디자인 패턴

1) 명령패턴, 커맨드 패턴 (Command Pattern)

요청 자체를 캡슐화 하는 것이다. 이를 통해, 서로 다른 사용자(client)를 매개변수로 만들고, 요청을 대기시키거나 로깅하여, 되돌릴 수 있는 연산을 지원한다.

*로깅 : 문제가 발생하였을 때, 로그를 생성하도록 시스템을 작성하는 활동.

즉; 메서드 호출을 실체화한 것이다.
어떤 개념을 변수에 저장하거나, 함수에 전달할 수 있도록 데이터 즉 객체로 바꿀 수 있다는 것을 의미함.

함수 호출을 객체로 감싼다는 것이다. (콜백을 객체지향적으로 표현한 것이다.)
*콜백 : 다른함수의 인자로써 이용되는 함수, 어떤 이벤트에 의해 호출되어지는 함수.
<콜백 설명>

```
// The callback method
function meaningOfLife() {
    log("The meaning of life is: 42");
}


// A method which accepts a callback method as an argument
// takes a function reference to be executed when printANumber completes
function printANumber(int number, function callbackFunction) {
    print("The number you provided is: " + number);
}

// Driver method
function event() {
   printANumber(6, meaningOfLife);
}
```

printANumber는 두 번째 매개변수로 function 타입의 callbackFunction을 인자로 받는다.

결론 : **명령 패턴은 함수 호출을 객체로 감싼다는 뜻으로, 콜백을 객체 지향적으로 표현한 것**

입력 키 변경

모든 게임에는 버튼이나 키보드등 유저 입력을 읽는 코드가 있고, 이런 코드는 입력을 받아서 의미있는 행동으로 전환한다.

```
void InputHandler::handleInput()
{
   if (isPressed(BUTTON_X)) jump();
   else if(isPressed(BUTTON_Y)) fireGun();
   else if(isPressed(BUTTON_A)) swapWeapon();
   else if(isPressed(BUTTON_B)) kneel();
}
```

위의 코드를 예로 들을 수 있다.
이러한 함수는 게임 루프에서 매 프레임 호출된다.

위의 함수들을 직접호출하지 않고, 교체 가능한 무언가로 바꿔 주어야한다.

<공통 상위 클래스>

```
class Command {
public:
   virtual ~Command() { }
   virtual void excute() = 0;
};
// 인터페이스에 반환 값이 없는 메서드 하나밖에 없다면 명령 패턴일 가능성이 높다.
```

<각 행동별로 하위 클래스>

```
class JumpCommand : public Command {
public:
   virtual void execute() { jump(); }
};

class FireCommand : public Command {
public:
   virtual void execute() { fireGun(); }
};

// ... more
```

```
class InputHandler {
public:
    void handleInput();
    // 명령을 바인드(bind)할 메서드들...

private:
    Command* buttonX_;
    Command* buttonY_;
    Command* buttonA_;
    Command* buttonB_;
};
```

입력 핸들러 코드에는 각 버튼별로 command 클래스 포인터를 저장한다.

```
void InputHandler::handleInput()
{
    if (isPressed(BUTTON_X)) buttonX_->excute();
    else if(isPressed(BUTTON_Y)) buttonY_->excute();
    else if(isPressed(BUTTON_A)) buttonA_->excute();
    else if(isPressed(BUTTON_B)) buttonB_->excute();
}
```

직접함수를 호출하던 코드 대신, 한겹 우회하는 계층이 생김.

위의 코드들도 잘 동작하지만, 암시적으로 플레이어 객체만을 움직이게 할 수 있다는 가정이 있다.
그러므로, 제어하려는 객체를 함수에서 찾지 말고, 밖에서 찾아주게끔 한다.

```
class Command {
public:
   virtual ~Command() { }
   virtual void excute(GameActor& actor) = 0;
};
```

```
class JumpCommand : public Command {
public:
   virtual void execute(GameActor& actor) {
      actor.jump();
   }
};
```

command를 상속받은 클래스는 execute()가 호출될 때, gameActor 객체를 인수로 받는다.
-> 원하는 액터의 메서드를 호출할 수 있다.

```
void InputHandler::handleInput()
{
   if (isPressed(BUTTON_X)) return buttonX_;
   if(isPressed(BUTTON_Y)) return buttonY_;
   if(isPressed(BUTTON_A)) return buttonA_;
   if(isPressed(BUTTON_B)) return buttonB_;

   // 아무것도 누르지 않았다면, 아무것도 하지 않는다.
   return NULL;
}
```

<명령 객체를 받아서 플레이어를 대표하는 GameActor객체에 적용>

```
Command* command = inputHandler.handleInput();
if(command)
   command->excute(actor);
```