



디자인패턴 (+GoF)

잘 설계된 S/W란?

고객의 요청들

1. 제작 요청 : 이거 만들어주세요.
2. 기능 추가 요청 : 이거 기능 추가해주세요.
3. 변경 요청 : 이 부분 수정해주세요.

잘 설계된 S/W 조건

4. 고객 요구사항 대로 개발이 되었는가? □ 기본
5. 구조가 쉽게 파악이 되는가?
6. 기능 추가를 쉽게 할 수 있는가?
7. 변경이 편리한가?

고객의 요구사항

잠만 잘수있게 해주세요.

- 저는 잠에서 잘 안깨요.
- 나중에 수정사항은 전달드릴게요.

이렇게 만들어진 결과물

오른쪽 집은,
변경 요청에 대응이 어렵다.

- 구조 파악이 힘들다.
- 창문 교체 요청
(집 전체가 무너질 수 있음)



수정이 편리한 S/W 결과물

조립식 건물

- 지붕 교체 가능
- 문 교체 가능
- 창문 교체 가능



하지만, 아무리 잘 만든 S/W 라도 ...

당연하게도

무리한 고객 변경 요구사항은 수용 불가

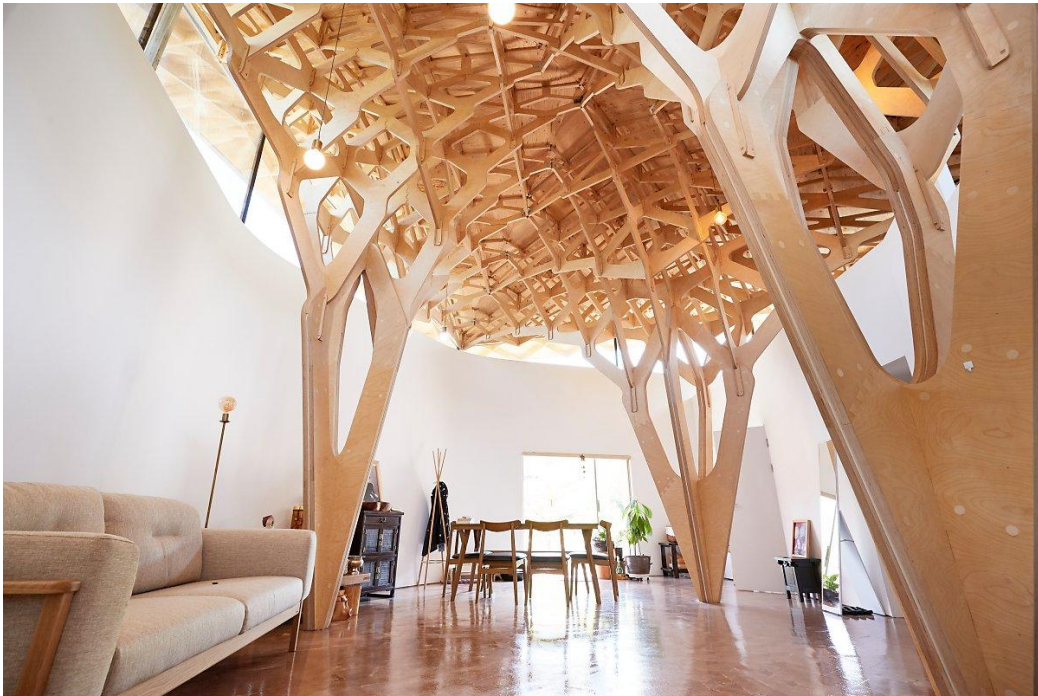


변경요청
수용불가



아름다운 구조? 복잡하다.

구조가 단순해야, 유지보수가 된다.



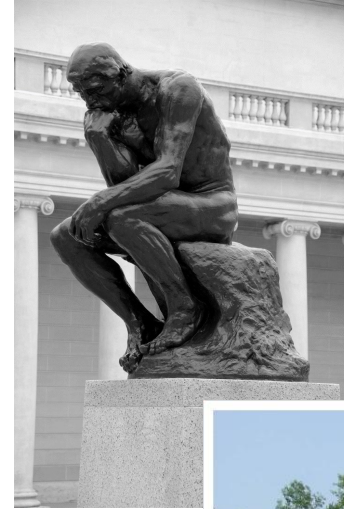
유지보수가 힘든 복잡한 구조의 천장



오랜시간이 지나도, 유지보수가 용이한 구조

깔끔한 구조로 OOP 개발 하기 위한 훈련

1. 객체지향 원칙을 학습하는 방법
 - SOLID 원칙
2. **깔끔한 코딩 뼈대를** 학습하는 방법
 - 디자인 패턴
3. 리팩토링
 - 깔끔한 구조로, 지속적인 개선하는 훈련



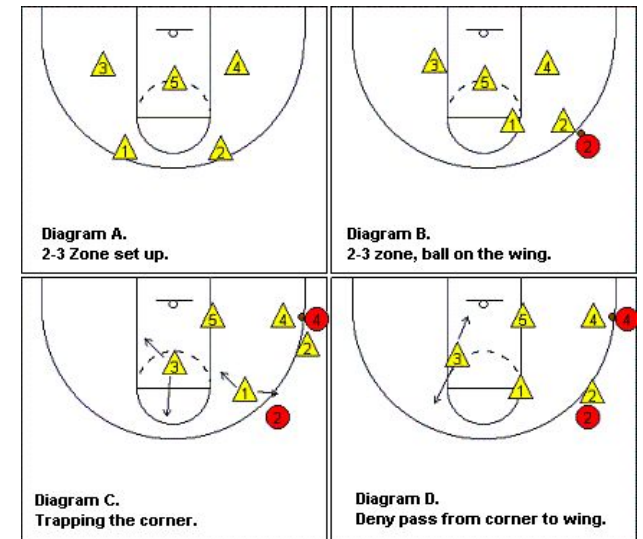
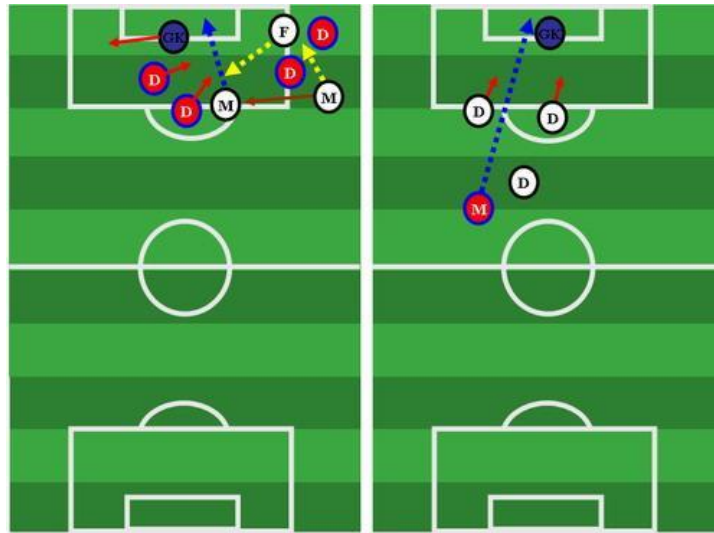
GoF 디자인 패턴 소개

Gang of Four 가 만든, 소스코드 패턴



디자인 패턴이란?

자주 사용되는 설계를 일반화하여,
패턴으로 정리한 것



디자인 패턴의 장단점

장점

- 개발자간 의사소통
- S/W 구조 파악이 쉽다.
- 설계 변경 요청에 대해 유연한 대처 가능

디자인 패턴의 장단점

단점

- 개발에 참여하는 팀원 전체가 학습되지 않으면, 유지보수가 더 어려워진다.
- 구현 난이도가 올라간다.

GoF 디자인 패턴

소프트웨어 설계에 공통된 문제에 대한 표준 해법

- 네 명의 학자가 쓴 책이기에 Gang of Four, GoF 디자인패턴으로 불리운다.

Factory Method Pattern

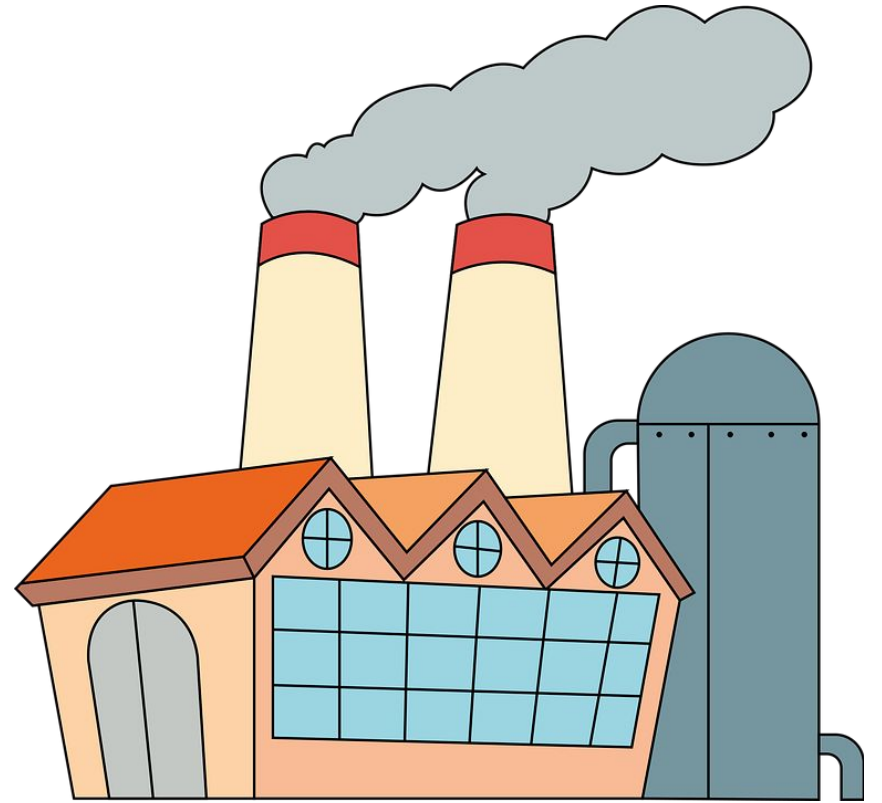
객체를 생성해주는 공장

Simple Factory 형태

객체를 생성해주는 곳이 한곳이다.

- 한 곳에서 생성, 관리 가능

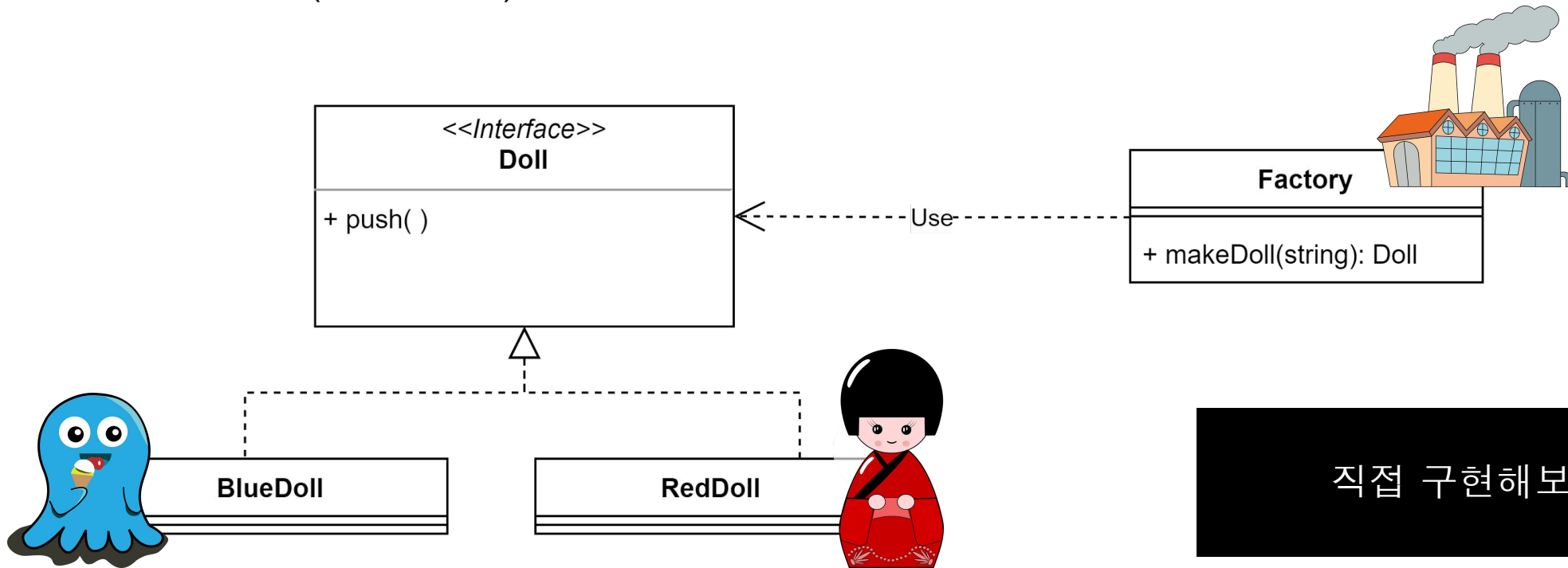
생성코드를 Client에 공개하지 않는다.



UML 분석 1 (Factory 형태)

makeDoll(string)

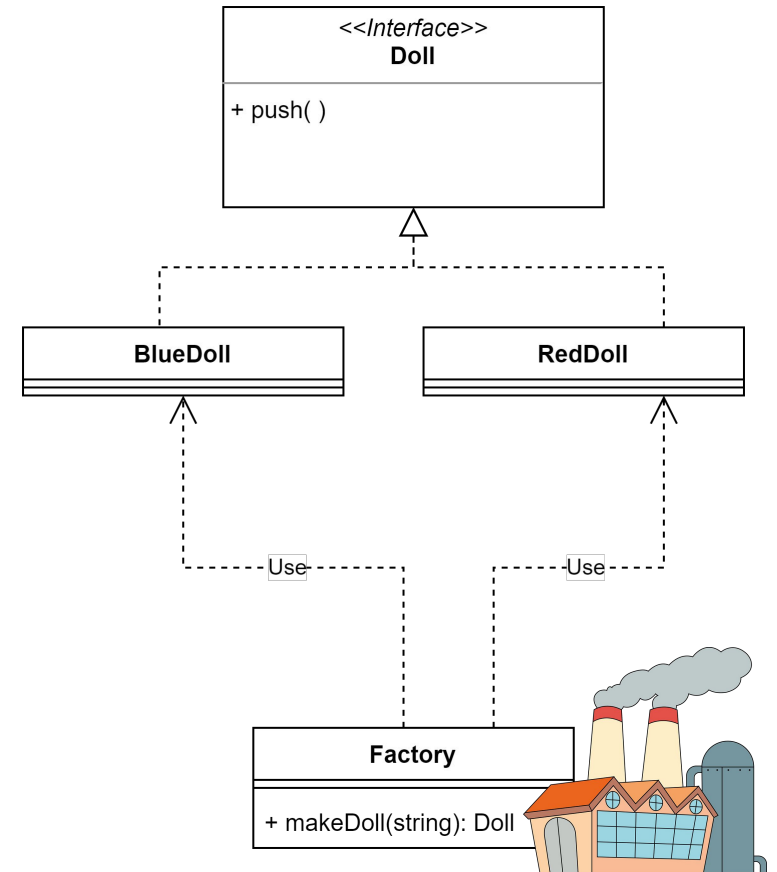
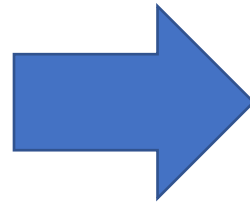
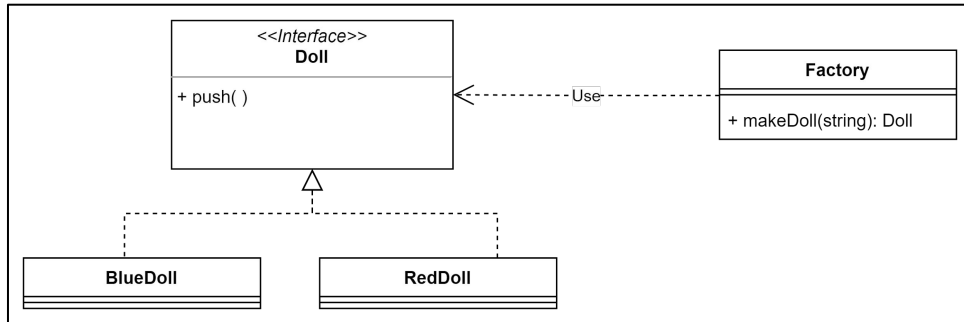
- makeDoll("RED") : Red 인형 생성
- makeDoll("BLUE") : Blue 인형 생성



직접 구현해보자

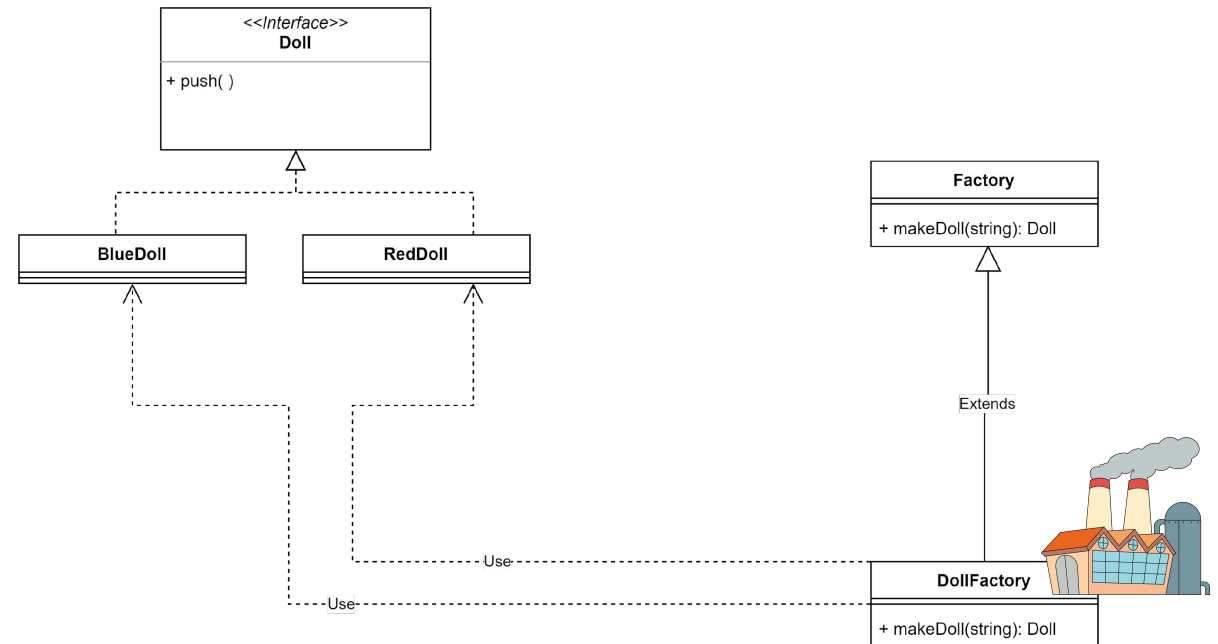
UML 분석 2 (Factory 형태)

더 정확한 UML 형태



Factory Method Pattern

여러개의 Factory 생성 가능



내용 출처 :

https://ko.wikipedia.org/wiki/%ED%8C%A9%ED%86%A0%EB%A6%AC_%EB%A9%94%EC%84%9C%EB%93%9C_%ED%8C%A8%ED%84%B4

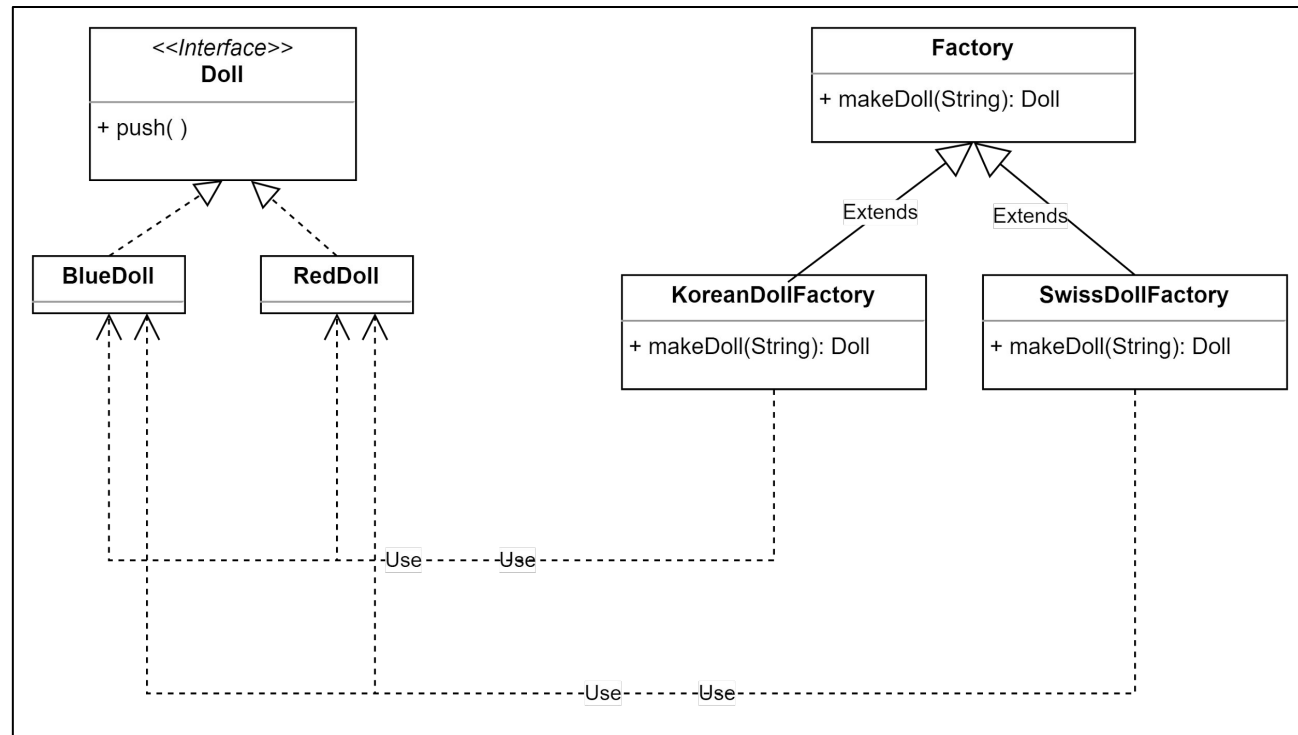
[도전] Factory Method Pattern 구현

KoreanDollFactory

- 한국어로 된 음성이 나온다.
 - BlueDoll : “파랑파랑”
 - RedDoll : “레드레드”

SwissDollFactory

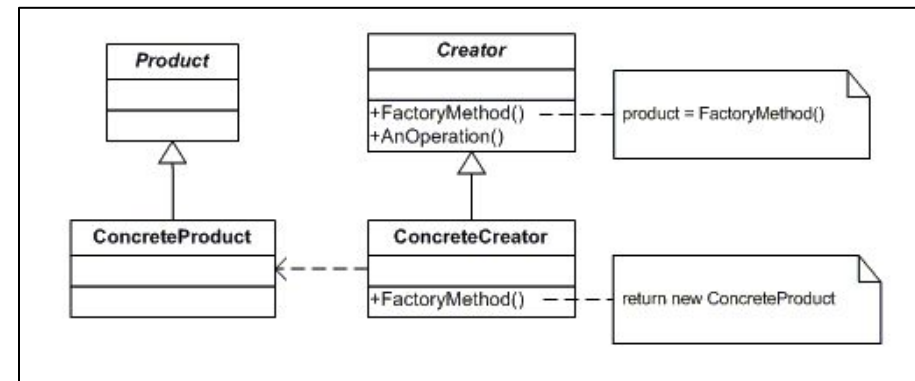
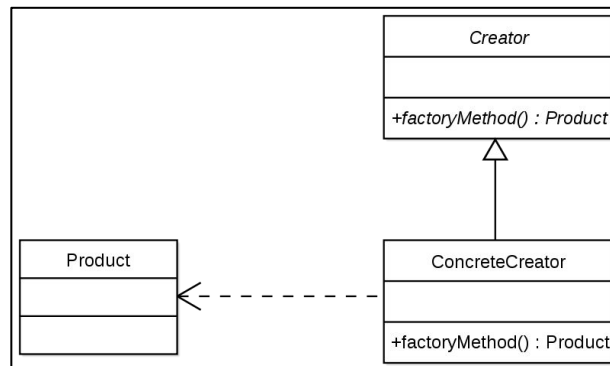
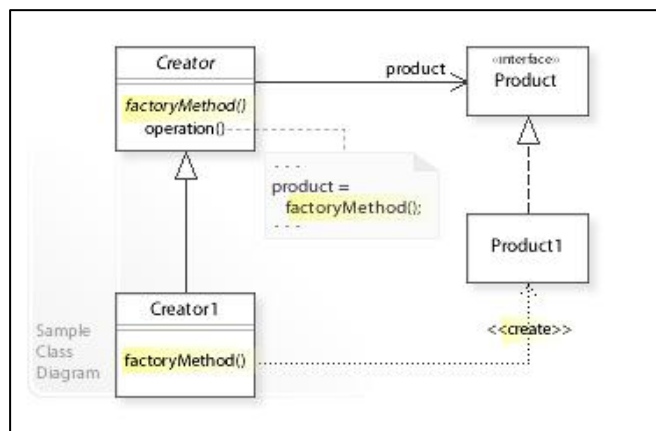
- 독일어로 된 음성이 나온다.
 - BlueDoll : “BuruGten”
 - RedDoll : “RetGten”



UML 표현 예시

Creator : Factory

ConcreteCreator : 구체(구상)적인 Factory



Builder Pattern

생성을 더 간단하게.

두 가지 Builder Pattern

Effective Java의 Builder Pattern



- 생성자 인자가 많을때 사용한다. 생성자 오버로딩 대체 수단.
- 메서드 체인

GoF 의 Builder Pattern

- 생성 절차가 복잡할 때 쓴다. 두 부분을 나누기 위해 사용
- 생성을 담당하는 builder
- 절차를 담당하는 director

헤어컷

Client 구현

- Style명 : “군인컷”
 - 앞머리 : 2cm
 - 뒷머리 : 1cm
 - 옆머리 : 1cm
- Style명 : “아나운서컷”
 - 앞머리 : 6cm
 - 뒷머리 : 3cm
 - 옆머리 : 7cm



일반적인 객체 생성 방법

평범한 객체 생성과 setter 제작

```
int main() {  
    Cut gunCut{ "군인컷", 2,1,1 };  
}
```

각 Argument가 무슨 뜻인지,
이 코드만으로는 알 수 없음

```

class Cut {
public:
    Cut(std::string style, int front, int back, int side)
        : Cut(style, front, back, side, 0) {}

    Cut(std::string style, int front, int back, int side, int guretnaru)
        : Cut(style, front, back, side, guretnaru, 0) {}

    Cut(std::string style, int front, int back, int side, int guretnaru, int mustache) {
        //...
    }
};

```

생성자 overloading 으로 구현(java 에서는 default 매개변수 X)

```

class Cut {
public:
    Cut(std::string style, int front, int back, int side, int guretnaru = 0, int mustache = 0) {
        //...
    }
};

```

디폴트 매개변수로 구현

```
class Cut {
public:
    void setStyle(string str) {
        //...
    }
    void setFront(int ml) {
        //...
    }
    void setBack(int ml) {
        //...
    }
    void setSide(int ml) {
        //...
    }
private:
    string style;
    int front;
    int back;
    int side;
};
```

```
int main() {
    Cut cut;
    cut.setStyle("군인컷");
    cut.setFront(2);
    cut.setBack(1);
    cut.setSide(1);

    //...

    return 0;
}
```

[참고] 메서드 체이닝

```
class Chaining {  
public:  
    Chaining& a() {  
        cout << "A" << " ";  
        return *this;  
    }  
    Chaining& b() {  
        cout << "B" << " ";  
        return *this;  
    }  
    Chaining& c() {  
        cout << "C" << " ";  
        return *this;  
    }  
};
```

```
int main() {  
    Chaining& something = (*new Chaining).a().b().c();  
  
    return 0;  
}
```

```

class Builder {
public:
    Builder& setFront(int ml) {
        frontHair = ml;
        return *this;
    }

    Builder& setBack(int ml) {
        backHair = ml;
        return *this;
    }

    Builder& setSide(int ml) {
        sideHair = ml;
        return *this;
    }

    Builder& setStyleName(string name) {
        style = name;
        return *this;
    }

    Cut build() {
        return Cut(style, frontHair, backHair, sideHair);
    }

private:
    string style;
    int frontHair;
    int backHair;
    int sideHair;
};

```



```

class Cut {
public:
    class Builder { ... };

    void showInfo() {
        cout << "style:" << style
              << "\nfront:" << frontHair
              << "\nback:" << backHair
              << "\nside:" << sideHair
              << endl;
    }

private:
    Cut(string style, int frontHair, int backHair, int sideHair)
        : style{style},
          frontHair{frontHair},
          backHair{backHair},
          sideHair{sideHair}
    { }

    string style;
    int frontHair;
    int backHair;
    int sideHair;
};

```

```

int main() {
    Cut cut = Cut::Builder()
        .setFront(3)
        .setBack(2)
        .setSide(1)
        .build();

    cut.showInfo();
}

```


Singleton Pattern

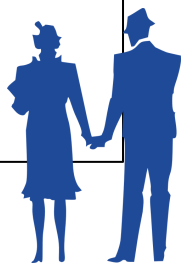
단 하나의 Instance만을 사용한다.

싱글톤이란

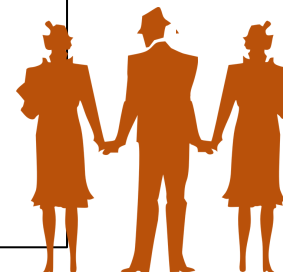
프로그램에서 하나의 인스턴스만을 사용하도록 하는 경우 사용하는 디자인 패턴

```
int main() {  
    Wife wife{};  
    wife.date();  
}
```

문제없는 소스코드



```
int main() {  
    Wife wife1{};  
    Wife wife2{};  
  
    wife1.date();  
    wife2.date();  
  
    Client의 옳지 못한 사용  
}
```



static 멤버 함수 이해하기

Instance를 생성하지 않고, 접근 가능함

```
class Something {  
public:  
    static void run() {  
        cout << "WOW" << endl;  
    }  
};  
  
int main() {  
    Something::run();  
}
```

Singleton	
-	<u>singleton : Singleton</u>
-	Singleton()
+	<u>getInstance() : Singleton</u>

1. 생성자를 private로 만들어, Client에서 생성하지 못하도록 막는다.
2. getInstance() 메서드를 통해서 해당 객체를 얻는다

```
class Singleton {  
public:  
    static Singleton& getInstance() {  
        static Singleton instance;  
        return instance;  
    }  
  
private :  
    Singleton() {  
    }  
};
```

static 멤버함수: instance 생성 없이 접근 가능
static 지역변수 : 지연된 초기화

private 생성자로 외부에서 생성 X

문제점

복사 생성자, 복사 대입 연산자로 인한 문제

딱 하나의 인스턴스만 허용해야 한다.

한 가지 접근 방법만을 제공해야 한다.

```
int main() {  
    Singleton& s1 = Singleton::getInstance();  
    Singleton s2{ s1 }; // 복사 생성자  
    Singleton& s3 = Singleton::getInstance();  
    s3 = s1;           // 복사 대입  
}
```

개선

복사 생성자, 복사 대입 연산자 삭제

```
class Singleton {  
public:  
    static Singleton& getInstance() {  
        static Singleton instance;  
        return instance;  
    }  
  
private :  
    Singleton() {  
    }  
    Singleton& operator=(const Singleton& other) = delete;  
    Singleton(const Singleton& other) = delete;  
};
```

싱글톤 직접 구현해보기

Cursor, AudioManager
등등 특정 클래스를 가정해서
만들어 본다.

```
class Singleton {  
public:  
    static Singleton& getInstance() {  
        static Singleton instance;  
        return instance;  
    }  
  
private :  
    Singleton() {  
    }  
    Singleton& operator=(const Singleton& other) = delete;  
    Singleton(const Singleton& other) = delete;  
};
```


[도전] Logger 제작하기

Logger 클래스 제작하기

- Write 메서드 제공
- 싱글톤으로 구현하기

```
class Logger {
public:
    Logger() {
        // 파일 open("ABC.txt")
    }

    void write(string str) {
        // 파일에 str 쓰기
    }
};

int main() {
    Logger log1{};
    log1.write("HI");

    Logger log2{}; // 문제 발생!
    log2.write("HI");
}
```

Pattern 을 사용한 개발 PJT 1

Singleton + Factory Method Pattern



도전 과제

1. Factory Method Pattern (또는 Simple Factory)
2. Singleton 을 모두 사용하여 개발한다.
 - Factory + Singleton = 자주 사용 됨

JewelryFactory 만들기

만들어내는 제품

- Gold, Ruby, Diamond

공장은 Singleton 으로 만든다.

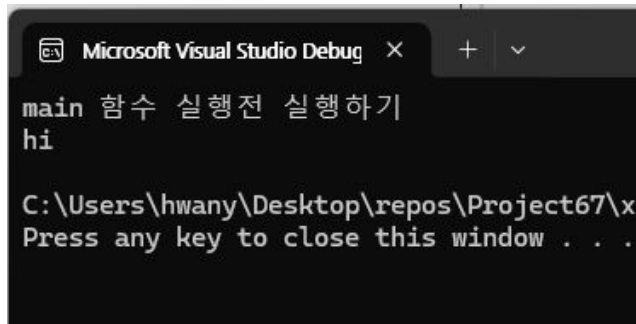
각 제품들은 showInfo() 메서드를 갖는다.

- 제품의 이름, 가격이 출력된다.



[참고] main 함수 전에 실행

- main 함수보다 생성자 호출을 먼저하기 위해서 오른쪽과 같이 작성하면 된다.



```
Microsoft Visual Studio Debug x + v
main 함수 실행전 실행하기
hi
C:\Users\hwany\Desktop\repos\Project67\...
Press any key to close this window . . .
```

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

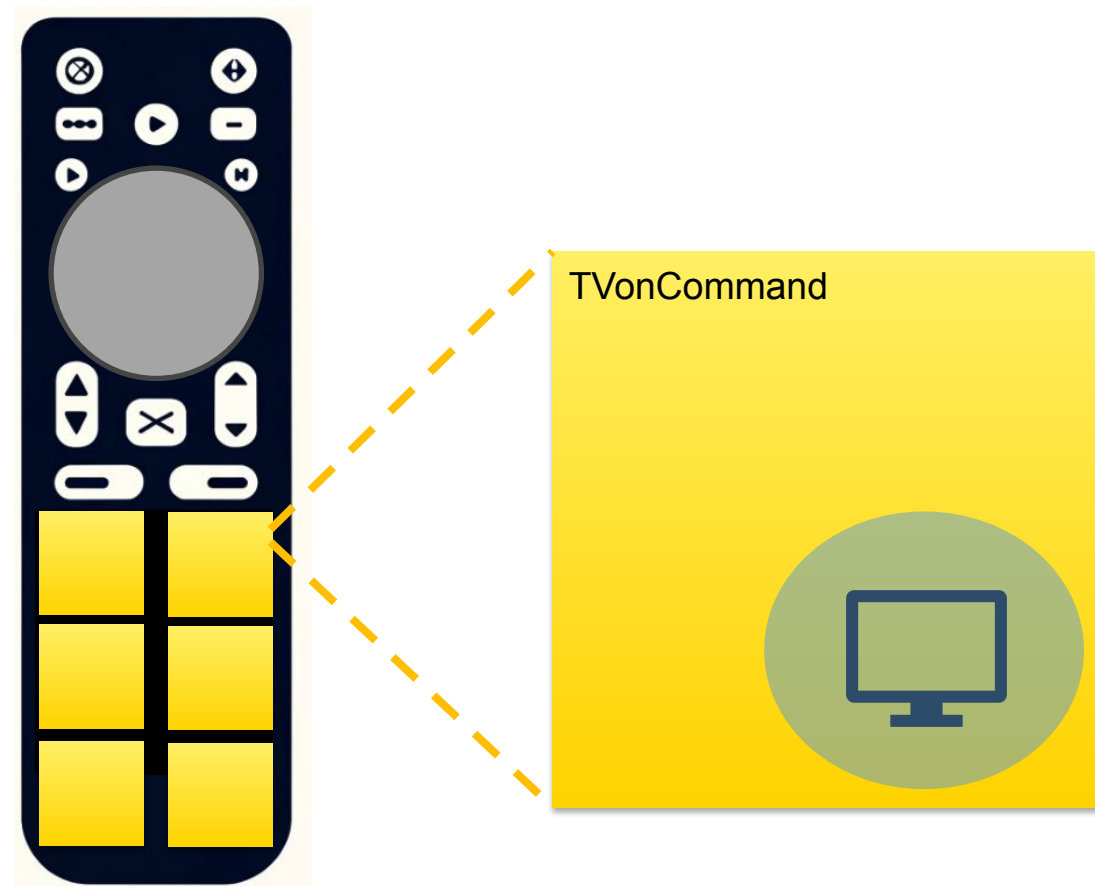
class ExcuteBeforeMain {
public:
    ExcuteBeforeMain() {
        cout << "main 함수 실행전 실행하기" << endl;
    }
    static ExcuteBeforeMain t; // 외부에서 초기화
};

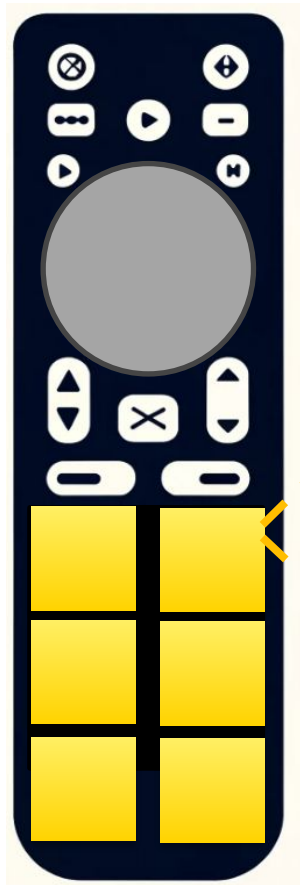
ExcuteBeforeMain ExcuteBeforeMain::t{};

int main() {
    cout << "hi" << endl;
}
```

커맨드 패턴

명령을 하는 측과 수행을 하는 측을 분리하는 패턴





Invoker



```
class TV {  
public:  
    void turnOn() {  
        // tv가 켜진다  
    }  
};
```



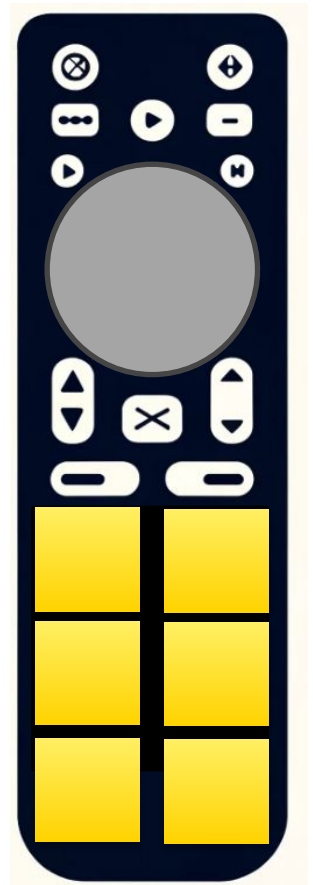

```
class Command {
public :
    virtual void excute() = 0;
};

class TVonCommand : public Command {
public :
    TVonCommand(TV* tv) {
        myTv = tv;
    }
    void excute() override{
        myTv->turnOn();
    }
private :
    TV* myTv;
};
```

Command

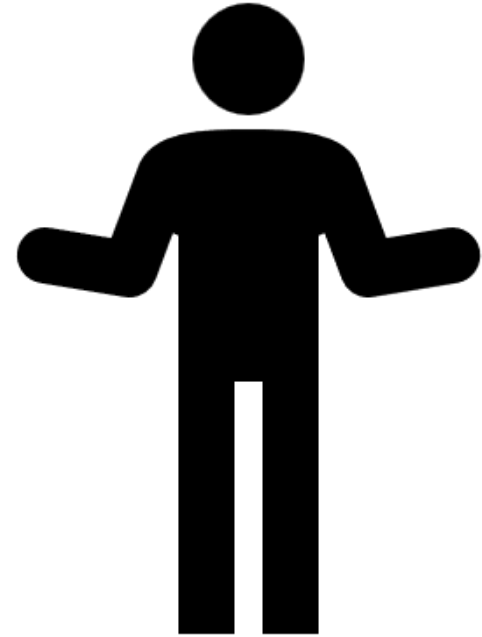


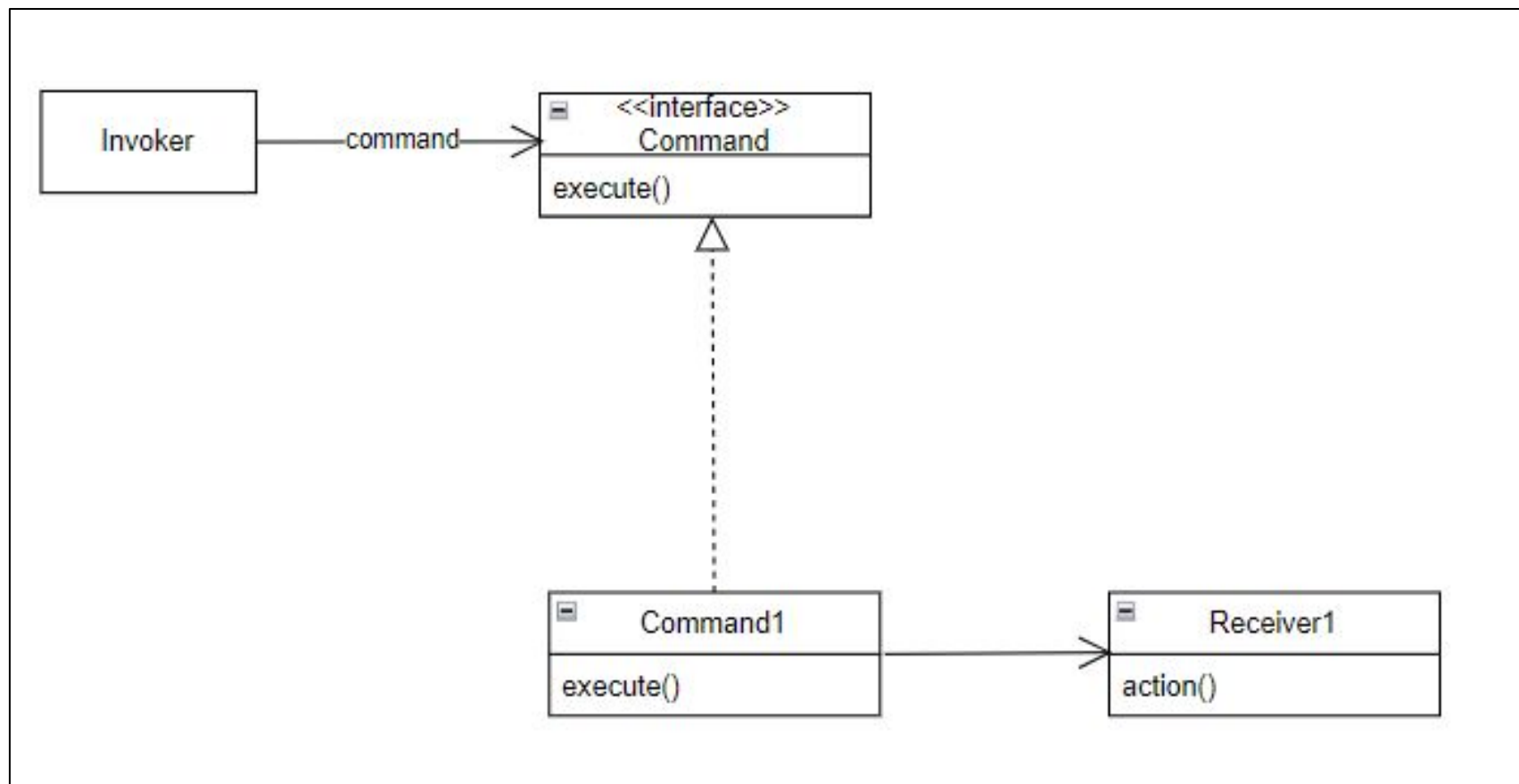
```
class Invoker {  
public:  
    void setCommand(Command* command) {  
        this->command = command;  
    }  
    void run() {  
        command->execute();  
    }  
private:  
    Command* command;  
};
```



Invoker

```
int main() {  
    TV receiver;  
    TVonCommand tvcmd(&receiver);  
  
    Invoker invoker;  
    invoker.setCommand(&tvcmd);  
  
    invoker.run();  
}
```





명령어 기반 프로그램

명령을 받고 수행하는 프로그램

- LEFT 명령어
 - Box가 왼쪽으로 이동
- RIGHT 명령어
 - Box가 오른쪽으로 이동
- SHOW 명령어
 - Box가 화면에 출력되는 프로그램

```
int main() {  
    Box box{};  
    while (true) {  
        cout << "명령어 입력: " << endl;  
        string cmd;  
        cin >> cmd;  
  
        if (cmd == "left") {  
            LeftMove(box);  
        }  
        if (cmd == "right") {  
            RightMove(box);  
        }  
        Show(box);  
    }  
}
```

박스 준비

박스 클래스를 하나 준비한다.

- 최소값 : 0
- 최대값 : 10
- 초기 박스의 위치 : 5

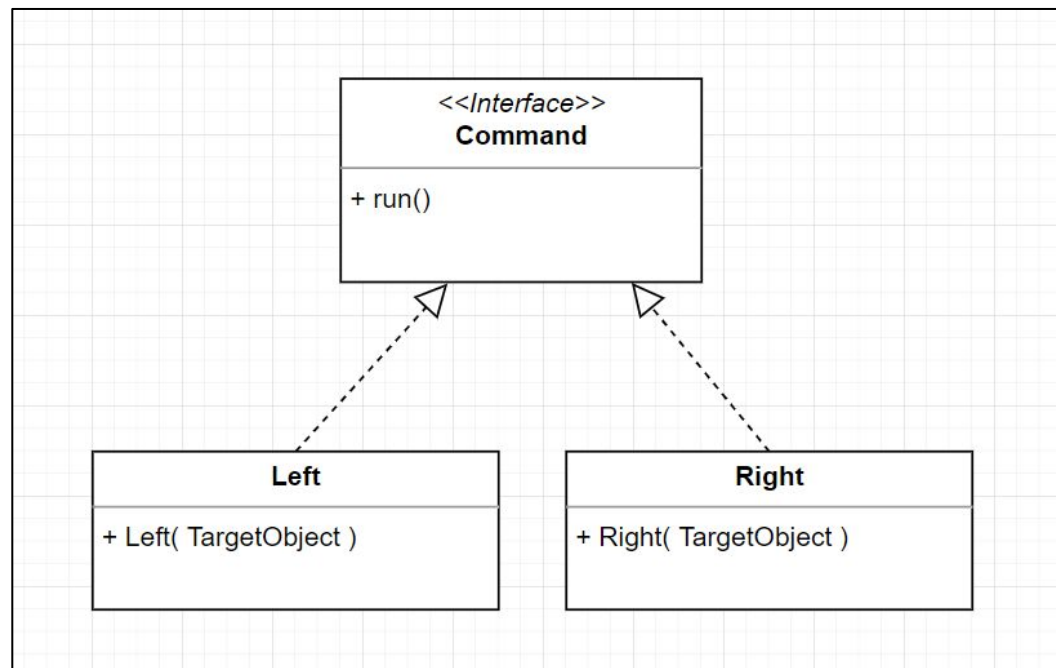


```
class Box {  
public:  
    Box()  
        :position{ 5 }  
    { }  
  
    int getPos() const {  
        return position;  
    }  
  
    void setPos(int x) {  
        if (x < 0 || x > 10) return;  
        position = x;  
    }  
  
private :  
    int position;  
};
```

[도전] Command 객체 만들기

1. Left, Right 객체를 만든다.

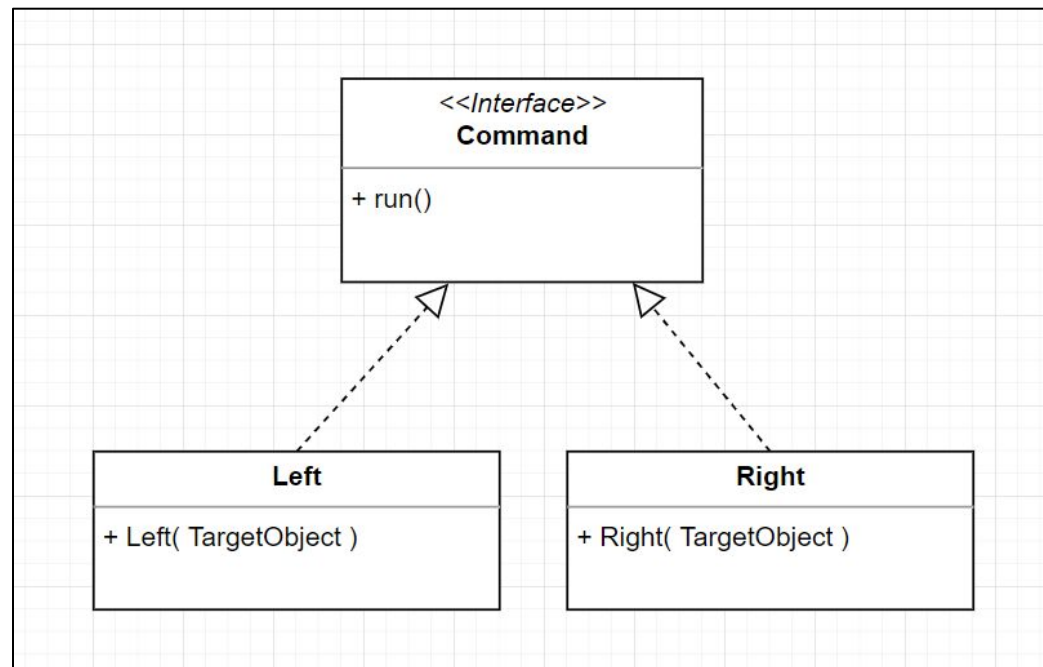
2. 기존 소스코드를
해당 객체로 치환한다.



함수가 아닌, Command 객체로 만들기

1. Command 인터페이스 준비
2. ConcreteCommand
 - Receiver 설정
 - excute 구현
3. Command 실행 코드

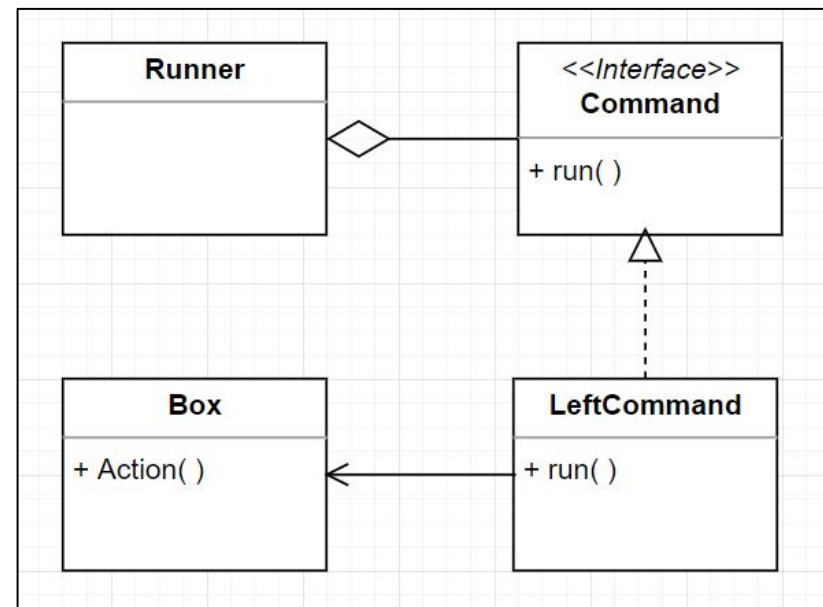
메서드 대신
Command 객체로 만들어보자.



Command Pattern 구현하기

Command를 함수로 구현하지 않고,
객체로 만들어서 관리하는 패턴

1. Command를 객체화
2. Invoker 에서 Command 관리
 - Command 로깅
 - Undo 나 Redo
 - Command 목록 자동화
 - Command 스케줄링



Client에서 동작 수행은 Runner를 통해 이뤄진다.