**프로그래밍 언어론 과제**

**2011903054 행정학과**

**문석현**

**1. strategy pattern**

(1) 개념

Strategy pattern의 의미는 알고리즘 군을 정의하고 (strategysort 추상 클래스) 같은 알고리즘(버블 정렬, 퀵 정렬, 선택 정렬 등)을 각각 하나의 클래스로 캡슐화한 (quicksort 클래스, selectsort클래스, bubblesort 클래스) 다음, 필요할 때 서로 교환해서 사용할 수 있게 해준다. 즉 스트래티지 패턴의 알고리즘군의 경우 바뀌는 부분만을 빼서 캡슐화를 진행한 것이라 볼 수 있다.

(2). 문제 설명

1) duck 문제

가) duck 예제

Duck 예제의 경우에는 오리라는 하나의 부모 클래스를 생성해준다 이 duck 클래스의 경우에는 quack() , swim(), perfomfly() 등 자식 클래스에 공통적으로 상속되는 부분을 선언해준다 이를 통해 우리는 자식 클래스에서 해당 기능을 새로 구현하지 않아도 <상속>이라는 과정을 통해 효율적으로 구현해낼 수 있다. 그러나 서브 클래스에 코드가 중복되거나, 모든 오리의 행동을 예측할 수 없다는 점, 코드를 변경했을 때 자식클래스에 원하지 않는 영향을 미칠 수 있다는 단점이 있다. 따라서 이를 부모클래스, 자식클래스, interface로 분리하여 해당 부분을 해결할 수 있다.

나). 클래스 다이어그램(UML)

DUCK

+swim()

+quak()

+perfomfly()

<<interface>>

Flyable

FlyNoWithwings

+fly()

FlyWithWings

+fly()

Rubber Duck

+display()

Toy duck

+display()

Red duck

+display()

3) 소스코드

Public abstract class Duck {

Swim, Quack, performFly;

} // Duck 클래스가 부모클래스이므로 abstract를 사용

Public abstract void display(); // display 함수의 경우 부모 클래스와 자식클래스의 display가 다르므로 클래스에서 따로 분리해서 사용

Public void performFly(){

Flybehavir.fly();

}

Interface Duck{

Void fly();

}

Class flywithwings implements Duck{

Public void fly(){

System.out.println(“오리가 날고 있다.”);

}

}

Class flynowithwings implements Duck{

Public void fly(){

System.out.println(“오리가 날지 못한다.”);

}

}

Class ruberduck{

Public void fly(){

System.out.println(“고무오리가 보인다.”);

}

}

Class red duck{

Public void fly(){

System.out.println(“빨간오리가 보인다.”);

}

}

Class Toy duck{

Public void fly(){

System.out.println(“장난감오리가 보인다.”);

}

}

2) 스트리트 파이터 예제

가) 예제 설명

여러 파이터들의 행동을 구현한다. Fighter의 행동은 punch, kick jump,roll 등으로 구현되고 jumpbehavir kickbehavior가 interface로 구현된다 fighter class의 자식 클래스로서 ryu ken이 있다

나) 클래스 다이어그램 구현

<<interface>>

kickBehavior

Fighter

+punch()

+kick()

+jump() , +roll()

TornadoKick

+kick()

Lightningkick

+kick()

RYU

+display()

Ken

+display()

<<interface>>

jumpBehavior

longjump

kickBehavior

shortjump

+jump()

3)소스 코드

Public interface KickBehavior {

Public void kick();

}

Public interface jumpBehavior{

Public void jump();

}

Public class LightningKick implements KickBehavior

@override

Public void kick(){

System.out.println(“lightning kick”);

}

}

Public class TornadoKick implements KickBehavior

@override

Public void kick(){

System.out.println(“Tornado kick”);

}

}

Public class Longjump implements jumpbehavior{

@override

Public void jump(){

System.out.println(“Long jump”);

}

}

Public class shortjump implements jumpBehavior{

@overrid

Public void jump(){

System.out.println(“short jump);

}

}

Public abstract class Fighter{

kickBehavior kickBehavior;

jumpbehavir jumpbehavior;

public Fighter(kickBehavior kickBehavior, jumpBehavior jumpBehavior)

{

This.kickBehavior = kickBehavior;

This.jumpbehavior = jumpbehavior;

}

Public void punch(){

System.out.printlm(“Default punch”);

}

Public void kick(){

kickBehavior.kick();

}

Public void jump(){

jumpBehavior.jump();

}

Public void roll(){

System.out.println(“Default Roll”);

}

Public void setKickBehavior(kickbehavior kickbehavior)

{

This.kickBehavior = kickBehavior;

}

Public void setjumpbehavior(jumpbehavior jumpbehavior){

This.jumpbehavior = jumpbehavior;

}

Public abstract void display();

}

Public class Ryu extends Fighter{

Public Ryu(kickBehavior kickbehavior, jumpbehavior jumpbehavior){

Super(kickbehavior, jumpbehavior);

}

@override

Public void display(){

System.out.println(“RYU”);

}

}

Public class Ken extends Fighter{

Public ken(kickBehavior kickBehavior, jumpBehavior jumpBehavior)

Super(kickBehavior, jumpBehavior);

}

@override

Public void display(){

System.out.println(“ken”);

}

}

3) 판타지 게임 예제

가) 예제 개념

person이라는 class를 만든다 person은 masician과 swordsman 이라는 자식 클래스를 가지고 euipweapon 그리고 attack 을 interface로 가진다

나) 클래스 다이어그램

<<interface>>

Attack

Person

+getAttack()

+setAttack()

+doAttack() ,+seEquiWeapon()

attackPhysical

attackMasical

<<interface>>

EquipWeapon

Swordman

+super()

Masician

+super()

equipWeaponstaff

equipWeaaponsword

다) 코드 구현

public interface Attack {

    public void action();

}

public class AttackMasical implements Attack {

    @Override

    public void action() {

        System.out.println("마법 공격");

    }

}

public class AttackPhsical implements Attack{

    @Override

    public void action() {

        System.out.println("물리 공격");

    }

}

public interface EquipWeapon {

    public void action();

}

public class EquipWeaponStaff implements EquipWeapon{

    @Override

    public void action() {

        System.out.println("지팡이 장착");

    }

}

public class EquipWeaponSword implements EquipWeapon{

    @Override

    public void action() {

        System.out.println("검 장착");

    }

public class Person {

    private Attack attack;

    private EquipWeapon equipWeapon;

    private String name;

    public Person(String job, EquipWeapon equipWeapon, Attack attack) {

        this.name = job;

        this.equipWeapon = equipWeapon;

        this.attack = attack;

        System.out.println("직업 : " + job);

    }

    public Attack getAttack() {

        return attack;

    }

    public void setAttack(Attack attack) {

        this.attack = attack;

    }

    public EquipWeapon getEquipWeapon() {

        return equipWeapon;

    }

    public void setEquipWeapon(EquipWeapon equipWeapon) {

        this.equipWeapon = equipWeapon;

    }

    public void doAttack() {

        attack.action();

    }

    public void doEquip() {

        equipWeapon.action();

    }

}

public class Masician extends Person{

    public Masician() {

        super("마법사", new EquipWeaponStaff(), new AttackMasical());

    }

}

public class Swordman extends Person{

    public Swordman() {

        super("검사", new EquipWeaponSword(), new AttackPhsical());

    }

}

(4). discussion

스트레티지 패턴은 상속이라는 연결을 사용해서 알고리즘을 용이하게 교환할 수 있다는 장점이 존재한다. 그리고 Runtime에서도 변경이 가능하다는 장점이 있다 하지만 단점으로는 상황에 맞는 적절한 로직을 모두 알고 있어야 하며, 관리해야하는 클래스의 수가 늘어난다는 단점이 존재한다.

public class Person {

    private Attack attack;

    private EquipWeapon equipWeapon;

    private String name;

    public Person(String job, EquipWeapon equipWeapon, Attack attack) {

        this.name = job;

        this.equipWeapon = equipWeapon;

        this.attack = attack;

        System.out.println("직업 : " + job);

    }

    public Attack getAttack() {

        return attack;

    }

    public void setAttack(Attack attack) {

        this.attack = attack;

    }

    public EquipWeapon getEquipWeapon() {

        return equipWeapon;

    }

    public void setEquipWeapon(EquipWeapon equipWeapon) {

        this.equipWeapon = equipWeapon;

    }

    public void doAttack() {

        attack.action();

    }

    public void doEquip() {

        equipWeapon.action();

    }

}

public class Masician extends Person{

    public Masician() {

        super("마법사", new EquipWeaponStaff(), new AttackMasical());

    }

}

public class Swordman extends Person{

    public Swordman() {

        super("검사", new EquipWeaponSword(), new AttackPhsical());

    }

}

https://blog.naver.com/wowzzin/221397812115

2. decorator pattern

(1) 개념

데코레이터 패턴은 주어진 상황에 따라 어떤 객체에 책임을 덧붙이는 패턴으로, 기능 확장이 필요할 때 서브클래스 대신 쓸 수 있는 유연한 대안이 될 수 있다. 즉, 데코레이터 패턴은 기존의 코드를 건ㄴ드리지 않고 확장하도록 한다.

(2) 문제설명

1) 음료 예제

특정 음료부터 시작해서 첨가물로 그 음료를 장식한다 예를 들어 어떤 손님이 모카하고 휘핑크림을 추가한 다크 로스트 커피를 주문한다고 가정한다

2) 클래스 다이어그램

Component

methodA()

methodB()

concreteComponent

methodA()

methodB()

decorator

methodA()

methodB()

concretedecoratorB

methodA()

method()

concreteDecoratorA

methodA

method

newbehavior

3) 코드 구현

Public abstract class Beverage{

String description = “제목 없음”;

Public string getdescription(){

Return description ;

}

Public abstract double cost();

}

Public abstract class conditimentdecorator extends beverage{

Public abstract string getdescription();

}

Public class espresso extends beverage{

Public espresso(){

Description = “에스프레소”;

}

Public double coat(){

Return 1.99;

}

}

Public class Houseblend extends Beverage{

Public Houseblend(){

Description = “하우스 블랜드 커피”

}

Public double cost(){

Return = .89;

}}

Public class Mocha extends condimentDecorator {

Beverage beverage;

Public Mocha(Beverage beverage){

This.beverage = beverage;

}

Public string getdescription(){

Return beverage.getdescription() + “, 모카”;

}

Public double cost(){

Return . 20 +beverage.cost();

}

3) 음료예제에 우유 두유 모카 휘핑크림에 대한 추가 예제

Public class Beverage {

Milkcost, soycost, mochacost

Public float cost(){

{

Float condimentcost = 0.0;

If(hasmilk(){

Condimentcost= milkcost;

}

If(hassoy(){

Condimentcost += soycost;

}

If(hasmocha()){

Condimentcost += mochaCost;

}

If(haswhip(){

Condimentcost += whip cost;

}

Return condiment cost;

}}

Public class darkroast extends beverage{

Public darkRoast(){

Description = “최고의 다크 로스트”

}

Publc floast cost(){

Return 1.99 + super cost();

}

}

4) 쿠키에 얹을 토핑 만들기

Public class decoratormain{

Public static void main (string[] args){

Cookie bravecookie = new bravecookie();

Cookie milkBravecookie = new milktopping(bravecookie);

Cookie chocolatemilkBravecookie = new chocolatetopping(milkBravecookie);

Cookie chocolateDoublemilkBravecookie = newmilktopping

System.out.println(chocolatedoublemilkBravecookie.getName());

Cookie sodacookie = new cookie(){

@override

Public string getName(){

Return “소다 쿠키”;

}

};

Sodacookie = new chocolateTopping(new chocolateTopping(sodaCookie));

System.out.println(sodaCookie.getName());

}

}

(3) discussion

데코레이터 패턴의 경우 객체에 기능을 추가하고자 할 때 정적인 상속관계를 이용하는것보다 훨씬 유연하고 동일한 기능을 반복하는 것이 간편하나 단점으로 클래스 수는 줄어드는 반면에 객체에 수는 많아진다는 단점이 존재한다

1. 팩토리 패턴  
(1) 개념

팩토리 패턴은 객체지향 디자인 패턴이다 . 팩토리 패턴은 부모 클래스에 알려지지 않은 구체클래스를 생성하는 패턴이며 자식 클래스가 어떤 객체를 생성할지를 결정하도록 하는 패턴이기도 하다. 부모 클래스 코드에 구체 클래스 이름을 감추기 위한 방법으로도 사용된다

(2) 클래스 다이어 그램

Pizzastore

Orderpizza()

Simplepizzafactory orderpizza()

Bake()

Cull()

Pizza

Prepare()

Bake()

Cull()

pepperronipizza

cheespizza

clarrpizza

vegglepizza

(3)팩토리 예제

1) 피자 팩토리 예제

피자 관련 부모 클래스를 만들고 이하 자식 클래스로 여러 피자 종류를 선언한다 피자 스토어와 심플피자 팩토리를 이용하여 피자 클래스를 직접 참조하는 부분을 둔다

2) 소스코드

Public class NYpizzaStore extends pizzastore{

Pizza createpizza(string item) {

If ((item.equals (“cheese”)){

Return new nystylecheesepizza ();}

Else if (item.equals(“veggle”)){

Return new nystyleveggiepizza();

{else if (item.equals(“clam”)){

Return new nystyleclampizza();

}else if (item.equals(“pepproni”)){

Return new nystylepepprionipizza();

}

Else return null;

}

}

Public abstract class pizzastore{

Public pizza orderpizza(string type)

{

Pizza pizza;

Pizza = createpizza(type);

Pizza.prepare();

Pizza.bake();

Pizza.cut();

Pizza.box();

Return pizza;

}

Protected abstract pizza createpizza(string type);

}

Public abstract class pizza{

Public static pizza pizzafactory(pizzatype pizzatype) throws throwable{

Switch(pizzatype){

caseHammushroom;

return new hamandmushroompizza();

case deluxe;

return new deluxepizza();

case seafood;

returm new seafoodpizza();

} recognition”

Throw new exception(The pizza type “ + pizzatype.tostring()+”is not recognition”)

}

3) 두번째 예제

public abstract class Computer {

public abstract String getRAM();

public abstract String getHDD();

public abstract String getCPU();

@Override

public String toString(){

return "RAM= "+this.getRAM()+", HDD="+this.getHDD()+", CPU="+this.getCPU();

}

}

public class PC extends Computer {

private String ram;

private String hdd;

private String cpu;

public PC(String ram, String hdd, String cpu){

this.ram=ram;

this.hdd=hdd;

this.cpu=cpu;

}

@Override

public String getRAM() {

return this.ram;

}

@Override

public String getHDD() {

return this.hdd;

}

@Override

public String getCPU() {

return this.cpu;

}

}

public class Server extends Computer {

private String ram;

private String hdd;

private String cpu;

public Server(String ram, String hdd, String cpu){

this.ram=ram;

this.hdd=hdd;

this.cpu=cpu;

}

@Override

public String getRAM() {

return this.ram;

}

@Override

public String getHDD() {

return this.hdd;

}

@Override

public String getCPU() {

return this.cpu;

}

}

|  |  |
| --- | --- |
|  | public class ComputerFactory { |
|  |  |
|  | public static Computer getComputer(String type, String ram, String hdd, String cpu){ |
|  | if("PC".equalsIgnoreCase(type)) |
|  | return new PC(ram, hdd, cpu); |
|  | else if("Server".equalsIgnoreCase(type)) |
|  | return new Server(ram, hdd, cpu); |
|  |  |
|  | return null; |
|  | } |
|  | } |

4) 로봇 클래스 예제

package pattern.factory;

public abstract class Robot {

public abstract String getName();

}

package pattern.factory;

public class SuperRobot extends Robot {

@Override

public String getName() {

return "SuperRobot";

}

}

package pattern.factory;

(3) discussion

public class PowerRobot extends Robot {

@Override

public String getName() {

return "PowerRobot";

}

}

package pattern.factory;

public abstract class RobotFactory {

abstract Robot createRobot(String name);

}

package pattern.factory;

public class SuperRobotFactory extends RobotFactory {

@Override

Robot createRobot(String name) {

switch( name ){

case "super": return new SuperRobot();

case "power": return new PowerRobot();

}

return null;

}

}

package pattern.factory;

public class ModifiedSuperRobotFactory extends RobotFactory {

@Override

Robot createRobot(String name) {

try {

Class<?> cls = Class.forName(name);

Object obj = cls.newInstance();

return (Robot)obj;

} catch (Exception e) {

return null;

}

}

}

(3) discussion

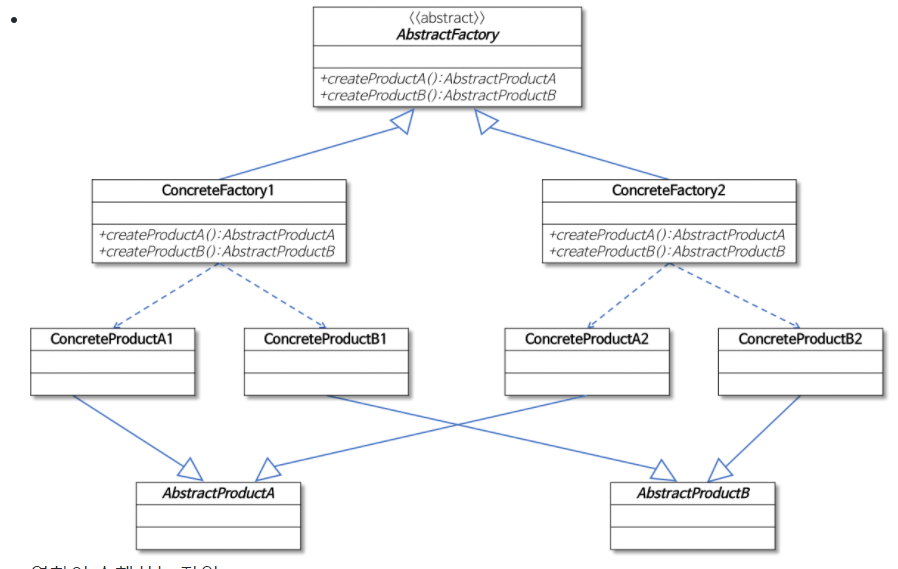
팩토리 패턴의 경우 리턴하는 객체의 클래스가 입력 매개변수에 따라 매번 다를 수 있다는 장점이 존재하나 단점으로는 public 또는 protected 생성자 없이 public static 메소드만 제공하는 클래스는 상속할 수 없다는 단점이 존재한다

1. 추상 팩토리 패턴

(1) 개념

추상 팩토리 패턴이란 구체적인 클래스에 의존하지 않고 서로 연관되거나 의존적인 객체들의 조합을 만드는 인터페이스를 제공하는 패턴을 의미한다.

(2) 다이어그램 구조



(3) 코드 예제

1) 앨라배이터 부품 업체 변경하기

여러 제조 업체의 부품을 사용하더라도 같은 동작을 지원할 수 있게한다 즉엘리베이터 프로그램의 변경을 최소화 하기 위해 추상 클래스로 motor door을 정의하고 motor클래스의 핵심 기능인 이동은 move() 메서드로 정의한다

1-2) 코드 예제

public void move(Direction direction) {

// 1) 이미 이동 중이면 무시한다.

// 2) 만약 문이 열려 있으면 문을 닫는다.

// 3) 모터를 구동해서 이동시킨다. -> 제조 업체에 따라 다르다.

// 4) 모터의 상태를 이동 중으로 설정한다.

}

public void open() {

// 1) 이미 문이 열려 있으면 무시한다.

// 2) 문을 닫는다. -> 제조 업체에 따라 다르다.

// 3) 문의 상태를 '닫힘'으로 설정한다.

}

public abstract class Door {

private DoorStatus doorStatus;

public Door() { doorStatus = DoorStatus.CLOSED; }

public DoorStatus getDoorStatus() { return doorStatus; }

// primitive 또는 hook 메서드

protected abstract void doClose();

// 템플릿 메서드: 문을 닫는 기능 수행

public void close() {

// 이미 문이 닫혀 있으면 아무 동작을 하지 않음

if(doorStatus == DoorStatus.CLOSED)

return;

// primitive 또는 hook 메서드. 하위 클래스에서 오버라이드

doClose(); // 실제로 문을 닫는 동작을 수행

doorStatus = DoorStatus.CLOSED; // 문의 상태를 닫힘으로 기록

}

// primitive 또는 hook 메서드

protected abstract void doOpen();

// 템플릿 메서드: 문을 여는 기능 수행

public void open() {

// 이미 문이 열려 있으면 아무 동작을 하지 않음

if(doorStatus == DoorStatus.OPENED)

return;

// primitive 또는 hook 메서드. 하위 클래스에서 오버라이드

doOpen(); // 실제로 문을 여는 동작을 수행

doorStatus = DoorStatus.OPENED; // 문의 상태를 열림으로 기록

}

}

public class LGDoor extends Door {

protected void doClose() { System.out.println("close LG Door"); }

protected void doOpen() { System.out.println("open LG Door"); }

}

2) 두번째예제

public class FactoryCreator {

static ViewFactory createViewFactory(String OS) {

if (OS.equals("window")) {

return new WindowViewFactory();

} else if (OS.equals("linux")){

return new LinuxViewFactory();

} else {

return null;

}

}

}

public class WindowViewFactory implements ViewFactory {

@Override

public ButtonView createButtonView() {

return new WindowButtonView();

}

@Override

public EditTextView createEditTextView() {

return new WindowEditTextView();

}

}

public class LinuxViewFactory implements ViewFactory {

@Override

public ButtonView createButtonView() {

return new LinuxButtonView();

}

@Override

public EditTextView createEditTextView() {

return new LinuxEditTextView();

}

}

public class Main {

public static void main(String[] args) {

ViewFactory viewFactory = FactoryCreator.createViewFactory("window");

ButtonView buttonView = viewFactory.createButtonView();

EditTextView editTextView = viewFactory.createEditTextView();

buttonView.doButtonSomething();

editTextView.doEditTextSomething();

viewFactory = FactoryCreator.createViewFactory("linux");

buttonView = viewFactory.createButtonView();

editTextView = viewFactory.createEditTextView();

buttonView.doButtonSomething();

editTextView.doEditTextSomething();

}

}

3) 세번째 예제

Suit suit = null;

switch(type){

case("space"):

suit = new SpaceSuit();

break;

case("hydro"):

suit = new HydroSuit();

break;

case("stealth"):

suit = new StealthSuit();

break;

default:

suit = new CombatSuit();

}

//슈트 추상 클래스

public abstract class Suit {

public abstract String getName();

}

//스텔스 슈트 클래스

public class StealthSuit extends Suit {

@Override

public String getName() {

return "StealthSuit";

}

}

//슈트 팩토리 추상클래스

public abstract class SuitFactory {

public abstract Suit createSuit(String type);

}

//슈트 팩토리 구현클래스

public class TypeSuitFactory extends SuitFactory{

@Override

public Suit createSuit(String type) {

Suit suit = null;

switch(type){

case("space"):

suit = new SpaceSuit();

break;

case("hydro"):

suit = new HydroSuit();

break;

case("stealth"):

suit = new StealthSuit();

break;

default:

suit = new CombatSuit();

}

return suit;

}

}

3) discussion

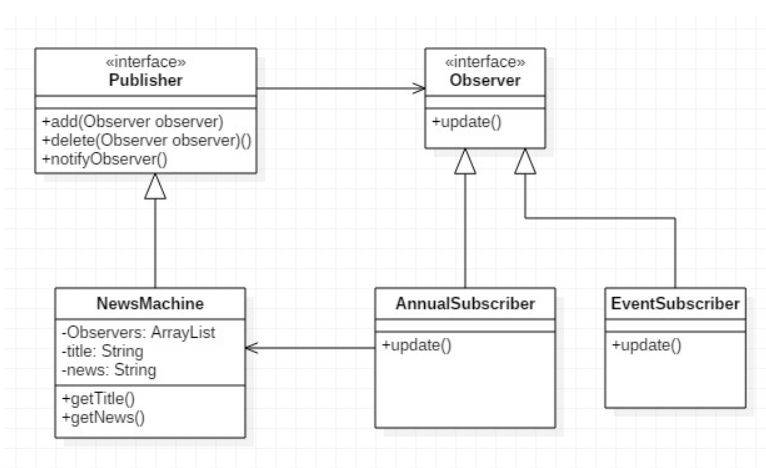
장점으로는 구체적인 클래스를 사용자로부터 분리할 수 있고 제품군을 쉽게 대체할 수 있다는 장점이 존재하지만 새로운 종류의 제품을 추가하기 어렵다는 단점이 존재한다

**1.observer pattern**

(1) 개념

옵저버 패턴이란 어떤 클래스의 변화가 일어났을 때, 이를 감지하여 다른 클래스에 통보해주는 패턴을 의미한다

(2) 다이어 그램



(3) 예시

1) 뉴스 구독

매일매일 새로 구독하는 뉴스를 제공하는 머신이 있다고 할 때 이 머신은 새로운 뉴스를 취합하는 정보를 가지고 있고 이 뉴스머신에서 나온 뉴스를 구독하고 싶어하는 observer가 있다고 가정할 때 뉴스머신이 새로운 뉴스를 생성할 때 마다 소식을 들을 수 있는 형태

2) 코드

public interface Observer { public void update(String title, String news); }

]

public interface Publisher { public void add(Observer observer); public void delete(Observer observer); public void notifyObserver(); }

public class NewsMachine implements Publisher { private ArrayList<Observer> observers; private String title; private String news; public NewsMachine() { observers = new ArrayList<>(); } @Override public void add(Observer observer) { observers.add(observer); } @Override public void delete(Observer observer) { int index = observers.indexOf(observer); observers.remove(index); } @Override public void notifyObserver() { for(Observer observer : observers) { observer.update(title, news); } } public void setNewsInfo(String title, String news) { this.title = title; this.news = news; notifyObserver(); } public String getTitle() { return title; } public String getNews() { return news; } }

public class AnnualSubscriber implements Observer { private String newsString; private Publisher publisher; public AnnualSubscriber(Publisher publisher) { this.publisher = publisher; publisher.add(this); } @Override public void update(String title, String news) { this.newsString = title + " \n -------- \n " + news; display(); } private void display() { System.out.println("\n\n오늘의 뉴스\n============================\n\n" + newsString); } }  
  
public class EventSubscriber implements Observer { private String newsString; private Publisher publisher; public EventSubscriber(Publisher publisher) { this.publisher = publisher; publisher.add(this); } @Override public void update(String title, String news) { newsString = title + "\n------------------------------------\n" + news; display(); } public void display() { System.out.println("\n\n=== 이벤트 유저 ==="); System.out.println("\n\n" + newsString); } }

public class MainClass { public static void main(String[] args) { NewsMachine newsMachine = new NewsMachine(); AnnualSubscriber as = new AnnualSubscriber(newsMachine); EventSubscriber es = new EventSubscriber(newsMachine); newsMachine.setNewsInfo("오늘 한파", "전국 영하 18도 입니다."); newsMachine.setNewsInfo("벛꽃 축제합니다", "다같이 벚꽃보러~"); } }

3)여러 방식으로 성적 출력하기

입력된 성적갑을 출력하고 다시 받는 형식의 코드를 구현

/\* 입력된 점수를 저장하는 클래스 (1. 출력형태: 목록) \*/

public class ScoreRecord {

private List<Integer> scores = new ArrayList<Integer>(); // 점수를 저장함

private DataSheetView dataSheetView; // 목록 형태로 점수를 출력하는 클래스

public void setDataSheetView(DataSheetView dataSheetView) { this.dataSheetView = dataSheetView; }

// 새로운 점수를 추가하면 출력하는 것에 변화를 통보(update())하여 출력하는 부분 갱신

public void addScore(int score) {

scores.add(score); // scores 목록에 주어진 점수를 추가함

dataSheetView.update(); // scores가 변경됨을 통보함

}

// 출력하는 부분에서 변화된 내용을 얻어감

public List<Integer> getScoreRecord() { return scores; }

}

/\* 1. 출력형태: 목록 형태로 출력하는 클래스 \*/

public class DataSheetView {

private ScoreRecord scoreRecord;

private int viewCount;

public DataSheetView(ScoreRecord scoreRecord, int viewCount) {

this.scoreRecord = scoreRecord;

this.viewCount = viewCount;

}

// 점수의 변경을 통보받음

public void update() {

List<Integer> record = scoreRecord.getScoreRecord(); // 점수를 조회함

displayScores(record, viewCount); // 조회된 점수를 viewCount 만큼만 출력함

}

private void displayScores(List<Integer> record, int viewCount) {

System.out.println("List of " + viewCount + " entries: ");

for (int i = 0; i < viewCount && i < record.size(); i++) {

System.out.println(record.get(i) + " ");

}

System.out.println();

}

}

public class Client {

public static void main(String[] args) {

ScoreRecord scoreRecord = new ScoreRecord();

// 3개까지의 점수만 출력함

DataSheetView dataSheetView = new DataSheetView(scoreRecord, 3);

scoreRecord.setDataSheetView(dataSheetView);

for (int index = 1; index <= 5; index++) {

int score = index \* 10;

System.out.println("Adding " + score);

// 10 20 30 40 50을 추가함, 추가할 때마다 최대 3개의 점수만 출력함

scoreRecord.addScore(score);

}

}

}

4) 세번째예제

public abstract class Generator {

    //등록된 옵저버들을 관리하기 위한 ArrayList

    private ArrayList<Observer> observers = new ArrayList<Observer>();

    public void AddObserver(Observer observer)    //옵저버 등록

    {

        observers.add(observer);

    }

    public void NotifyObservers()                //상태 변화에 따른 옵저버들에게 통지해주는 메서드

    {

        for(int i=0;i<observers.size();i++)

        {

            observers.get(i).update(this);

        }

    }

    public abstract String getString();

    public abstract void execute();

}

public class StringGenerator extends Generator{

    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

    private String Message;        //상태 정보를 가지는 String형 변수

    public String getString()    //현재 관찰 대상자의 상태 정보를 반환

    {

        return this.Message;

    }

    public void execute()

    {

        while(true)

        {

            try {

                Message = in.readLine();    //키보드 입력이 있었다는 건 상태 정보가 변한 것을 의미

                NotifyObservers();            //등록된 옵저버들에게 통지

            }catch(Exception e) {}

        }

    }

}

|  |  |
| --- | --- |
|  | public interface Observer {      public void update(Generator generator);  //관찰 대상자와 상태 정보를 동기화  } |

public class StringObserver implements Observer {

    String Message;

    public void update(Generator generator)

    {

        Message = generator.getString();        //관찰 대상자의 상태정보를 가져옴

        System.out.println("StringObserver : " + Message); //화면에 출력

    }

}

public class StringCountObserver implements Observer {

    int count;

    public void update(Generator generator)

    {

        count = generator.getString().length();        //관찰 대상자의 상태 정보를 가져와 문자열 길이를 저장

        System.out.println("StringCountObserver : " + count);

    }

}

(3) discussion

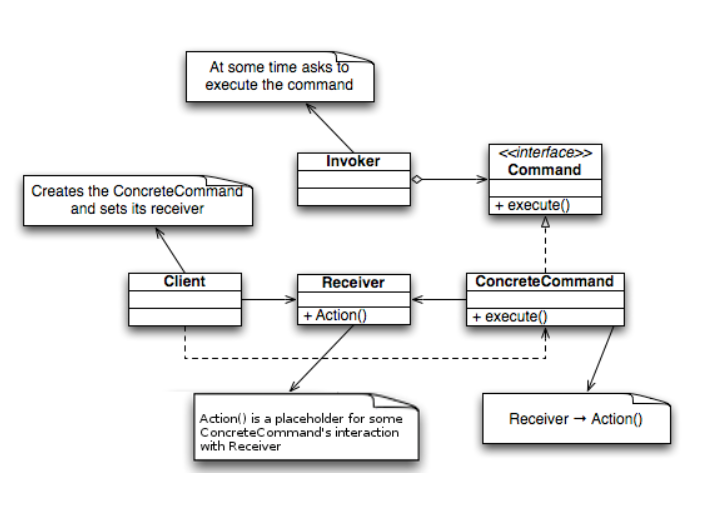
옵저버 패턴은 클래스이기에 이를 상속받는 서브클래스를 만들어야 하고 이미 다른수퍼클래스를 확장하고 있는 서브클래스라면 observable 기능을 추가할 수 없어 재사용성의 제약이 있다

1. command 패턴

(1) 개념

커맨드 패턴이랑 요청을 객체의 형태로 캡슐화하여 사용자가 보낸 요청을 나중에 이용할 수 있도록 매서드 이름 매개변수 등 요청에 필요한 정보를 저장또는 로깅 취소할 수 있게 하는 패턴이다

(2) 다이어그램



(3) 커맨드 패턴 예시

1)첫 번째 예시 : 버튼이 눌리면 램프에 불이 켜지는 형식

ublic class Lamp {

public void turnOn(){ System.out.println("Lamp On"); }

}

public class Button {

private Lamp theLamp;

public Button(Lamp theLamp) { this.theLamp = theLamp; }

public void pressed() { theLamp.turnOn(); }

}

public class Client {

public static void main(String[] args) {

Lamp lamp = new Lamp();

Button lampButton = new Button(lamp);

lampButton.pressed();

}

}

(2) 천장팬 작동 예시

// 모든 커맨드 객체에서 구현해야 하는 인터페이스(Interface)

// 모든 명령은 execute() 메소드 호출을 통해 수행되며, 이 메소드에서는 Receiver에 특정 작업을 처리하라는 지시를 전달한다.

public interface Command {

 public void execute();

 public void undo();

}

// ConcreteCommand

// 이 클래스는 천장팬 속도를 높이는 행동과 Receiver인 CeilingFan 사이를 bind한다.

public class CeilingFanHighCommand implements Command {

 CeilingFan ceilingFan;

 int prevSpeed;

 public CeilingFanHighCommand(CeilingFan ceilingFan) {

  this.ceilingFan = ceilingFan;

 }

 // Invoker에서 이 메소드를 호출하면 천장팬 속도를 높인다.

 // 이 때, undo 기능을 구현하기 위해 이전에 있던 속도 값을 저장해둔다.

 @Override

 public void execute() {

  prevSpeed = ceilingFan.getSpeed();

  ceilingFan.high();

 }

 // 이전 속도를 바탕으로 속도를 재조정한다.

 @Override

 public void undo() {

  if (prevSpeed == CeilingFan.HIGH) {

   ceilingFan.high();

  } else if (prevSpeed == CeilingFan.MEDIUM) {

   ceilingFan.medium();

  } else if (prevSpeed == CeilingFan.LOW) {

   ceilingFan.low();

  } else if (prevSpeed == ceilingFan.OFF) {

   ceilingFan.off();

  }

 }

}

public class CeilingFanOffCommand implements Command {

 CeilingFan ceilingFan;

 int prevSpeed;

 public CeilingFanOffCommand(CeilingFan ceilingFan) {

  this.ceilingFan = ceilingFan;

 }

 @Override

 public void execute() {

  prevSpeed = ceilingFan.getSpeed();

  ceilingFan.off();

 }

 @Override

 public void undo() {

  if (prevSpeed == CeilingFan.HIGH) {

   ceilingFan.high();

  } else if (prevSpeed == CeilingFan.MEDIUM) {

   ceilingFan.medium();

  } else if (prevSpeed == CeilingFan.LOW) {

   ceilingFan.low();

  } else if (prevSpeed == CeilingFan.OFF) {

   ceilingFan.off();

  }

 }

}

// NoCommand 객체는 일종의 Null Object(널 객체)이다.

// 리턴할 객체는 없지만 클라이언트 쪽에서 null을 처리하지 않아도 되게 하고 싶을 때 활용하면 좋다.

public class NoCommand implements Command {

 @Override

 public void execute() {

 }

 @Override

 public void undo() {

 }

}

// Receiver

// 천장팬을 작동시키는 클래스

public class CeilingFan {

 public static final int HIGH = 3;

 public static final int MEDIUM = 2;

 public static final int LOW = 1;

 public static final int OFF = 0;

 String location;

 int speed;

 public CeilingFan(String location) {

  this.location = location;

  speed = OFF;

 }

 public void high() {

  speed = HIGH;

 }

 public void medium() {

  speed = MEDIUM;

 }

 public void low() {

  speed = LOW;

 }

 public void off() {

  speed = OFF;

 }

 public int getSpeed() {

  return speed;

 }

}

// Invoker

public class RemoteControl {

 Command[] onCommands;

 Command[] offCommands;

 Command undoCommand;

 public RemoteControl() {

  onCommands = new Command[7];

  offCommands = new Command[7];

  Command noCommand = new NoCommand();

  for (int i = 0; i < 7; i++) {

   onCommands[i] = noCommand;

   offCommands[i] = noCommand;

  }

  undoCommand = noCommand;

 }

 // setCommand() 메소드로 리모컨의 각 슬롯에 특정 명령을 지정해놓는다.

 public void setCommand(int slot, Command onCommand, Command offCommand) {

  onCommands[slot] = onCommand;

  offCommands[slot] = offCommand;

 }

 // 특정 슬롯의 ON 버튼이 눌러지면, 해당 슬롯의 ONCommand의 execute() 메소드를 호출한다.

 public void onButtonWasPushed(int slot) {

  onCommands[slot].execute();

  undoCommand = onCommands[slot];

 }

 public void offbuttonWasPushed(int slot) {

  offCommands[slot].execute();

  undoCommand = offCommands[slot];

 }

 public String toString() {

  StringBuffer stringBuff = new StringBuffer();

  stringBuff.append("\n-------Remote Control--------\n");

  for (int i = 0; i < onCommands.length; i++) {

   stringBuff.append("[slot " + i + "] " + onCommands[i].getClass().getName() + "   "+offCommands[i].getClass().getName() + "\n");

  }

  return stringBuff.toString();

 }

 public void undoButtonWasPushed() {

  undoCommand.undo();

 }

}

// Client

public class RemoteLoader {

 /\*\*

  \* @param args

  \*/

 public static void main(String[] args) {

  RemoteControl remoteControl = new RemoteControl();

  CeilingFan ceilingFan = new CeilingFan("Living Room");

  CeilingFanHighCommand ceilingFanHigh = new CeilingFanHighCommand(ceilingFan);

  CeilingFanOffCommand ceilingFanOff = new CeilingFanOffCommand(ceilingFan);

  remoteControl.setCommand(2, ceilingFanHigh, ceilingFanOff);

  System.out.println(remoteControl);

  remoteControl.onButtonWasPushed(0);

  remoteControl.offbuttonWasPushed(0);

  remoteControl.undoButtonWasPushed();

  remoteControl.onButtonWasPushed(2);

  remoteControl.offbuttonWasPushed(2);

  remoteControl.undoButtonWasPushed();

 }

}

(3) 3번째 예제코드  
  
public interface Command { public abstract void excute(); }  
public class Sword\_attack{ public void attack() { System.out.println("Sword Attack"); } }  
public class Mountain\_retreat { public void retreat() { System.out.println("Retreat to Mountain"); } }  
  
public class Attack\_command implements Command { private Sword\_attack mAk; Attack\_command(Sword\_attack ak){ mAk = ak; } @Override public void excute() { mAk.attack(); } }  
  
public class Retreat\_command implements Command { private Mountain\_retreat mRt; Retreat\_command(Mountain\_retreat rt){ mRt = rt; } @Override public void excute() { mRt.retreat(); } }  
public class Strategy { private Command mA\_Command; private Command mR\_Command; Strategy(Command a\_command,Command r\_command){ mA\_Command = a\_command; mR\_Command = r\_command; } public void setCommnad(Command a\_command,Command r\_command){ mA\_Command = a\_command; mR\_Command = r\_command; } public void attack\_Command(){ mA\_Command.excute(); } public void retreat\_Commnad(){ mR\_Command.excute(); } }  
  
public class General { public static void main(String[] args){ Sword\_attack s\_ak = new Sword\_attack(); Attack\_command a\_c = new Attack\_command(s\_ak); Mountain\_retreat m\_rt = new Mountain\_retreat(); Retreat\_command r\_c = new Retreat\_command(m\_rt); Strategy g\_command = new Strategy(a\_c,r\_c); g\_command.attack\_Command(); g\_command.retreat\_Commnad(); } }  
  
(4) discussion

커맨드 패턴의 경우 요청부와 동작부를 분리시켜주기에 시스템의 결함도를 낮추고 각 객체들이 수정되어도 다른 객체가 영향을 받지 않짐만 리시버의 동작이 추가된다면 그 동작에 대한 클래스를 만들어야 하기에 다소 많은 클래스가 추가될 수 있다는 단점이 존재한다

1. 어댑터 패턴

(1) 개념

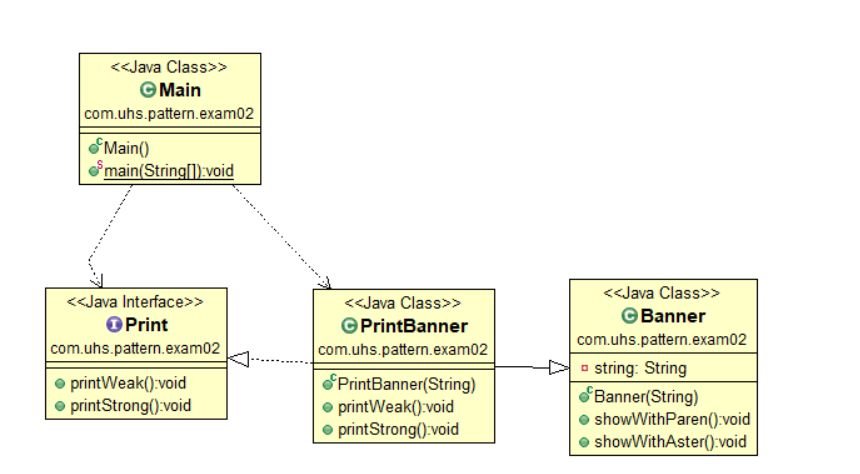
: 제공되고 있는 것과 필요한 것 사이를 연결해주는 것,

무엇인가를 한번 포장해서 다른 용도로 사용할 수 있게 교환해주는 것

한 클래스의 인터페이스를 클라이언트에서 사용하고자 하는 다른 인터페이스로 변환한다.

이 패턴을 사용하면 다른 인터페이스와의 호환성 문제를 해결할 수 있다

(2) 다이어그램



(3) 예시

package AdapterPattern;

public interface TypeA {

public void type();

public void shape();

}

package AdapterPattern;

public interface TypeC {

public void type();

public void shape();

}

**[출처]** [어댑터 패턴(Adapter Pattern)](https://blog.naver.com/adamdoha/222092596000)|**작성자** [DolphaGo](https://blog.naver.com/adamdoha)

package AdapterPattern;

public class USB1 implements TypeA {

@Override

public void type() {

System.out.println("A Type.......");

}

@Override

public void shape() {

System.out.println("Large Rectangle......");

}

}

package AdapterPattern;

public class USBAdapter implements TypeA {

package AdapterPattern;

public class AdapterTest {

public static void main(String[] args) {

USB1 usb1=new USB1();

USB2 usb2=new USB2();

System.out.println("----------USB1--------");

checkUSB(usb1);

System.out.println("----------USB2--------");

usb2.type();

usb2.shape();

System.out.println("-----Adapter 적용-----");

TypeA usbAdapter = new USBAdapter(usb2);

//어댑터 적용 후 USB의 타입

checkUSB(usbAdapter);

}

public static void checkUSB(TypeA usb){

usb.type();

usb.shape();

}

}

TypeC typeC;

public USBAdapter(TypeC typeC){

this.typeC=typeC;

}

@Override

public void type() {

typeC.type();

}

@Override

public void shape() {

typeC.shape();

}

}

**[출처]** [어댑터 패턴(Adapter Pattern)](https://blog.naver.com/adamdoha/222092596000)|**작성자** [DolphaGo](https://blog.naver.com/adamdoha)