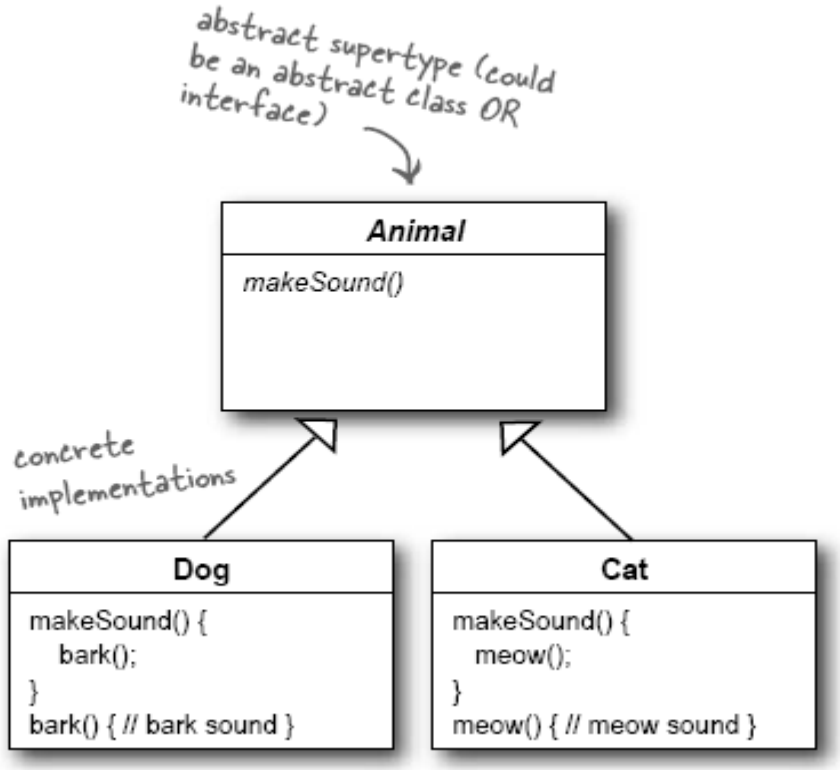
Goff디자인패턴

1. GoF패턴 개요

- 누군가 이미 우리들의 문제를 해결해 놓았다. 똑같은 문제를 경험하고, 그문제를 해결했던 다른 개발자들이 익혔던 지혜와 교훈을 활용하는 방법을 배운다. 이러한 패턴을 사용하는 가장 좋은 방법은 패턴을 머리 속에 집어 넣은 다음 자신의 디자인 및 기본 어플리케이션 어디에 적용할수 있는지 파악하는 것이다. 디자인 패턴은 코드를 재사용하는 것과 마찬가지로 경험을 재사용하는 것이다.
- 객체 지향 설계 시 고려하는 두가지 사항은 **재사용**과 **성능**이다

- Gang of Four의 이니셜을 딴 것
- 디자인 패턴이란 것은 크리스토퍼 알렉산더란 건축가가 여러 환경에서 축물을 만드는데 몇가지 패턴을 이야기한 책의 제목 이었다. 그런 개념을 객체 지향언어에 접목 시킨 사람들인 Erich Gamma, Richard Helm, Ralp Johnson, Jone Vlissides 가 만든 것으로 네 사람을 존경의 뜻으로 Gang of Four란 애칭으로 사용
- 자바 라이브러리설계, C# 라이브러리 설계에 이용됨
- 오늘날 객체 지향 설계에 널리 이용됨

- 인터페이스에 맞춰서 프로그래밍 한다.
- 실행시에 쓰이는 객체가 코드에 의해서 고정되지 않고, 어떤 상위 형식에 맞춰서 프로그래밍함으로써 다형성을 활용해야 한다는 의미 이다



객체지향의 기초

추상화
캡슐화
다형성
상속

객체 지향의 원칙

바뀌는 부분은 캡슐화
상속보다는 구성을 활용
구현이 아닌 인터페이스

패턴 – Strategy
알고리즘 군을 정의하고, 각각의
캡슐화하여 변경가능하게

설계 패턴 종류

◆ **Creational Patterns**

(객체 생성관련 패턴)

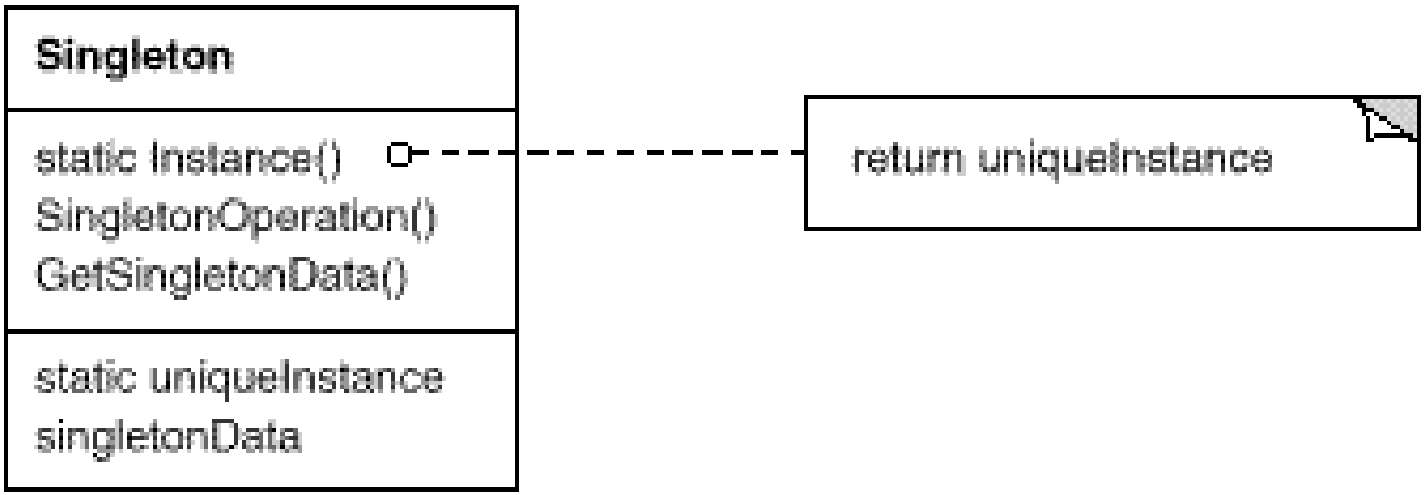
◆ **Structural Patterns**

(객체 구조관련 패턴)

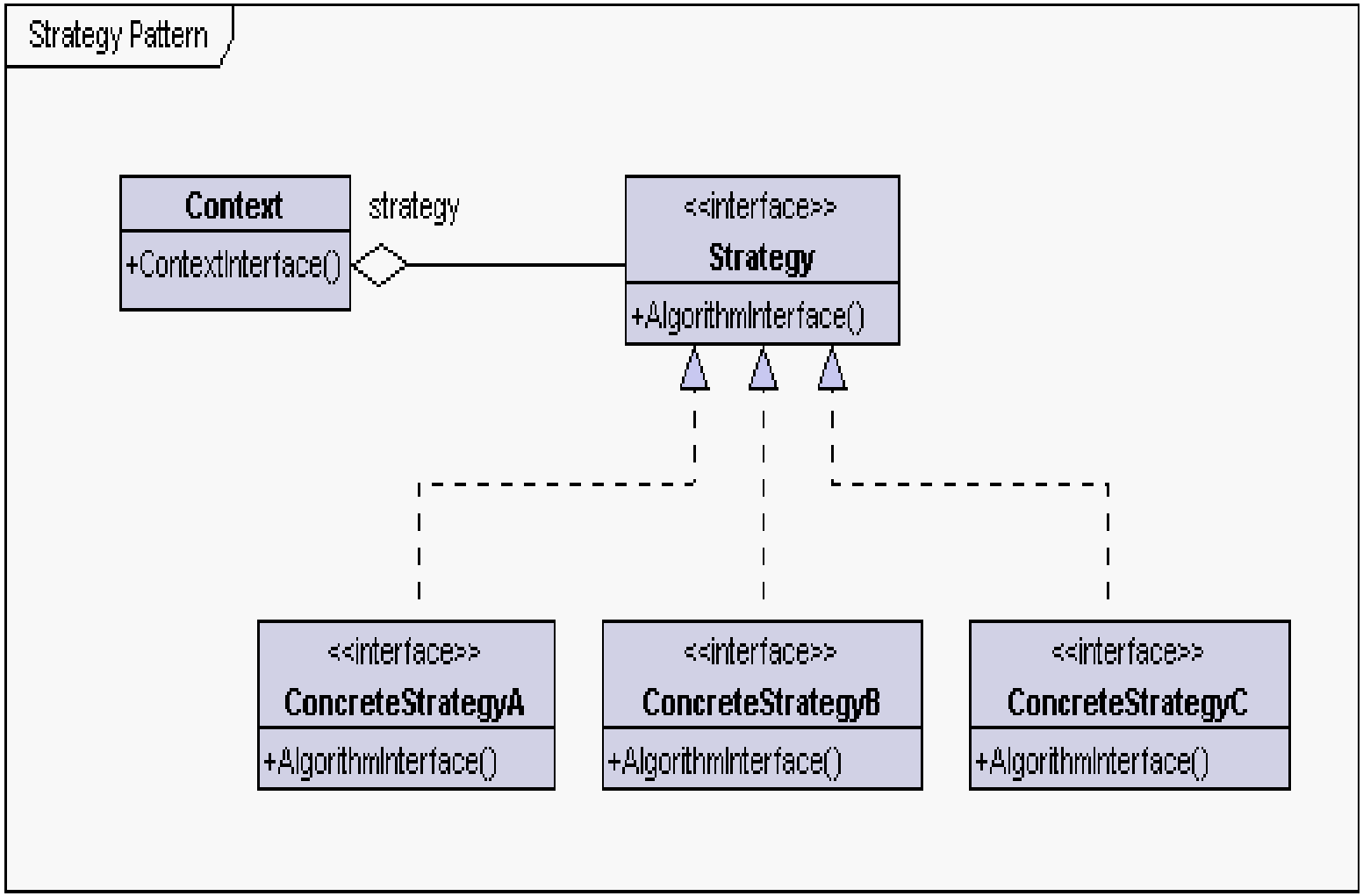
◆ **Behavioral Patterns**

(객체 메소드 관련 패턴)

- 단일객체
- 장치제어 클래스등 하나의 객체만 필요로 하는 경우 사용
- 불필요한 객체생성으로 메모리 수집 연산으로 인한 성능저하 방지
- 객체생성 패턴


Singleton
static Instance()
SingletonOperation()
GetSingletonData()
static uniqueInstance
singletonData
return uniqueInstance

- Strategy Pattern - 전략 패턴
- 동적으로 알고리즘을 교체할 수 있는 구조
- 알고리즘 인터페이스를 정의하고, 각각의 알고리즘 클래스별로 캡슐화하여 각각의 알고리즘을 교체 사용 가능하게 한다
- 즉, 하나의 결과를 만드는 목적(메소드)은 동일하나, 그 목적을 달성할 수 있는 방법(전략, 알고리즘)이 여러가지가 존재할 경우
- 기본이 되는 템플릿 메서드(Template Method Pattern) 패턴과 함께 가장 많이 사용되는 패턴 중에 하나이다


Strategy Pattern
Context
+ContextInterface()
strategy
<<interface>>
Strategy
+AlgorithmInterface()
<<interface>>
ConcreteStrategyA
+AlgorithmInterface()
<<interface>>
ConcreteStrategyB
+AlgorithmInterface()
<<interface>>
ConcreteStrategyC
+AlgorithmInterface()

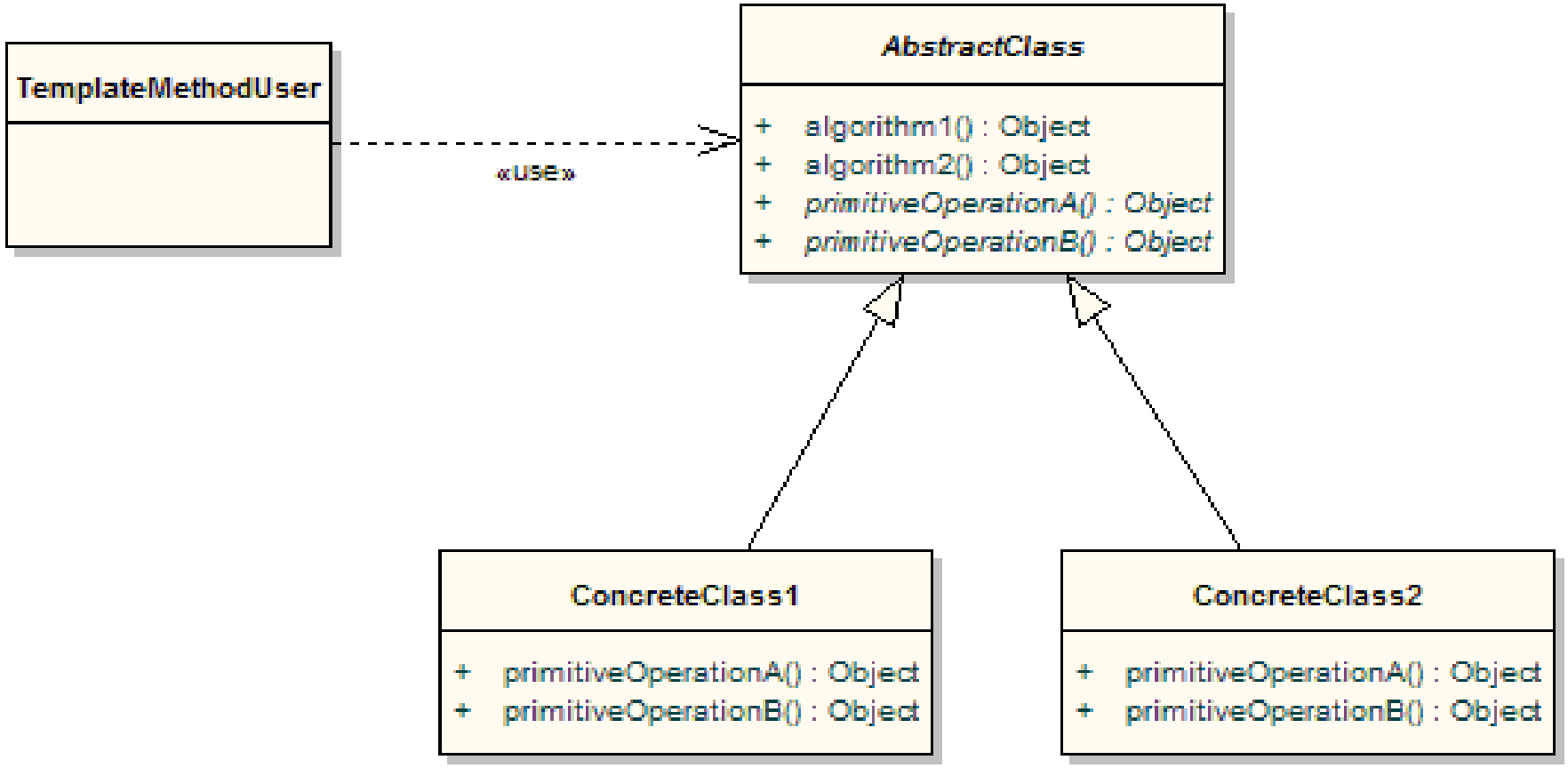
전략 패턴(strategy) 구조

```
#include <iostream>
using namespace std;
//------------------------------------------------------
// Strategy Interface class
class Strategy
{
public:
          virtual void AlgorithmInterface() = 0;
};
//------------------------------------------------------
// Strategy Algorithm A
class ConcreteStrategyA : public Strategy
{
public:
          void AlgorithmInterface() override
          {
                    cout << "Processed by Strategy A" <<
endl;
          }
};
//------------------------------------------------------
// Strategy Algorithm B
class ConcreteStrategyB : public Strategy
{
public:
          void AlgorithmInterface() override
          {
                    cout << "Processed by Strategy B" <<
endl;
          }
};
```

```
// Strategy Algorithm C
class ConcreteStrategyC : public Strategy
{
public:
          void AlgorithmInterface() override
          {
                    cout << "Processed by Strategy C"
<< endl;
          }
};
class Context
{
public:
          Context() : m_pStrategy(0) {}
          ~Context() {
                    if (m_pStrategy)
                              delete m_pStrategy;
          }
public:
          void ChangeStrategy(Strategy* pStrategy)
          {
                    if (m_pStrategy)
                              delete m_pStrategy;
                    m_pStrategy = pStrategy;
          }
          void ContextInterface()
          {
                    m_pStrategy ->
AlgorithmInterface();
          }
private:
          Strategy* m_pStrategy;
};
```

```
int main()
{
        Context* pContext = new Context();
        pContext->ChangeStrategy(new ConcreteStrategyA());
        pContext->ContextInterface();
        pContext->ChangeStrategy(new ConcreteStrategyB());
        pContext->ContextInterface();
        pContext->ChangeStrategy(new ConcreteStrategyC());
        pContext->ContextInterface();
        delete pContext;
        return 0;
}
```

- 완전히 동일한 절차를 가진 코드를 작성할 경우
- 일부 과정의 구현만 다를뿐 나머지 구현은 동일한 경우
- 상위클래스에서 흐름을 제어하고 하위클래스에서 처리 내용을 구체화한다
- 여러 클래스 공통된 부분은 상위클래스에 정의하고 상세한부분은 하위클래스에 구현한다
- 코드 중복을 줄이고, 전략패턴과 가장 많이 쓰인다.


TemplateMethodUser
«use»
AbstractClass
+ algorithm1() : Object
+ algorithm2() : Object
+ primitiveOperationA() : Object
+ primitiveOperationB() : Object
ConcreteClass1
+ primitiveOperationA() : Object
+ primitiveOperationB() : Object
ConcreteClass2
+ primitiveOperationA() : Object
+ primitiveOperationB() : Object

```
#include <iostream>
using namespace std;
class FriedRice
{
public:
		void Fry()
		{
			cout << "Fry Rice" <<endl;
		}
		void CookingProc()
		{
			InputSeasoning();
			Fry();
		}
		virtual void InputSeasoning()=0;
};

class KimchiFriedRice: public FriedRice
{
public:
		virtual void InputSeasoning()
		{
			cout << "input kimchi" << endl;
		}
};
```

```
class ShrimpFriedRice: public FriedRice
{
public:
		virtual void InputSeasoning()
		{
			cout << "input Shrimp" << endl;
		}
};

int  main()
{
		FriedRice* p_rice = new ShrimpFriedRice;
		p_rice->CookingProc();
		return 0;
}
```

2. 설계 스펙

◆ 아래의 문제를 전략패턴을 적용하여 클래스를 설계하시요

1) Permanenet(급여), Temporary(시간당급여, 일한날짜), Mangager(연봉, 담당부서) 클래스를 정의 하고 사원정보(이름, 주소, 급여, 직급 등) 을 출력하고자 한다.

2) Feature phone, smart phone, home phone 클래스를 정의하고
각각 폰의 기능을 출력하고자 한다