1. template method

(1) 문제설명

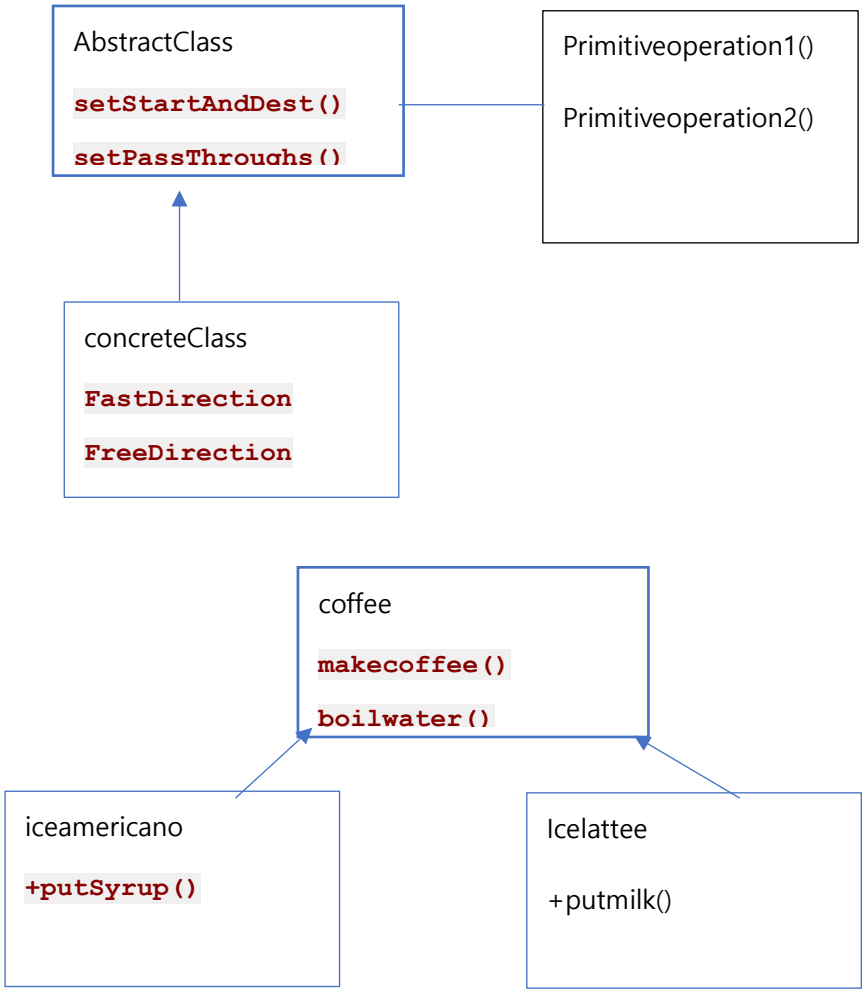
템플릿 메소드 패턴은 알고리즘의 구조를 메소드에 정의하고 하위 클래스에서 알고리즘 구조의 변경없이 알고리즘을 재정의 하는 패턴이다. 같은 역할을 하는 메소드지만 여러곳에서 다른 형태로 사용이 필요한 경우 유용한 패턴이며 상속을 통해 슈퍼클래스 기능을 확장할 때 사용하는 가장 대표적인 방법이다.

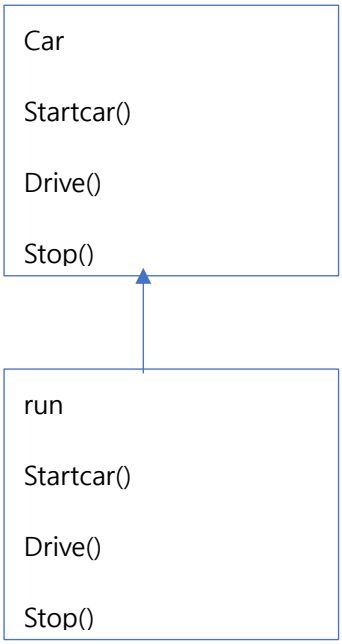
첫 번째 사례는 템플릿 메소드를 통해 서울부터 부산까지 경유지를 거쳐 길을 찾는 구조에 해당한다.

두 번째 사례는 아이스 아메리카노와 아이스 라떼를 만드는 방식의 차이를 구별하는 구조이다.

세 번째 사례를 차에 시동걸기 운행 멈춤 시동 끄기 관련한 예제이다.

(2) 클래스 다이어그램(UML)





(3) 소스코드 구현

<템플릿클래스>

```
public abstract class Direction
{ // 출발, 도착 세팅 final void setStartAndDest(String start, String end)
{ System.out.println("출발지 : " + start); System.out.println("도착지 : " + end); }
// 경유지 세팅 final void setPassThroughs(String... points) { for (String point : points)
{ System.out.println("* 경유지 : " + point); } }
// 길 찾기 abstract void getDirection(); }
```

<구현클래스>

```
public class FastDirection extends Direction { @Override void getDirection() { System.out.println("
빠른 길을 찾습니다."); } }
public class FreeDirection extends Direction { @Override void getDirection() { System.out.println("
무료 길을 찾습니다."); } }
public class CloseDirection extends Direction { @Override void getDirection()
{ System.out.println("가까운 길을 찾습니다."); } }
```

public static void main(String[] ar

gs)

{ FreeDirection direction = **new** FreeDirection(); direction.setStartAndDest("서울", "부산");

direction.setPassThroughs("천안", "대전", "대구"); direction.getDirection(); }

<템플릿클래스>

package TemplateMethodPattern

; public abstract class Coffee { final void makeCoffee() { boilWater(); putEspresso(); putIce(); putExtra(); } // SubClass에게 확장/변화가 필요한 코드만 코딩하도록 한다.

abstract void putExtra();

// 공통된 부분은 상위 클래스에서 해결하여 코드 중복을 최소화 시킨다.

private void boilWater() { System.out.println("물을 끓인다."); }

private void putEspresso() { System.out.println("끓는 물에 에스프레소를 넣는다."); }

private void putIce() { System.out.println("얼음을 넣는다."); } }

<IceAmericano 클래스>

package TemplateMethodPattern;

public class IceAmericano extends Coffee{ @Override void putExtra() { System.out.println("시럽을 넣는다."); } }

<icelattee클래스>

package TemplateMethodPattern; public class IceLatte extends Coffee{ @Override void putExtra() { System.out.println("우유를 넣는다."); } }

<coffeemain 클래스>

package TemplateMethodPattern

; public class CoffeeMain {

public static void main(String[] args)

```
{ IceAmericano americano = new IceAmericano();
IceLatte latte = new IceLatte(); americano.makeCoffee();
System.out.println("==="); latte.makeCoffee(); } }
```

<자동차 관련 예제 소스코드3>

```
public abstract class car{
protected abstract void startcar();
protected abstract void drive();
protected abstract void stop();
protected abstacet void turnoff();

public final void run(){
startcar();
drive();
stop();
turnoff();
}
```

(4) discussion

Template method 패턴은 소스코드 재사용을 위한 기법으로 클래스 라이브러리 구축 시 공통된 부분을 추출해내기 위한 수단으로 사용된다 또한 상위클래스의 템플릿 메소드에 알고리즘이 기술되어 있기에 하위 클래스에서는 알고리즘을 일일이 구현할 필요없이 해당부분만 구현하면 되는 장점이 존재한다.

2. 이터레이터와 컴포지트 패턴

(1) 문제 설명

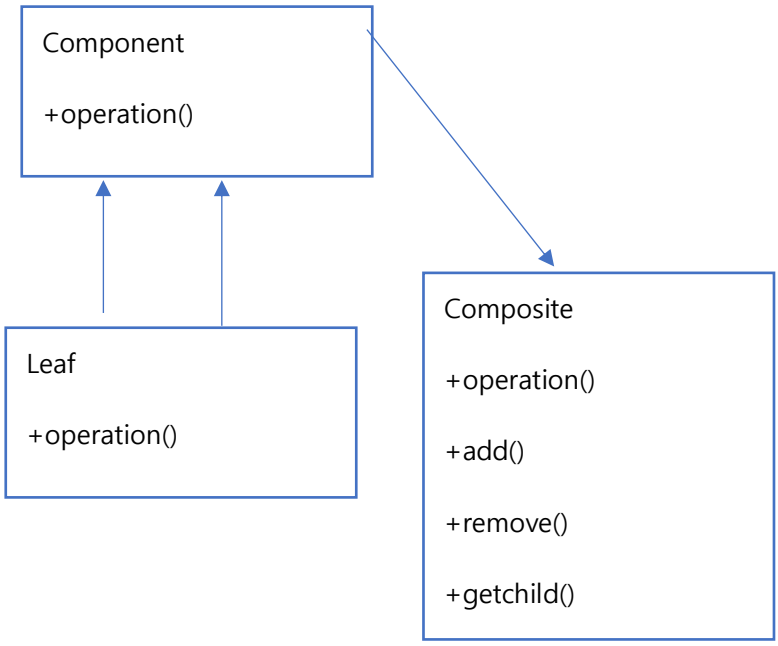
이터레이터 패턴은 컬렉션 구현 방법을 노출시키지 않으면서도 그 집합체 안에 들어있는 모든 항목에 접근할 수 있도록 해주는 방법을 제공하는 패턴을 의미하며 컬렉션들을 공통된 인터페이스를 구현하게 함으로써 공통된 방법으로 모든 항목에 접근할 수 있는 방법이다.

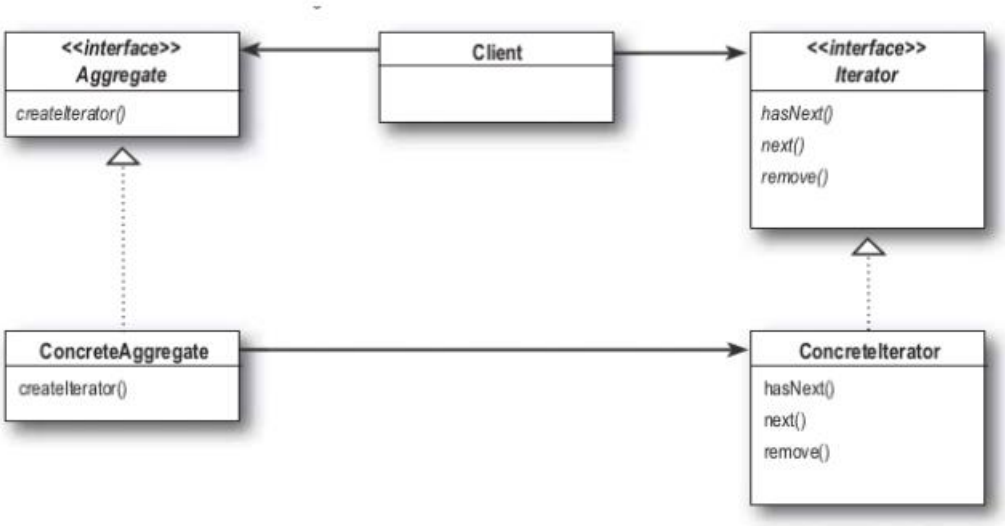
첫 번째 사례에 경우 펜케이크하우스 런치메뉴, 카페메뉴, 브런치 메뉴 디저트 메뉴를 나눠서 각각 메뉴와 설명에 대해 나와있는 예제이다.

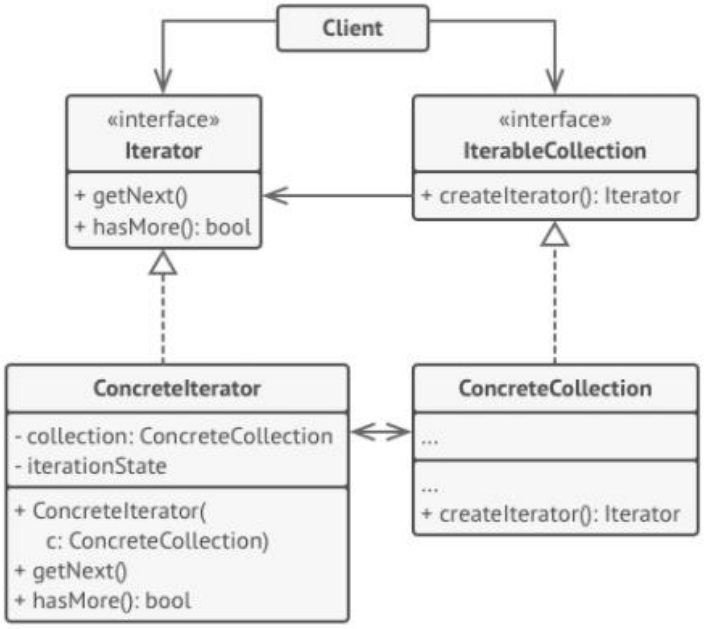
두 번재 사례에 경우 런치 메뉴와 저녁식사메뉴를 나누고 점심은 토스트 세트 및 아보카도 토스트 세트를 저녁메뉴는 알리오 파스타와 마르게리타 피자를 나눠서 화면에 출력하는 예제이다

세 번째 사례의 경우 concreteiterator literableCollection concreteCollection으로 나누어 화면에 출력하는 예제이다

(2) 클래스 다이어그램







(3) 소스코드

<MenueComponent클래스>

Public abstract class menuComponent{

Public void add(menucomponent menucomponent){

Throw new unsupportedoperationexception();

}

Public void print(){

```
Throw new unsupportedOperationException();
}
}
```

<menu item 클래스>

```
Public class meunitem extends menucomponent{
Private String name;
Private String description;
Double price
Public void print(){
System.out.println("-"+getName()+";" + getprice()));
}
```

<menu 클래스>

```
Public class menu extends menucomponent{
Arraylist menucomponents = new AraayList();
String name;
String description;

Public void print(){
System.out.println(getName()+","+getdescription());
System.out.println("--------------------");
Iterator iterator = menuComponents.Iterator();
While (iterator.hasNext()){
MenuComponent menucomponent =
(Menucomponent.print():
}
System.out.println();
```

}

}

<main 클래스>

Public class Main{string[] args}{

Menucomponent pancakeHouseMenu = new Menu (" 팬케이스 하우스 메누", "런치 메뉴");

Menucomponent cafeMenu = new Menu (" 카페메뉴", "브런치 메뉴");

Menucomponent dessertMenu = new Menu (" 디저트메뉴");;

allMenus.print();

}

}

ConcreteAggregate

```
public class MenuItem {
    private String name;
    private String description;
    private double price;

    public MenuItem(String name, String description, double price) {
        this.name = name;
        this.description = description;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public double getPrice() {
        return price;
    }
}
```

LunchMenu

```
public class LunchMenu implements Menu {

    private List<MenuItem> menuItems;

    public LunchMenu() {
        menuItems = new ArrayList<>();

        addItem("토스트 세트"
                , "기본 토스트에 아메리카노 포함"
                , 4000);

        addItem("아보카도 토스트 세트"
                , "아보카도와 다양항 야채들이 들어간 샌드위치에 아메리카노 포함"
                , 5000);

    }

    private void addItem(String name, String description, int price) {
        MenuItem menuItem = new MenuItem(name, description, price);
        menuItems.add(menuItem);
    }


    @Override
    public Iterator createIterator() {
        return menuItems.iterator();
    }
}
```

DinnerMenu

```
public class DinerMenu implements Menu {

    private Map<String, MenuItem> menuItems;

    public DinerMenu() {
        menuItems = new HashMap<>();

        addItem("알리오 올리오 파스타"
                , "베이컨, 마늘, 핫페퍼가 들어간 파스타"
                , 10000);

        addItem("마르게리타 피자"
                , "토마토, 모차렐라, 바질이 들어간 피자"
                , 12000);

    }

    private void addItem(String name, String description, int price) {
        MenuItem menuItem = new MenuItem(name, description, price);
        menuItems.put(menuItem.getName(), menuItem);
    }


    @Override
    public Iterator createIterator() {
        return menuItems.values().iterator();
```

Client

```
public class Waitress {
    private Menu lunchMenu;
    private Menu dinerMenu;

    public Waitress(Menu lunchMenu, Menu dinerMenu) {
        this.lunchMenu = lunchMenu;
        this.dinerMenu = dinerMenu;
    }

    public void printMenu() {
        Iterator lunchIterator = lunchMenu.createIterator();
        Iterator dinerIterator = dinerMenu.createIterator();
        System.out.println("===== 점심 식사 메뉴 =====");
        printMenu(lunchIterator);
        System.out.println("===== 저녁 식사 메뉴 =====");
        printMenu(dinerIterator);
    }

    private void printMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = (MenuItem) iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getDescription() + ", ");
            System.out.println(menuItem.getPrice());
        }
    }
}
```

```
using System;
using System.Collections;
using System.Collections.Generic;
namespace RefactoringGuru.DesignPatterns.Iterator.Conceptual
{
abstract class Iterator : IEnumerator
{
object IEnumerator.Current => Current();
public abstract int Key();
public abstract object Current();
public abstract bool MoveNext();
public abstract void Reset();
}
abstract class IteratorAggregate : IEnumerable
{
public abstract IEnumerator GetEnumerator();
}
class AlphabeticalOrderIterator : Iterator
{
private WordsCollection _collection;
private int _position = -1;
private bool _reverse = false;
public AlphabeticalOrderIterator(WordsCollection collection, bool reverse = false)
```

```
{
this._collection = collection;
this._reverse = reverse;
if (reverse)
{
this._position = collection.getItems().Count;
}
}
public override object Current()
{
return this._collection.getItems()[_position];
}
public override int Key()
{
return this._position;
}
public override bool MoveNext()
{
int updatedPosition = this._position + (this._reverse ? -1 : 1);
if (updatedPosition >= 0 && updatedPosition < this._collection.getItems().Count)
{
this._position = updatedPosition;
return true;
}
else
{
return false;
}
}
public override void Reset()
{
this._position = this._reverse ? this._collection.getItems().Count - 1 : 0;
}
}
class WordsCollection : IteratorAggregate
{
List<string> _collection = new List<string>();
bool _direction = false;
public void ReverseDirection()
```

```
{
_direction = !_direction;
}
public List<string> getItems()
{
return _collection;
}
public void AddItem(string item)
{
this._collection.Add(item);
}
public override IEnumerator GetEnumerator()
{
return new AlphabeticalOrderIterator(this, _direction);
}
}
class Program
{
static void Main(string[] args)
{
var collection = new WordsCollection();
collection.AddItem("하나");
collection.AddItem("둘");
collection.AddItem("셋");
Console.WriteLine("정방향 순회:");
foreach (var element in collection)
{
Console.WriteLine(element);
}
Console.WriteLine("₩n 역방향 순회:");
collection.ReverseDirection();
foreach (var element in collection)
{
Console.WriteLine(element);
}
}
}
}
```

(4) discussion

이터레이터 및 컴포지트 패턴은 구조가 복잡한 집합 객체 내에서 어떤 식으로 일이 처리되는지에 대해 전혀 모르는 상태에서 그안에 들어있는 모든항목들에 대해 반복작업을 수행할 수 있고 컬렉션 내에 들어있는 모든 항목에 저브간하는 방식이 통일되어 있으므로 어떤 종류의 집합체에 대해서도 사용할 수 있는 다형적인 코드를 만들 수 있다.

3. State Pattern(상태 패턴)

(1)문제 설명

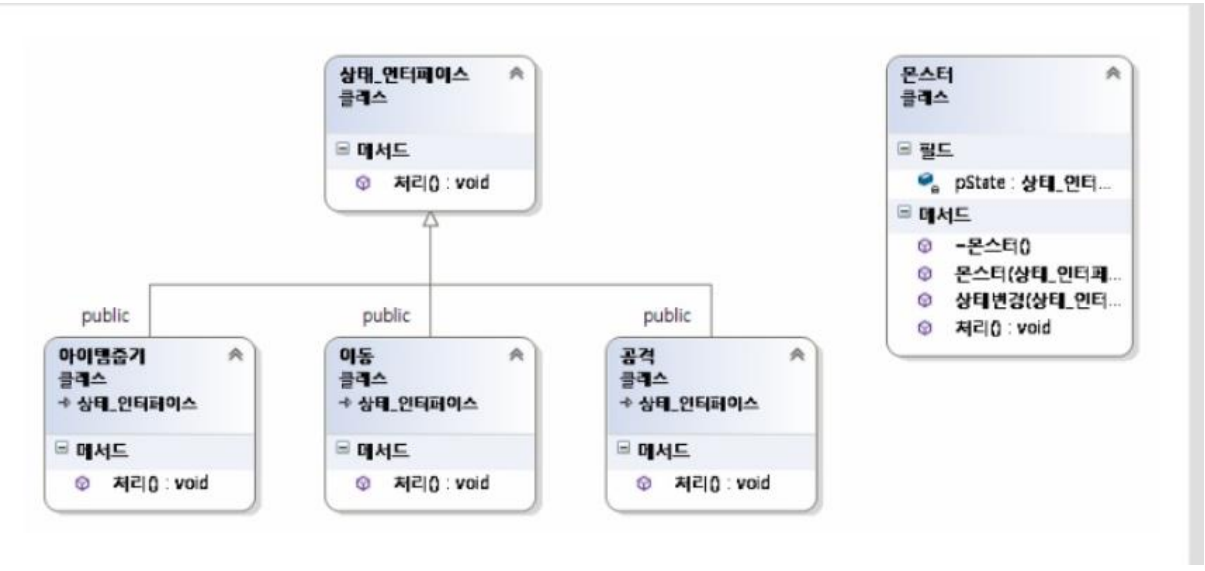
상태 패턴이란 객체의 상태에 따라 각각의 행위를 변경할 수 있게 캡슐화하는 패턴을 의미한다. 동적으로 행동을 교체할 수 있으며 전략 패턴과 구조는 거의 동일하나 쓰임의 용도가 다르다는 것이 큰 차이점이다.

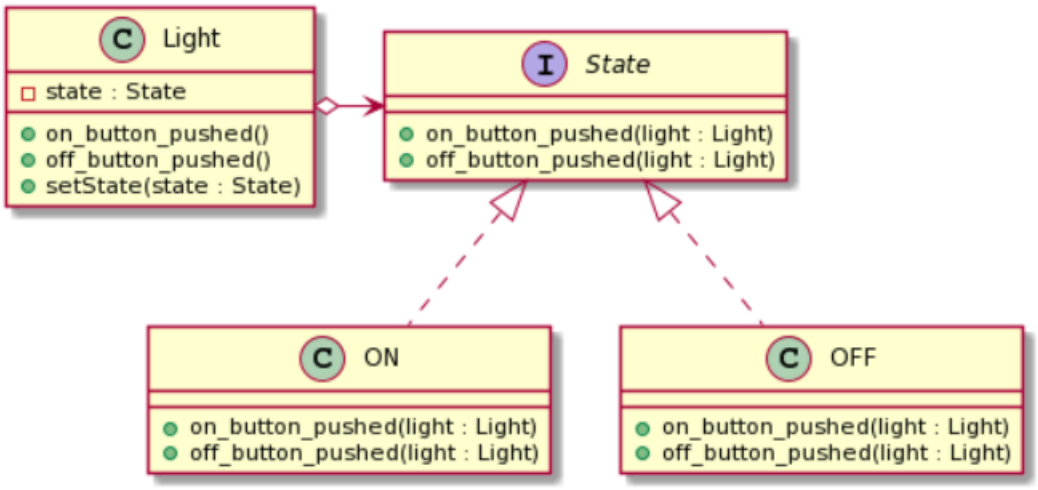
첫 번째 예제는 몬스터 처리 예제이고 아이템을 줍고 이동하며 상대방을 공격해서 몬스터를 쓰러트리는 에제이다

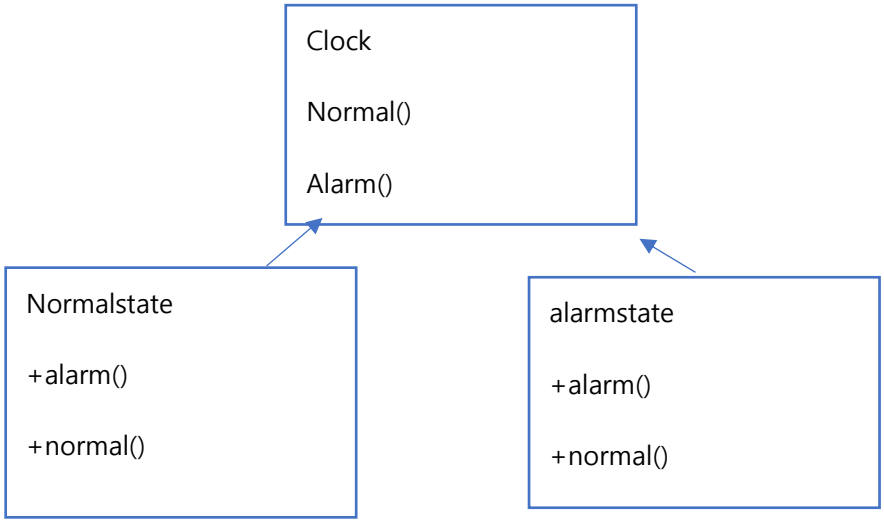
두 번째 예제는 형광등 클래스 예제이며 형광등을 키면 on 버튼을 누르고 형광등이 꺼져있으면 off 버튼을 누르는 예제이다

세 번째 예제는 시계의 상태에 관한 예제이며 normal 일때와 alarm이 울릴 때를 나눠서 작동하는 예제이다

(2) 클래스 다이어그램







(3) 소스코드

Class state_inteface

{

Public:

Virtual void handling() = 0;

};

Class move : public state_interface

```
{
Public :
Void handling() @override{cout<<"이동"<<endl;}
};
Class Attack : public state_inerface
{
Public :
Void handling() @override {cout<<"공격"<<endl;}
};

Class itemget :public state_interface
{
Public :
Void handling() @override {cout<< " 아이템줍기"<<endl;}
};
Class monster
{
Public :
Monster(state_inerface* state):pstate(state){
~monster(){
If (pstate ) delete p statae: }

Public :
Void stateinvariante(state_interface* state){if (pstate delete pstate: pstate = state);
Void handling(){pstate->handling();}
Private:
```

```
State_interface* pstate;
};

Int main(int arge, _TCHAR* argv[])
{
Monster* pMonster = new monster (new move());
pMonster -> handling();
pMonster -> statevarianet(new attack());
pMonster -> handling();

delete pMonster;
return 0;
}


Public class Light{
Private static int ON = 0;
Private static int OFF = 1;
Private int state;

Public Light(){
State = OFF;
}

Public void on_buttin_pushed(){
If(state == ON){
```

```
System.out.println{"반응없음"};
}else {
System.out.println("Light on!");
State = ON;
}
}
Public void off_button_pushed(){
If(state == OFF){
System.out.println("반응  없음");
}else {
System.out.println("light Off!");
State = OFF;
}
}
}

Public class Clock{
Private State state;
Public Clock(){
State = new NormalState();
}
Public void setState(State state){
This.state = state;
}
Public void normal(){
State.normal(this);
```

```
}
Public void alarm(){
State.alarm(this);
}
}

Class NormalState implements State{
Clock clock;
NormalStaate(){
System.out.println("Normal 상태");
}
@override
Public void normal (Clock clock){
System.out.println("normal normal Method 호출");
}
@override
Public void alarm(Clock clokc){
System.out.println("normal alarm Method 호출");
System.out.println("Alarm을 울립니다.");
Clock.setState(new AlarmState());
}
}
Class AlarmState implements State{
Int min;
Final int ALARM_TIME=5;
Public AlarmState(){
```

```
System.out.println("Alarm 상태");

Min = 0;

}

@override

Public void normal(Clock clokc){

While (min <= ALARM_TIME){

Sysytem.out.println("min : "+min);

Min++;

}

Clock.setState(new NormalState());

}

@override

Publc void alarm(Clock clock){}

}
```

(4) discussion

스테이트 패턴은 전략패턴과 많이 유사한 것을 확인할 수 있다. 그러나 두 패턴에는 차이점이 존재하는데 전략 패턴의 경우 전략 변경에 일정함이 없지만, 스테이트 패턴의 경우 각 state 간에 일정한 규칙을 가지고 변한다는 점에 차이점이 있으면 state 객체는 하나만 존재하더라도 상관없다는 특성을 지니기에 state 객체들은 싱글턴 패턴을 적용하여 구현하면 좀 더 효율적으로 코드를 개선할 수 있다

4. proxy pattern

(1) 문제설명

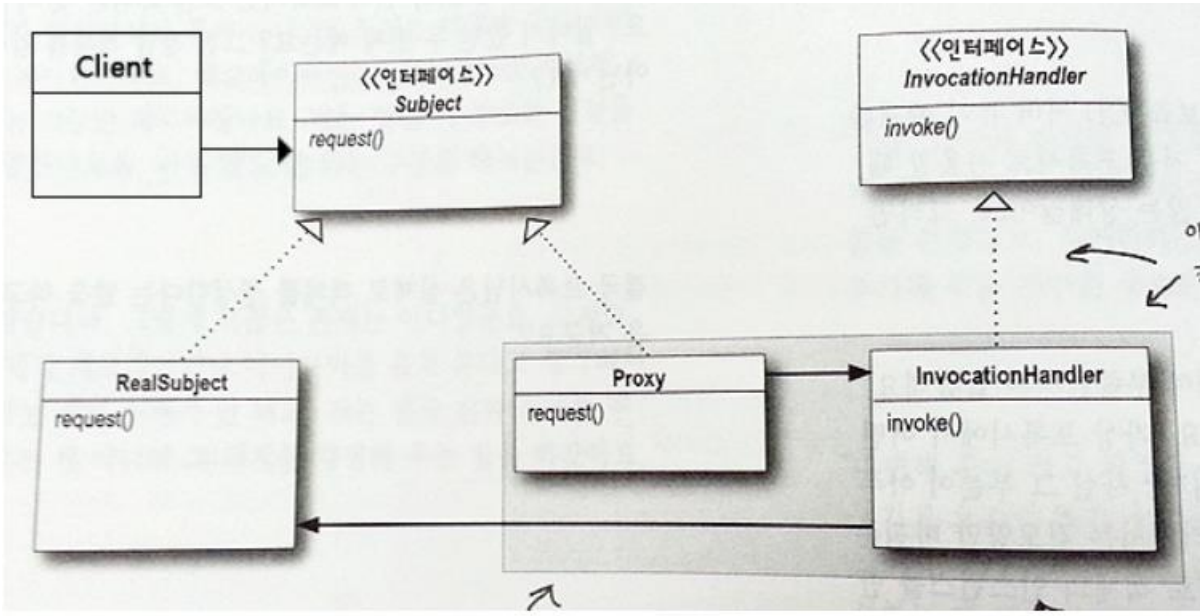
프록시 패턴이란 실제 기능을 수행하는 객체 대신 가상의 객체를 사용해 로직의 흐름을 제어하는 디자인 패턴을 의미한다.

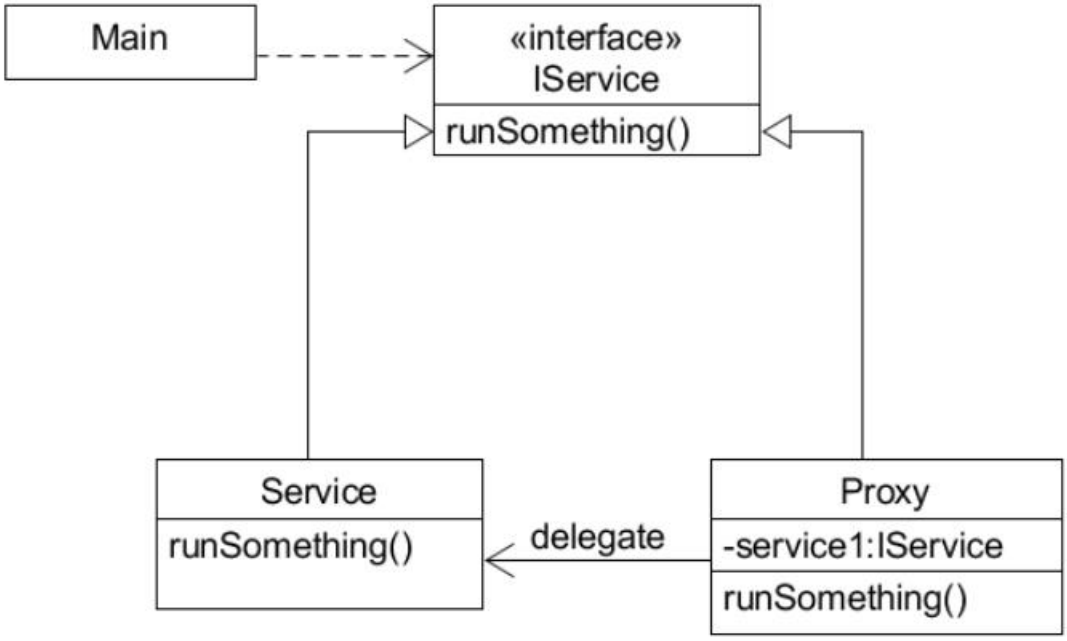
첫 번째 예제에 경우 보호 프록시 패턴의 예제이며 결혼 정보회사의 회원 정보 관련 예제로서 회원 정보인 당사자의 경우에는 기본정보 사항에 대한 set,get 권한만을 주고 일반 정보회사 이용자들에게는 get 권한과 해당 회원에 대한 점수만 매길 수 있도록 권한을 준다

두 번째 예제의 경우 main에서 service의 runSomething()매서드를 직접 호출하지 않고 proxy에게 대신시키며 proxy는 service의 인스턴스를 가지며 직접service의 runsomething()메서드를 호출하는 구조이다

세 번째 예제의 경우 실제 서비스, 프록시, 클라이언트로 구분하여 프록시 패턴을 구현한 예제이다

(2) 클래스 다이어그램(UML)





(3) 소스코드

```
public interface PersonBean {
    String getName();
    String getGender();
    String getInterest();
```

```
    int getHotOrNotRating();

    void setName(String name);
    void setGender(String gender);
    void setInterest(String interest);
    void setHotOrNotRating(int rating);
}<font face="맑은 고딕, sans-serif"><span style="white-space: normal;">
</span></font>


public class NonOwnerInvoctionHandler implements InvocationHandler {
    PersonBean person;

    public  NonOwnerInvoctionHandler(PersonBean person) {
        this.person = person;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
IllegalAccessException {
        try {
            if(method.getName().startsWith("get") == true) {
                return method.invoke(person, args);
            }
            else if (method.getName().equals("setHotOrNotRating") == true) {
                return method.invoke(person, args);
            }
            else {
                System.out.println("IllegalAccessException");
                return new IllegalAccessException();
            }
        }
        catch (InvocationTargetException e) {
            e.printStackTrace();
        }
        return null;
    }
}
public class OwnerInvoctionHandler implements InvocationHandler {
    PersonBean person;
```

```
    OwnerInvoctionHandler(PersonBean personBean) {
        this.person = personBean;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
IllegalAccessException {
            try {
                if(method.getName().startsWith("get")) {
                    return method.invoke(person, args);
                }
                else if(method.getName().equals("setHotOrNotRating")) {
                    System.out.println("IllegalAccessException");
                    new IllegalAccessException();
                }
                else if(method.getName().startsWith("set")) {
                    return method.invoke(person, args);
                }

            }
            catch (InvocationTargetException e) {
                e.printStackTrace();
            }
            return null;
        }
}
public static void main(String[] args) {
        Main main = new Main();
        PersonBean person1 = new PersonBeanImpl();

        PersonBean ownerProxy = main.getOwnerProxy(person1);
        ownerProxy.setName("홍길동");
        ownerProxy.setGender("남자");
        ownerProxy.setInterest("무협소설 읽기");
        ownerProxy.setHotOrNotRating(10); // ownerProxy 이기 때문에 실제 실행이 안되는 것을 볼
수 있다.

        PersonBean nonOwnProxy = main.getNonOwnerProxy(person1);
```

```
        nonOwnProxy.setName("박길동");  // nonOwnProxy 이기 때문에 setName 은  실행이 안되는 것을 볼 수 있다.
        ....
    }

    PersonBean getOwnerProxy(PersonBean person)
    {
        return (PersonBean) Proxy.newProxyInstance(
                person.getClass().getClassLoader(),
                person.getClass().getInterfaces(),
                new OwnerInvoctionHandler(person));
    }

    PersonBean getNonOwnerProxy(PersonBean person)
    {
        return (PersonBean) Proxy.newProxyInstance(
                person.getClass().getClassLoader(),
                person.getClass().getInterfaces(),
                new NonOwnerInvoctionHandler(person));
    }
```

```
public interface IService {

    String runSomething();
}


public class Service implements IService{

    @Override
    public String runSomething() {
        return "서비스 짱!!!";
    }

}
public class Proxy implements IService{

    IService service1;
```

```
    @Override
    public String runSomething() {
        System.out.println("호출에 대한 흐름 제어가 주목적, 반환 결과를 그대로 전달");

        service1 = new Service();
        return service1.runSomething();
    }

}
public class Main {

    public static void main(String[] args) {
        //직접 호출하지 않고 프록시를 호출한다.
        IService proxy = new Proxy();
        System.out.println(proxy.runSomething());
    }
}
```

```
Public interface Service{

Void requestapi();

}


Public class Service implements Service{

@override

Public void requestapi(){

}

}


Public class Proxy implements Service{

Private Service mService;

@override
```

```
Public void requestApi(){
mService = new Service();
mService.requestApi();
}
}

Public class Client{
Public statice void main(final String[] args){
Final proxy proxy = new proxy();
Proxy.requestApi();
}
}
```

(4) discussion

프록시 패턴의 경우 단순히 보안상의 이유만의 문제에서 더 나아가 프록시 서버는 프록시 서버에 요청된 내용들을 캐시를 이용해 저장해 둔다. 이렇게 캐시를 해두고 난 후에 만약 클라이언트 쪽에서 캐시 안에 있는 정보를 요구하는 요청을 할 경우, 프록시 서버는 원격 서버에 접속하여 데이터를 가져올 필요가 없게 된다. 이는 전송 시간을 절약할 수 있음과 동시에 불필요하게 외부와의 연결을 하지 않아도 된다는 장점을 갖게 된다.

5. compound pattern

(1) 문제설명

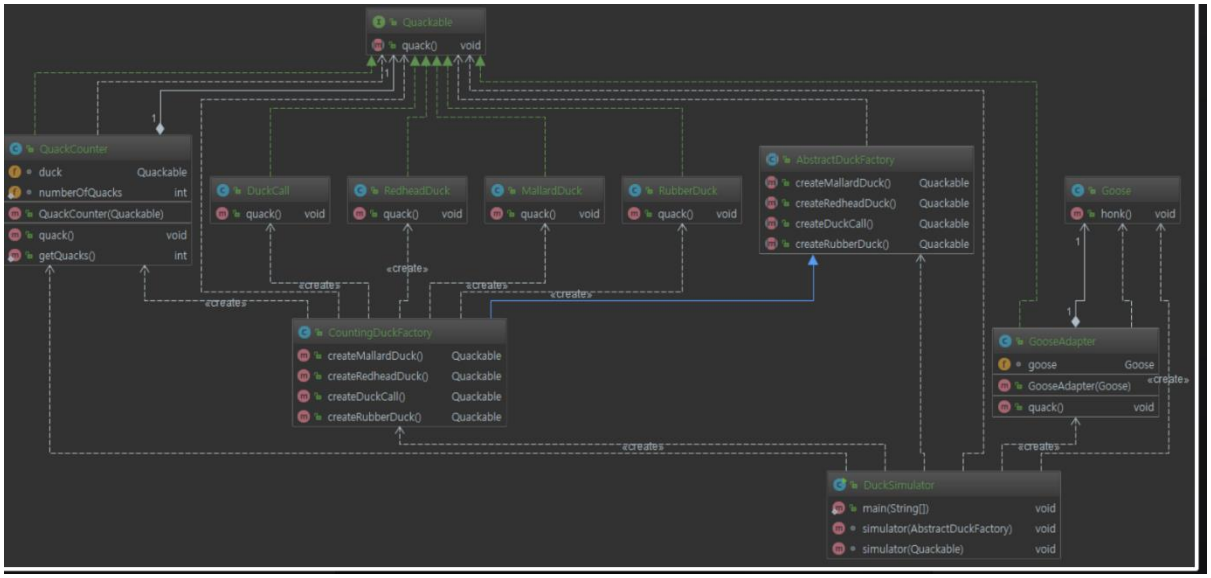
컴파운드 패턴이란 두 개 이상의 패턴을 결합하여 일반적으로 자주 등장하는 문제들에 대한 해법을 제공하는 패턴을 의미한다.

첫 번째 예제는 팩토리 패턴과 컴포지트 패턴 그리고 옵저버 패턴을 이용하여 오리가 내는 소리와 duck을 관리하고 객체를 만드는 방식의 예제이다

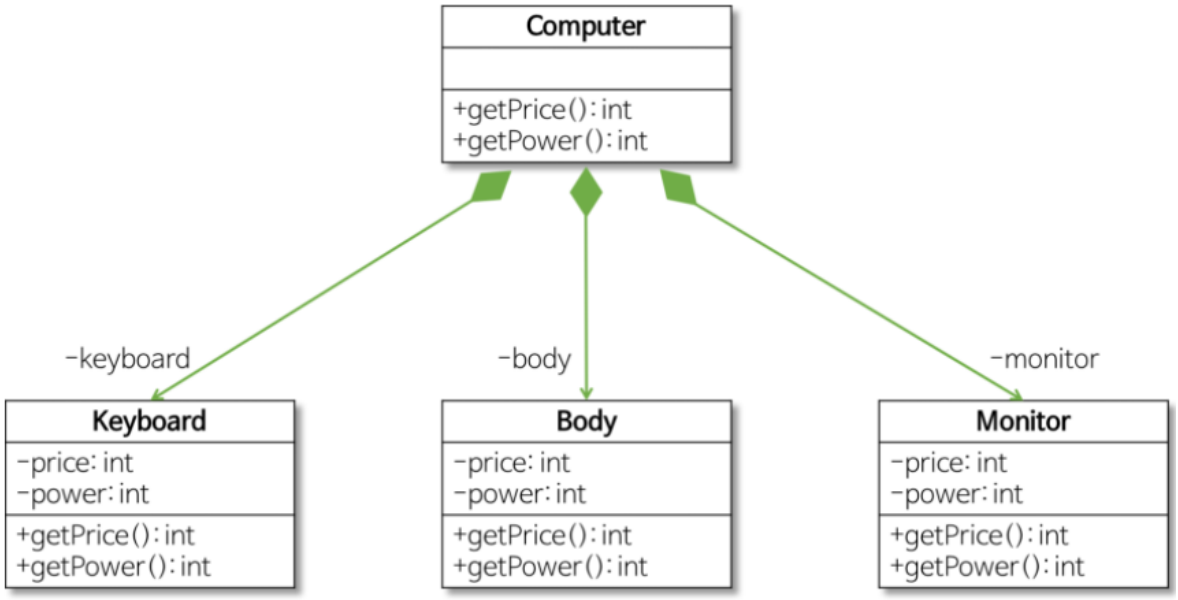
두 번째 예제는 인터페이스 내에 각각 도형마다 색을 채우는 예제에 해당한다

세 번째 예제는 컴퓨터에 키보드 본체 모니터 클래스를 추가하는 예제에 해당한다.

(2) 클래스 다이어그램



•



(3) 소스 코드

```
Public interface Quackable{

Public void quack();

}

Public class MallarDuck implements Quackable{

@override

Public void quack(){

System.out.println(“Quack”);

}

}

Public class RedheadDuck implements Quackable{

@override

Public void quack(){

Sysstem.out.println(“quack”);

}
```

```
}
Public class RubberDuck implements Quackable{
@overrid
Public void quack(){
System.out.prinln(“Squeak”);
}
}
Public class DuckCall implements Quakcable{
@override
Public void quack(){
System.out.printlm(“Kwak”);
}
}

Public class GooseAdapter implements Quackable{
Goose goose;
Public GooseAdapter(Goose goose){
This.goose = goose;
}
@override
Public void quack(){
Goose.honk();
}
}
Public class Goose{
Public void honk(){
```

```
System.out.println(“Honk”);
}
}

Public abstract class AbstractDuckFactory{
Public abstact Quackable createMallarDuck();
Public abstract Quackable createRedheadDuck();
Public abstract Quackable createDuckCall();
Public abstract Quackable createRubberDuck();
}

Public class DuckSimulatoer{

Public static void main(String[] args){
DuckSimulator simulator = new Duck Simulator();
AbstractDuckFactory duckFactory = new CountingDuckFactory();
Simulator.simulator(duckFactory);
}

Void simulator(AbstrackDuckFactory duckFactory){
Quackalbe mallardDuck = duckFactory.createMallarDuck();
Quackalbe redheadDuck = duckFactory.createredheadDuck()
Quackalbe duckCall = duckFactory.createDuckcall();
Quackalbe rubberDuck = duckFactory.createrubberDuck();
Quackalbe gooseDuck = duckFactory.createGooseAdapter();
Sysyem.out.println(“₩nDuck Simulator”);
```

```
}
```

```
public interface Shape {

    public void draw(String fillColor);

}
    public class Triangle implements Shape {

        @Override
        public void draw(String fillColor) {
            System.out.println("Drawing Triangle with color "+fillColor);
        }
    }
public class Circle implements Shape {

    @Override
    public void draw(String fillColor) {
        System.out.println("Drawing Circle with color "+fillColor);
    }
}
public class Drawing implements Shape {

    //collection of Shapes
    private List<Shape> shapes = new ArrayList<Shape>();

    @Override
    public void draw(String fillColor) {
        for(Shape sh : shapes) {
```

```
            sh.draw(fillColor);
        }
    }

    //adding shape to drawing
    public void add(Shape s) {
        this.shapes.add(s);
    }

    //removing shape from drawing
    public void remove(Shape s) {
        shapes.remove(s);
    }

    //removing all the shapes
    public void clear() {
        System.out.println("Clearing all the shapes from drawing");
        this.shapes.clear();
    }
}
    public class TestCompositePattern {

        public static void main(String[] args) {
            Shape tri = new Triangle();
            Shape tri1 = new Triangle();
            Shape cir = new Circle();

            Drawing drawing = new Drawing();
```

```
                drawing.add(tri1);
                drawing.add(tri1);
                drawing.add(cir);

                drawing.draw("Red");

                List<Shape> shapes = new ArrayList<>();
                shapes.add(drawing);
                shapes.add(new Triangle());
                shapes.add(new Circle());

                for(Shape shape : shapes) {
                      shape.draw("Green");
                }
          }
     }
```

```
public class Keyboard {
   private int price;
   private int power;
   public Keyboard(int power, int price) {
      this.power = power;
      this.price = price;
   }
   public int getPrice() { return price; }
   public int getPower() { return power; }
}
public class Body { 동일한 구조 }
public class Monitor { 동일한 구조 }
public class Computer {
   private Keyboard Keyboard;
```

```
    private Body body;
    private Monitor monitor;

    public addKeyboard(Keyboard keyboard) { this.keyboard = keyboard; }
    public addBody(Body body) { this.body = body; }
    public addMonitor(Monitor monitor) { this.monitor = monitor; }

    public int getPrice() {
        int keyboardPrice = keyboard.getPrice();
        int bodyPrice = body.getPrice();
        int monitorPrice = monitor.getPrice();
        return keyboardPrice + bodyPrice + monitorPrice;
    }
    public int getPower() {
        int keyboardPower = keyboard.getPower();
        int bodyPower = body.getPower();
        int monitorPower = monitor.getPower();
        return keyboardPower + bodyPower + monitorPower;
    }
}
public class Client {
    public static void main(String[] args) {
        // 컴퓨터의 부품으로 Keyboard, Body, Monitor 객체를 생성
        Keyboard keyboard = new Keyboard(5, 2);
        Body body = new Body(100, 70);
        Monitor monitor = new Monitor(20, 30);
```

```
// Computer 객체를 생성하고 부품 객체들을 설정
Computer computer = new Computer();
computer.addKeyboard(keyboard);
computer.addBody(body);
computer.addMonitor(monitor);

// 컴퓨터의 가격과 전력 소비량을 구함
int computerPrice = computer.getPrice();
int computerPower = computer.getPower();
System.out.println("Computer Price: " + computerPrice + "만원");
System.out.println("Computer Power: " + computerPower + "W");
```

(4) discussion

컴포지트 패턴의 장점은 기본객체와 구성 객체를 특별히 구분하지 않고 소스코드를 작성 가능하고 모두 동일한 자료형으로 구성되어 있기에 새로운 클래스 추가가 용의하다는 장점이 존재하지만 단점으로는 설계를 일반화시켜 객체간의 구별하기 어려우며 특별히 제약을 가하기 힘들다는 단점이 존재한다