C++ 디자인패턴

참고: 위키피디아, dotnetcoders.com

C++ 11

- **C++ 11**(**C++0x**라고도 알려짐)은 [1] ISO가 2011년 8월 12일에 승인한 C++ 프로그래밍 언어의 최신판이다. [2] 2000년대의 한 시점에 공개될 것으로 예상해 C++0x으로 불려 왔으나, 2011년에야 국제표준으로 확정되면서 최종적으로 C++11로 이름이 확정되었다.
- C++11은 핵심 언어에 여러가지를 추가하고 C++ 표준 라이브러리를 확장하고, C++ 기술 보고서 1(TR1)의 라이브러리의 수학적 특수 함수의 라이브러리 예외를 통합했다.[3] C++11은 ***ISO/IEC 14882:2011***라고 출판되었다.
- (위키피디아)

디자인 패턴 소개

누군가 이미 우리들의 문제를 해결해 놓았습니다. 똑같은 문제를 경험하고, 그문제를 해결했던 다른 개발자들이 익혔던 **지혜와 교훈**을 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 재사용**하는 것이다.

소프트웨어 개발에 있어서 변하지 않는 것 ....

변 화

기존코드에 미치는 영향은 최소한으로 줄이면서 작업을 할 수 있도록 만들수 있는 방법이 있으면 좋겠죠?

Gof 패턴

- Gang of Four의 이니셜을 딴 것
- 디자인 패턴이란 것은 크리스토퍼 알렉산더란 건축가가 여러 환경에서 건축물을 만드는데 몇가지 패턴을 이야기한 책의 제목 이였습니다.
- 그런 개념을 객체지향언어에 접목 시킨 사람들인 Erich Gamma, Richard Helm, Ralph Johnson, Jone Vlissides 가 만든 것으로 네사람을 존경의 뜻으로 Gang of Four란 애칭으로 사용
- 자바 라이브러리설계에 이용됨

디자인 패턴 도구들…

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

객체 지향프로그램 과 패턴은 메모리구조를 잘알아야

Creational Patterns (객체 생성과 관련된 패턴)

- Abstract Factory
- Builder
- Factory Method
- Prototype
- **Singleton(중요)**

Structural Patterns(객체 구조관련 패턴)

- Adapter
- Bridge
- **Composite(중요)**
- Decorator
- Façade
- Flyweight
- Proxy

Behavioral Patterns

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- **Strategy(중요)**
- **Template Method(중요)**
- Visitor

UML

- UML(Unified Modeling Language): 애플리케이션을 구성하는 클래스들간의 관계를 그리기 위하여 사용

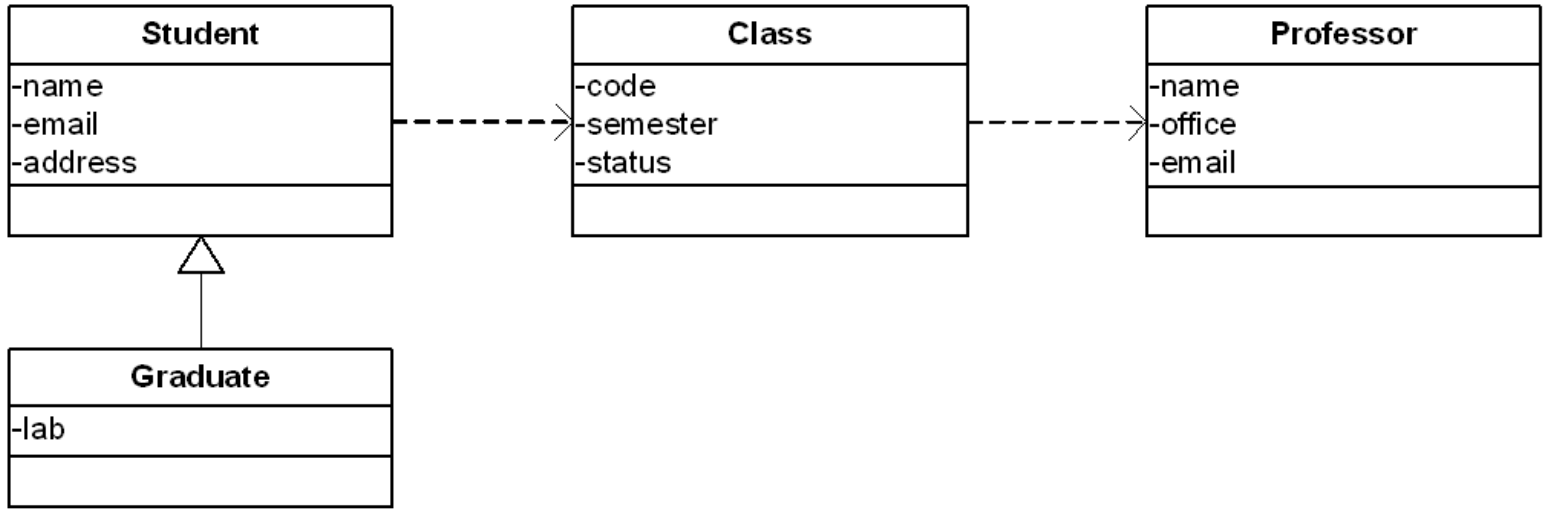


그림 8.9 UML의 예

Singleton

Type: Creational

What it is:
Ensure a class only has one instance and provide a global point of access to it.

무분별한 객체생성을 방지하고, 1개의 객체만 생성하여 이용하는 프로그램 코딩에 유용하게 적용할 수 있다.

템플릿 메소드 패턴

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

```java
public class Tea {
    void boilWater(){
        System.out.println("물끓여요!");
    };
    void steepTeaBag(){
        System.out.println("녹차를우려요");
    };
    void purInCup(){
        System.out.println("컵에 따라요");
    };
    void addLemon(){
        System.out.println("레몬을 추가해요");
    };
}
```

```java
public class Coffee {
    void boilWater(){
        System.out.println("물끓여요!");
    };
    void brewCoffeeGrinds(){
        System.out.println("커피를우려요");
    };
    void purInCup(){
        System.out.println("컵에 따라요");
    };
    void addSugarAndMilk(){
        System.out.println("설탕과 유우를 추가해요");
    };
}
```

```java
public abstract class MakeDrink {

    void boilWater(){
        System.out.println("물끓여요!");
    };

    abstract void brew();

    void purInCup(){
        System.out.println("컵에 따라요");
    };

    abstract void addMore();

}
```

```
class 밥
{
public:
		void 볶는다()
		{
			cout << "밥을 볶는다" <<endl;
		}
		void 요리진행()
		{
			밥첨가물넣기();
			볶는다();
		}
		virtual void 밥첨가물넣기()=0;
};

class 김치볶음밥: public 밥
{
public:
		virtual void 밥첨가물넣기()
		{
			cout << "김치를 넣는다" << endl;
		}
};

class 새우볶음밥: public 밥
{
public:
		virtual void 밥첨가물넣기()
		{
			cout << "새우를 넣는다" << endl;
		}
};

class 야채볶음밥: public 밥
{
public:
		virtual void 밥첨가물넣기()
		{
			cout << "야채를 넣는다" << endl;
		}
};

int  main()
{
		밥* p_rice = new 김치볶음밥;
		p_rice->요리진행();
		return 0;
}
```

Strategy 패턴

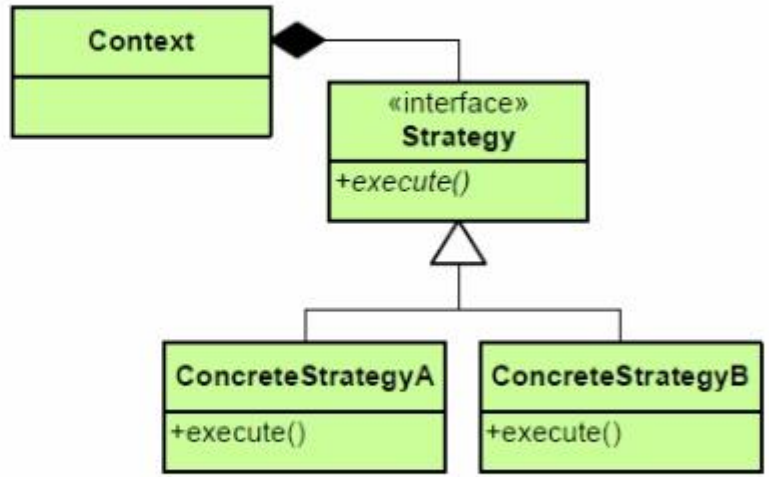
- **Strategy Pattern - 전략 패턴**

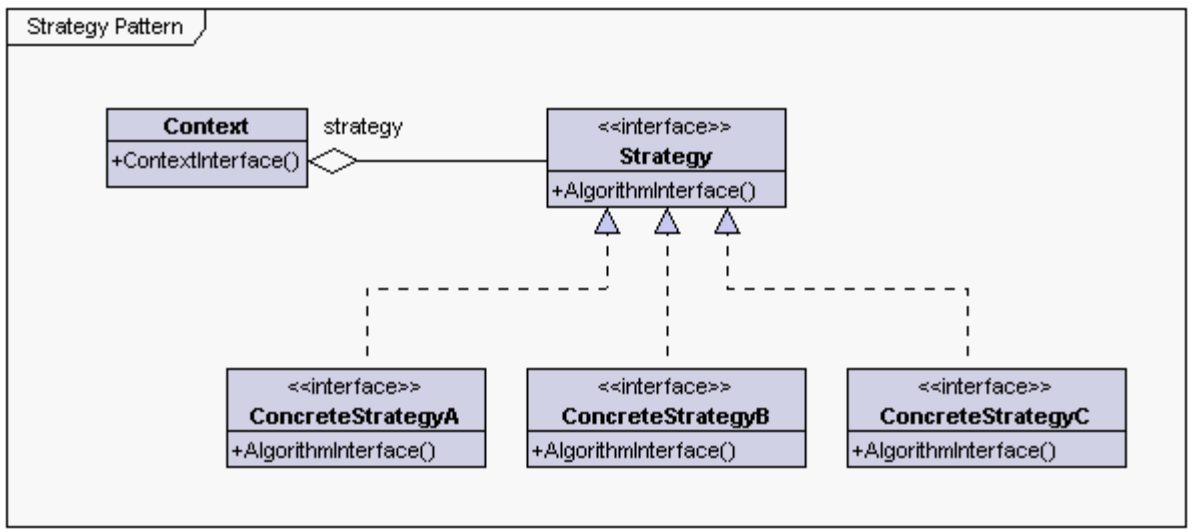
- 동적으로 알고리즘을 교체할 수 있는 구조
- 알고리즘 인터페이스를 정의하고, 각각의 알고리즘 클래스별로 캡슐화하여 각각의 알고리즘을 교체 사용 가능하게 한다
- 즉, 하나의 결과를 만드는 목적(메소드)은 동일하나, 그 목적을 달성할 수 있는 방법(전략, 알고리즘)이 여러가지가 존재할 경우
- 기본이 되는 템플릿 메서드(Template Method Pattern) 패턴과 함께 가장 많이 사용되는 패턴 중에 하나이다

Strategy

Type: Behavioral

What it is:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from clients that use it.





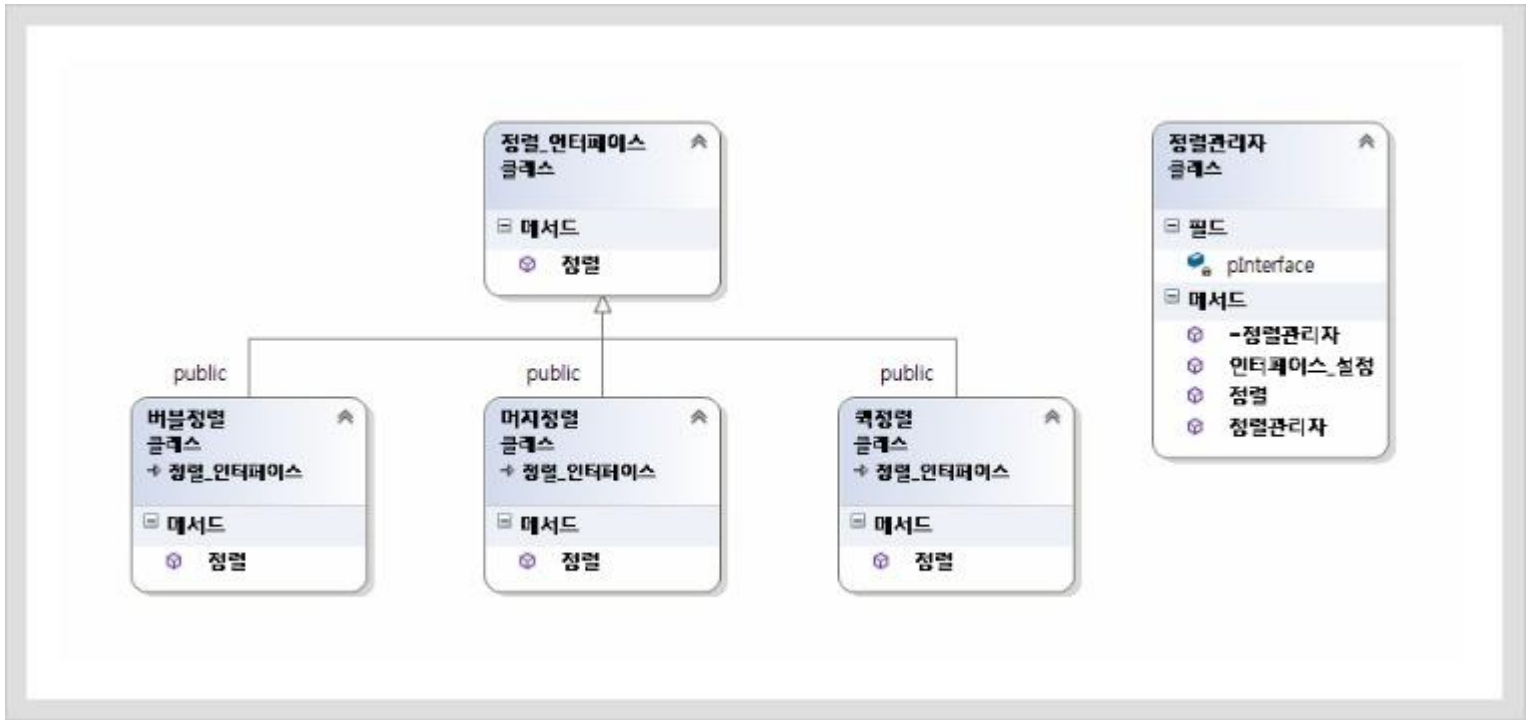
매장에서의 할인 정책에 따른 클래스를 구현한다고 하자

```
class Calculator
{
  int sum = 0;
  for(Item item : items){
    if(firstGuest)
            sum += (int)(item.getPrice() * 0.9);        // 첫 손님 10% 할인
    else if(!item.isFresh())
            sum += (int)(item.getPrice() * 0.8);        // 덜 신선한 것 20% 할인
    else
            sum += item.getPrice();                     // 첫 손님 10% 할인
  }
  return sum;
}
```

**위의 클래스에서 보면 만약에 가격정책이 더 늘어나면 늘어날때마다 else if 블럭이 증가하게
될것이다. 그리고 코드가 점점 더 길어져서 코드 분석을 어렵게 만들것이다.
전략패턴은 이런 변화 될수 있는 부분들을 추상화시켜 코드분석과 유지보수가 용이한 코드로
바꿔주는 역할을 하는 디자인 패턴 중 하나이다.
다음은 전략 패턴을 적용한 클래스이다.**

```
public class Calculator{
private DiscountStrategy discountStrategy;
public Calculator(DiscountStrategy discountStrategy){
this.discountStrategy = discountStrategy;
}
public int calculate(List<Item> items){
int sum = 0 ;
for(Item item : items){
sum += discountStrategy.getDiscountPrice(item);
}
return sum;
}
}
public interface DiscountStrategy{
int getDiscountPrice(Item item);
}
Public Class FirstGuestDiscountStratey implements
DiscountStrategy{
@Override
public int getDiscountPrice(Item item){
return (int)(item.getPrice() * 0.9;
}
}
```

여기서 가격할인을 추상화하고 있는 DiscountStrategy를 전략이라고 부르고,
가격 계산 기능 자체의 책임을 갖고 있는 Calculator 클래스를 콘텍스트(Context)
라고 부른다. 이렇게 특정 콘텍스트에서 기능을 별도로 분리하는 설계 방법이
전략 패턴이다.


정렬_인터페이스
클래스
메서드
정렬
정렬관리자
클래스
필드
pInterface
메서드
-정렬관리자
인터페이스_설정
정렬
정렬관리자
public
public
public
버블정렬
클래스
정렬_인터페이스
메서드
정렬
머지정렬
클래스
정렬_인터페이스
메서드
정렬
퀵정렬
클래스
정렬_인터페이스
메서드
정렬

```
//-------------------------------------------------------------
-------
// 정렬 인터페이스
class 정렬_인터페이스
{
public:
virtual void 정렬() = 0;
};
//-------------------------------------------------------------
-------
// 퀵 정렬 알고리즘 클래스
class 퀵정렬 : public 정렬_인터페이스
{
public:
void 정렬() override { cout << "퀵 정렬" << endl; }
};
//-------------------------------------------------------------
-------
// 버블 정렬 알고리즘 클래스
class 버블정렬 : public 정렬_인터페이스
{
public:
void 정렬() override { cout << "버블 정렬" << endl; }
};
//-------------------------------------------------------------
-------
// 머지 정렬 알고리즘 클래스
class 머지정렬 : public 정렬_인터페이스
{
public:
void 정렬() override { cout << "머지 정렬" << endl; }
};
```

```
//-------------------------------------------------------------
// 정렬 관리자 클래스
class 정렬관리자
{
public:
정렬관리자() : pInterface(0) {}
~정렬관리자() { if (pInterface) delete pInterface; }
public:
void 정렬() { pInterface->정렬(); }
void 인터페이스_설정(정렬_인터페이스* _interface)
{
if (pInterface) delete pInterface;
pInterface = _interface;
}
private:
정렬_인터페이스* pInterface;
};
//-------------------------------------------------------------
// Main
int _tmain(int argc, _TCHAR* argv[])
{
정렬관리자 *pManager = new 정렬관리자();
pManager->인터페이스_설정(new 버블정렬());
pManager->정렬();
pManager->인터페이스_설정(new 퀵정렬());
pManager->정렬();
pManager->인터페이스_설정(new 머지정렬());
pManager->정렬();
delete pManager;
return 0;
}
```