl;
    }
};

// 메인 코드
int main()
{
    PrinterRun printer;//프린터 객체
    printer.startPrinting();
```

```
    printer.stopPrinting();


    return 0;
}
```

파사드 패턴은 복잡한 서브시스템을 간단한 인터페이스로 추상화하여 사용자가 서브시스템의 세부 사항에 신경 쓰지 않고 사용할 수 있게 한다. 예제에서는 PrinterRun이 파사드 역할을 하며, 프린터의 각 부품(베드, 익스트루더, 모터)을 제어하는 여러 기능을 간단한 클래스로 묶어서 제공한다.

어댑터 패턴

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


// 기존의 라이트닝 인터페이스
class LightningConnector
{
public:
    virtual void connectLightning() // 라이트닝 연결 함수
    {
        cout << "Connecting Lightning." << endl;
    }
};


// 기존의 C타입 인터페이스
class usbCconnector
{
public:
```

```
    virtual void connectTypeC() // USB Type-C 연결 함수
    {
        cout << "Connecting USB Type-C." << endl;
    }
};

// USB Type-C를 라이트닝으로 변환해주는 어댑터 클래스
class LightningAdapter : public LightningConnector
{
private:
    usbCconnector* typeCConnector; // 어댑터가 연결할 Type-C 장치포인터

public:
    LightningAdapter(usbCconnector* typeC) : typeCConnector(typeC) {} // 어댑터 생성자

    void connectLightning() override 라이트닝 연결 함수를 Type-C로 변환하는 함수로 바꾸어 구현
    {
        cout << "USB Type-C to Lightning." << endl;
        typeCConnector->connectTypeC(); // USB Type-C 연결 메서드 호출
    }
};

// 메인 코드
int main()
```

```
{
    // USB Type-C 장치 생성
    usbCconnector* typeCDevice = new usbCconnector();

    // 어댑터를 통해 라이트닝으로 변환
    LightningAdapter* adapter = new LightningAdapter(typeCDevice);

    // 어댑터를 통해 라이트닝 연결
    adapter->connectLightning(); // Lightning으로 연결

    // 메모리 할당 해제
    delete adapter;
    delete typeCDevice;

    return 0;
}
```

어댑터 패턴은 호환되지 않는 인터페이스들을 연결해주는 패턴이다. 예제에서는 usbCconnector 클래스를 LightningConnector와 호환되도록 LightningAdapter가 역할을 수행한다. 클라이언트는 어댑터를 통해 기존에 존재하는 인터페이스와 새로운 인터페이스 간의 차이를 알 필요 없이 사용할 수 있다.

상태 패턴

```
#include <iostream>
#include <Windows.h>//sleep()사용

using namespace std;
```

```
// 바리케이드 상태 인터페이스
class BaricadeState
{
public:
	virtual void handleState() = 0; // 상태를 표현하는 추상함수
	virtual ~BaricadeState() {}
};

// 바리케이드가 올라간 상태  자식 클래스
class BaricadeUpState : public BaricadeState
{
public:
	void handleState() override //상태 표현 함수 구현
	{
		cout << "Baricade is up." << endl;
	}
};

// 바리케이드가 내려간 상태 자식 클래스
class BaricadeDownState : public BaricadeState
{
public:
	void handleState() override
	{
		cout << "Baricade is down." << endl;
```

```
    }
};

// 컨텍스트 클래스: 바리케이드
class Baricade
{
private:
    BaricadeState* currentState;//바리케이드 상태 포인터
public:
    // 생성자, 초기 상태 설정
    Baricade(BaricadeState* initialState) : currentState(initialState) {}

    // 상태 변경 함수
    void setState(BaricadeState* state)
    {
        currentState = state;
    }



    void performAction()
    {
        currentState->handleState(); // 현재 상태에 따른 동작 표현
    }
};
```

```
// 메인 코드
int main()
{
    // 상태 객체
    BaricadeUpState upState;//바리케이드가 올라간 상태의 객체
    BaricadeDownState downState;//바리케이드가 내려간 상태의 객체

    // 바리케이드가 내려가 있는 상태로 시작
    Baricade baricade(&downState);

    int i=0;
    while (1)
    {
        if (i % 2 == 0)
        {
            baricade.setState(&upState); // 상태를 '올라간 상태'로 변경
        }
        else
        {
            baricade.setState(&downState); // 상태를 '내려간 상태'로 변경
        }
        i++;
        Sleep(5000);//5초 대기
    }
    return 0;
```

```
}
```

상태 패턴은 객체의 상태에 따라 행동이 달라지는 상황에서 상태를 객체로 분리해 관리하는 패턴이다. 예제에서 바리케이드는 현재 상태에 따라 동작이 달라지며, BaricadeUpState와 BaricadeDownState는 각각의 상태를 정의한다. 상태가 변경되면 그에 맞는 행동을 수행하게 된다.

프록시 패턴

```
#include <iostream>

#include <string>


using namespace std;


// 이미지 인터페이스

class Image

{

public:

    virtual void display() = 0; // 이미지를 출력하는 추상함수

};


// 실제 이미지 클래스

class RealImage : public Image

{

private:

    string filename; // 이미지 파일 이름


    void loadFromDisk()// 실제 이미지 로드 함수

    {
```

```
        cout << "Loading image from disk: " << filename << endl;
    }

public:
    RealImage(string filename) : filename(filename)
    {
        loadFromDisk(); // 생성 시 이미지 로드
    }

    void display() override // 이미지 출력 함수
    {
        cout << "Displaying image: " << filename << endl;
    }
};

// 이미지 프록시 클래스
class ImageProxy : public Image
{
private:
    RealImage* realImage; // 실제 이미지 객체
    string filename;

public:
    ImageProxy(string filename) : realImage(nullptr), filename(filename) {}
```

```
    ~ImageProxy()
     {
        delete realImage; // 프록시가 소멸될 때 실제 이미지 객체도 해제
    }

    void display() override// 이미지 출력 함수
 {
        if (realImage == nullptr)
{
            // 실제 이미지를 처음 요청할 때만 로드
            realImage = new RealImage(filename);
        }
        realImage->display(); // 이미지를 출력
    }
};

// 메인 코드
int main() {
    // 프록시를 통해 이미지에 접근
    Image* image = new ImageProxy("test_image.jpg");

    // 이미지가 처음 요청될 때 실제 이미지가 로드되고 출력됨
    cout << "First call to display:" << endl;
    image->display();
```

```
    // 두 번째 호출에서는 이미 로드된 이미지가 출력됨
    cout << "₩nSecond call to display:" << endl;
    image->display();

    // 메모리 해제
    delete image;

    return 0;
}
```

프록시 패턴은 실제 객체에 대한 대리 객체를 제공하여, 필요할 때만 실제 객체를 생성하거나 접근할 수 있게 한다. 예제에서는 ImageProxy가 실제 이미지를 대신하며, RealImage는 필요할 때만 로드되어 메모리를 절약한다.보통 객체가 직접 사용되었을 때 보안상 문제가 생기거나 이미지를 로드해야 하는등 객체가 메모리를 많이 사용해야 할 때 사용한다

빌더 패턴

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

// Product 클래스: 만들고자 하는 복잡한 객체 (House)
class House
{
public:
    std::string walls;
    std::string roof;
    std::string windows;
```

```
    void show() const
    {
        std::cout << "Walls: " << walls << "₩n"
            << "Roof: " << roof << "₩n"
            << "Windows: " << windows << std::endl;
    }
};

// Builder 자식 클래스, 집을 만드는 과정
class HouseBuilder
{
public:
    virtual ~HouseBuilder() {}
    virtual void buildWalls() = 0; // 벽을 만드는 추상함수
    virtual void buildRoof() = 0;//지붕
    virtual void buildWindows() = 0;//창문
    virtual House* getHouse() = 0;//완성된 집 객체 반환
};

// 빌더 클래스, 집을 만드는 과정 코드
class ConcreteHouseBuilder : public HouseBuilder
{
private:
    House* house;
```

```
public:
    ConcreteHouseBuilder() //빌더 생성자, 집 객체 생성
    {
        this->house = new House();
    }

    ~ConcreteHouseBuilder()
    {
        delete house; //메모리 해제
    }

    void buildWalls() override //벽 생성
    {
        house->walls = "Brick walls";
    }

    void buildRoof() override //지붕 생성
    {
        house->roof = "Metal roof";
    }

    void buildWindows() override //창문 생성
    {
        house->windows = "Double-glazed windows";
    }
```

```
    House* getHouse() override //집 반환
    {
        return house;
    }
};

// Director 클래스 : 객체 생성 과정을 실행
class HouseDirector
{
private:
    HouseBuilder* builder;

public:
    HouseDirector(HouseBuilder* builder) //생성자에서 빌더 지정
    {
        this->builder = builder;
    }

    void constructHouse()
    {
        builder->buildWalls();
        builder->buildRoof();
        builder->buildWindows();
    }
```

```
};

int main()
{
    // Builder 생성
    ConcreteHouseBuilder* builder = new ConcreteHouseBuilder();

    // Director 생성
    HouseDirector director(builder);

    // House 생성
    director.constructHouse();

    // 완성된 집 객체 얻기
    House* house = builder->getHouse();

    // 집 정보 출력
    house->show();

    // 메모리 정리
    delete builder;

    return 0;
}
```

빌더 패턴은 복잡한 객체를 단계별로 생성할 때 사용된다. 예제에서는 House라는 복잡

한 객체를 생성하기 위해 ConcreteHouseBuilder가 각 부분을 단계별로 만들고, HouseDirector가 그 과정을 관리한다.