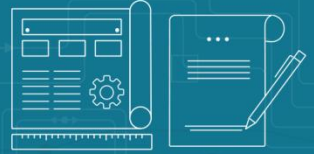


화면 구현

part 1

소프트웨어 구조 패턴과 행동 패턴



```
$('.menuBtn > .go').on('click', function(){
    var menuId = $(this).attr('id');
    switch (menuId) {
        case "goAbout":
            $('.slideWrap').slick('slickGoTo', 1);
            break;
        case "goContact":
            $('.slideWrap').slick('slickGoTo', 2);
            break;
        case "goHistory":
            $('.slideWrap').slick('slickGoTo', 3);
            break;
        case "goMain":
            $('.slideWrap').slick('slickGoTo', 0);
            break;
    }
    break;
});
```



학습목표

- 소프트웨어 구조 패턴을 이해하고, 이를 적용할 수 있다.
- 소프트웨어 행동 패턴을 이해하고, 이를 적용할 수 있다.

학습내용

- 소프트웨어 구조 패턴
- 소프트웨어 행동 패턴

구조 패턴

1. 적응자(Adapter) 패턴

1) 정의 및 개념

인터페이스가 호환되지 않는 클래스들을
함께 이용할 수 있도록
타 클래스의 인터페이스에 덧씌움

클라이언트와 구현된
인터페이스 분리 가능

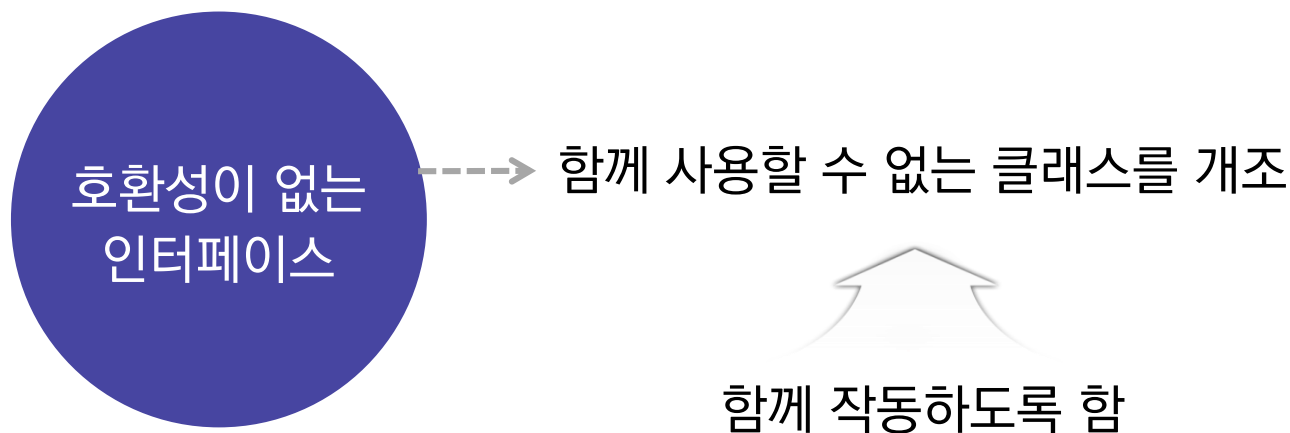
→ 인터페이스가 바뀌더라도 **변경 내역은 어댑터에 캡슐화 됨**

클라이언트는 바뀔 필요가 없음

구조 패턴

1. 적응자(Adapter) 패턴

1) 정의 및 개념



2) 고려사항

- 대체할 수 있는(재사용성) 어댑터 사용
- 개조되는 두 클래스의 인터페이스 모두를 상속받아 정의
- 클래스 방식, 객체 방식 두 가지 구현 방법을 상황에 맞게 적용

구조 패턴

1. 적응자(Adapter) 패턴

3) Adapter 패턴 선언

[정보 제공 클래스]

Person 클래스에 데이터 입력

```
public class Person {  
    private String name;  
    private String phone;  
  
    public Person(String name, String phone) {  
        this.name = name;  
        this.phone = phone;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public String getPhone() {  
        return phone;  
    }  
}
```

구조 패턴

1. 적응자(Adapter) 패턴

3) Adapter 패턴 선언

[메세지 보내는 기본 함수 클래스]

SendMessage() : Person 1명에게만 메세지 보내는 함수

```
public class SendMessage {  
    public void sendMessage(Person personObj){  
        System.out.println("To : "+personObj.getPhone()+"\n"+"Message : Hello "+personObj.getName()+"\n");  
    }  
}
```

[메세지 보내는 Push Adapter 인터페이스 선언]

```
public interface PushAdapter {  
    public void PushMessage(Person personObj);  
    public void PushMessages(ArrayList<Person> personObjList);  
}
```

구조 패턴

1. 적응자(Adapter) 패턴

3) Adapter 패턴 선언

[Adapter 클래스1]

클래스에 의한 Adapter 패턴

- 상속을 활용하기 때문에 유연하지 못함
- Adapter 전체를 다시 구현할 필요가 없음

```
public class PushAdapterImpl extends SendMessage implements PushAdapter {  
  
    @Override  
    public void PushMessage(Person personObj) {  
        // TODO Auto-generated method stub  
        this.sendMessage(personObj);  
    }  
  
    @Override  
    public void PushMessages(ArrayList<Person> personObjList) {  
        // TODO Auto-generated method stub  
        for(Person personObj : personObjList){  
            this.sendMessage(personObj);  
        }  
    }  
}
```

구조 패턴

1. 적응자(Adapter) 패턴

3) Adapter 패턴 선언

[Adapter 클래스2]

인스턴스를 선언하여 Adapter 클래스 선언

- 어댑터 클래스의 코드 대부분을 구현해야 함
- 구성(Composition)을 사용하기 때문에 유연함

```
public class PushAdapterImpl extends SendMessage implements PushAdapter {  
  
    @Override  
    public void PushMessage(Person personObj) {  
        // TODO Auto-generated method stub  
        this.sendMessage(personObj);  
    }  
  
    @Override  
    public void PushMessages(ArrayList<Person> personObjList) {  
        // TODO Auto-generated method stub  
        for(Person personObj : personObjList){  
            this.sendMessage(personObj);  
        }  
    }  
}
```


구조 패턴

1. 적응자(Adapter) 패턴

4) 사용법

```
public class Main {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        ArrayList<Person> personObjList = new ArrayList<Person>();
        personObjList.add(new Person("A","010-1234-0001"));
        personObjList.add(new Person("B","010-1234-0002"));
        personObjList.add(new Person("C","010-1234-0003"));

        PushAdapterImpl pushObj = new PushAdapterImpl();
        pushObj.PushMessage(personObjList.get(0));
        pushObj.PushMessages(personObjList);

        System.out.println("\n");

        PushAdapterImpl2 pushObj2 = new PushAdapterImpl2();
        pushObj2.PushMessage(personObjList.get(1));
        pushObj2.PushMessages(personObjList);

    }

}
```

[클래스에 의한
Adapter패턴 결과]

To : 010-1234-0001
Message : Hello A

To : 010-1234-0001
Message : Hello A

To : 010-1234-0002
Message : Hello B

To : 010-1234-0003
Message : Hello C

[인스턴스에 의한
Adapter패턴 결과]

To : 010-1234-0002
Message : Hello B

To : 010-1234-0001
Message : Hello A

To : 010-1234-0002
Message : Hello B

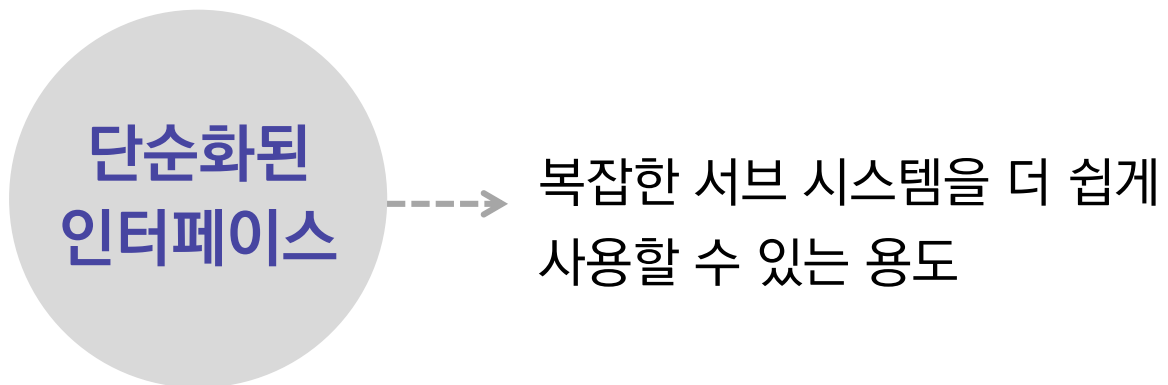
To : 010-1234-0003
Message : Hello C

구조 패턴

2. 퍼사드(Facade) 패턴

1) 정의 및 개념

서브 시스템에 있는 인터페이스 집합에
하나의 인터페이스를 제공



2) 효과

서브 시스템

- 구성요소 보호
- 서브 시스템과 클라이언트 코드 간의 결합도 ↓

구조 패턴

2. 퍼사드(Facade) 패턴

3) Facade 패턴 선언

[Login Process 클래스 선언]

- 복잡한 로그인 과정이 있음
- 디비 접속 → 유저 ID, PW 가져오기 → 입력된 값을 비교 → 로그인 결과 → 디비 접속 끊기
- 클라이언트에서는 전체 로그인 과정을 직접 수행하지 않음
- 복잡한 순서를 단순하게 사용할 수 있는 login 인터페이스를 생성
- 클라이언트 : login을 이용하여 작업 진행

구조 패턴

2. 퍼사드(Facade) 패턴

3) Facade 패턴 선언

```
public class LoginProcess {  
  
    public LoginProcess(){}  
  
    public void login(String id, String pw){  
        this.getDBConnection();  
        this.getUserIDfromDB();  
        this.getUserPWfromDB();  
        this.checkUser(id, pw);  
        this.closeDBConnection();  
    }  
  
    private void getDBConnection(){  
        System.out.println("Connect DB success");  
    }  
  
    private void getUserIDfromDB(){  
        System.out.println("User ID is XXX");  
    }  
  
    private void getUserPWfromDB(){  
        System.out.println("User PW is XXXXXX");  
    }  
  
    private void checkUser(String id , String pw){  
        System.out.println("Check id success");  
        System.out.println("Check pw success");  
        System.out.println("Success Login");  
    }  
  
    private void closeDBConnection(){  
        System.out.println("Disconnect DB Success");  
    }  
  
}
```

구조 패턴

2. 퍼사드(Facade) 패턴

4) 사용법

```
public class Main {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        LoginProcess loginObj = new LoginProcess();
        loginObj.login("id", "pw");
    }
}
```

Connect DB success
User ID is XXX
User PW is XXXXXX
Check id success
Check pw success
Success Login
Disconnect DB Success

클라이언트에서는 **login 인터페이스를 이용**하여
간단하게 로그인 작업을 수행

3. 프록시(Proxy) 패턴

1) 정의 및 개념

접근 조절

비용 절감

복잡도 감소

접근이 힘든 객체에 대한 **대역 제공**

↓
객체를 감싸서 그 객체에 대한
접근을 제어

구조 패턴

3. 프록시(Proxy) 패턴

1) 정의 및 개념

- 인터페이스를 사용하고 실행시킬 클래스에 대한 객체가 들어갈 자리에 대리자 객체를 대신 투입

클라이언트

실제 실행시킬 클래스에 대한
객체를 통해 메서드를
호출하고 반환 값을 받는지?

대리자 객체를 통해 메서드를
호출하고 반환 값을 받는지?

... 모르게 처리

구조 패턴

3. 프록시(Proxy) 패턴

2) 특징

- 프록시는 실제 서비스와 같은 이름의 메서드를 구현 (인터페이스 사용)
- 실제 서비스의 같은 이름을 가진 메서드를 호출하고 그 값을 클라이언트에게 돌려줌
- 메서드 호출 전후에도 별도의 로직을 수행할 수도 있음

3) 프록시 패턴 선언

[인터페이스 선언]

실제 서비스와 같은 이름의 메서드

```
public interface IServerModule {  
    public void execute();  
}
```

구조 패턴

3. 프록시(Proxy) 패턴

3) 프록시 패턴 선언

[실제 서비스 클래스 선언]

실제 서비스 클래스에 기능 구현

```
public class ServerModule implements IServerModule {

    @Override
    public void execute() {
        // TODO Auto-generated method stub
        System.out.println("Run Server Module");
    }

}
```

[프록시 클래스 선언]

실제 서비스와 같은 이름의 메서드를 구현한 클래스 선언

- 실제 서비스 실행 전후에 다른 작업을 할 수 있음

```
public class Proxy implements IServerModule {
    IServerModule server;

    @Override
    public void execute() {
        // TODO Auto-generated method stub
        System.out.println("ServerModule의 execute() 실행 전에 proxy에서 무언가를 할 수 있음");

        server = new ServerModule();
        server.execute();
    }
}
```


구조 패턴

3. 프록시(Proxy) 패턴

4) 사용법

```
public class Main {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("직접 호출");
        ServerModule smObj = new ServerModule();
        smObj.execute();

        System.out.println("\n" + "Proxy 를 통한 호출");
        IServerModule proxy = new Proxy();
        proxy.execute();
    }
}
```

직접 호출

Run Server Module

Proxy 를 통한 호출

ServerModule의 execute() 실행 전에 proxy에서 무언가를 할 수 있음

Run Server Module

[직접 호출하는 경우와 proxy를 통한 호출]

구조 패턴

4. 컴포지트(Composite) 패턴

1) 정의 및 개념

- 클라이언트에서 객체 컬렉션과 개별 객체를 똑같이 다룰 수 있도록 함
- 0개 1개 혹은 그 이상의 객체를 묶어 하나의 객체로 이용할 수 있음
- 객체들의 관계를 트리구조로 구성하여 부분-전체 계층을 표현
- 단일 객체와 복합 객체를 동일하게 다룰 수 있음
- 개별적인 객체들과 객체들의 집합 간의 처리방법의 차이가 없을 경우 사용

구조 패턴

4. 컴포지트(Composite) 패턴

2) 컴포지트 패턴 선언

[컴포넌트 인터페이스 선언]

객체와 복합 객체가 구현해야 할 메소드를 정의한 인터페이스

```
public interface ElectronicProductComponent {  
    public void powerOn();  
    public void powerOff();  
}
```

구조 패턴

4. 컴포지트(Composite) 패턴

2) 컴포지트 패턴 선언

[컴포넌트 클래스 선언]

모든 컴포넌트 메서드를 구현, 컴포넌트객체들을 자식으로 가짐

```
public class EletronicProductComposite implements ElectronicProductComponent {

    private ArrayList<ElectronicProductComponent> productList = new ArrayList<ElectronicProductComponent>();

    public EletronicProductComposite() {
    }

    public void addProductList(ElectronicProductComponent product){
        productList.add(product);
    }

    public void removeProductList(ElectronicProductComponent product){
        productList.remove(product);
    }

    @Override
    public void powerOn() {
        // TODO Auto-generated method stub
        System.out.println("Power on...");
        for(ElectronicProductComponent product : productList){
            product.powerOn();
        }
    }

    @Override
    public void powerOff() {
        // TODO Auto-generated method stub
        System.out.println("Power off...");
        for(ElectronicProductComponent product : productList){
            product.powerOff();
        }
    }
}
```

구조 패턴

4. 컴포지트(Composite) 패턴

2) 컴포지트 패턴 선언

[컴포넌트 클래스 객체]

컴포넌트의 모든 메서드를 구현

```
public class ProductTV implements ElectronicProductComponent {
    private String name;
    public ProductTV(String name) {
        this.name = name;
    }

    @Override
    public void powerOn() {
        // TODO Auto-generated method stub
        System.out.println("TV power on - "+this.name);
    }

    @Override
    public void powerOff() {
        // TODO Auto-generated method stub
        System.out.println("TV power off - "+this.name);
    }
}
```

```
public class ProductAudio implements ElectronicProductComponent {
    private String name;
    public ProductAudio(String name) {
        this.name = name;
    }

    @Override
    public void powerOn() {
        // TODO Auto-generated method stub
        System.out.println("Audio power on - "+this.name);
    }

    @Override
    public void powerOff() {
        // TODO Auto-generated method stub
        System.out.println("Audio power off - "+this.name);
    }
}
```

구조 패턴

4. 컴포지트(Composite) 패턴

3) 사용법

```
public class Main {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        ProductTV pdTV1 = new ProductTV("TV_1");
        ProductTV pdTV2 = new ProductTV("TV_2");
        ProductAudio pdAudio1 = new ProductAudio("Audio_1");
        ProductAudio pdAudio2 = new ProductAudio("Audio_2");

        EletronicProductComposite composite = new EletronicProductComposite();

        composite.addProductList(pdTV1);
        composite.addProductList(pdAudio1);
        composite.addProductList(pdTV2);
        composite.addProductList(pdAudio2);

        composite.powerOn();

        composite.powerOff();
    }
}
```

Power on...

- TV power on - TV_1
- Audio power on - Audio_1
- TV power on - TV_2
- Audio power on - Audio_2

Power off...

- TV power off - TV_1
- Audio power off - Audio_1
- TV power off - TV_2
- Audio power off - Audio_2

컴포넌트(Component) 클래스를 선언,
해당 컴포지트에 생성된 **컴포넌트 추가 및 삭제 관리**

행동 패턴

1. 옵저버(Observer) 패턴

1) 정의 및 개념



옵저버(Observer) 패턴이란?

상태 변화를 알려주도록 하는 패턴



- 1대 n의 의존성을 가짐
- 이벤트 핸들링 시스템 구현에 활용

행동 패턴

1. 옵저버(Observer) 패턴

2) Observer 패턴 선언

“자바 내장 API(`java.util.Observable`, `java.util.Observer`) 사용”

Observable 클래스 선언

- 등록된 옵저버들을 관리
- 새로운 데이터가 들어오면, 옵저버에게 데이터 전달
- `setChanged()` 함수는 전달할 새로운 데이터가 있음을 설정
- `notifyObservers()` 함수는 등록된 `Observer` 클래스에게 순차적으로 데이터가 변경되었음을 전달

행동 패턴

1. 옵저버(Observer) 패턴

2) Observer 패턴 선언

```
public class TemperatureData extends Observable {  
    private float temperature;  
  
    public TemperatureData() {  
        // TODO Auto-generated constructor stub  
    }  
  
    public void setMeasurements(float temperature){  
        this.temperature = temperature;  
        measurementsChanged();  
    }  
  
    public void measurementsChanged(){  
        setChanged();  
        notifyObservers();  
    }  
  
    public float getTemperature(){  
        return temperature;  
    }  
}
```

행동 패턴

1. 옵저버(Observer) 패턴

2) Observer 패턴 선언

“자바 내장 API(java.util.Observable, java.util.Observer) 사용”

Observer 인터페이스 선언

- Observer 클래스에서 사용할 함수를 선언
- Observer 클래스에서 온도 데이터를 전달받으면 온도 값에 의해 호출될 함수 선언(power On(), power Off())

```
public interface AirConditionerElement {  
    public void powerOn();  
    public void powerOff();  
}
```

행동 패턴

1. 옵저버(Observer) 패턴

2) Observer 패턴 선언

“자바 내장 API(`java.util.Observable`, `java.util.Observer`) 사용”

[Observer 클래스 선언]

AirConditionerMachine 1, 2 선언

- `AirConditionerElement`, `Observer`를 이용
- `Observable`의 `addObserver()` 함수를 이용하여 현재 선언한 `Observer` 등록
- `update()` 함수는 `Observable`에 값이 변경되면 `Observer`의 `update()`를 호출하여 변경된 값을 갱신
- 변경된 온도 데이터를 판단하여 `power On()`, `powerOff()` 함수를 호출

행동 패턴

1. 옵저버(Observer) 패턴

2) Observer 패턴 선언

```
public class AirConditionerMachine1 implements AirConditionerElement, Observer {

    Observable observable;

    private final float BaseTemperature = 26.5f;
    private float currentTemperature = 0.0f;

    public AirConditionerMachine1(Observable observable) {
        // TODO Auto-generated constructor stub
        this.observable = observable;
        this.observable.addObserver(this);
    }

    @Override
    public void update(Observable obs, Object arg) {
        // TODO Auto-generated method stub
        if(obs instanceof TemperatureData){
            TemperatureData tempData = (TemperatureData) obs;
            currentTemperature = tempData.getTemperature();
            if(BaseTemperature < currentTemperature){
                this.powerOn();
            }else{
                this.powerOff();
            }
        }
    }

    @Override
    public void powerOn() {
        // TODO Auto-generated method stub
        System.out.println("=====");
        System.out.println("에어콘 1번");
        System.out.println("설정 온도 : "+this.BaseTemperature+"도 || "+ "현재 온도 : "+this.currentTemperature+"도");
        System.out.println("에어콘 가동 시작 !!");
        System.out.println("=====");
    }

    @Override
    public void powerOff() {
        // TODO Auto-generated method stub
        System.out.println("=====");
        System.out.println("에어콘 1번");
        System.out.println("설정 온도 : "+this.BaseTemperature+"도 || "+ "현재 온도 : "+this.currentTemperature+"도");
        System.out.println("에어콘 가동 중지 !!");
        System.out.println("=====");
    }
}
```

행동 패턴

1. 옵저버(Observer) 패턴

2) Observer 패턴 선언

```
public class AirConditionerMachine2 implements AirConditionerElement, Observer {

    Observable observable;

    private final float BaseTemperature = 28.0f;
    private float currentTemperature = 0.0f;

    public AirConditionerMachine2(Observable observable) {
        // TODO Auto-generated constructor stub
        this.observable = observable;
        this.observable.addObserver(this);
    }

    @Override
    public void update(Observable obs, Object arg) {
        // TODO Auto-generated method stub
        if(obs instanceof TemperatureData){
            TemperatureData tempData = (TemperatureData) obs;
            currentTemperature = tempData.getTemperature();
            if(BaseTemperature < currentTemperature){
                this.powerOn();
            }else{
                this.powerOff();
            }
        }
    }

    @Override
    public void powerOn() {
        // TODO Auto-generated method stub
        System.out.println("=====");
        System.out.println("에어콘 2번");
        System.out.println("설정 온도 : "+this.BaseTemperature+"도 || "+ "현재 온도 : "+this.currentTemperature+"도");
        System.out.println("에어콘 가동 시작 !!");
        System.out.println("=====");
    }

    @Override
    public void powerOff() {
        // TODO Auto-generated method stub
        System.out.println("=====");
        System.out.println("에어콘 2번");
        System.out.println("설정 온도 : "+this.BaseTemperature+"도 || "+ "현재 온도 : "+this.currentTemperature+"도");
        System.out.println("에어콘 가동 중지 !!");
        System.out.println("=====");
    }
}
```

행동 패턴

1. 옵저버(Observer) 패턴

3) 사용법

- Observable 클래스 - TemperatureData 선언
- Observer 클래스 - AirConditionerMachine1, 2 선언
- changeTemperature() 함수를 이용하여 온도 데이터를 변경
- 각각의 Observer 클래스는 온도에 따라 상태 변화

```
public class MainController {
    static TemperatureData tempData;
    static AirConditionerMachine1 airConditionerMachine1;
    static AirConditionerMachine2 airConditionerMachine2;

    public static void MainController(){
        tempData = new TemperatureData();

        airConditionerMachine1 = new AirConditionerMachine1(tempData);
        airConditionerMachine2 = new AirConditionerMachine2(tempData);
    }

    public static void changeTemperature(float temp) {
        tempData.setMeasurements(temp);
    }

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        MainController();

        System.out.println("----- 온도변화 -----");
        changeTemperature(27.7f);
        System.out.println("");

        System.out.println("----- 온도변화-----");
        changeTemperature(27.1f);

        System.out.println("");
    }
}
```

----- 온도변화 -----

=====

에어콘 2번

설정 온도 : 28.0도 || 현재 온도 : 27.7도

에어콘 가동 중지 !!

=====

=====

에어콘 1번

설정 온도 : 26.5도 || 현재 온도 : 27.7도

에어콘 가동 시작 !!

=====

-----온도변화-----

=====

에어콘 2번

설정 온도 : 28.0도 || 현재 온도 : 27.1도

에어콘 가동 중지 !!

=====

=====

에어콘 1번

설정 온도 : 26.5도 || 현재 온도 : 27.1도

에어콘 가동 시작 !!

=====

행동 패턴

2. 이터레이터 (Iterator) 패턴

1) 정의 및 개념



이터레이터(Iterator) 패턴이란?

- 무엇인가 많이 모여 있는 것 중 하나씩 열거하며, 전체를 검색하면서 처리하는 패턴
- 어떤 목록을 순차적으로 처리하기 위한 디자인 패턴
- 문법 규칙을 클래스 화한 구조를 가지며, SQL언어나 통신 프로토콜 같은 것을 개발할 때 사용
- 내부 구현은 상관없이 항목들을 탐색 할 수 있게 함
- 자료 집합체 탐색이 가능(for 문 등을 순차 처리)

행동 패턴

2. 이터레이터 (Iterator) 패턴

2) Iterator 패턴 선언

[데이터 클래스 선언]

Book 클래스를 선언하여 데이터로 이용

```
public class Book {  
    private String name;  
  
    public Book(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```


행동 패턴

2. 이터레이터 (Iterator) 패턴

2) Iterator 패턴 선언

[이터레이터 인터페이스 선언]

다음 데이터로 이동하기 위한 함수 선언

```
public interface Iterator {  
    public boolean hasNext();  
    public Object next();  
}
```

[이터레이터 집합체 인터페이스 선언]

새로운 형태의 Iterator를 만들기 위한 함수 선언

```
public interface Aggregate {  
    public Iterator createIterator();  
}
```

행동 패턴

2. 이터레이터 (Iterator) 패턴

2) Iterator 패턴 선언

[서점 이터레이터 클래스 선언]

Iterator 기능을 새롭게 정의

```
public class LibraryIterator implements Iterator {
    private Library Lib;
    private int index;

    public LibraryIterator(Library Lib){
        this.Lib = Lib;
        this.index = 0;
    }

    @Override
    public boolean hasNext() {
        // TODO Auto-generated method stub
        if(index < Lib.getSize()){
            return true;
        }else{
            return false;
        }
    }

    @Override
    public Object next() {
        // TODO Auto-generated method stub
        Book book = Lib.getBook(index);
        index++;
        return book;
    }
}
```

행동 패턴

2. 이터레이터 (Iterator) 패턴

2) Iterator 패턴 선언

[서점 클래스 선언]

- 서점 클래스를 선언하여 book 인스턴스를 List에 저장
- createIterator() 함수를 정의 하여 iterator 기능을 제공

```
public class Library implements Aggregate {
    private ArrayList<Book> books;

    public Library() {
        this.books = new ArrayList<Book>();
    }

    public int getSize(){
        return books.size();
    }

    public void addBook(Book book){
        this.books.add(book);
    }

    public Book getBook(int index){
        return books.get(index);
    }

    @Override
    public Iterator crateIterator() {
        return new LibraryIterator(this);
    }
}
```

행동 패턴

2. 이터레이터 (Iterator) 패턴

3) 사용법

- 6개의 책 객체를 생성하여 서점 클래스에 저장
- 서점 클래스는 저장된 데이터를 iterator 기능을 통해 데이터를 나열

```
public class Main {

    public static void main(String[] args) {
        Library lib = new Library();
        lib.addBook(new Book("A"));
        lib.addBook(new Book("B"));
        lib.addBook(new Book("C"));
        lib.addBook(new Book("1"));
        lib.addBook(new Book("2"));
        lib.addBook(new Book("3"));

        Iterator it = lib.createIterator();
        while(it.hasNext()){
            Book book = (Book)it.next();
            System.out.println(book.getName());
        }
    }

}
```

A
B
C
1
2
3

행동 패턴

3. 커맨드(Command) 패턴

1) 정의 및 개념



커맨드(Command) 패턴이란?

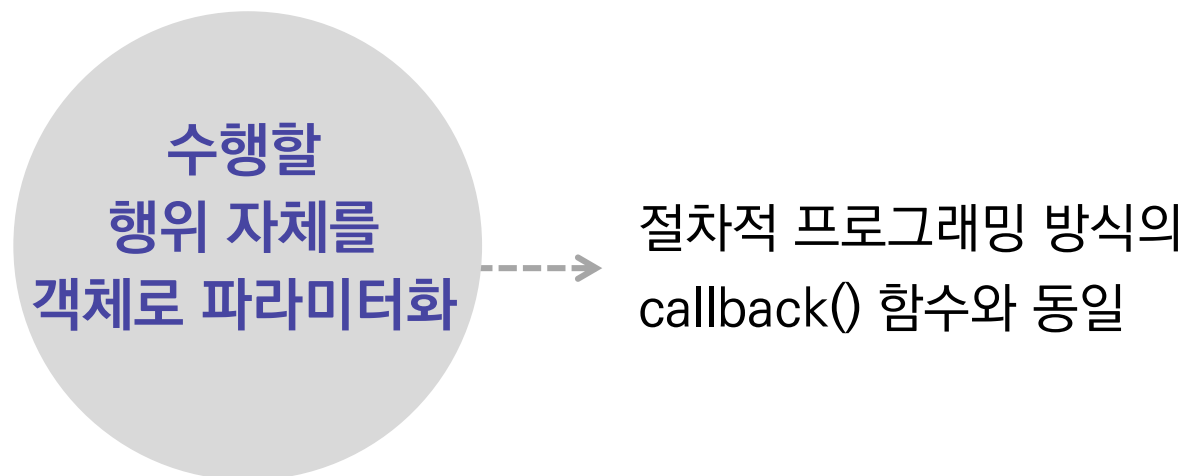
명령을 클래스로 표현하는 **구조, 요청을 객체의 형태로 캡슐화**하는 디자인 패턴

- 서로 요청이 다른 사용자의 매개변수, 요청 저장, 로깅, 연산 취소를 지원
- 하나의 추상 클래스에 메서드를 하나 만들고, 각 명령이 들어오면 그에 맞는 서브 클래스가 선택되어 실행
- 명령어를 조합해서 다른 명령어 제작 가능

행동 패턴

3. 커맨드(Command) 패턴

2) 활용



- 서로 다른 시간에 요청 명시, 저장, 수행 가능
- 명령어 객체 자체에 실행 취소에 필요한 상태를 저장 가능
- 기본적인 오퍼레이션 조합으로 고난도의 오퍼레이션 구성 가능(트랜잭션 모델링 가능)

행동 패턴

3. 커맨드(Command) 패턴

2) 활용

TV 클래스 선언

- TV에 대한 동작을 함수로 표현

```
public class TV {  
  
    public TV(){  
    }  
  
    public void turnOn(){  
        System.out.println("Turn On TV");  
    }  
  
    public void turnOff(){  
        System.out.println("Turn Off TV");  
    }  
  
    public void volumeUp(){  
        System.out.println("Volume Up");  
    }  
  
    public void volumeDown(){  
        System.out.println("Volume Down");  
    }  
}
```

행동 패턴

3. 커맨드(Command) 패턴

2) 활용

Command 인터페이스 선언

- Command 객체를 만들기 위한 인터페이스 선언

```
public interface Command {  
    public void execute();  
}
```

Command 객체 선언

- Command 인터페이스를 이용하여 커맨드 객체를 선언
- 전원 on/off , 볼륨 up/down 커맨드 객체 선언

행동 패턴

3. 커맨드(Command) 패턴

2) 활용

```
public class ActionTurnOnCommand implements Command {
    private TV tvObj;

    public ActionTurnOnCommand(TV tvObj){
        this.tvObj = tvObj;
    }

    @Override
    public void execute() {
        // TODO Auto-generated method stub
        this.tvObj.turnOn();
    }
}
```

```
public class ActionTurnOffCommand implements Command {
    private TV tvObj;

    public ActionTurnOffCommand(TV tvObj){
        this.tvObj = tvObj;
    }

    @Override
    public void execute() {
        // TODO Auto-generated method stub
        this.tvObj.turnOff();
    }
}
```

행동 패턴

3. 커맨드(Command) 패턴

2) 활용

```
public class ActionVolumeUpCommand implements Command {
    private TV tvObj;

    public ActionVolumeUpCommand(TV tvObj){
        this.tvObj = tvObj;
    }

    @Override
    public void execute() {
        // TODO Auto-generated method stub
        this.tvObj.volumeUp();
    }
}
```

```
public class ActionVolumeDownCommand implements Command {
    private TV tvObj;

    public ActionVolumeDownCommand(TV tvObj) {
        this.tvObj = tvObj;
    }

    @Override
    public void execute() {
        // TODO Auto-generated method stub
        this.tvObj.volumeDown();
    }
}
```

행동 패턴

3. 커맨드(Command) 패턴

2) 활용

Command를 처리할 객체 선언

- Command 객체 실행

```
public class ActionCommand {  
    public ActionCommand() {  
    }  
  
    public void commandExecute (Command cmd){  
        cmd.execute();  
    }  
}
```

행동 패턴

3. 커맨드(Command) 패턴

3) 사용법

- Command 객체 선언 → Command 실행 객체 이용 → Command 실행
- 네 가지 커맨드를 순서대로 ActionCommand 클래스에서 처리
- ActionCommand에서 Command를 저장하여 순서대로 스케줄링 가능

```
public class Main {

    public static void main(String[] args) {
        TV tv = new TV();
        Command TvTurnOn = new ActionTurnOnCommand(tv);
        Command VolumeUp = new ActionVolumeUpCommand(tv);
        Command VolumeDown = new ActionVolumeDownCommand(tv);
        Command TvTurnOff = new ActionTurnOffCommand(tv);

        ActionCommand ac = new ActionCommand();
        ac.commandExecute(TvTurnOn);
        ac.commandExecute(VolumeUp);
        ac.commandExecute(VolumeDown);
        ac.commandExecute(TvTurnOff);
    }
}
```

Turn On TV
Volume Up
Volume Down
Turn Off TV

행동 패턴

4. 메디에이터(Mediator) 패턴

1) 정의 및 개념



메디에이터(Mediator) 패턴이란?

중재자를 통해 처리하는 구조 패턴

- 객체 간의 상호작용을 캡슐화해서 하나의 객체 안에 정의하고, 참조 관계를 직접 정의하기보다는 이를 독립된 객체가 관리
- 통일된 인터페이스 집합을 제공
- 여러 참조 발생 시 한곳으로 몰아서 관리 함

행동 패턴

4. 메디에이터 (Mediator) 패턴

2) 활용

여러 객체가 잘 정의된 형태나
복잡한 상호작용을 하는 경우

여러 객체가 잘 정의된 형태나
복잡한 상호작용을 하는 경우

다른 객체와의 연결 관계의 복잡함으로
객체의 재사용을 방해 받을 때

여러 클래스에 분산된 행위들이
상속 없이 수정될 때

행동 패턴

4. 메디에이터 (Mediator) 패턴

2) 활용

Peer 추상 클래스 선언

- Peer 객체가 수행할 항목들을 선언하여 캡슐화

```
public abstract class Peer {
    public IMediator mediator;

    public void setMediator(IMediator Imed){
        this.mediator = Imed;
    }

    public void callEvent(String to, String from, String call){
        this.mediator.sendCall(to, from, call);
    }

    abstract public void Call(String to, String call);
    abstract public void receiveCall(String from, String call);
    abstract public String getName();
}
```

Peer 추상 클래스 선언

- Mediator 인터페이스 선언 → sendCall() 함수를 이용하여 Mediator 패턴 사용

```
public interface IMediator {
    public void sendCall(String to, String from, String call);
}
```

행동 패턴

4. 메디에이터 (Mediator) 패턴

2) 활용

Mediator 클래스 선언

- Mediator 클래스를 이용하여 처리하는 구조
- send Call()을 통해 들어오는 요청을 해당 Peer 클래스로 수행
- Mediator는 싱글톤 패턴을 이용하여 하나의 객체를 사용

```
public class Mediator implements IMediator {
    private Mediator(){
    }

    private static class SingletonHolder {
        public static final Mediator SingletonInstance = new Mediator();
    }

    public static Mediator getInstance(){
        return SingletonHolder.SingletonInstance;
    }

    private ArrayList<Peer> peerList = new ArrayList<Peer>();

    public void addPeer(Peer peerObj){
        peerObj.setMediator(this);
        peerList.add(peerObj);
    }

    @Override
    public void sendCall(String to, String from, String call) {
        for(Peer peer : peerList){
            if(peer.getName().equals(to)){
                peer.receiveCall(from, call);
            }
        }
    }
}
```


행동 패턴

4. 메디에이터 (Mediator) 패턴

2) 활용

Peer 클래스 선언

- Peer 추상 클래스를 상속 받아 기능을 추가적으로 구현
- 싱글톤 패턴을 이용 → 선언된 Mediator 객체를 가져와서 각각의 Peer들이 Mediator 통해 중재 되도록 함

```
public class PeerA extends Peer {
    private String name = "A";

    private Mediator med = Mediator.getInstance();

    public PeerA() {
    }

    @Override
    public void Call(String to, String call) {
        med.sendCall(to, name, call);
    }

    @Override
    public void receiveCall(String from, String call) {
        System.out.println("To peer : " + name + "\t From peer : " + from + "\t call : " + call);
    }

    @Override
    public String getName() {
        return this.name;
    }
}
```

행동 패턴

4. 메디에이터 (Mediator) 패턴

2) 활용

```
public class PeerC extends Peer {

    private String name = "C";

    private Mediator med = Mediator.getInstance();

    public PeerC() {
    }

    @Override
    public void Call(String to, String call) {
        med.sendCall(to, name, call);
    }

    @Override
    public void receiveCall(String from, String call) {
        System.out.println("To peer : " + name + "\t From peer : " + from + "\t call : " + call);
    }

    @Override
    public String getName() {
        return this.name;
    }
}
```

```
public class PeerB extends Peer {

    private String name = "B";

    private Mediator med = Mediator.getInstance();

    public PeerB() {
    }

    @Override
    public void Call(String to, String call) {
        med.sendCall(to, name, call);
    }

    @Override
    public void receiveCall(String from, String call) {
        System.out.println("To peer : " + name + "\t From peer : " + from + "\t call : " + call);
    }

    @Override
    public String getName() {
        return this.name;
    }
}
```

행동 패턴

4. 메디에이터 (Mediator) 패턴

2) 활용

- Peer 객체를 생성하여 Mediator에 추가
- PeerC 객체가 PeerA에게 Mediator의 sendCall()을 이용하여 Hello메시지 전송
- PeerA객체에서 Call() 함수를 호출, Mediator의 sendCall()을 이용하여 PeerB에게 메시지 전송

```
public class Main {
    public static void main(String[] args) {
        Peer peerA = new PeerA();
        Peer peerB = new PeerB();
        Peer peerC = new PeerC();

        Mediator med = Mediator.getInstance();
        med.addPeer(peerA);
        med.addPeer(peerB);
        med.addPeer(peerC);

        //PeerC -> PeerA
        med.sendCall(peerA.getName(), peerC.getName(), "Hello");
        //PeerA -> PeerB
        peerA.Call(peerB.getName(), "Hi");
    }
}
```

To peer : A

To peer : B

From peer : C call : Hello

From peer : C call : Hi

핵심정리

1. 소프트웨어 구조 패턴

- 클래스나 객체의 합성에 관한 패턴
- Adapter 패턴
 - ✓ 패턴 클래스의 인터페이스를 사용자가 기대하는 다른 인터페이스로 변환하는 패턴
 - ✓ 호환성이 없는 인터페이스 때문에, 함께 동작할 수 없는 클래스들이 함께 작동하도록 해주는 패턴
- Facade 패턴
 - ✓ 서브 시스템에 있는 인터페이스 집합에 통합된 하나의 인터페이스를 제공, 서브 시스템을 좀 더 쉽게 사용하기 위해 고수준의 인터페이스를 정의
- Proxy 패턴
 - ✓ 어떤 다른 객체로 접근하는 것을 통제하기 위해 그 객체의 매니저 또는 자리 채움자를 제공하는 패턴
- Composite 패턴
 - ✓ 객체들의 관계를 트리 구조로 구성하여 부분-전체 계층을 표현하는 패턴으로, 사용자가 단일 / 복합객체 모두 동일하게 다루도록 하는 패턴

핵심정리

2. 소프트웨어 행동 패턴

- 클래스나 객체들이 상호작용하는 방법과 책임을 분산하는 방법을 정의하는 패턴
- Observer 패턴
 - ✓ 객체들 사이에 1 : N의 의존관계를 정의하여 어떤 객체의 상태가 변할 때, 의존관계에 있는 모든 객체들이 통지 받고 자동으로 갱신될 수 있게 만드는 패턴
- Iterator 패턴
 - ✓ 내부 표현부를 노출하지 않고 어떤 객체 집합의 원소들을 순차적으로 접근할 방법을 제공하는 패턴
- Command패턴
 - ✓ 요청을 객체로 캡슐화하여 서로 다른 사용자의 매개변수화, 요청 저장 또는 로깅, 연산의 취소를 지원하게 만드는 패턴
- Mediator 패턴
 - ✓ 한 집합에 속해있는 객체들의 상호작용을 캡슐화하는 객체를 정의하는 패턴
 - ✓ 중재자는 객체들이 직접 서로 참조하지 않도록 함으로써 객체들 간의 느슨한 연결을 촉진시키며 객체들의 상호작용을 독립적으로 다양화 가능