목차

[1. 중재자 패턴(Mediator Pattern) 2](#_Toc491244922)

[1-1. 정의 2](#_Toc491244923)

[1-2. 중재자 패턴을 위한 요소 2](#_Toc491244924)

[**1-3.** UML 3](#_Toc491244925)

[1-4. 중재자 패턴의 장단점 3](#_Toc491244926)

[**2.중재자 패턴** 5](#_Toc491244927)

[**3 중재자 패턴 사용예제** 11](#_Toc491244928)

1. 중재자 패턴(Mediator Pattern)

## 정의

GoF의 정의에 따르면, 중재자 패턴은 "서로 전혀 다른 객체간에  상호 통신이 매우 복잡할때 중재자를 통해 객체간의 통신을 가능하도록 캡슐화하는 방법"이다.

좀 복잡하게 들릴수 있지만, 간단하게 말하면 두 객체를 직접 연결을 하지 않고, 제 3의 객체를 통해 느슨하게 연결하는 것이다.

가장 대표적인 자바의  중재자 구현 구현은 JMS(Java Message Server) 이다. JMS는 객체가 서로에 대해 레퍼런스를 가지고 있지 않더라도, 중간에 JMS서버를 두고 message를 받는 쪽은 subscribt, 보내는 쪽은 publcation 방식으로 메시지를 주고 받을 수 있다. 즉 Sender - JMS - receiver 를 통한 통신이 가능한 것이다.

또 다른 예는 java.util.concurrent.ExecutorService 가 있다. ExecutorService는 유저가 직접 쓰레드를 생성하는 방식이 가지고 있는 단점(전체 쓰레드 갯수 관리의 어려움)을 극복하기 위한 방식으로,  쓰레드 - ExecutorService - 워커쓰레드 방식의 연결을 해준다.

## 1-2. 중재자 패턴을 위한 요소

- 중재자 (Mediator)

: 실제 중재가 구현 로직에 대한 인터페이스로, 중재자의 서비스를 받기 바라는 클라이언트를 등록,실행하는 API가 정의되어있다.

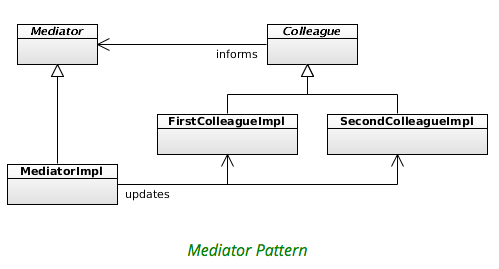
- 중재자에 대한 구현체(Concrete Mediator)

: 중재자 인터페이스를 실제로 구현한 클래스

- 중재자에 참여하는 동료 클래스들(Colleague Classes)

: 중재자에 의해 서비스를 받기를 원하는 대상 클래스들

## **1-3.** UML



## 1-4. 중재자 패턴의 장단점

장점:

효율적인 자원 관리(리소스 풀등)를 가능하게 한다.

객체간의 통신을 위해 서로간에 직접 참조할 필요가 없게 한다.

중재자 구현 클래스는 추후에 더 효율적인 클래스로 변경될수 있다.

단점:

객체간의 통신 로직이 복잡해지거나 객체의 형태가 자주 변경되는 경우 유지보수,관리가 어렵다.

**5. JDK 예제**

java.util.Timer.scheduleXXX()

:"새로운 쓰레드, TimerTask 를 schedule 함수"가 연결해 준다.

 java.util.concurrent.ExecutorService.execute()

: "새로운 쓰레드 - ExecutorService.excute - 워커쓰레드 " 간의 연결

java.lang.reflect.Method.invoke()

: "현재 쓰레드 - Method.invoke - 임의의 객체" 간의 연결을 해준다.

javax.jms.\*

: 응용프로그램간 비동기 메시징 서비스인 jms에서도 중재자 패턴을 사용한다. 정확히는 중재자 패턴과 옵저버 패턴이 적당히 섞여있다.

jms 의 경우 중재자 패턴과 옵져버 패턴이 적당히 섞인 경우이다.

jms에서는 메시지를 보내기 위한 중개서버인 jms 서버에 대한 연결 객체를 얻은 후,

메시지를 보내고 받기 위한 쓰레드인 Session 객체를 얻을수 있다.

그리고 이 session 쓰레드에는 메시지를 보낼수 있는 queue 와 메시지를 받을수 있는 queue를 설정해야 한다.

이후에는 session 쓰레드가 jms 서버를 통해 원격에 있는 객체와 통신을 할수 있게 되는 것이다. 즉

"session thread - session.createProducer - sender queue - remove jms receiver" 혹은

"session thread - session.createConsumer - receiver queue - remove jms sender"  의 결합이 가능하게 한다.

옵저버 패턴이 섞여 있다고 하는 것은 receiver queue 에 대한 처리를 할때 receiver 콜백함수 session에 등록하는 형식으로 프로그래밍을 많이 하기 때문이다.

**참조글)**

기본개념:

<http://www.oodesign.com/mediator-pattern.html>

구체적 코드예제:

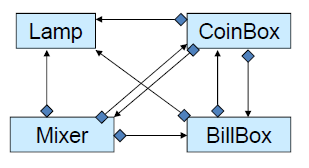
<http://www.java2s.com/Code/Java/J2EE/ThisexampleisasimpleJMSclientapplication.htm>

### **2.중재자 패턴**

**문제점**

  - 부품 객체들이 서로 강하게 연결되어 있음.

   \* 각 부품 객체들이 서로 필요한 객체들을 참조하고 있다.



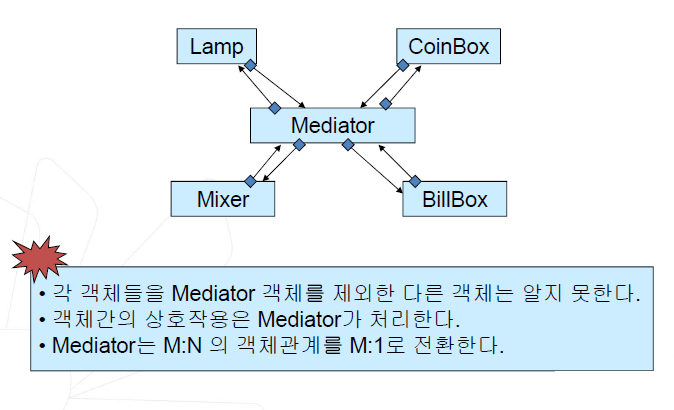
 - 한 부품의 class가 변겨오딘다면 연관된 class들의 수정이 필요하다.

 - 자판기 제어프로그램의 제어흐름이 각 부품 class마다 흩어져 있어서, 새로운 부품이 추가되었을 때 이를 위한 코드 변경이 어렵다.

**해결방안**

  - 부품 객체들 간의 연결을 느슨하게 만들어야 한다.

  - 각 부품 객체들간의 상호작용을 도맡아 처리하는 객체를 둔다.



**의도**

  - 여러 객체들 간으 ㅣ상호작용 자체를 encapsulation하는 객체를 정의한다.

  - 객체들끼리 직접 참조하는 것을 피함으로써 객체들 간의 연결 강도를 줄인다.

  - 객체들과 독립적으로 상호작용을 변경할 수 있다.

**적용범위**

  - 객체들간의 상호작용이 복잡해서 서로간의 의존관계가 구조화 되어있지 않고 이해하기 어려울 때

  - 하나의 객체가 많은 다른 객체들을 참조하고 있어 이것을 재사용하기 어려울 때

  - 여러 class에 분산되어 있는 행위를 많은 subclassing 없이 재구성해야 할 때

예제.

Colleague.java

|  |  |  |
| --- | --- | --- |
| 1  2  3  4  5  6  7  8 | package mediator;    public abstract class Colleague {      IMediator mediator;        public abstract void doSomething();  }    [*Colored by Color Scripter*](http://colorscripter.com/info#e) | [cs](http://colorscripter.com/info#e) |

IMediator.java

|  |  |  |
| --- | --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | package mediator;    public interface IMediator {      public void fight();        public void talk();        public void registerA(ColleagueA a);        public void registerB(ColleagueB b);  }    [*Colored by Color Scripter*](http://colorscripter.com/info#e) | [cs](http://colorscripter.com/info#e) |

ColleagueA.java

|  |  |  |
| --- | --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | package mediator;    public class ColleagueA extends Colleague {        public ColleagueA(IMediator mediator) {          this.mediator = mediator;      }        public void doSomething() {          this.mediator.talk();          this.mediator.registerA(this);      }    }    [*Colored by Color Scripter*](http://colorscripter.com/info#e) | [cs](http://colorscripter.com/info#e) |

ColleagueB.java

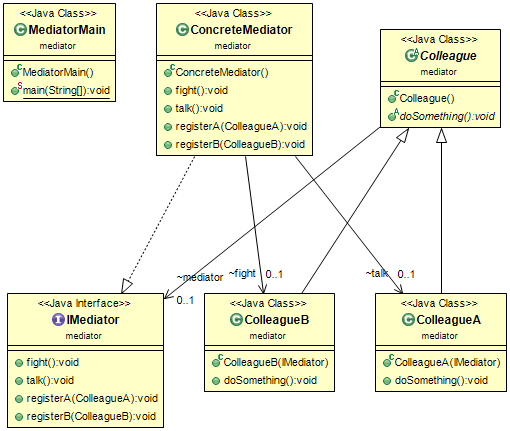
|  |  |  |
| --- | --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | package mediator;    public class ColleagueB extends Colleague {        public ColleagueB(IMediator mediator) {          this.mediator = mediator;          this.mediator.registerB(this);      }        public void doSomething() {          this.mediator.fight();      }    }    [*Colored by Color Scripter*](http://colorscripter.com/info#e) | [cs](http://colorscripter.com/info#e) |

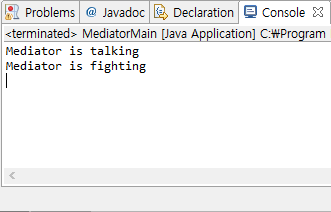
ConcreteMediator.java

|  |  |  |
| --- | --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | package mediator;    public class ConcreteMediator implements IMediator {        ColleagueA talk;      ColleagueB fight;        public void fight() {          System.out.println("Mediator is fighting");      }        public void talk() {          System.out.println("Mediator is talking");      }        public void registerA(ColleagueA a) {          talk = a;      }        public void registerB(ColleagueB b) {          fight = b;      }    }    [*Colored by Color Scripter*](http://colorscripter.com/info#e) | [cs](http://colorscripter.com/info#e) |

MediatorMain.java

|  |  |  |
| --- | --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | package mediator;    public class MediatorMain {      public static void main(String[] args) {          IMediator mediator = new ConcreteMediator();            ColleagueA talkColleague = new ColleagueA(mediator);          ColleagueB fightColleague = new ColleagueB(mediator);            talkColleague.doSomething();          fightColleague.doSomething();      }  }    [*Colored by Color Scripter*](http://colorscripter.com/info#e) | [cs](http://colorscripter.com/info#e) |



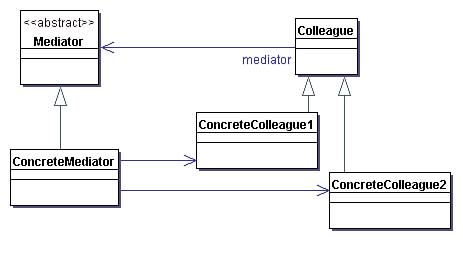


### **3 중재자 패턴 사용예제**

**중재자 패턴**은

객체간의 상호 작용을 세트로 묶어 캡슐화하는 객체를 정의하는 패턴입니다.

UML은 다음과 같습니다.



 Client는 중재자를 통해 메시지를 보내기도

다른 곳에서 메시지를 받기도 합니다.

import java.util.Date;

public class ChatRoom {

public static void showMessage(User user, String message){

System.out.println(new Date().toString() + " [" + user.getName() + "] : " + message);

}

}

public class User {

private String name;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public User(String name){

this.name = name;

}

public void sendMessage(String message){

ChatRoom.showMessage(this,message);

}

}

public class MediatorPatternDemo {

public static void main(String[] args) {

User robert = new User("Robert");

User john = new User("John");

robert.sendMessage("Hi! John!");

john.sendMessage("Hello! Robert!");

}

}

위 소스에서 ChatRoom이 중재자,

User가 동료가 되는 것이다.

추가적으로 ChatRoom이 User들을 자료구조를 통해 관리할 수도 있고

원한다면 인터페이스를 더 추가하고 많은 기능들을 캡슐화 해 둘 수도 있다.

중재자 패턴을 보면 *파사드 패턴이 생각난다*.

굉장히 흡사하기도 하지만

중재자 패턴은 행위 패턴임을 잊지 말자. in run-time 행위들을 묶어주는 거다.

중재자 패턴의 주 목적은 프로그램 내 존재하는 많~~~은 class들의 통신을 중재자로 몰아 유지보수성을 높이자는 것이다.

각 Class가 서로 직접 통신을 한다면 얼마나 복잡하겠는가..

결국 중재자 패턴을 통해 의존성을 낮추고 소스 리딩성을 올리고 **유지보수를 편하게** 하는 것이 주 사용 목적이다.

아쉬우니 예제를 하나 더 추가하겠습니다.

주제:

3개의 버튼이 있다. 1개의 버튼이 실행되면 실행된 버튼은 잠기고 나머지 버튼은 활성화가 된다.

각 버튼의 이벤트 부분에 이 동작을 넣는 다면.. 아마 일회용 코드가 될 것이다.

import java.awt.Font;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JPanel;

//Colleague interface

interface Command {

void execute();

}

//Abstract Mediator

interface Mediator {

void book();

void view();

void search();

void registerView(BtnView v);

void registerSearch(BtnSearch s);

void registerBook(BtnBook b);

void registerDisplay(LblDisplay d);

}

//Concrete mediator

class ParticipantMediator implements Mediator {

BtnView btnView;

BtnSearch btnSearch;

BtnBook btnBook;

LblDisplay show;

//....

public void registerView(BtnView v) {

btnView = v;

}

public void registerSearch(BtnSearch s) {

btnSearch = s;

}

public void registerBook(BtnBook b) {

btnBook = b;

}

public void registerDisplay(LblDisplay d) {

show = d;

}

public void book() {

btnBook.setEnabled(false);

btnView.setEnabled(true);

btnSearch.setEnabled(true);

show.setText("booking...");

}

public void view() {

btnView.setEnabled(false);

btnSearch.setEnabled(true);

btnBook.setEnabled(true);

show.setText("viewing...");

}

public void search() {

btnSearch.setEnabled(false);

btnView.setEnabled(true);

btnBook.setEnabled(true);

show.setText("searching...");

}

}

//A concrete colleague

class BtnView extends JButton implements Command {

Mediator med;

BtnView(ActionListener al, Mediator m) {

super("View");

addActionListener(al);

med = m;

med.registerView(this);

}

public void execute() {

med.view();

}

}

//A concrete colleague

class BtnSearch extends JButton implements Command {

Mediator med;

BtnSearch(ActionListener al, Mediator m) {

super("Search");

addActionListener(al);

med = m;

med.registerSearch(this);

}

public void execute() {

med.search();

}

}

//A concrete colleague

class BtnBook extends JButton implements Command {

Mediator med;

BtnBook(ActionListener al, Mediator m) {

super("Book");

addActionListener(al);

med = m;

med.registerBook(this);

}

public void execute() {

med.book();

}

}

class LblDisplay extends JLabel {

Mediator med;

LblDisplay(Mediator m) {

super("Just start...");

med = m;

med.registerDisplay(this);

setFont(new Font("Arial", Font.BOLD, 24));

}

}

class MediatorDemo extends JFrame implements ActionListener {

Mediator med = new ParticipantMediator();

MediatorDemo() {

JPanel p = new JPanel();

p.add(new BtnView(this, med));

p.add(new BtnBook(this, med));

p.add(new BtnSearch(this, med));

getContentPane().add(new LblDisplay(med), "North");

getContentPane().add(p, "South");

setSize(400, 200);

setVisible(true);

}

public void actionPerformed(ActionEvent ae) {

Command comd = (Command) ae.getSource();

comd.execute();

}

public static void main(String[] args) {

new MediatorDemo();

}

}