<JSTORM>

Design Pattern의 적용 - 2

- 자바 API 에서의 디자인패턴

중앙대학교 컴퓨터공학과
자바 동호회
JSTORM
http://www.jstorm.pe.kr

제목	디자인 패턴의 적용 –java API 에서의 디자인패턴	일련번호	
배포	마소 연재 기사	작성일	2000-08-14
분류	Public	변경일	2000-08-16
상태	Draft	필자	윤동빈(dbin@handysoft.co.kr), 백창현(pam@emotion.co.kr)

변경 HISTORY

변경한 사람	변경한 날짜	비고
윤준호 (junoyoon@orgio.net)	2001/4/15	11 월호와 12 월호 합본해서 jstorm 문서형식의 pdf 로 편집.

Table of Contents

마소 11 월호 기사

마소 12 월호기사

디자인 패턴의 적용
– java API 에서의 디자인패턴 11 월호

지난호에 이어

필자들은 지난 호까지 2 회에 걸쳐 패턴의 사용 경험을 기술하여 왔다. 그러나 지난 호까지 투고했던 내용들은 널리 알려진 시스템의 사례는 아닌, 필자들의 개인 경험에 의존한 사례들이었다. 패턴을 적용하면서 보고 느껴왔던 점들을 적은 글이기에 나름대로의 흥미 있는 주제가 되었으리라 기대한다.

그러나, 필자들이 만든 코드가 아닌 다른 유명한 시스템에서도 디자인 패턴이 정말 쓰이고 있는 것인가? 라고 의문을 가질 분들도 틀림 없이 있다. (한국 사람들은 원래 그런 것에 관심이 많아서 "어 여기서 쓰고 있네" 그래야지 좋다는 것을 깨닫는 습성이 있기도 하다. 필자들 또한 그렇다. – 개인 소견).

많은 디자인 패턴이 도입되어 있는 시스템에는 JAVA 가 있다. JAVA에서 사용되는 여러 클래스들에는 분명히 디자인 패턴의 기법들이 도입되어 있다. 물론 JAVA VM 을 만들고 JAVA를 설계한 사람들이 모두 디자인 패턴이라는 것을 의식하고 사용하였다는 것은 분명 아닐 것이다. JAVA 소스 코드나 VM 소스 코드의 어느 구석을 찾아 보아도 디자인 패턴이라는 어휘는 한마디도 나오지 않는다. 이번 기사를 쓰기 위해서 VM 의 일부분을 뒤지고 찾아 보았지만 디자인 패턴이라는 단어는 찾을 수 없었다.

그러나 자바의 부분들을 자세히 찾아보면 디자인 패턴의 덩어리라는 것을 알 수 있다. 특히나 자바의 환경은 JAVA에서 기본으로 제공하는 클래스들로 구성되어 있다는 사실과 이들 클래스들이 나름대로의 역할을 수행하고 상호 연동되는 과정에서 패턴이 사용된다. 알게 모르게 많은 부분들이 패턴으로 구현되어 있다는 사실은 이전에도 언뜻 생각이 들었지만, 이번 호의 원고를 준비하면서 정말 많은 부분들이 패턴으로 구현되어 있구나 라고 생각하게 되었다.

하나의 이론을 설명하는데 있어서 가장 좋은 방법은 분명 적절한 사례를 드는 것이다. 패턴의 원고를 기고하면서 이러한 실질적인 사례를 들고자 필자들은 많은 노력을 기울여 왔다. 그러나 그 노력이 필자들이 구현하였던 부분에 국한 되었다는 점 또한 사실이다. 이번 호에서는 JAVA에서 내부적으로 사용하고 있는 디자인 패턴의 사례를 들어 패턴이라는 것이 분명 널리 쓰이고 있다는 사실을 알리고자 한다.

JAVA 패턴을 설명

앞에서도 이야기 하였지만, 디자인 패턴은 자바에서도 널리 사용된다. 구체적으로 어떻게 사용되고 있기에 널리 사용되고 있다는 말을 할 수 있는가? 이때의 역할은 무엇이며 자바 내에서 어떻게 사용되고 있는가?

이러한 의문점들은 JAVA에서 사용된 패턴의 적절한 예제를 보면 하나 둘 씩 풀려나갈 것이다. 자바는 C 의 속성을 지니고 있는 C++과는 달리 애초부터 인터페이스, 상속 관계 등의 개념을 가지고 만들어진 객체지향 언어이기 때문에 더 많은 부분이 디자인 패턴을 이용하여 구성되어 있다. 구체적으로 어떠한 부분이 이렇게 패턴으로 구성되어 있을까?

이번호의 기사는 이의 설명을 위해서 2 가지 부분으로 나누었다. 하나는 패턴 중심으로 패턴을 설명하고 이 패턴이 어떻게 자바에서 사용되고 있는 가에 대한 부분이며, 다른 하나는 기능 중심에서 특정 기능이 어떻게 패턴을 사용하는가에 관한 방법이다.

패턴 중심의 자바 패턴 설명

패턴 중심의 자바 패턴 설명은 몇몇의 중요한 패턴이 자바에서 어떤 용도로 사용되고 있는가 하는 관점에서 접근한다. 이 부분에서는 모두 4 개의 패턴을 설명하려고 한다.

- Immutable Pattern

- Adaptor Pattern
- Bridge Pattern
- FlyWeight Pattern

각각의 패턴들을 기본적인 설명과 함께, 자바에서의 적용 사례와 사용하기 위한 개념을 설명하려 한다.

Immutable Pattern

이뮤터블 패턴

"이뮤터블 패턴은 다른 패턴들과 근본적으로 다르다. 이뮤터블 패턴은 적절한 곳에 적용될 때, 안정성과 유지 보수성이 비약적으로 향상된다." – from Pattern in JAVA.

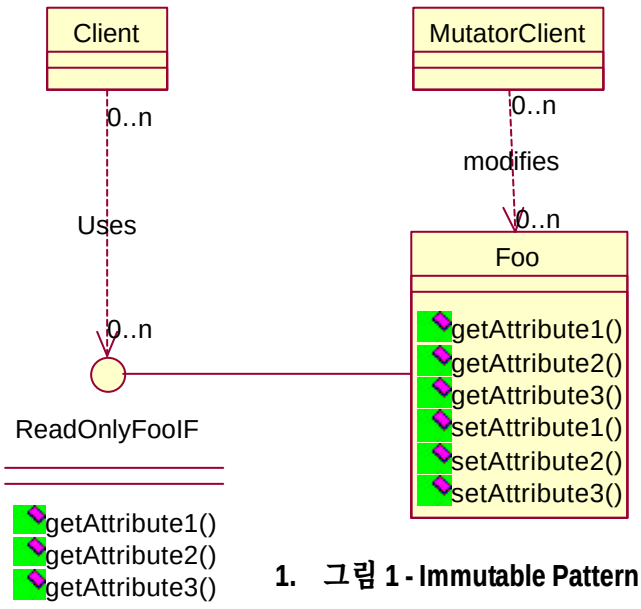
이뮤터블 패턴의 개념은 한마디로 표현할 수 있다. 한번 만든 객체는 변형하지 않으며, 변형이 발생할 때는 새로운 객체를 만든다는 것.

JAVA의 STRING 객체는 String이 변형될 때 새로운 객체가 만들어 진다는 사실을 알고 있는 독자들도 있을 것이다. 이를 간단히 설명하기 위해 다음을 보자.

```
String A = 'HELLO';
String B = 'HI';
String C = A.concat(B);
```

2개의 스트링을 하나로 만드는 작업인 concat을 수행하는 시점에서 A객체에 B의 내용이 덧붙여 C에 할당되는 것이 아니라, 새로운 스트링 객체가 만들어져 C로 할당된다.

한번 생성된 스트링 객체는 결코 변하지 않는다. 객체는 변하지 않으며 다만 새로운 객체가 만들어질 뿐이다. 이뮤터블 패턴은 다음과 같은 구조로 구성된다.



1. 그림 1 - Immutable Pattern

외부의 클라이언트 객체는 읽기 전용의 인터페이스만 사용할 수 있으며, 몇몇

의 허가된 클래스만이 본 객체를 변화시킬 수 있다. 객체에 변화가 발생하면 이뮤터블 패턴은 기존 객체의 내용을 변화시키지 않고 새로운 객체를 만들어 변화된 값을 할당한다. 이러한 방식으로 멀티 쓰레드에서 발생하는 여러 가지 문제들을 원천적으로 해결할 수 있다.
"어, 변형 시에 무조건 객체를 새로 생성하는 것이 무슨 패턴이야?" 라고 의문점을 표시하는 독자들도 있으리라 생각된다. 이러한 의문점처럼 이뮤터블 패턴은 복잡한 클래스 구조를 가지는 패턴은 아니다.
오히려 객체의 사용 전략에 관한 방법론에 가깝다. 한가지 분명한 것은 이뮤터블 패턴은 몇몇 장소에서 매우 유용하다는 것이다. 이들 장소 중 가장 유용한 곳은 바로 JAVA의 String 객체 표현이 아닐까 한다.

Immutable in JAVA

"스트링 클래스의 인스턴스는 이뮤터블이다. 스트링 객체가 표현하는 문자들은 객체가 생성될 때 결정된다. 스크링 클래스는 스트링 객체가 표현하는 문자들을 변경하기 위한 어떠한 메서드도 제공하지 않는다. 스트링 객체에 존재하는 toLowerCase, subString 등의 메서드들은 새로이 만들어지는 문자들을 새로이 생성된 스트링 객체에 할당되어 반환된다." – from Pattern in JAVA

Immutable Pattern 이 사용되는 곳은 단연코 String 객체이다.
String 객체에서 스트링의 일부를 잘라내거나, 대소문자의 변경과 같은 스트링에 변형이 발생하면 새로운 스트링 객체가 생성되고 변형된 결과는 새로 생성된 스트링 객체에 반영된다.
다음의 함수들을 보자.

```
public String concat(String str) {
        int otherLen = str.length();
        if (otherLen == 0) {
            return this;
        }
        char buf[] = new char[count + otherLen];
        getChars(0, count, buf, 0);
        str.getChars(0, otherLen, buf, count);
        return new String(0, count + otherLen, buf);
}

public String substring(int beginIndex, int endIndex) {
        if (beginIndex < 0) {
            throw new StringIndexOutOfBoundsException(beginIndex);
        }
        if (endIndex > count) {
            throw new StringIndexOutOfBoundsException(endIndex);
        }
        if (beginIndex > endIndex) {
            throw new StringIndexOutOfBoundsException(endIndex -
beginIndex);
        }
        return ((beginIndex == 0) && (endIndex == count)) ? this :
            new String(offset + beginIndex, endIndex - beginIndex, value);
}
```

위의 2 가지 함수들은 JDK 의 String 클래스의 소스 코드에 포함되어 있는 2 가지 함수이다.
위에서 보듯이 concat 과 subString 의 2 가지 스트링을 변형시키는 함수는 내부적으로 새로운 스트링 버퍼를 구성하고 스트링 객체를 만드는 여러 작업들을 수행하고 마지막에는 반드시 새로운 스트링을 만들어서 반환한다.

사용하려면?

이뮤터블 패턴을 어떠한 곳에 사용할 수 있을까? JAVA 의 STRING 객체는 Immutable 패턴을 사용하기 위한 매우 적절한 예제이다. 그러나 필자들은 지금까지 프로그램을 짜면서 이뮤터블을 사용할만한 적절한 예제를 찾은 적이 없다.

적절한 예제를 찾을 수 없었던 원인중의 하나는 JAVA 에서는 가비지 컬렉션이 자동으로 발생하기 때문에 이를 사용할 수 있지만, C++이나 다른 가비지 컬렉션을 지원하지 않는 언어에서는 사용하기 힘들다는 문제점이 있다. 새로이 생성한 객체와 이전의 객체를 어느 시점에서 메모리에서 삭제할 것인가? 이 문제를 명확하게 처리하지 않으면 Immutable 객체는 사용하기 쉽지 않다.

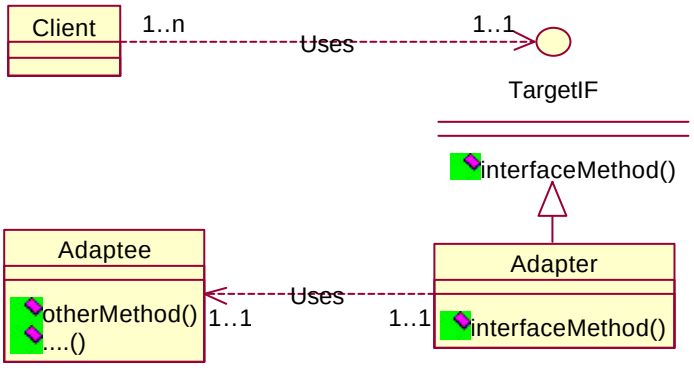
Adaptor Pattern

어댑터 패턴

"어뎁터 클래스는 인터페이스를 구현을 통하여 클라이언트에 알려지지 않은 클래스 인스턴스를 억세스 가능하게 한다. 어뎁터 객체는 어떠한 객체가 인터페이스를 구현하였나에 관계 없이 인터페이스에 의해 정의된 기능을 제공한다." – from Patterns in JAVA

간단히 말하자면 객체를 인터페이스와 구현 클래스로 분리하여 여러 개의 객체들이 하나의 인터페이스에 의해서 사용될 수 있도록 정의하는 것이다. 왜 이렇게 하는 것일까? 프로그램들을 재사용하기 위한 방법 중의 하나는 인터페이스와 구현 클래스의 분리이다. 기능을 이용하기 원하는 클라이언트 클래스들은 오직 인터페이스만을 사용한다.

인터페이스와 구현 클래스의 분리는 최상의 캡슐화 효과를 가져온다. 단지 인터페이스만을 사용하는 클라이언트 클래스에서는 인터페이스를 구현한 구현 클래스의 내부적인 상태에 접근할 방법이 없다.



2. 그림 2 - Adapter Pattern

위의 그림에서 어뎁터는 인터페이스를 제공하여 외부에 노출시킨다. 정작 어뎁터 클래스 자신은 작업을 수행하기 위해서 어뎁티(Adaptee) 클래스를 사용한다. 이 부분은 어뎁터 클래스가 단순한 클래스 인터페이스 분리와 다르다는 것을 암시한다. 어뎁터 클래스에는 구현이 거의 존재하지 않으며 이름 그대로의 어뎁터 역할을 담당한다. 어뎁터는 클라이언트 클래스에 기능을 제공하기 위한 중간 클래스의 역할을 담당한다.

결과적으로는 기능을 구현한 어뎁티 클래스와 기능을 사용하는 클라이언트 클래스는 상호 독립적으로 존재하게 된다. 그리고 어뎁터 클래스를 이용하여 어떤 메서드가 호출될지를 다이나믹하게 설정할 수 있다. 데이터 베이스 데이터를 가지고 오는 클래스가 구현되어 있고 이를 어뎁터를 사용하여 이용하였을 때, 데이터 베이스가 아닌 XML 과 같은 다른 데이터 소스에서 정보를 가져올 때도 클라이언트가 어뎁터만 변경하면 동일한 인터페이스로 접근할 수 있다.

Adaptor in JAVA

"자바 API는 당장 사용할 수 있는 어떠한 공용 어뎁터 객체도 포함하지 않고 있다. 다만, java.awt.event.WindowAdapter와 같은 직접적으로 사용되기 보다는 간접적으로 상속받아 사용될 수 있게 하기 위한 클래스들을 포함한다. 이는 WindowListener와 같은 여러 개의 메서드를 선언하는 어떤 이벤트 리스너가 존재한다는 사상이다. 많은 경우에 있어서 모든 메서드들이 구현될 필요는 없다. WindowListener 인터페이스는 모두 8가지의 서로 다른 종류의 윈도우 이벤트에 따라 호출되기 위한 8개의 모서드를 선언한다. 이중에서 1개 내지 2개의 이벤트만이 주로 관심의 대상이 된다. 관심의 대상이 아닌 이벤트에 상응하는 인터페이스들은 do-nothing 구현 방법으로 처리된다. WindowAdapter 클래스는 WindowListener 인터페이스를 구현하지만, 이를 모두 Do-Nothing 방식으로 구현한다. WindowAdapter 클래스를 상속받는 어뎁터 클래스는 관심 대상이 되는 주 이벤트를 처리하는 인터페이스만 구현하면 된다. 나머지는 do-Nothing 으로 상속되어 처리된다.

```
addWindowListener(new WindowAdapter() {
        public void windowClosing(WindoeEvent e) {
                exit();
        } // windowClosing(WindowEvent)
};
```

위의 코드 예제에서 익명 어뎁터(Anonymous Adapter)클래스는 WindowAdapter 클래스를 상속 받는다. 이는 단지 windowClosing 매서드만을 구현한다. 나머지 7개의 이벤트는 WindowAdapter 클래스의 do-Nothing 구현으로 처리된다." – from Pattern in JAVA

AWT,SWING의 경우 Window UI, Button, Box 등의 객체 들과 관련되어 많은 부분에서 패턴이 사용되고 있다. 어뎁터 패턴도 역시 AWT의 윈도우 어뎁터에서 사용된다.

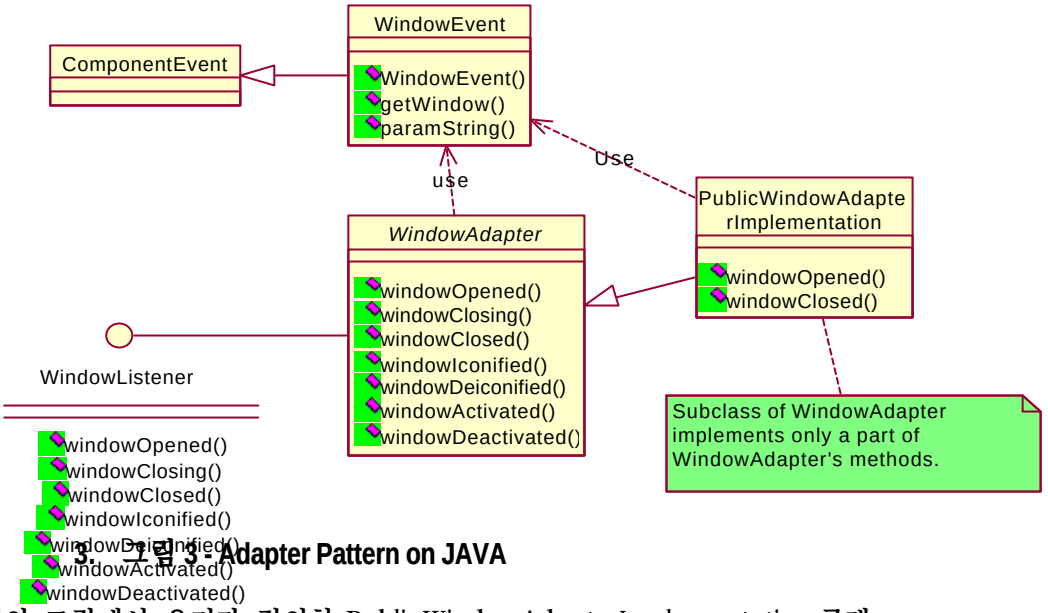


그림 3. Adapter Pattern on JAVA

상기의 그림에서 유저가 정의한 PublicWindowAdapterImplementation 클래스 부분이 어뎁터를 구현하는 클래스이다. 이 클래스는 Abstract Class 인 WindowAdapter 를 상속 받아 인터페이스를 구현한다. 구현은 다른 적절한 클래스를 호출하여 정의된다. 여기서 클라이언트 어플리케이션은 이벤트를 생성하는 곳이며, 생성된 이벤트에 의해서 인터페이스가 호출되면 어뎁터를 구현한 PublicWindowAdapterImplementation 클래스의 메서드가 수행된다.

사용하려면?

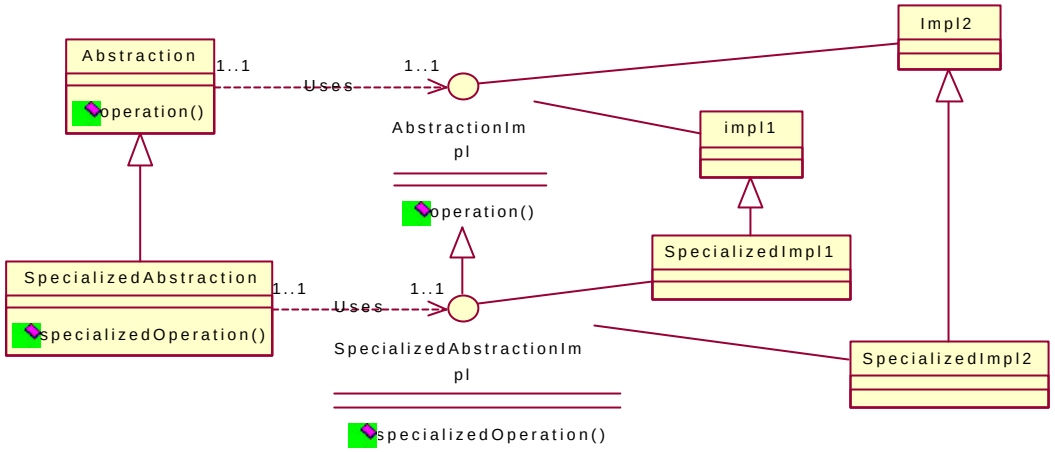
어뎁터 패턴을 사용하는 것은 코드의 재활용성을 높일 수 있다. 그러나 어뎁터를 사용하는 것은 물리적으로는 서로 다른 기능을 논리적인 개념의 인터페이스로 묶어 사용하는 것으로 적절한 인터페이스를 정의하는 것이 쉽지는 않다. 예를 들어 XML 에서 데이터 억세스하는 부분과 DB 에서 데이터 억세스하는 것을 하나로 통합하기 위해서 어뎁터로 구성하는 부분에서 공용 인터페이스로 정의할 수 있는 부분도 있으나 그렇지 못하고 데이터 소스에 따라 별도로 분리하여 정의되어야 할 부분도 있다. 이러한 차이점을 극복하고 적절한 인터페이스를 정의한다는 것은 쉽지 않은 일이다. 그렇다고 공통되는 부분만을 인터페이스로 구현하면 그렇지 않은 부분은 어떻게 적절히 설정할 것인가? 이 부분의 해결에는 경험과 노하우가 필요하다.

Bridge Pattern

브리지 패턴

"브리지 패턴은 추상의 계층관계와 상응하는 추상의 구현이 존재할 때 유용하다. 추상화 클래스와 구현 클래스를 여러 개의 구분된 클래스로 나누어 구현하지 않고, 추상클래스와 구현클래스를 별개의 클래스로 구현하여 이들이 다이나믹하게 조합될 수 있게 한다." – from Patterns in JAVA

브리지 패턴은 추상화에 관계된 클래스들이 이들 추상 객체를 구현하는 클래스들과 분리될 수 있도록 한다. 추상 클래스들의 계층관계가 존재할 때, 이들을 구현하는 클래스들의 계층관계도 존재한다.



4. 그림 4 - Bridge Pattern

위의 그림에서 추상 객체들간의 계층관계는 이를 상속 받아 구현하는 객체들의 추상관계에 그대로 반영된다. 이 같은 브리지 패턴은 하나의 클래스만이 추상관계를 이용하여 독립적으로 분리되는 개념이 아닌, 일련의 클래스의 집합들이 동등한 계층관계유지하며 유지된다는 점에서 어뎁터 패턴과의 차별성을 지닌다.

한마디로 추상 객체를 1 개가 아닌 여러 개를 정의하고, 이들을 사용하여 프로그램을 구성한 다음, 이들을 구현하는 여러 개의 구현 클래스를 만들어 프로그램을 완성한다. 구현 클래스는 하나의 상황에 대한 것만이 아니며, 여러 개의 상황에 대해서 여러 벌의 구현 클래스 집합이 적절하게 구성된다.

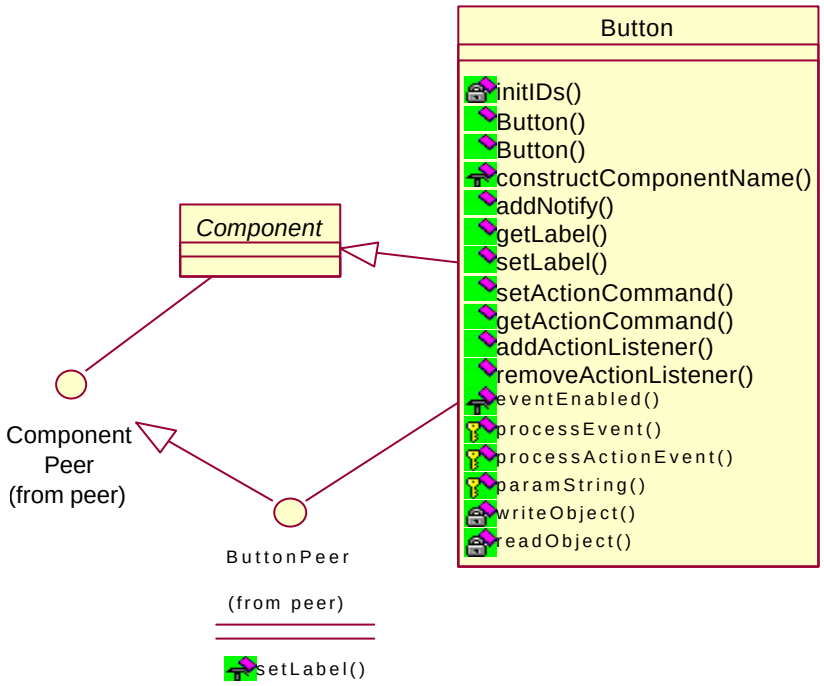
예를 들어, 데이터 억세스를 위한 일련의 추상(Abstract)클래스들을 정의하였을 때, 이를 상속 받는 XML 데이터 억세스 클래스들과 DB 데이터 억세스 클래스들의 2 가지 클래스 집합을 구현할 수 있다. 추상 클래스가 비록 여러 개이지만, 2 가지 클래스 집합에서 어느 집합을 사용하는가에 따라 클라이언트의 코드 변화 없이 기능 변경이 가능하다.

Bridge in JAVA

"자바 API 는 java.awt 패키지를 포함하고 잇다. 이 패키지는 Component 클래스를 포함한다. Component 클래스는 모든 GUI 컴포넌트들의 공용 로직을 캡슐화하는 추상 클래스이다. 콤포넌트 클래스는 Button, List, TextField 와 같은 서브 클래스들을 가지고 있으며, 이들은 운용환경에 의존적인 GUI 컴포넌트들을 캡슐화한다.

Java.awt.peer 패키지는 ComponentPeer, ButtonPeer, ListPeer, TextFieldPeer 등의 인터페이스들을 가지고 있으며, 이들 인터페이스는 컴포넌트 클래스의 하위 클래스들이 구현될 때 요구되는 운용 환경에 의존적인 메서드들이 선언되어 있다. 컴포넌트 클래스의 하위 클래스들은 이들을 구현하는 객체들을 생성하기 위해 Abstract Factory 패턴을 사용한다. Java.awt.toolkit 클래스는 Abstract Factory 의 역할을 담당하는 추상 클래스이다. 운용환경은 구현 클래스들을 생성하기 위한 Concrete Factory 클래스와 구현 객체를 제공한다." – from Pattern in JAVA

다음의 클래스 다이어그램은 자바 소스에서 추출한 것이다.



5. 그림 5 - Bridge Pattern On JAVA (Button on AWT)

위의 클래스 다이어그램에서 추상 객체인 Component 와 ComponentPeer 를 구현 객체인 Button 과 ButtonPeer 가 상속 받아 실질적인 동작을 담당한다. 여기서 주목할 만한 점은 Component와 ComponentPeer 의 관계가 Button 과 ButtonPeer 라는 상속받은 객체에서도 동일하게 유지된다는 점이다.

사용하려면?

추상(Abstract) 클래스들의 메커니즘을 미리 정의하고, 이를 상속 받아 구현하는 클래스들을 여러 개 구현한다는 것은 간단하지 않다. 일단 추상 클래스들의 레벨에서 위와 같이 동작하는 것을 생각하기에는 힘들다.

"Pattern in JAVA"책에서 나온 예제로는 센서들의 집합을 예로 들었다. 2개 회사의 센서들을 사용하는 경우를 예를 들어 설명한다. 센서는 여러 종류가 있지만 이들간의 상호 연관관계를 브리지 패턴을 이용하여 추상관계로 이끌어 내어, 센서 집합의 사용을 구현하는 회사와 독립적으로 가져갈 수 있게 하였다.

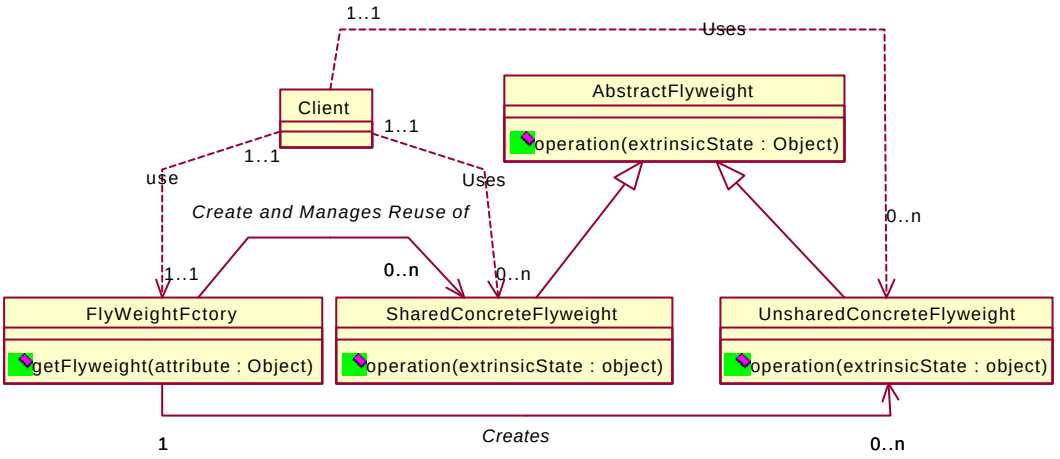
이러하듯, 브리지 패턴은 둘 이상의 객체의 연관관계가 추상화 될 필요가 있을 때 유용하다. 반면, 이런 관계는 의식적으로 구현하려고 노력하지 않으면 생각하는 것이 쉽지 않기 때문에 적용이 힘들다는 단점도 있다.

FlyWeight Pattern

플라이 웨이트 패턴

"동일한 정보를 가지고 있는 객체의 인스턴스들이 상호 교환되어 사용될 수 있다면, 플라이 웨이트 패턴은 하나의 객체로 정보를 공유하여 여러 개의 객체 인스턴스로 인한 비용 소모를 피할 수 있다." – from Patterns in JAVA

동일한 정보를 가지는 객체들을 재활용하여 객체의 수를 최소화 하자는 것이 플라이 웨이트 패턴의 개념이다. 이들 공용 객체는 공용 플라이 웨이트 풀(POOL)에 저장되며, 이를 사용하는 클라이언트 객체는 해당 객체를 객체 풀(POOL)에서 찾아 레퍼런스 만을 유지한다. 만약 적절한 객체가 존재하지 않으면 객체를 생성하고 이를 객체 풀(POOL)에 저장한다.



6. 그림 6 - FlyWeight Pattern

위의 그림에서 플라이 웨이트 객체들은 공용적으로 사용될 객체(SharedConcreteFlyweight)과 공용적으로 사용되지 아니할 객체(UnsharedConcreteFlyweight)의 2 가지의 플라이 웨이트 객체로 구현된다. 서버에서는 이들 객체를 풀에 넣고 재활용한다. 플라이 웨이트를 사용하는 클라이언트 클래스는 모든 플라이 웨이트 객체를 팩토리 클래스를 이용하여 생성한다. 생성시점부터 도일한 정보를 가지는 객체가 존재하는지의 여부를 플라이 웨이트 풀에서 확인한다.

플라이 웨이트 패턴에서 흔히 나오는 예제는 워드프로세스의 구현에서 문자 집합 부분이다. 문자 객체는 한 문자 당 1 개만이 생성되며, 이를 사용하는 문자열이나 문장에서는 문자 객체에 대한 레퍼런스만을 유지하며 새로이 생성하지 않는다.

FlyWeight in JAVA

"자바는 문자열 어휘로 스트링 객체를 표현하는데 플라이 웨이트 패턴을 사용한다. 만약 프로그램내에 동일한 문자순서로 구성된 어휘들이 존재한다면, 자바 가상 머신은 동일한 스트링 객체로 이들 어휘들을 표현한다. 스트링 클래스의 intern 메서드에서 모든 문자 어휘들을 표현하는 스트링 객체들을 관장한다." – from Patterns in JAVA

자바에서 사용하는 플라이 웨이트 패턴은 스트링 객체의 생성부분에서 나타난다. 하나의 스트링을 생성하라는 요청이 들어오면 String 객체의 intern 이라는 메서드를 통하여 스트링을 생성하는데, 이는 자바 VM 의 String Flyweight Pool 을 검색하여 동일한 스트링이 있는지를 확인한다. POOL 에 동일 스트링이 존재하면 존재하는 스트링을 그대로 반환한다. 그렇지 아니하면 POOL 에 새로운 스트링을 생성하여 저장하고 새로운 스트링을 반환한다.

String 은 매우 빈번하게 생성되는 객체이며, 생성시에 Intern 은 매번 호출 되

어야 한다. 속도의 향상을 위해서 Intern 은 내이티브(Native) 매서드로 구현되어 있다.

JVM 내부의 String Native Implementation (from Win32 JDK1.2.2 Source)

```
Hjava_lang_String *
internString(Hjava_lang_String *str)
{
    Hjava_lang_String *result;
    struct Classjava_lang_String *strObj;
    int length;
    unicode *chars;
    int index;
    string_bucket_type *bucket;
    sys_thread_t *self = sysThreadSelf();

    HEAP_LOCK(self);
    strObj = unhand(str);
    chars = unhand(strObj->value)->body + strObj->offset;
    length = strObj->count;
   index = string_hash_fun(chars, length) % HASH_TABLE_SIZE;
    for (bucket = string_hash_table[index]; bucket; bucket = bucket->next) {
        if (bucket->string == str ||
            stringEqual(bucket->string, chars, length)) {
            result = bucket->string; // 스트링 객체의 존재 여부를 HashSet에서 확인
            goto unlock;
        }
    }

    if (free_string_buckets) {
        bucket = free_string_buckets;
        free_string_buckets = free_string_buckets ->next;
        n_free_string_buckets --;
    } else {
        bucket = (string_bucket_type *)sysMalloc(sizeof(string_bucket_type));
        if (bucket == NULL) {
            result = NULL;
            goto unlock;
        }
    }

    bucket->string = str;
    bucket->next = string_hash_table[index];
    string_hash_table[index] = bucket;
```

```
    result = bucket->string;  // String 객체를 HashSet에 추가.

 unlock:
    HEAP_UNLOCK(self);
    return result;
}
```

위에 나온 소스 코드는 Win32 JDK 내부의 자바 VM 의 스트링 객체 처리 부분에서 Intern 함수를 구현한 C 언어 소스 코드이다. 위의 예제에서 보듯, 일단 새로운 스트링 생성의 요청이 들어오면 Pool 의 HashSet을 이용하여 객체가 존재하는지를 확인한다. JDK 의 소스 코드가 C++이 아닌 C로 구현된 함수들로 이루어 져 있다. 한가지 명심할 것은 패턴을 어떤 언어로 구현하는가는 중요한 것이 아니다. 중요한 것은 패턴의 개념을 명백히 이해하고, 그 개념을 도입하여 프로그램을 만드는 일이다.

사용하려면?

자바의 String 객체는 플라이 웨이트 패턴의 매우 적절한 예제이다. 플라이 웨이트 패턴을 사용하기 위해서는 반복적으로 빈번하게 사용되면서도 공유될 수 있는 객체들이 존재해야 한다. 워드 프로세서에서 스트링 처리를 위한 문자 객체나, HTML 구문 처리를 위한 ELEMENT 객체 등은 플라이 웨이트 패턴이 사용될 수 있는 장소이다.

많은 경우에 있어서 플라이 웨이트 패턴이 이뮤터블 패턴과 함께 적절히 사용된다는 점을 감안하면 프로그램 작성시에 플라이 웨이트 패턴을 적용할 만한 부분을 찾기는 쉽지 않은 일이다.

기능중심의 패턴 사용 설명

다음은 JAVA 의 패키지와 기능 별로 디자인 패턴이 어떻게 사용되고 있는가를 소개한다. 소개하는 부분은 다음의 3 가지 부분이다.

- IO Package 에서 IO 처리
- RMI 에서 Proxy 패턴의 사용
- Collection 객체에서의 Single Threaded Execution

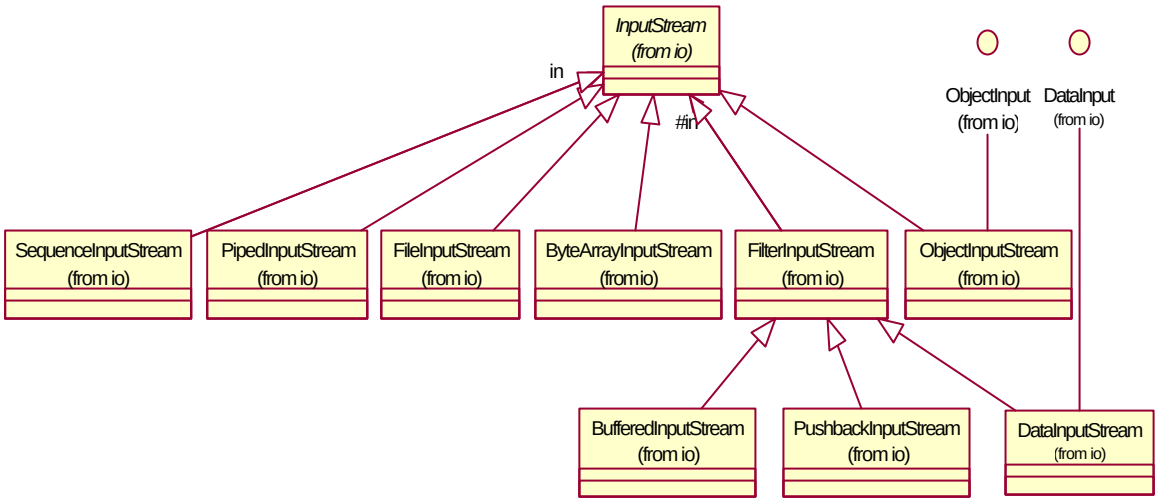
각 부분에서의 어떤 기능이 패턴을 어떠한 방식으로 사용하고 있는 가를 설명한다.

IO 패키지 (java.io)

IO 패키지는 자바에서 가장 많이 사용되는 동시에 사용법이 간단하여 가장 쉽게 이해할 수 있는 패키지중의 하나이다. 심지어 "Hello World"를 화면에 처음 출력해 보는 초보자에게도 필요한 패키지이다. 이처럼 친숙한 IO 패키지내에 숨어 있는 디자인 패턴을 점검해 보기로 하자.

이 패키지는 기둥이 되는 네가지의 추상 클래스(Abstract Class)가 존재한다. 즉 InputStream 과 Reader, OutputStream 과 Writer 가 그것이다. InputStream 과 OutputStream 은 바이트를 처리의 기본 요소로 하며 Reader 와 Writer 는 캐릭터를 기본 요소로 한다. 좀더 자세히 살펴 보기 위해 그림 7 의 InputStream 클래스 계층도를 보자. 이 클래스 계층도에서는 deprecation

된 메소드인 StringBufferInputStream, LineNumberInputStream 은 제외하였다.



7. 그림 7 - InputStream Class Diagram

최상위에는 InputStream 가 존재하는데 이 InputStream 은 바로 Template Method 패턴을 따르는 추상 클래스이다.

박스 1 Template Method 패턴

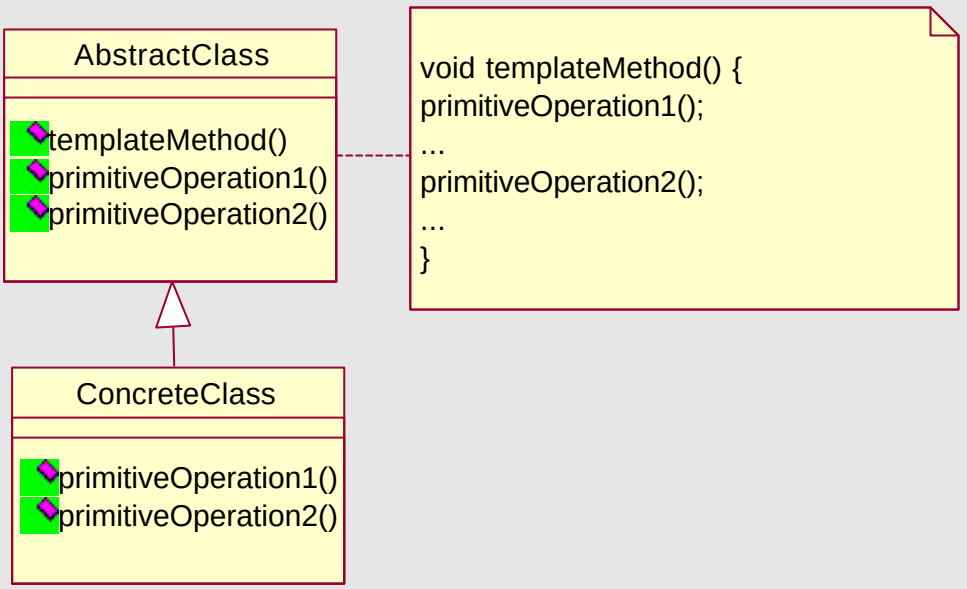
클래스들을 작성해 나가다 보면 공통적인 행위(Operation, Method)를 갖는 클래스 집합에 도달하는 경우가 많다. 능력이 있고 경험많은 개발자들은 디자인 단계에서도 이 클래스집합을 잡아 낼 수도 있으나 대부분은 구현 단계에서 드러나게 되어 있다. 왜냐면 디자인 단계에서 알 수 있는 클래스들의 모습은 스펙에서 유추된 것이기 때문에 구현단계의 클래스의 모습과는 동떨어져 있는 경우가 많다. 이러한 공통적인 행위를 갖는 집합은 보통 하나의 추상 클래스에 공통적인 행위들을 뽑아 내는 기법을 쓴다. 여기까지가 우리가 일반적으로 알고 있는 상속(Inheritance)이다.

Template Method 패턴에서의 추상 클래스내의 메소드는 Primitive Method(아래 그림에서의 primitiveOperation1, primitiveOperation2)와 Template Method(아래 그림에서의 templateMethod)로 나뉜다. Primitive Method 는 각 서브 클래스들마다 다른 행위들을 하는 메소드들을 나타내며 abstract 로 선언된다. Template Method 는 이러한 Primitive Method 들을 내부적으로 사용하여 알고리즘이나 절차들을 기술하는데에 사용된다.

정리해 보면 변하지 않는 알고리즘이나 절차들은 Template Method 내에서 기술하고 이때 서브 클래스마다 상이하게 동작할 수 있는 Primitve Method 들을 사용한다. 이 Template Method 패턴을 사용하게 되면 물론 상속의 이점인 코드 중복을 막는다는 면도 가지게 되지만 가장 큰 이점은 새로운 행위의 변화가 생겼을 때 Primitive Method 를 갖는 구현 클래스만

추가하면 된다는 점이다. 기존의 절차형 언어(Procedural Language)에서는 새로운 행위의 추가 때문에 알고리즘자체의 변화(분기문의 추가라든지, 배열값의 추가등)를 가했어야 되었던 것에 비하면 유지보수가 매우 쉬워짐을 알 수 있을 것이다.

[그림] Template Method 패턴 클래스 다이어그램



추상 클래스인 InputStream에서 abstract로 선언되어진 메소드는 단 하나 read 메소드이다.

```
public abstract class InputStream {
...
        public abstract int read() throws IOException;
...
}
```

예를 들어 skip() 메소드와 read(byte[], int, int) 메소드는 내부적으로 read() 메소드를 사용하여 구현되어 있다. 나머지 메소드들은 Template Method에 속하며 read 메소드는 서브 클래스에서 구현해야 될 Primitive Method라고 할 수 있다. 이처럼 Template Method 패턴을 사용하게 되면 알고리즘과 같은 공통적인 절차를 독립화시킬 수 있게 되는 장점이 생긴다.

이제 하위 클래스들을 점검해 보도록 하자. InputStream을 바로 상속받고 있는 클래스들은 SequenceInputStream, PipedInputStream, FileInputStream, ByteArrayInputStream, FilterInputStream, ObjectInputStream이 있다. 이중 FilterInputStream은 추상 클래스로 존재하며 다시 FilterInputStream을 상속받는 서브 클래스로는 BufferedInputStream, PushbackInputStream,

DataInputStream 이 있다.

이때 잠깐 살펴 볼 것이 왜 InputStream 을 직접 상속받지 않고 FilterInputStream 을 상속받는 클래스들이 존재하는가에 대한 것이다. 그 해답은 FilterInputStream 의 구조를 들여다 보면 어느정도는 풀린다. InputStream 은 추상 클래스인 반면 FilterInputStream 은 일반 클래스라는 차이점이 우선 눈에 띈다. 그러나 더 중요한 차이는 FilterInputStream 의 생성자에 있다. FilterInputStream 의 유일한 생성자는 다음과 같이 정의된다.

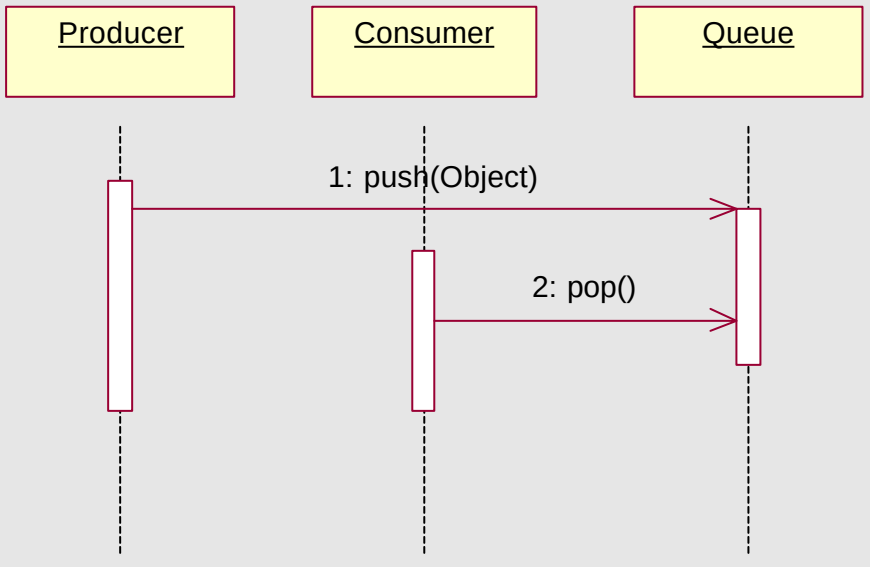
```
public FilterInputStream(InputStream in)
```

즉 FilterInputStream 에서는 입력받은 InputStream 을 사용해서 모든 행위가 이루어진다. 일종의 delegation 이라고 보면 된다. 따라서 기존에 이미 있는 InputStream 을 기반으로 행위가 벌어지기 때문에 Filter 를 앞에 붙여 놓은 것이라고 할 수 있다. FilterInputStream 을 상속받는 세개의 클래스 모두 독립된 데이터 소스를 만들기 보다는 이미 만들어져 있는 데이터 소스를 변형하는 역할을 하고 있음을 확인해 보기 바란다.

다시 패턴 이야기로 가보자. PipedInputStream 은 PipedOutputStream 과 더불어 Producer-Consumer 패턴을 이룬다. Producer-Consumer 패턴은 Producer 와 Consumer, 그리고 Queue 라고 불리는 버퍼 이렇게 세가지 요소로 구성되어 있는데, 자바 구현에서는 Queue 와 Consumer 의 역할을 PipedInputStream 이 수행하고, Producer 역할은 PipedOutputStream 이 수행하게 된다.

박스 2 Producer-Consumer 패턴

[그림 Producer Consumer Sequence Diagram]



Producer-Consumer 패턴에서 Producer 는 데이터의 생성(a data source)을 담당한다. 생성된 데이터는 Consumer 에 의해 소비된다(a data sink). Queue 는 중간의 버퍼역할을 담당하게 된다. Producer 는 데이터를 Queue 에 push()한후 다음 작업을 계속하게 된다. Consumer 는 Queue 에서 pop()을 시도하는데 만일 데이터가 없다면 Producer 에 의해 생성될때까지 대기하게 된다.
Queue 에서는 보통 버퍼 크기가 한정되어 있고 이 버퍼 크기를 넘어선 데이터가 들어오게 되면 에러가 나게 된다. 모든 데이터를 주고 받는 모듈에는 이 Producer-Consumer 패턴이 사용된다.

이 패턴을 사용할 때 염두에 두어야 할 것은 Consumer 가 blocking I/O 로 동작한다는 점이다. 즉 pop()이 호출된 후 이용가능한 데이터가 없다면 데이터가 생길때까지 대기하게 된다. 이러한 대기 상태를 막기 위해서는 Queue 에 이용가능한 데이터가 있는지 검사해 보는 size()나 available()같은 메소드를 만들어 pop()을 호출하기 전에 조건 검사를 하도록 한다.

PipedOutputStream 은 내부적으로 PipedInputStream 을 지니고 있어서 write()에 의해 데이터가 도착할때마다 PipedInputStream 의 receive()를 호출하여 버퍼에 데이터를 저장하게 된다. PipedInputStream 의 read() 메소드에서는 다음의 코드에서 볼 수 있듯이 데이터의 유무에 따라 대기 여부를 결정한다.

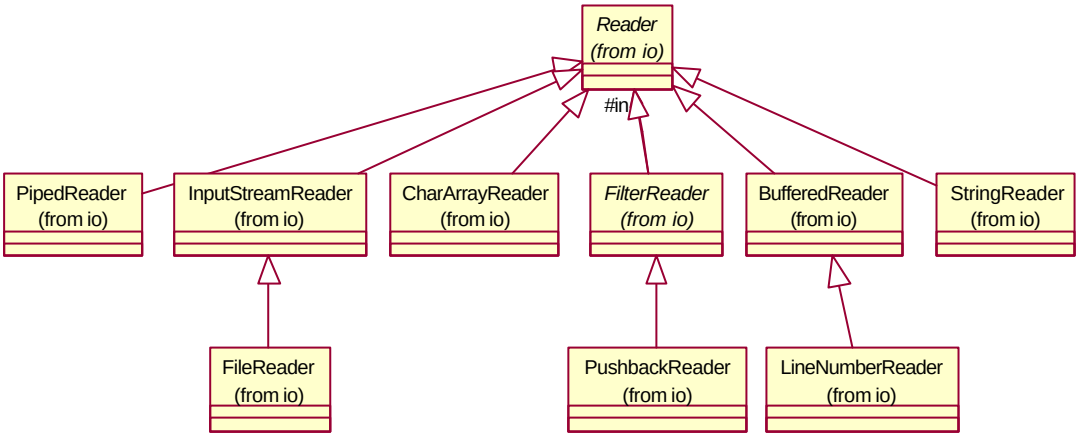
```
public synchronized int read()  throws IOException {
        ..............
        while (in < 0) {
            // no data
            ....
            /* might be a writer waiting */
            notifyAll();
            try {
                wait(1000);
            } catch (InterruptedException ex) {
                    throw new java.io.InterruptedIOException();
            }
        }
        int ret = buffer[out++] & 0xFF;
        ..............
        return ret;
}
```

Reader 클래스의 계층도는 그림처럼 InputStream 의 계층도와 아주 유사하다. 차이점을 정리해 보면 다음과 같다.

- Character 중심인 Reader 계열에서는 DataInputStream 과 ObjectInputStream 에 해당하는 클래스가 없다.
- Byte 스트림에서 Character Encoding 을 해주는 InputStreamReader 가 추가된다. 이에 따라 FileReader 는 InputStreamReader 를 상속하게 된다.

- BufferedReader 가 Reader 를 바로 상속한다.

OutputStream 과 Writer 의 계층도는 InputStream 과 Reader 와 유사하므로 이글에서는 따로 다루지 않겠다.



8. 그림 8 - Reader Class Diagram

가끔 자바를 처음 배우기 시작한 프로그래머들에게서 다음과 같은 코드를 본다.

```
public void doOperation(FileWriter writer)
{
        String data = ……; // data allocation
        writer.write(data);
}
```

아무 이상이 없는가? 물론 동작에는 문제가 없다. 그러나 활용성이 떨어진다. 바로 FileWriter 를 사용했기 때문이다. doOperation 내에서 사용되는 writer 가 만일 FileWriter 의 특정한 메소드를 사용하지 않는다면 doOperation 의 signature 는 *doOperation(Writer writer)* 로 바꾸어주는 것이 좋다. 그렇게 되면 FileWriter 뿐만 아니라 다른 Writer 를 상속받은 클래스들을 가지고도 doOperation 을 호출할 수 있게 되기 때문이다.
한걸음 더 나아가 메소드를 디자인할 때 가급적 클래스 계층도의 상위 클래스 즉 Super 클래스(혹은 인터페이스)를 패러미터로 사용하게 되면 그 메소드는 광범위하게 쓰일 수 있게 될 것이다. 이것이 바로 객체 지향 언어에서의 다형성(Polymorphism)의 힘이다.

RMI 와 Proxy 패턴

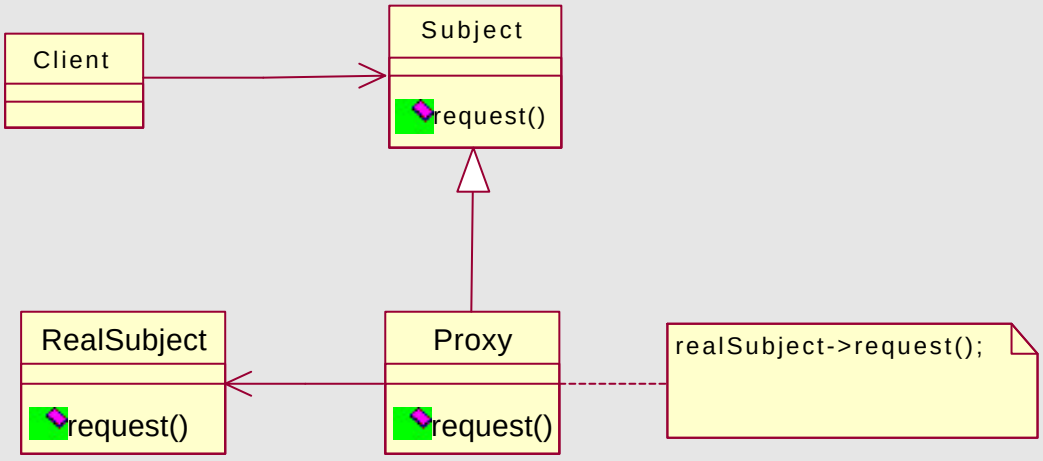
요즘은 RMI 를 사용해서 직접 프로그래밍하는 예는 거의 없다. 그것은 Sun 이

제공하는 RMI 런타임의 Scalability 가 떨어져서이기도 하지만 EJB 가 서버쪽의 자바 컴포넌트 표준으로 거의 굳어진 것이 더 큰 이유가 될 것이다. 그렇다면 RMI는 완전히 사라졌을까? 엔터프라이즈 레벨의 컴포넌트 표준인 EJB 나 임베디드 시스템 레벨의 표준인 JINI 모두 기본 사상은 RMI에서 출발했다고 할 수 있다. EJB 내부에서 사용되는 프로토콜중의 하나는 JRMP(RMI의 트랜스포트 프로토콜로써 EJB 가 지원하는 다른 하나의 프로토콜은 IIOP 임)이며 EJB 의 클래스 설계는 홈 인터페이스를 제외하면 RMI와 유사하다. JINI는 내부적으로 RMI 매커니즘을 사용하여 구현되어 있다.

RMI에서는 인터페이스에서부터 스텁 클래스(stub class)와 스켈리턴 클래스(skeleton class)가 rmic 에 의해 자동 생성된다. 마치 EJB 에서 ejbc(Weblogic EJB Container 의 경우)라고 불리는 내부 구현 클래스 자동 생성툴이 있는 것처럼 말이다.

이제 HelloWorld 라는 RMI 객체를 예를 들어 보자. HelloWorld RMI 객체의 어느 부분이 프록시 패턴의 어느 부분에 해당하는지 확인해 보도록 한다.

박스 3 Proxy패턴

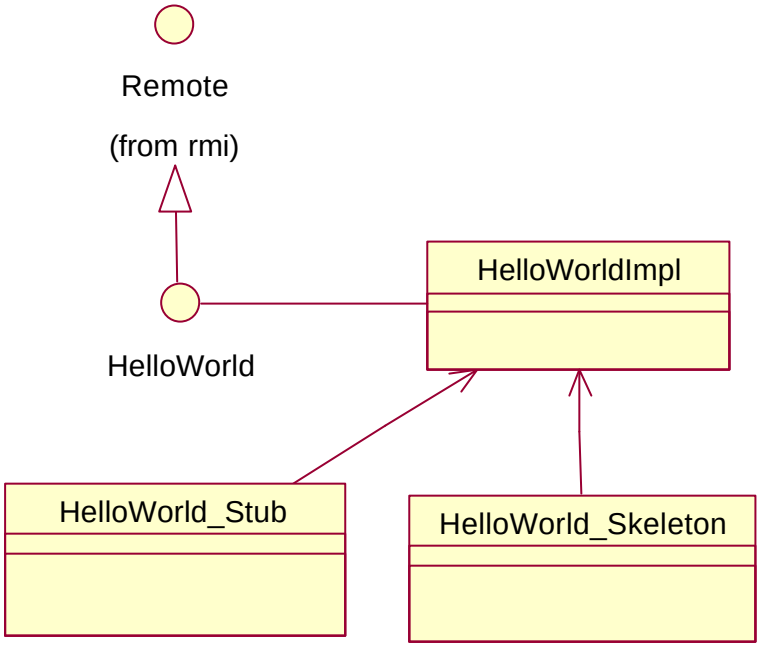


프록시(Proxy) 패턴에서 클라이언트는 직접 구현 클래스 즉 위의 그림에서는 RealSubject를 접근하지 않는다. 대신 프록시라고 불리는 대리자를 통해서 접근하게 된다. 위의 다이어그램에서는 프록시도 Subject라는 인터페이스를 통해서 구현되었다.

프록시 패턴은 굉장히 광범위하게 적용되고 있는데 GoF 책에서는 모두 네가지로 프록시 패턴을 분류하고 있어서 여기에 소개한다.

- 원격 프록시(remote proxy) – 자바 RMI에서처럼 다른 어드레스 스페이스를 지닌 객체에 대한 로컬 대리자를 제공한다.
- 가상 프록시(virtual proxy) – 실제 객체의 생성 비용이 높은 경우 가상 프록시를 먼저 생성한 후 실제 객체가 사용될 때까지 실제 객체의 생성을 늦출 때 사용한다.

- 보호 프록시(protection proxy) – 실제 객체에대한 직접적인 접근을 제한할 때 보호 프록시를 통하도록 한다.
- 스마트 레퍼런스 (smart reference) – reference 를 대치하는 것으로 부가적인 작업을 한다. 예를 들어 ATL 에서의 스마트 레퍼런스는 스스로에 대한 참조 카운트를 유지하며 0 가 되었을 때 자동으로 free 가 된다.



9. 그림 9 - HelloWorld RMI Object Class Diagram

위의 그림에서처럼 HelloWorld 라는 RMI 객체를 만들었다고 치자. 클라이언트 입장에서는 HelloWorld 하나만을 사용하게 된다. 그러나 내부적으로는 HelloWorld_Stub 이라는 스텁이 구동되는 것이고 다시 이 스텁은 원격지의 HelloWorld_Skeleton 을 호출하게 된다. 다시 HelloWorld_Skeleton 은 HelloWorldImpl 을 사용하여 비로소 클라이언트가 요청한 일을 수행하게 된다. 그렇다면 프록시는 무엇이었을까? 바로 rmic 에 의해 자동생성된 HelloWorld_Stub 인 것이다. 프록시 패턴과 HelloWorld 예제와의 매핑 관계를 정리해 보면 다음과 같다.

- Subject <-> HelloWorld
- Proxy <-> HelloWorld_Stub
- RealSubject <-> HelloWorld_Skeleton, HelloWorldImpl

그리고 RMI 는 서로 상이한 어드레스 스페이스의 자바 객체를 호출할 수 있게 해 주므로 프록시 패턴중에서 원격 프록시에 해당한다. 프록시 패턴이라는 개

념은 분산 환경에서 두고 두고 쓰이므로 이번 기회에 패턴을 잘 익혀 두면 활용할 기회가 많이 있을 것으로 생각된다.

Collection 과 Single Threaded Execution 패턴의 관계

Collection Framework 을 정리하면 다음의 테이블로 나타낼 수 있다. 인터페이스와 구현이 분리되어 유연하게 디자인되어 있음을 확인해 볼 수 있다. Set 과 List 인터페이스는 Collection 인터페이스를 상속하는데 반해 Map 인터페이스는 상속하는 인터페이스가 존재하지 않음에 유의한다.[1]

		구현			
		Hash Table	**Resizable Array**	**Balanced Tree**	**Linked List**
인터페이스	**Set**	HashSet		TreeSet	
	List		ArrayList		LinkedList
	Map	HashMap		TreeMap	

JDK 1.0 에서부터 존재하던 Hashtable 과 Vector 는 단일 쓰레드만이 접근할 수 있는 구조로 되어 있었다. 이처럼 단일 쓰레드만의 접근을 허용하는 것을 Single Threaded Execution 패턴(이하 줄여서 STE 패턴이라 함)이라고 한다. 자바에서 STE 패턴은 synchronized 메소드나 synchronized 블록을 사용해서 구현되어진다.

Collection Framework 에 속한 Set, List, Map 은 모두 STE 패턴을 따르도록 디자인되어 있지 않으므로 STE 를 따르도록 하려면 다음처럼 해줘야 한다.

```
Collection c = Collections.synchronizedCollection(myCollection);
   ...
 synchronized(c) {
    Iterator i = c.iterator();
    while (i.hasNext())
      foo(i.next());
 }
```

List, Set, Map 인터페이스를 명시하고 싶은 경우 각각 synchronizedList(), synchronizedSet(), synchronizedMap()을 사용도록 한다.

[1] Collection Framework 에서 Map 은 keys, values, pairs 세가지 Collection 의 View라고 해석되기 때문이다.

이번호의 마무리.

지금까지 자바 클래스와 자바 VM 에서 사용된 패턴들의 예제를 보아왔다. 분명 자바에서도 많은 부분이 디자인 패턴을 이용하여 구현되었다. 이는 클래스에 기반 한다는 자바의 속성때문이기도 하다.

이 글은 2 명의 필자가 함께 쓰고 있다. 이번 호도 2 명의 필자가 함께 쓰고 있지만 이중에서 이번호의 마무리 글을 쓰고 있는 "나"는 현재 이모션의 개발 실장으로 있다. 잠시 나의 입장에서 마무리 글을 써 보려고 한다.

이모션의 개발 실장인 나는 최근에 개발부의 연구원 2 명으로부터 2 가지 흥미 있는 질문을 받았다. 우리나라에서는 MS-SQL 이나 ORACLE, Windows 2000 과 같은 훌륭한 소프트 웨어를 만들수가 없느냐는 것과 앞으로 계속 프로그래머의 길을 가려면 무엇을 어떻게 공부해야 하는가에 관한 질문이었다.

이들 질문은 새삼스러운 것은 아니었다. 글을 쓰는 나도 진작부터 그것을 고민하고 있었다. 그러나 다시 한번 새삼스럽게 젊은 연구원 2 명으로부터 그러한 질문을 받게 되니 감회가 새로왔다. 그리고 그들에게 나 자신이 생각해왔던 바를 이야기 해 주었다.

"나도 그 문제를 고민해 왔습니다. 프로그래머의 길을 10 년이고 20 년이고 가고 싶다고 생각하는 것은 저도 동일합니다. 한국에서 멋진 소프트웨어를 만들고 싶기도 하고.

그것을 아십니까? 객체 지향의 개념이 나온 지 이미 20 년이 다 되었다는 사실을 말입니다. 그러나 아직 프로그래머 중에는 객체 지향의 개념을 가지고 프로그램을 만드는 사람은 그다지 많지 않습니다. COM+, EJB 의 시대를 말하지만 정작 객체 지향의 개념을 이해하고 개념에 따라 프로그램을 구성하는 사람은 많지 않았습니다. 저는 수십 명의 프로그래머들과 이들이 만든 많은 코드들을 보아왔습니다.

다른 프로그래밍 선진국에는 많은 프로그래머들이 있습니다. 그들은 적어도 우리보다는 안정성 있고, 신뢰성 있는 제품을 만들어 내는 것 같습니다. 나사의 우주 정거장 건설 프로젝트의 프로그램 소스 라인이 몇 라인일까요. 무려 10 억 라인이나 된다고 들었습니다. 지금은 더욱 많을 것입니다. 이러한 프로젝트가 오차 없이 동작하려면 얼마나 많은 노력과 고민을 하였을까요. 단순히 잘짜는 것만으로는 부족합니다.

체계적으로 이론을 공부하고 지속적인 관심을 가져야 합니다. 그래서 스스로가 확신을 가질 수 있는 코드를 만들어야 합니다. 자신이 구현한 코드에 만든 자신이 자신감을 가지고 있어야 합니다. 코드를 만들 자신감을 가지려면, 코드가 동작하는 원리를 명백하게 이해하고 코드를 작성해야 합니다. 개념이 확립되지 않은 상태에서 주먹 구구식으로 코드를 구현하면 언젠가는 문제를 일으킵니다."

위의 대답을 하여 주면서 디자인 패턴의 이야기를 하고 싶었다. 개념의 설립과 동작 원리의 이해를 위한 최상의 방법중의 하나는 디자인 패턴을 이용하는 것이라고 말하고 싶었으나, 아직 그런 말을 할 단계는 아니어서 위의 대답으로 마무리 지었다. 패턴 도입 효과의 좀더 객관적인 자료를 준비하고 패턴을 언급하여야겠다는 생각에서 였다. 이제, 자바 VM 과 기반 클래스에서 디자인 패턴이 사용된 사례를 이번 호에 기고하여, 나에게 질문을 한 이들에게 실질적 증거에 근거한 패턴의 유용성을 설명할 수 있게 되어 조금은 흐뭇한 느낌이다.

필자의 생각은 이렇다. 프로그램을 수학에 비교하자만, 디자인 패턴은 인수분해나 사인/코사인 함수에 해당한다고. 자신이 좀더 수준 있는 개발자가 되기 위한 단계로 디자인 패턴은 반드시 익혀야 하며, 이를 익힌 프로그래머는 익히지 않은 프로그래머 보다 월등한 프로그램 도구를 가지는 것이다. 개발이라는 전쟁터에 나가는 일반 프로그래머가 권총을 가지고 나갈 때, 패

턴을 익힌 프로그래머는 자동소총을 들고 가는 것과 같다. JAVA에서의 예를 보라. 정말 많은 부분에 패턴이 사용되고 있다. 이번 호에 소개한 부분은 그 일부에 지나지 않으며 다음 호에서도 자바에서의 디자인 패턴 사용을 계속 다룰 것이다. 비단 자바 뿐만이 아니다, 샌프란시스코, COM 등의 많은 부분에 패턴이 사용되고 있다. 이들은 패턴이라는 무기를 들고 전쟁에서 지금까지 승리하여 왔던 것이다.

디자인 패턴의 적용
– java API 에서의 디자인패턴 12 월호

이번호에 앞서

지난 호까지 디자인 패턴에 관해서 여러 이야기들을 하여 왔다.

이번 호는 마지막으로 자바에서의 디자인 패턴의 적용을 정리하고 4 회에 걸친 디자인 패턴에 관련된 맺음말로 글을 마무리 하려 한다.

이번 호에서 소개하는 부분은 2 가지이다. 하나는 자바 스페이스에서의 분산 이벤트 모델의 디자인 패턴이며 다른 하나는 커넥터 아키텍쳐에서의 디자인 패턴이다. 전자는 분산 객체 환경인 자바 스페이스에서 나타나는 이벤트 모델 내부에서 사용되는 디자인 패턴에 관련된 설명이며, 후자는 커넥터라는 이름이 의미하는 것처럼 임의의 EIS(Enterpirse Information System)에 대한 접근을 일반화하시키자는 데에 그 목적이 있다.

커넥터와 자바스페이스 모두 독자들에게 생소한 기술인 관계로 이해에 다소 어려움이 있더라도 양해를 바란다.

다자인 패턴 IN JAVASPACE

자바 스페이스는 JINI 의 일 부분으로 분산 객체가 존재하는 공간이다. 자바 스페이스에는 객체들이 저장되고 변경되기 위한 몇 가지 인터페이스를 제공한다. 자바 스페이스의 개념은 매우 간단하다.

- 자바 스페이스라 불리우는 공간이 있다.
- 객체는 공간에 저장될 수 있다.
- 공간에 존재하는 객체를 사용하기 위한 어플리케이션은 객체를 공간에서 꺼내와야 한다.
- 어플리케이션이 사용 후 변경된 객체를 공간에 반영하기 위해서는 객체를 공간에 저장하여야 한다.
- 트랜잭션은 객체를 공간에 저장하는 시점에서 발생한다.

로 요약될 수 있다.

원격지에서 동떨어진 어플리케이션들은 자바 스페이스를 주시하고, 객체들이 추가되거나 삭제되는 등의 액션에 따라 함께 반응한다.

지금까지의 연재 기사를 읽어온 독자분들 중에서 자바 스페이스에서 디자인 패턴의 개념이 사용되었을까라는 질문에 "NO"라고 답할 분은 아마 존재하지 않으실 것이다.

그러하다. 자바 스페이스에서도 당연히 디자인 패턴이 존재한다. 자바 스페이스는 분산 컴퓨팅 환경에서 객체들이 상호 연동되는 연동의 장이다.

이러한 분산 객체 환경에서 디자인 패턴이 어떻게 사용될까? 디자인 패턴을 사용할 수 있는 개념을 적용할 수 있는 분야에는 분산 객체 환경이 포함된다.

디자인 패턴이 진정으로 중요한 이유중의 하나는 새로운 기술들인 분산 객체 환경에서도 디자인 패턴이 여지 없이 사용되고 있다는 사실 그 자체이다. 많은 사람들이 하나의 기술을 배우면 2~3 년 내에 사장되어 버리고 새로운 기술을 배울 수 없는 것이 아닌가? 하는 걱정을 내심 가지고 있다. C 언어는 이미 사장 되어 가는 기술이며, PHP,CGI 등의 기술도 이미 JSP 나 ASP 등의 신기술에 맞물려 조금씩 사라지고 있다.

그러나, 원천적인 기술은 변하지 않는다. 디자인 패턴은 이미 94 년도에 최초로 널리 알려진 기법이지만 지금까지도 그다지 변하지 않는 기술이며, 앞으로도 핵심적인 기술로 사용될 것이

다.

자바 스페이스에서의 이벤트 모델에 있어 디자인 패턴이 어떻게 적용되어 왔는가를 발견하는 과정을 정리하여 다음에 기술한다. 필자가 자바 스페이스를 공부하면서 이중에서 디자인 패턴을 발견하고 시스템을 더욱 이해하게 되는 과정을 상세하게 기술하여, 디자인 패턴이 프로그램을 이해하는 측면에서 어떻게 활용되는가를 설명한다.

JAVA SPACE & Distributed Event Model

자바 스페이스는 분산 객체에 중점적으로 맞추어져 있다는 사실을 명심하라.
분산 환경에서는 강하게 연결된 (Tightly Coupled) 커뮤니케이션이 어렵다.
분산 환경에서 발생하는 주요한 이슈중의 하나는 분산 이벤트 모델이다. 하나의 객체에서 발생하는 이벤트를 어떻게 효과적으로 주고 받을 수 있는 환경을 만들 수 있는가?
자바의 최신 기술 중에서 자바 스페이스는 매우 재미있는 분산 이벤트 모델 부분들을 포함하고 있다.

JAVA 에서의 분산 이벤트 모델의 이해의 시작.

자바 스페이스의 개념 설명에 등장하는 "자바 스페이스는 이벤트들이 이벤트 핸들러에게 다이나믹하게 전달될 수 있는 장을 마련한다."라는 어휘에서 필자는 다음의 몇가지 항목들을 생각할 수 있었다.

1. 이벤트를 처리하는 객체는 미리 정의되어 있는 것이 아니라, 런타임 시점에서 다이나믹하게 정의된다. → Dynamic Linkage Pattern

2. 이벤트를 처리하는 객체는 이미 추상 클래스 혹은 인터페이스로 운용 방법이 지정되어 있다. → Adaptor Pattern

이러한 개념은 디자인 패턴에서 나타나는 개념으로 프로그램의 경험이 어느 수준이상 되는 개발자라면 어느 정도 짐작이 갈 부분들이다.

필자는 여기서 "Dynamic Linkage Pattern 과 Adaptor Pattern 의 개념으로 이벤트를 처리하는 과정을 이해하자"라고 어느 정도 예상하고 있었다.[Dynamic Linkage Pattern 과 Adaptor Pattern 에 관한 상세한 내용은 박스기사를 참고하기 바란다.]

이벤트 시스템의 개념을 대략적으로 나마 파악하고 시스템을 이해하기 시작하는 과정은 프로그램의 구조에 대한 개념이 전혀 없는 상태에서 이해하기 시작하는 것보다 시스템 전반적인 상황을 예측하고 이해를 시작하는 과정이 2 배 이상의 수월한 작업이다.

분산 이벤트 모델과 디자인 패턴

자바 스페이스에서의 이벤트 모델은 색다른 구조를 가지고 있다. 자바 스페이스의 분산 이벤트 모델에서 이벤트는 자바 스페이스 내부에서 발생한다. 발생된 이벤트는 분산 객체에 의해서 처리된다.

이벤트의 개념을 간단히 이야기 하면, 특정 사건에 의해서 이벤트가 발생하고

발생된 이벤트는 사전에 등록된 이벤트 리스너에 의해서 핸들링 된다는 개념이다.
자바 스페이스의 이벤트 모델에서의 다음의 순서로 발생한다.

1. 이벤트 리스너 생성
2. JavaSpace 의 Notify 메소드를 이용한 이벤트 종류와 핸들링하는 이벤트 리스너 등록
3. JavaSpace 에 이벤트 종류로 등록된 객체가 저장되어 이벤트가 발생
4. 발생된 이벤트는 등록된 이벤트 리스너에 의해서 처리된다.

의 4 가지 순서로 표현된다.

여기서 한가지 눈 여겨서 볼 것은 이벤트라는 것이 별도로 지정되는 것이 아니라는 점이다. 이벤트는 외부의 어플리케이션이 명시적으로 지정하여 발생시키는 것이 아니며, 이벤트 객체를 생성하여 던지는 것도 아니다. 오직 자바 스페이스에 객체가 생성되어 저장(Write)되는 순간 발생한다. 만약 자바 스페이스에 저장되는 객체가 이벤트로 등록이 된 객체라면, 해당 이벤트가 내부적으로 발생하게 되며, 이를 처리하는 이벤트 처리기가 이벤트의 처리를 담당할 것이다.

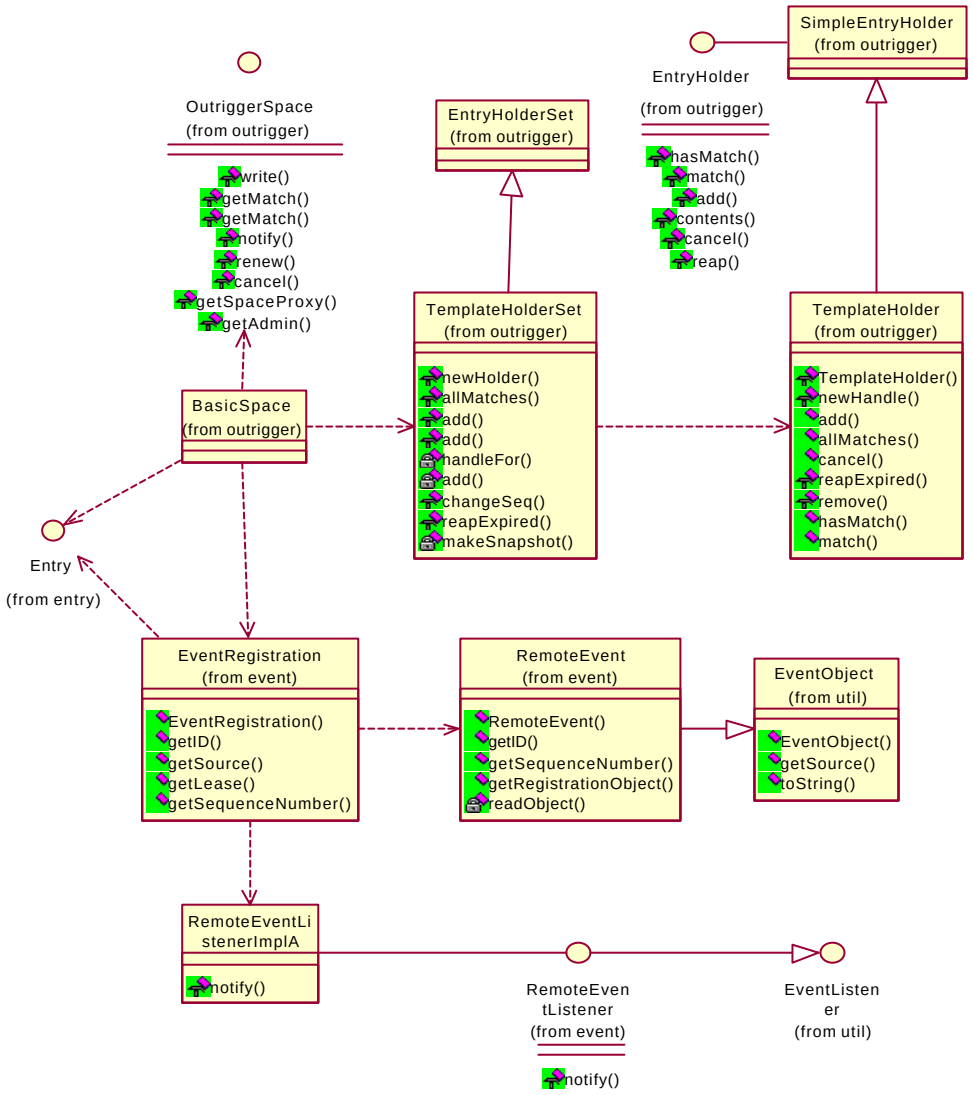
분산 이벤트 모델에서 이들을 구현하기 위해서 자바 스페이스들은 다음의 구성 요소들을 제공한다.

Event Listener 생성에서는 미리 정의된 이벤트 리스너의 인터페이스에 맞추어 이벤트를 정의한다. 이벤트의 등록 시점에서는 새로이 생성할 이벤트의 인스턴스를 임의로 생성하여 자바 스페이스에 리스너와 함께 등록한다. 이 시점에서 해당 이벤트 클래스의 이벤트 리스너는 등록된 리스너로 지정된다.

분산 이벤트 모델 & Dynamic Linkage Pattern

자바 스페이스 뿐만이 아니라 다른 대부분의 이벤트 모델에서는 동적 연결 패턴이 흔히 사용된다. 이벤트의 종류와 이를 처리하기 위한 이벤트 리스너는 정적인 연결 관계로 정의될 수 없기 때문에 이를 동적으로 연결하기 위한 방안으로 동적 연결 패턴을 사용한다.

이를 이용한 이벤트 모델은 이벤트의 종류와 이를 처리하기 위한 이벤트 리스너의 연관 관계가 해쉬테이블이나 Mapping 테이블의 형태로 저장되며, 이벤트가 발생하는 시점에서 리스너의 정보를 테이블에서 찾아 리스너를 수행시키는 형태로 동작한다.



10. 그림 10 – 이벤트 등록 관계 클래스 다이어그램

위의 클래스 다이어그램은 이벤트 등록에 관계된 여러 클래스들의 연관 관계를 보여준다. 이벤트 들은 TemplateHolderSet이라는 집합에 등록된다. 이벤트들은 모두 자바 스페이스에 저장되는 객체들의 종류인 Entry 객체를 상속받는다. TemplateHolderSet에서는 이들 Entry 객체의 종류를 저장한다.

```
public EventRegistration notify (EntryRep tmpl, Transaction tr,
RemoteEventListener listener, long leaseTime, MarshalledObject
handback) throws TransactionException, RemoteException {
        ---
}
```

이벤트의 등록은 자바스페이스의 Notify 메소드를 이용하여 처리된다. 여기서 Notify 는 이벤트가 발생되었을 때 호출되는 함수가 아님을 명심해야 한다. 자바 스페이스의 Notify 메소드는 분명 이벤트 종류와 이벤트 리스너를 등록하기 위한 것으로 이벤트 리스너와 이벤트 객체를 파라메터로 포함하고 있다.

트랜잭션과 마샬링에 관한 설명은 이번 호의 범위를 벗어난 것으로 생략한다.

```
/**
 * Private implementation of add for the handle and chit pair.
 */
private void a d d (TemplateHandle handle, NotifyChit chit) {
        EntryRep tmpl = handle.rep();
        if (tmpl.id() == tmpl.ID_UNSET) {
            synchronized (this) {                    // protect this id field
                  if (tmpl.id() == tmpl.ID_UNSET)          // in case someone
beat us to it
                        tmpl.id(BasicSpace.nextID());
            }
        }
        // Now that the chit is all set up we can add it to our tables
        idMap.put(chit.getCookie(), chit); ← 이벤트 처리기는 이벤트 객체의
종류와 함께 Chit( 증명서) 라는 형태로 가공되어 ID Mapping Hashtable 에 저
장된다.
        handle.add(chit);
}
```

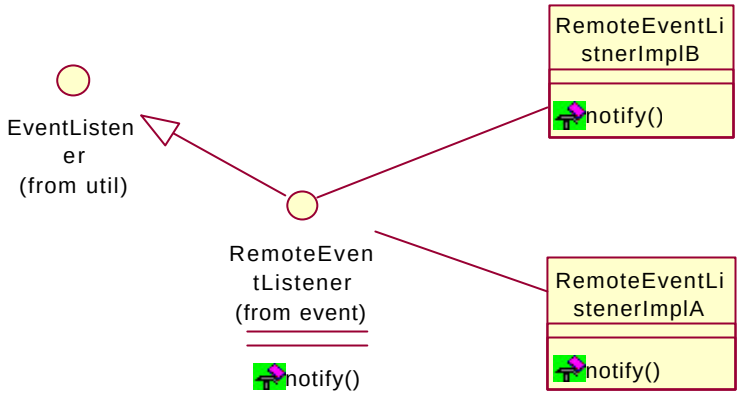
위의 예제는 Sun 의 JavaSpace 의 소스코드에서 EventListener 가 등록되는 부분인 Add 메소드의 소스 코드이다. idMap.put에서 결국 Dynamic Linkage Pattern 으로 귀결된다.
이벤트가 생성되면, idMap 상의 이벤트 이벤트 종류의 ID 에서 이를 처리하는 이벤트 리스너의 정보를 검색하여 해당 이벤트 리스너의 Notify 메서드를 호출한다. 이벤트 리스너는 모두 Notify[2]메소드를 구현해야 한다.

분산 이벤트 모델 & Adaptor

동적 연결 패턴은 당연하게 Adaptor 패턴과 연관 되어 사용된다.
자바 스페이스에서의 이벤트 처리 내부 동작은 동적 연결 패턴을 이용하여 등록된 이벤트 리스너를 획득하여 자바 스페이스 시스템이 해당 이벤트 리스너의 Notify 메소드를 호출하여 처리된다.

리모트 이벤트 리스너의 Constructor 와 Notify 메소드의 구현은 분명 Adaptor Pattern 으로 모두 RemoteEventListener 인터페이스를 상속하여 구현된다.

[2] JavaSpace.notify 와 EventListener.notify 의 역할이 명확히 다르다는 사실을 염두에 두기를 바란다. 전자는 이벤트 등록 과정이며 후자는 이벤트 처리과정이다.



11. 그림 11 – Adaptor 패턴을 이용한 이벤트 리스너 구현 클래스 다이어그램

자바 스페이스와 자바 스페이스의 이벤트를 연결하여 주는 RemoteEventListner 는 Interface 로 정의되어 있지만, 이를 구현하는 객체는 RMI 를 이용하여 RemoteEventListener 를 정의한다. 이는 다음의 예제를 참조하라.

```
public class ExampleListener implements RemoteEventListener {
        private JavaSpace space;

        public Listener(JavaSpace space) throws RemoteException {
                this.space = space;
                UnicastRemoteObject.exportObject(this);
        }

        public void notify(RemoteEvent ev) {
                Message template = new Messgae();
                try {
                        Message result =

        (Message)space.read(template,null,long.MAX_VALUE);
                        System.out.println(result.content);
                } catch( Exception e) {
                        e.printStackTrace();
                }
        }
}
```

위의 예제에서 Constructor 는 RMI 의 처리를 담당하고, Notify 에서 이벤트의 처리를 수행한다.

분산 이벤트 모델과 RMI

자바 스페이스 분산 이벤트 모델에서 분산 이벤트의 처리는 RMI(Remote Method Invocation)에 근간을 두고 있다. 하나의 자바 스페이스는 다른 RemoteEventListener 인터페이스를 통하여 다른 자바 스페이스에 존재하는

객체들을 이벤트 리스너로 등록 할 수 있다. 이때, RMI Proxy 를 이용하여 RMI의 Proxy 객체를 이벤트가 발생하는 자바 스페이스에 등록한다.

자바 스페이스에서 발생된 이벤트는 Proxy 객체를 통하여 원격지에 있는 객체에게 전송되고, 원격지에서 이를 처리한다.[RMI 와 Proxy 패턴의 상호관계에 관한 설명은 11 월 호의 기사를 참조하기 바란다.]

분산 이벤트 모델의 이해와 디자인 패턴

패턴에 대한 사전 지식이 자바 스페이스의 이해에 도움을 분명 주었을까?
필자는 아직까지 자바 스페이스에 관해서 심도 있는 이해를 하고 있지 않다. 다만 분산 컴퓨팅환경에서 객체들을 담아두고 관리하기 위한 저장소 정도의 개념으로만 생각하고 있었다.
그러나 분산 컴퓨팅 환경 중에서 중요한 요소인 분산 이벤트 처리의 방법을 다루면서 프로그램의 이해에 많은 고민과 고통이 뒤따라 왔다. 몇 일을 밤을 새면서 글을 읽고 있으면, 어느 순간인가 갑자기 이해가 되는 부분들이 있다.
대개의 경우, 이러한 부분들은 패턴의 개념이 적용된 부분들이다. 리모트 이벤트 리스너 등에서 사용된 Proxy 기법까지의 패턴이 파악되는 순간, 분산 컴퓨팅 환경에서 이벤트 처리가 어떻게 이루어 지는지에 관한 기본 수준의 이해를 하게 되었다.

패턴은 분명 새로운 기술과 프로그래밍 개념의 이해를 돕는데 많은 도움이 된다.

커넥터 아키텍처(Connector Architecture)와 디자인 패턴

커넥터 아키텍쳐란 무엇인가?

커넥터 아키텍쳐는 이 글을 쓰는 현재 JCP(Java Community Process)의 Final Draft 단계 에 있으며 같이 진행되고 있는 J2EE 1.3 스펙에 포함될 예정으로 있다. 커넥터 아키텍쳐는 이름이 의미하는 것처럼 임의의 EIS(Enterpirse Information System)에 대한 접근을 일반화하시키자는 데에 그 목적이 있다. 시스템을 개발할때에 매일 사용하고 있는 데이터베이스에서부터 ERP, TP Monitor, Groupware, KMS 등도 모두 EIS 라고 할 수 있다. 자 그럼 지금까지 EIS 와 연동되는 프로그램 혹은 서버를 개발하기 위해 어떤 일을 해 왔는지 한번 상기해 보는 의미에서 그림 3 을 살펴 보도록 한다.

[그림 3 ERP, TP System 에 대한 기존 개발 구조도]

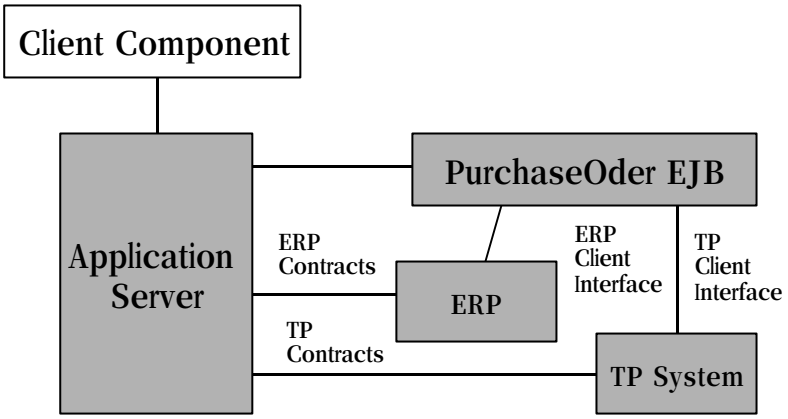


그림 3 은 구매 시스템에서의 PurchaseOrder EJB 를 개발했을때의 구조도이다. 이 그림에서의 애플리케이션 서버란 EJB Container 와 JSP/Servlet Container 등을 모두 포함하는 J2EE 기반의 애플리케이션 서버이다. PurchaseOrder EJB 에서는 ERP 와 TP System 에 접근을 해야 한다. 이것은 보통 ERP 나 TP System 에서 제공하는 SDK 와 같은 API Set을 통해 이루어진다. 즉 비즈니스 로직을 개발할때는 EIS 에서 개별적으로 제공하는 API 를 이용하므로 EIS 갯수만큼의 인터페이스 로직이 필요하게 된다.

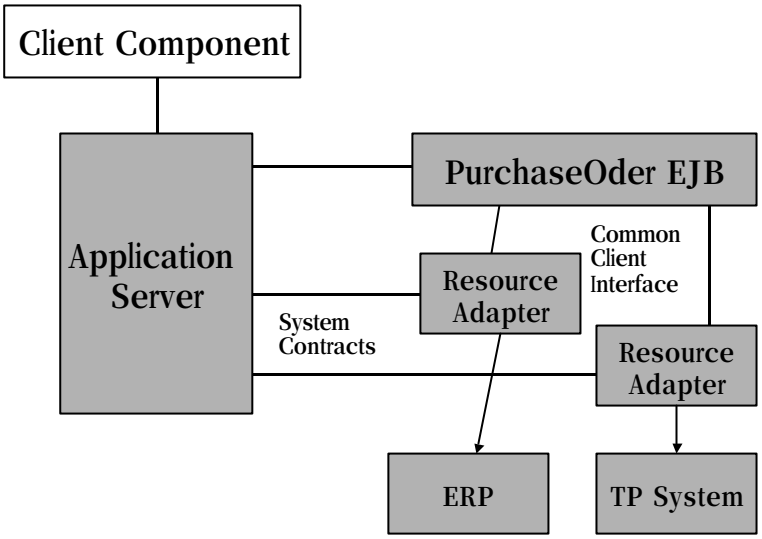
이번에는 애플리케이션 서버입장에서 살펴 보자. 트랜잭션 조절이나 Connection Pool 과 같은 관리의 이유로 애플리케이션 서버 또한 EIS 에 접근할 필요가 있다. 역시 이경우에도 EIS 에서 제공하는 Administration API 를 이용해야 하기 때문에 EIS 개수만큼의 인터페이스 로직이 필요하게 된다.

아마도 필자를 포함해 많은 사람들은 이러한 것을 당연시 여겨왔다. 이것은 두가지 이유가 있는데 첫째는 EIS 에 접근하기 위한 일반화된 연결고리가 정의되어져 있지 않았다는 것이고 두번째는 EIS 의 다양한 기능을 위한 일반화된 접근법이 정의되어질 수 있겠느냐는 의문이 있어 왔다는 점이다.

하나의 예를 들어 보자. 많은 DB 벤더들은 나름대로의 특화된 API 를 제공하였고 자신의 제품이 모든 DB 에서 동작한다고 선전하는 회사의 개발팀에서는 별개의 DB 마다 릴리즈하기 위해 특별한 밤들을 추가적으로 새워야 했다. ODBC/JDBC 의 등장으로 개발자들의 많은 역할들이 드라이버 벤더들에게 넘어갔고 덕분에 제품 개발팀은 매우 빠른시간에 복수 DB 에 대한 개발을 가능해 졌다.

과연 다양한 EIS 에 대해서도 일반화된 연결고리가 정의되어질 수 있을 것인가? 동시에 각 EIS 마다 특화된 기능들을 자연스럽게 사용할 수 있을까? 이 질문의 해답에 대한 도전이 바로 커넥터 아키텍쳐이다. 그림 4 를 보자.

[그림 4 커넥터 아키텍쳐상에서의 ERP, TP System 에 대한 개발 구조도]

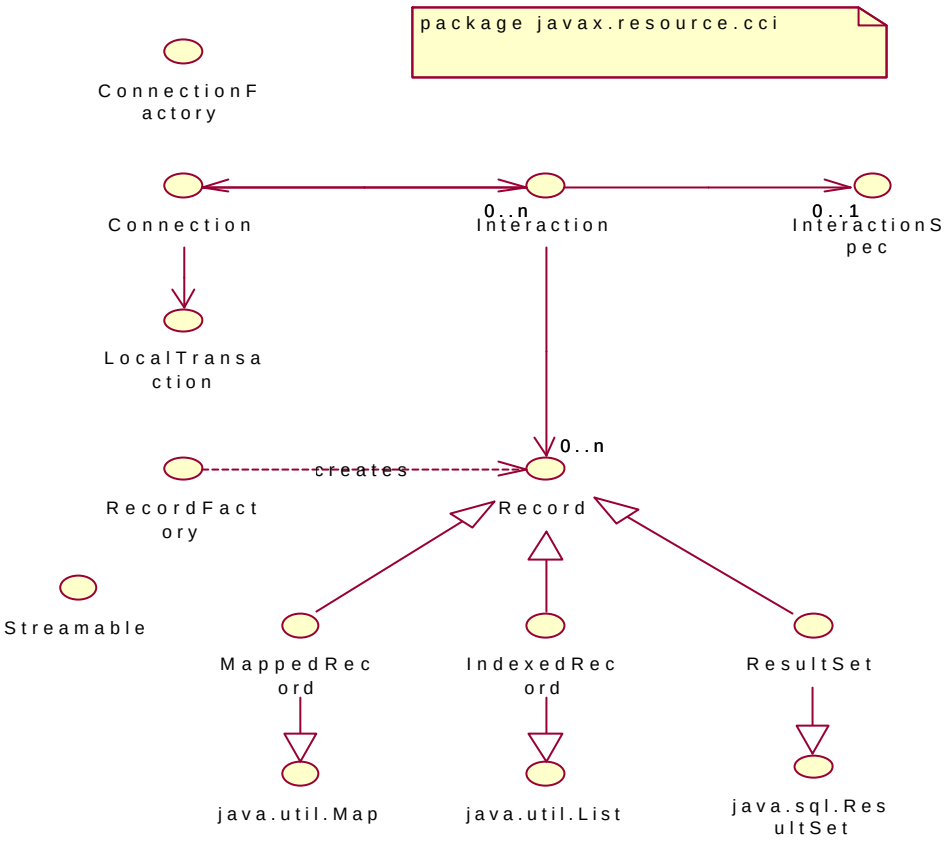


모든 것에 대한 열쇠는 리소스 어댑터(Resource Adapter)에 있다. 리소스 어댑터는 이 어댑터를 사용하는 비즈니스 로직에 대해서 CCI(Common Client Interface)라고 불리는 공통적인 인터페이스를 제공한다. PurchaseOrder EJB 에서는 CCI 를 통해 단일한 로직으로 ERP, TP System 을 바라볼 수 있게 된다. 애플리케이션 서버에서는 System Contracts 라고 불리는 리소스 어댑터에서 정의되어 있는 기능을 이용해 어떠한 리소스 어댑터도 애플리케이션 서버에 플러그인되어 작동되어 질 수 있다. 물론 애플리케이션 서버내에서는 System Contracts 를 지키면 트랜잭션도 같이 조절되게 된다. 이처럼 m x n 매트릭스의 인터페이스들이 1 x 1 인터페이스로 간략히 될 수 있다는 것이 커넥터 아키텍쳐를 통해 얻을 수 있는 장점인 것이다. 자 이제 CCI 와 System Contracts 에 대해 자세히 알아 보자.

CCI (Common Client Interface)란 무엇인가?

우선 CCI 는 EIS 에 연결할 수 있는 Conneciton 기능, 특정 기능을 호출할 수 있는 Interaction 기능 그리고 특정 정보를 주고 받을 수 있는 데이터 표현 기능, 마지막으로 EIS 의 리소스 어댑터의 정보를 얻을 수 있는 메터데이터 기능이 있다. 그림 5 은 javax.resource.cci 패키지의 클래스 다이어그램이며 CCI 의 기능들을 한 눈에 살펴 볼 수 있다.

[그림 5 CCI Class Diagram]



Connection 인터페이스는 EIS 에 대한 커넥션을 총괄하게 된다. 인터액션(Interaction) 인터페이스는 EIS 에 대한 특정 명령을 실행하게 해준다. 이때 필요한 인자들 즉 실행될 함수 이름, 수행모드(동기 혹은 비동기)같은 실행 정보들은 인터액션 스펙(InteractionSpec) 인터페이스를 통해 인터액션 인터페이스에 전달된다. Record 인터페이스는 EIS 와 주고 받는 정보를 나타내게 해준다. Record 인터페이스는 총 세가지 서브 인터페이스를 가지는데 이 세가지 서브 인터페이스를 통해 Map, List, Result의 형태로 EIS 정보가 표현된다.JDBC를 사용해 본 독자분들은 많은 추상화 아이디어가 JDBC와 유사한 형태를 띄고 있음을 알 수 있을 것이다.

System Contracts 란 무엇인가?

EIS 를 대리하는(delegation) 리소스 어댑터가 애플리케이션 서버내에서 동작하기 위해서는 애플리케이션 서버가 리소스 어댑터를 조절할 수 있는 어떤 표준이 존재해야 한다. 이 표준을 커넥터 아키텍쳐에서는 System Contracts 라고 부르며 모든 EIS 벤더가 리소스 어댑터를 제공할 때 CCI와 더불어 구현해야 하는 두번째 기능이 된다. 이 System Contracts 에서는 다음의 세가지 기능을 정의하고 있다.

Connection Management

EIS 에 대한 커넥션 풀을 유지하며 애플리케이션 컴포넌트로 하여금 EIS 에 연결을 할 수 있게 해 준다.
Transaction Management
EIS 에 대한 트랜잭션 조절을 가능하게 한다. 이것은 상이한 EIS 들이 같은 트랜잭션내에서 조절될 수 있게 한다.
Security Management
EIS 에 대한 접근(Access)을 조절한다.

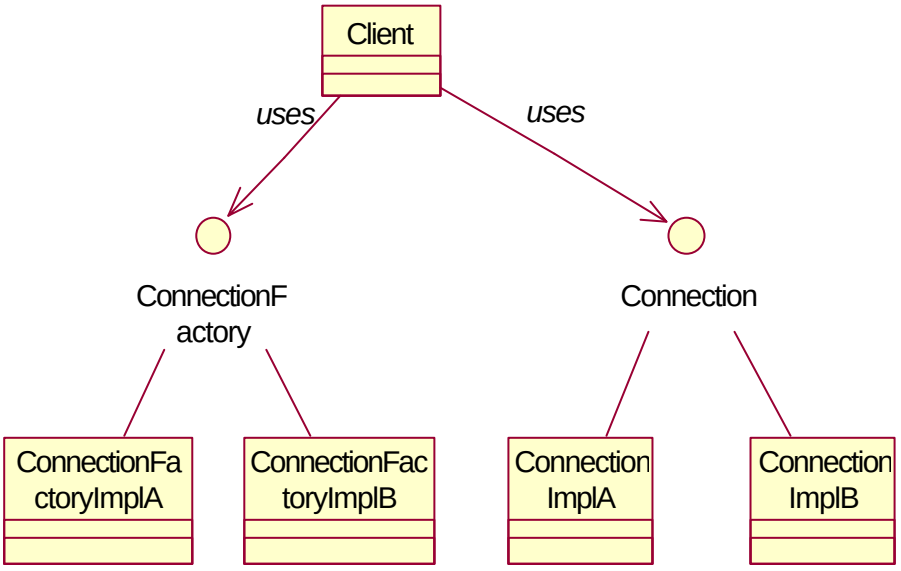
커넥터 아키텍처와 디자인 패턴

커넥터 아키텍처는 다른 J2EE 의 구성요소들과 마찬가지로 인터페이스에 대한 규약이다. 리소스 어댑터는 CCI 와 System Contracts 를 EIS 에 맞게 구현해야 한다. 이 커넥터 아키텍처와 관계된 디자인 패턴에 대해 알아 보자.

추상 팩토리 패턴 (Abstract Factory Pattern)

CCI 의 Connection 을 얻는 매커니즘이 바로 Abstract Factory 패턴을 따르고 있다. 비단 CCI 뿐만 아니라 JDBC 의 Connection 도 이 Abstract Factory 패턴을 가진다. 그림 6 를 보자.

[그림 6 CCI 의 Connection 매커니즘 – Abstract Factory Pattern]



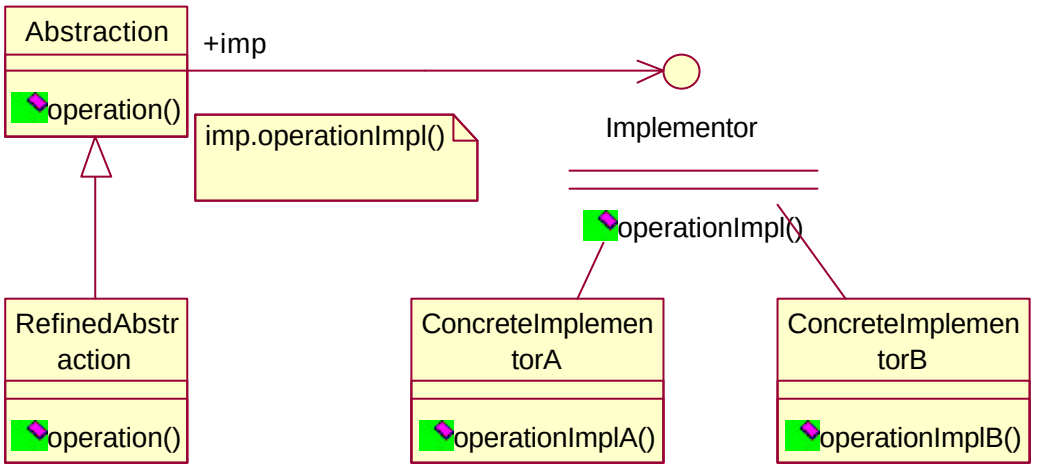
ConnectionFactory 는 Connection 을 얻는 통로 역할을 수행하는 인터페이스이며 보통 JNDI 를 통해서 얻어진다. Factory 구현 클래스를 얻을 때는 얻고자 하는 클래스명을 명시한다. 각각의 Factory 구현 클래스(ConnectionFactoryImpl)는 getConnection() 메소드를 통해 각각에 맞는 ConnectionImpl 을 돌려주게 된다. Abstract Factory Pattern 입장에서 봤을 때 ConnectionFactory 는 Abstract Factory 역할을 하며 Connection 은 Abstract Product 의 역할을 수행한다. 커넥터 아키텍처는 추상 팩토리 패턴을 이용함으로써 각기 상이한 Connection 과 Connection Factory 가 독립적

인 벤더들에 의해 독립적으로 구현될 수 있게 한다. A와 B 외에 C라는 벤더의 구현은 A, B 의 구현에 영향을 미치지 않게 되는 것이다. 커넥터 아키텍쳐의 Connection 매커니즘에 사용된 추상 팩토리 패턴에서는 Abstract Product이 Connection 하나로만 존재하지만 여러 개의 Abstract Prodct이 하나의 Abstract Factory 구현에서 얻어질 수도 있다.
정리하면 추상 팩토리 패턴은 이처럼 여러 개의 Concrete Factory(위의 예에서는 ConnectionFactoryImplA 와 ConnectionFactoryImplB) 와 여기서부터 얻어지는 Concrete Product들(ConnectionImplA 와 ConnectionImplB)을 하나의 Abstract Factory 와 Abstract Product로 사용될 수 있게 해준다.

브릿지 패턴 (Bridge Pattern)

브릿지 패턴은 커넥터 아키텍쳐에서는 직접 쓰이고 있지는 않는다. 그러나 필자는 지금부터 커넥터 아키텍쳐의 인터페이스들이 어떻게 브릿지 패턴에 사용될 수 있는지 보일려고 한다. 우선 브릿지 패턴의 정의부터 살펴 보자. 브릿지 패턴의 클래스 다이어그램은 그림 7 와 같다.

[그림 7 브릿지 패턴 – 클래스 다이어그램]

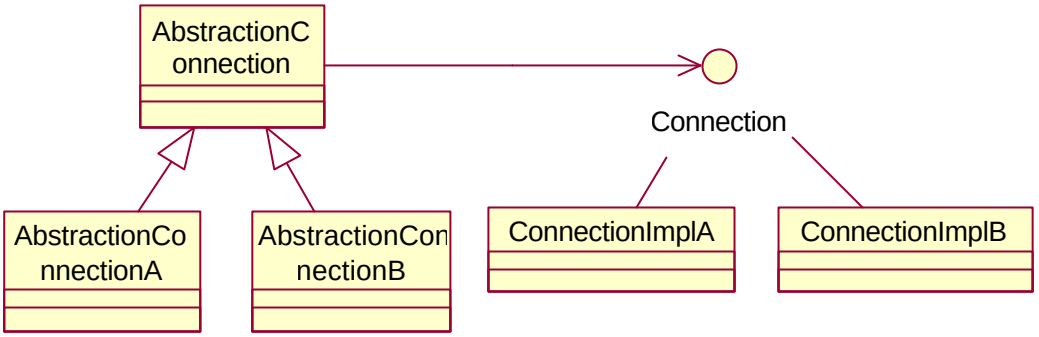


브릿지 패턴에서 중요한 관계는 바로 Abstraction 클래스와 Implementor 인터페이스의 관계이다. 이 둘의 관계가 브릿지(Bridge, 다리)와 같기 때문에 이 패턴의 이름이 브릿지 패턴이 되었다. Implementor 는 정상적인 인터페이스이다. 이 Implementor 를 구현하는 여러 구현클래스들(ConcreteImplementorA, ConcreteImplementorB)이 존재할 수 있다. Abstraction 의 모든 행위들은 Implementor 가 대리(delegation)하게 된다. 이렇게 되면 우리는 Implementor 와 그 구현 클래스들에 상관없이 얼마든지 Abstraction 을 Refine 시켜 나갈 수 있게 된다. 그림 5 에서처럼 RefinedAbstraction 은 얼마든지 존재할 수 있다. 정상적인 상속관계에서는 RefinedAbstraction 과 같은 클래스들이 존재할려면 구현클래스들이 변경이 되어야 했기 때문에 할일이 매우 많아진다.

이번에는 커넥터 아키텍쳐의 Connection 인터페이스에 브릿지를 만들어 보자.(그림 8)

[그림 8 브릿지 패턴을 사용해 커넥터 아키텍쳐상의 커넥션 매커니즘의 확장

]



필자는 Connection 에 대한 AbstractionConnection 을 만들었고 이 AbstractionConnection 클래스는 Connection 에 대한 delegator 역할을 한다. AbstractionConnection 에서는 Connection 의 여러 행위에 대한 로깅을 한다고 해 보자. 이후에 새로운 로깅 기능이 필요하게 된다면 이것은 ConnectionImpl 에 대해서 영향을 주지 않고 단순히 AbstractionConnection 들을 추가함으로써 상이한 로깅행위를 구현할 수 있게 된다.

java.awt 도 브릿지 패턴의 좋은 예이다. Button, List같은 클래스들은 모두 Component 라는 Abstraction 을 상속하며 java.awt.peer 에서는 Implementor 역할을 하는 ButtonPeer, ListPeer, ComponentPeer 를 제공하며 각 플랫폼에 의존적인 코드들이 존재한다. Button vs. ButtonPeer, List vs. ListPeer, Component vs. ComponentBeer 는 모두 Abstraction 과 Implemention 의 관계에 있는 브릿지들이라고 할 수 있다.

정리하면 브릿지 패턴은 구체적인 구현에 대한 상속관계와 이 디테일한 구현에 의존하지 않는 Abstraction 에 대한 상속 관계가 서로 분리되어져야 할 때 사용된다.

마무리의 글

지금까지 총 4 회에 걸쳐 부족하나마 디자인패턴의 실제 적용사례들을 필자들의 경험에 비추어 정리하여 보았다. 이제 패턴의 소개를 마치고 이들을 정리하려 한다.

디자인 패턴들과 패턴의 적용

디자인 패턴이란 매우 광대한 분야이기에 이 짧은 연재에서 언급되지 않은 디자인 패턴의 수도 매우 많다. 연재를 정리하면서 이번 원고를 쓰는 데 중요한 정보원이 되었던 Design Patterns(Elements of Reusable Object-Oriented Software)에 설명되어 있는 모든 패턴을 한번 나열해 보기로 한다.

생성 패턴 (Creational Pattern)

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

구조 패턴 (Structural Pattern)

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

행위 패턴 (Behavioral Pattern)

- Chain of Resposibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

패턴을 실제에 적용한다는 것은 그렇게 쉬운 작업만은 아니다. 패턴들은 몇가지가 뒤섞여서 나타나기도 하고 적용되어야 하는 패턴들이 명확하지 않은 상황도 많이 벌어진다. 때로는 사용자 요구 사항이 변하여 애써 구현했던 패턴 자체가 변화하기도 한다. 예를 들어 Factory Method 패턴은 요구 사항이 좀 더 복잡해 지면 Abstract Factory 패턴으로 종종 바뀐다. 패턴은 책에서 나온 형태보다 간략화되어 쓰이는 경우가 많으므로 어떨 때는 과연 이것이 그 패턴인지에 대한 의심이 들 정도가 되기도 한다.

국내에서 디자인 패턴을 적용하는데의 어려움중의 하나는 국내 개발자들사이에서는 패턴이 널리 알려지지 않았다라는 점이다. 아마도 실제 프로젝트에서는 여러분들은 패턴이 어떻게 사용될지 고민하기 보다 다른 팀원들에게 디자인 패턴이 무엇을 의미하는지 설명하기 위해 더 많은 시간을 사용해야 할지도 모르겠다. 그러나 팀원들 모두에게 디자인 패턴에 대한 공감대가 형성되지 않은 상태에서 일부에 의해 도입된 디자인 패턴은 거의 의미가 없을 것이므로 디자인 패턴의 도입전에 스터디 같은 행위를 통해 팀원들간의 지식의 레벨을 맞추는 것이 무엇보다 중요하다. 소스에 이 클래스는 Singleton Pattern 을 따른다고 커멘트를 달아 놓았는데 다른 팀원이 이 클래스에 잔뜩 static 메소드를 달아 놓는다면 어떻하겠는가?

디자인 패턴을 너무 복잡하게 생각하지는 말자. 본 연재의 처음에서도 언급했듯이 모든 디자인 패턴의 적용에는 다음의 세가지 원칙이 있다고 생각한다. 첫째는 언제나 인터페이스를 먼저 생각해 보라는 것이다. 그리고 인터페이스를 적극적으로 사용해야 한다. 그러나 유틸리티 같은 클래스에서조차 인터페이스를 사용할 필요는 없다. 인터페이스는 클래스들간의 커플링을 최소화시켜 줄 것이다. 두번째는 상속(Inheritance)보다는 Delegation 을 주로 사용하라는 것이다. Delegation 은 상속에서처럼 정적인 관계가 아니라 동적인 상속

관계를 만들어 주기 때문에 보다 유연한 관계를 만들어 줄 수 있다. 사실 많은 디자인 패턴들은 인터페이스와 Delegation, 그리고 상속을 기본 요소로 하고 있다. 셋째는 항상 Coupling 을 최소화하는 데 역점을 두어야 한다. 만약 OO 프로젝트에서 C 에서처럼 로직을 한 펑션에 주욱 기술하고 있다면 그것은 어딘가 잘 못된 것이다. 새로운 클래스와 인터페이스를 도입하고 delegation 과 inheritance 를 사용해서 클래스간의 관계를 설정하자. 언제나 요구 사항은 변화하고 기능 또한 변화하게 된다. 이러한 변화가 시스템 전체적으로 영향을 주지 않는 구조를 만드는 것은 모든 디자인 패턴의 목적중에 하나이다.

Design Pattern 과 소프트웨어의 제작

객체 지향 프로그래밍의 대가 중 한명으로 알려진 Grady Booch 의 글 중에서 소프트웨어를 건축물에 비교한 부분이 있다.

여러분들이 건축물을 만드는 건축가로써 활약하고 있다고 생각해보자. 여러분의 집이나 창고 같은 조그마한 건축물을 지었을 때, 그리고 조금 더 큰 건축물을 지었을 때, 나아가서 초대형 복합 건물을 건축하여야 하는 경우, 어떻게 하겠는가?

자신이 직접 망치와 톱을 들고, 나무를 썰어가면서 집을 지을 수 있을까? 결코 아니다. 오직 고양이나 개와 같은 작은 동물의 집만이 이러한 방식으로 지을 수 있다. 여러분들이 살아갈 조그마한 오두막을 건축하는 것에도 문과 방과 창문들을 만들고, 나무들을 적절히 자르고 운송하고 얽어 묶는 절차가 필요하다.

이제 초대형 주상 복합 건물을 여러분이 건축하게 되었다. 그럼 톱과 망치를 들고 나무 판자에 못 박는 것에서 건축을 시작하겠는가? 결코 그러한 방식으로는 건축할 수 없다.

굳이 초대형 주상 복합 건물과 같은 구조가 복잡한 건출물이 아닌, 가정집과 같은 자그마한 건축 과정에 있어서도 가장 필요한 것은 "설계"이다. 건축물을 어떻게 만들 것인가? 에 관한 설계가 탄생되고서부터, 벽돌을 쌓고, 하나씩 구성요소들을 만족시켜 나야 한다. 패턴은 건축물을 구성하는 구성요소이다. 소프트웨어 건축의 도구이며, 재료이다. 이들이 가져오는 영향들에 비하여 구성 원가가 저렴한 재료들이다.

반면 패턴들을 도입하여 나타나는 영향은 매우 크다. 필자들의 경우, 패턴을 도입하고 난 뒤부터 제작되는 소스 코드의 수준이 상당 수준으로 향상 되었음을 깨닫고 있다. 모든 부분에 패턴들이 사용되었기 때문이 아니라, 패턴을 공부하는 과정에서 함께 깨달았던 객체 지향의 여러 개념들이 소스 코드에 반영되기 때문이다. 여기에는 구현과 인터페이스의 분리, 페케이드(Façade) 클래스로 복합적인 여러 기능을 하나의 클래스로 통합하는 과정 등이 포함된다.
가끔씩 패턴과 이들 개념을 이용하여 3~4 일정도 프로그램을 작정하고 컴파일하고 작동시킬 때, 오류가 거의 발생하지 않는다는 사실을 발견하는 것은 큰 기쁨이었다.

필자들이 객체 지향 프로그래밍을 공부하면서 더욱 깨닫는 것은 패턴의 소중함이다. 객체 지향 소프트웨어 개발 라이프 사이클이서는 이터레이션이라는

과정이 있다. 결과물을 완성하기 위해서 몇 단계의 이터레이션을 반복하여 소프트웨어를 완성하는 것이 객체 지향 개발 방법론에서 각광 받고 있다. 예를 들어 5 개월에 걸쳐서 소프르웨어를 완성한다고 가정하여 보자. 1 개월 설계, 1 개월 1 차 제작, 1 개월 2 차 제작, 1 개월 3 차 제작, 1 개월 테스트의 과정으로 3 번의 제작 과정을 반복하는 것이 이터레이션이다. 디자인 패턴은 각각의 이터레이션에서 소프트웨어 구성 요소들을 재활용하는 단위로 사용된다. 각각의 이터레이션의 목표 설정에 관여하는 구성요소들이다.

가끔씩, "객체가 아닌 오라클이나 MS-SQL 같은 RDB 는 어떻게 표현해야 합니까?"라는 질문을 듣는다. 여기에 대해서 모든 것이 객체라는 설명을 하고 싶다. 관계형 데이터 베이스의 테이블도 하나의 객체이며,이를 대표하는 클래스로 표현될 수 있다. 따라서 객체 지향의 프레임 워크에서는 모든 것이 객체로 표현된다. 관계형 데이터 베이스도 OR MAPPING 의 과정을 거쳐서 객체로 표현되어, 객체 단위로 설계된다. 디자인 패턴은 이 객체 프레임 워크에서 객체들의 동작과 행위를 규정하는 중요한 역할을 담당한다. 어떤 객체들이 다른 객체에 접근할 때의 접근하는 행동 양식을 기술한 규범의 역할을 담당한다.

무엇이 기술인가?

진정한 소프트웨어의 기술은 어디에서 나타나는가? Windows2000, Linux, Netscape 등의 시스템을 만들어가고 구현하는 것이 바로 기술일까?
소니의 기술을 보라, IBM 의 기술을 보라. 간결하고 깔끔하면서도 편리함을 추구하는 소니의 제품들과 JAVA 를 기반으로 엔터 프라이즈를 구성하는 일련의 제품군들이 하나의 개념으로 연결되는 IBM 의 기술은 보고 있으면 절로 탄사를 자아내게 만든다.
마이크로 소프트의 COM 객체도 초창기에는 OLE (Object Linking & Embedding) 이라는 복잡하고 사용하기 힘든 기술이었으나, 시간이 지나면서 COM, ActiveX 로 더욱 간결한 개념과 사용하기 편리한 프로그래밍 수단으로 발전되어 같다.
분명히 이야기 하건데 복잡한 기술 요소들이 덕지 덕지 붙여 적절한 상호 연동 없이 복잡하기만 한 것은 결코 진정한 기술이 아니다. 진정한 기술은 간단함에서 부터 시작된다.
소프트웨어 설계자나 소프트웨어의 코드를 만드는 개발자들에게 기술은 설계와 개발을 간단한 컨셉으로 충분한 기능을 지원하게 만드는 것이다.
이러한 설계에서 간단한 것이 기술이다. 같은 기능을 수행하는 소프트웨어를 설계 하였으나, 한쪽이 다른 한쪽보다 설계가 복잡하고 구현하기 힘들다면 개발자는 어느쪽을 선택할 것인가?
설계에 있어서 간단하고도 강력한 개념에 기반을 둔, 확장성 있는 설계, 미리 검증된 소프트웨어의 설계를 위해서 디자인 패턴은 사용되어야 한다. 건축가가 건물을 지을 때, 방/거실/회의실/화장실등의 여러 개념들을 활용하여 건물을 구성한다. 개발자는 소프트웨어를 설계할 때, Proxy, Factory, Bridge 등의 여러 개념들을 활용하여 소프트웨어를 설계한다. 이들 기본이 되는 개념이 바로 지금까지 설명해온 디자인 패턴들이다.
많은 경우, 사람들은 오해한다. 복잡하고 어렵게 만드는 것이 어렵다고. 그러나, 더욱 어려운 것은 복잡한 시스템을 간결하게 구성하는 것이다. 좋은 시스템은 간결하고 접근하기 쉬운 개념으로 만들어 진다.
복잡하게 구성된 시스템이 정확하게 원하는 바대로 동작하려면, 그 핵심 구성 요소들이 간단하고 명백한 방식으로 정의되어 있어, 동작 원리에 따른 시스템의 행동이 파악되어야 한다. 복잡하고 어려운 구성요소가 편하게 사용될 수

있도록 진정 쉽게 설계되어야 한다. 간단하면서도 다양한 기능을 가지는 소프트웨어를 만드는 것인 쉽지 않다.
여러분들 중에는 소프트웨어의 제작을 단순한 기능이라 생각하시는 분들도 있을 것이다. 하지만, 대부분의 프로그래머들은 자신의 코드들에 자존심을 걸고 있다. (주 : 그래서 코드 리뷰를 그렇게 싫어하는지도 모른다.^^) 프로그래머가 작성한 소스 코드는 그 사람의 얼굴이자, 그 사람의 실력을 그대로 표현한다. 프로그래머에게 코드는 단순한 기술이 아닌 자신을 표현하는 수단이자 도구이다. 나름대로의 프로그래밍 철학과 노하우를 지닌 사람에게는 코딩은 기술이 아닌 예술로써 나타난다. 좋은 코드를 만드는 사람은 예술가의 정신, 장인의 정신을 가지고 나름대로의 세계를 탐미하여 새로운 작품을 창조한다. 코드는 그것에서부터 시작된다.
간단하고 단순명쾌하면서도 좋은 프로그램을 만들었을 때, 분명 희열을 느낄 것이다. 그렇다면 어떻게 간단하고 단순명쾌하면서도 좋은 프로그램을 만들 수 있을까? 디자인 패턴은 많은 프로그래밍의 선배님들이 자신들의 노하우를 후대에게 전수하고자 만들어 놓은 개발 기법이다. 여기에는 어떻게 하면 소프트웨어를 간단하면서도 단순 명쾌하게 만들 수 있는가?에 관한 명백한 방법이 포함되어 있다. 물론 디자인 패턴은 소프트웨어 제작의 전부는 아니며 일부에 불과하다. 하지만, 이 일부가 매우 중요한 일부라는 사실을 명심하기 바란다.

국내 개발자들에게 하고 싶은 말.

필자는 1994 년부터 사용되기 시작한 디자인 패턴이 국내에는 왜 유독 잘 알려지지 않았을까? 하는 의구심을 가지고 있었다. 필자가 추정한 사유로는

1. 디자인 패턴 책과 기타 자료는 경험 있는 프로그래머가 아니면 숙지하기 힘들다.
2. 디자인 패턴 책은 영어가 어렵다.

의 2 가지 주요한 요인이 있었으리라고 추정된다. 디자인 패턴은 프로그래머의 경험이 부족하면 이를 숙지하기 힘들다. 단순하게 머리 속으로 생각하는 것과 경험에 의거하여 패턴의 사상을 확연하게 깨닫는 것과는 많은 차이가 있다. 여기에 디자인 패턴 책이나 다른 객체 지향과 컴포넌트 제작 분야에서 널리 알려진 책들은 하나같이 영어가 매우 어렵다. 작가들의 소프트웨어의 철학을 담고 있기 때문일 것이다. 기회가 닿는다면 해외에서 경력과 어학 실력을 쌓아 두는 것도 많은 도움이 될 것이다.

마지막으로.

독자 분들이 진행하고 있는 모든 프로젝트에 디자인 패턴을 고려해 보라. 현재 진행되고 있는 프로젝트가 있다면 디자인 패턴을 사용해 Refactoring 을 해보고 완전히 새로운 프로젝트라면 클래스 디자인단계에서 디자인 패턴을 최대한 많이 사용해 보도록 하자. 잘 이해가 안되던 패턴들도 구현을 하면서 이해가 되기도 하고 동시에 패턴이 적용되면서 알고리즘이 더 깨끗하게 정리되는 것을 느낄 수 있을 것이다. 필자들이 느꼈던 많은 놀라움들을 독자 여러분들도 함께 공유하게 되기를 바라면서 본 연재를 마치겠다.

참고문헌

1. Patterns in Java Volume 1, Mark Grand, Wiley Press,1999

2. Design Patterns, Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1995

3. JavaSpaces Principles,Patterns, and Practice, Freeman, Hupfer, Arnold from Sun Microsystems.1999

4. Design Patterns by Contracts, Jean-Marc Jezequel, Addison-Wesley,2000

5. JINI 1.0 Source Code, Sun Microsoft Systems

6. JDK 1.2.2 Source Code, Sun Microsoft Systems

7. J2EE Connector Architecture Public Draft

8. J2EE Connector Architecture API Document

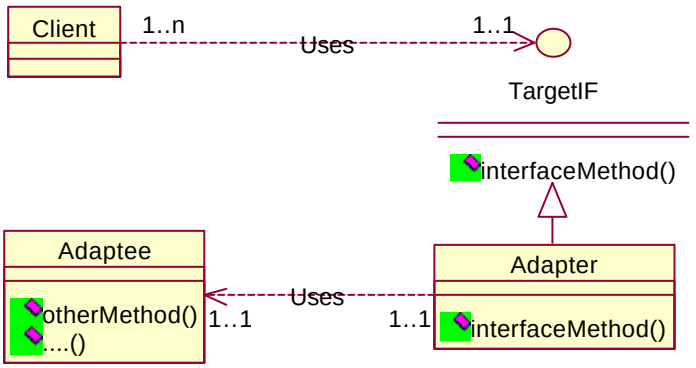
박스기사 1. Adaptor 패턴

어댑터 패턴

"어댑터 클래스는 인터페이스를 구현을 통하여 클라이언트에 알려지지 않은 클래스 인스턴스를 억세스 가능하게 한다. 어댑터 객체는 어떠한 객체가 인터페이스를 구현하였나에 관계 없이 인터페이스에 의해 정의된 기능을 제공한다." – from Patterns in JAVA

간단히 말하자면 객체를 인터페이스와 구현 클래스로 분리하여 여러 개의 객체들이 하나의 인터페이스에 의해서 사용될 수 있도록 정의하는 것이다. 왜 이렇게 하는 것일까? 프로그램들을 재사용하기 위한 방법 중의 하나는 인터페이스와 구현 클래스의 분리이다. 기능을 이용하기 원하는 클라이언트 클래스들은 오직 인터페이스만을 사용한다.

인터페이스와 구현 클래스의 분리는 최상의 캡슐화 효과를 가져온다. 단지 인터페이스만을 사용하는 클라이언트 클래스에서는 인터페이스를 구현한 구현 클래스의 내부적인 상태에 접근할 방법이 없다.



12. 그림 - Adapter Pattern

위의 그림에서 어댑터는 인터페이스를 제공하여 외부에 노출시킨다. 정작 어

덱터 클래스 자신은 작업을 수행하기 위해서 어뎁티(Adaptee) 클래스를 사용한다. 이 부분은 어뎁터 클래스가 단순한 클래스 인터페이스 분리와 다르다는 것을 암시한다. 어뎁터 클래스에는 구현이 거의 존재하지 않으며 이름 그대로의 어뎁터 역할을 담당한다. 어뎁터는 클라이언트 클래스에 기능을 제공하기 위한 중간 클래스의 역할을 담당한다.

결과적으로는 기능을 구현한 어뎁티 클래스와 기능을 사용하는 클라이언트 클래스는 상호 독립적으로 존재하게 된다. 그리고 어뎁터 클래스를 이용하여 어떤 메서드가 호출될지를 다이나믹하게 설정할 수 있다. 데이터 베이스 데이터를 가지고 오는 클래스가 구현되어 있고 이를 어뎁터를 사용하여 이용하였을 때, 데이터 베이스가 아닌 XML 과 같은 다른 데이터 소스에서 정보를 가져올 때도 클라이언트가 어뎁터만 변경하면 동일한 인터페이스로 접근할 수 있다.