12人 knowhow

개발 고수 12인이 말하는 **실전 노하우 ● 디자인 패턴 활용**

잘 쓰면 약, 못 쓰면 독

미워도 다시 보는 패턴 이야기

GoF 디자인 패턴(Design Pattern)을 시작으로 해서 패턴이 이 세상에 알려진지 어느덧 10년이 흘렀다. 하지만 국내에 수많은 패턴 관련 책이 출간되어 있음에도 불구하고, 실제 프로젝트에 적용되는 패턴의 수는 여전히 손에 꼽을 만큼 적다. 뿐만 아니라 지금의 현실과는 맞지 않다고 푸념하는 개발자들도 있고, 패턴의 남용이나 오용으로 인해 오히려 부정적으로 생각하는 이들도 쉽게 찾아볼 수 있다. 이 글에서는 개발자의 입장에서 패턴에 대한 잘못된 생각들을 바로 잡아 보고 간단한 예를 통해 그 올바른 사용법을 정리해 본다.

8 손 영 수
indigoguru@gmail.com

Microsoft MVP와 Devpia Architecture&Design 섹션의 시삽으로 활동 중이다. 부족한 실력이지만 가진 지식을 공유하는 자세만큼은 누구보다 부자라고 자평하고 있으며 지난 3년간 소프트웨어공학에 관련된 세미나와 스터디를 진행하고 있는 A&D Eva의 리더이다. 세계적인 패턴 학회인 PLOP을 국내에도 만드는 것이 꿈이다.

필자는 몇 년 전 어느 세미나에서 GoF, POSA 등의 패턴에 대해 발표한 직후, "패턴이 뭡니까? 간단히 설명해 주세요!"라는 질문을 받은 적이 있다. 그 당시 자세한 패턴 하나하나는 무엇인지 알고 있는 누군가가 정작 전체적인 그림인 패턴의 의미를 물어온다면 어떻게 대답해야 했을까? 필자는 그에 대한 정확한 답을 찾기 위해 먼저 그 어원에 대해 접근해 봤다.

디자인 패턴이란?

디자인(Design)은 De(away) + Sign(note)이라는 의미가 합성된 단어이다. De에는 '가볍게' 또는 '흐리다'(away) 라는 의미가 담겨 있고, Sign이라는 말은 표시(note)를 남긴다는 의미를 지니고 있다. 미술에서 조각상을 그릴 때 흔히 '데생한다'라는 말을 많이 사용하는데, 바로 이 '데생'과 디자인은 같은 어원을 가지고 있다. 어떤 그림을 그리기 전에 살짝 밑그림을 그리는 것을 데생이라고 하는데, 이는 완벽한 그림을 그리기 이전에 형태와 뼈대를 잡기 위한 것이다. 마찬가지로 소프트웨어 개발에서 디자인이라는 단어 역시 소프트웨어의 구조와 뼈대를 잡아가는 작업을 의미한다.

이어서 '패턴(Pattern)'이라는 단어는 흥미롭게도 영어의 'Father'이라는 단어에서 파생된 것을 알 수 있다. 우리가 아버지의 얼굴이나 신체적 특성, 성격 등을 상당수 물려받은 것처럼, 어떤 문제가 발생했을 때 패턴을 써서 '전에 이런 방법으로 해결했었지'라는 생각을 가지고 유사한 방법으로 해결할 수 있는 것이다.

그럼 이제 디자인과 패턴이라는 단어를 조합해 디자인 패턴의 의미를 정의할 수 있다. '프로그램이라는 작품을 만들기 이전에 반복적으로 자주 사용되는 밑그림이나 스케치 등을 마련해두는 것'. 이것이 바로 디자인 패턴의 정의라고 할 수 있다.

패턴에 대한 불신들

최근 들어 종종 패턴에 대한 여러 가지 불만 섞인 목소리를 들을 수 있다. 패턴을 사용하니 오히려 성능이 더 나빠졌다거나 가독성이 떨어져 유지 보수하기가 더 힘들어졌다는 것이 주를 이룬다. 왜 이런 현상이 발생하게 되었을까? 완벽한 분석은 아니겠지만 필자의 경험에 비춰 그 원인을 설명해 본다.

G●F의 23가지 패턴이 모든 문제를 해결해 줄 것이란 잘못된 환상

많은 초보 개발자들은 GoF 패턴이 패턴의 시작이며 끝인 것으로 알고 있다. 하지만 지구상에는 소프트웨어와 연관된 다양한 카테고리의 수십 가지 패턴 서적이 나와 있다. 예를 들어 거대한 시스템의 구조를 효율적으로 잡아주는 구조 패턴을 비롯해 분산 시스템을 위한 패턴, 리얼타임 시스템(Realtime System)의 QoS를 보장하기 위한 패턴, 보안 패턴, 단위 테스팅(Unit Testing)을 위한 패턴, 프로젝트를 제대로 관리하기 위한 패턴 등 여러 가지 관련 서적들이 소개되었고, 1994년 이후 매년 개최되는 PLOP 학회에서도 무수한 패턴들이 쏟아져 나오고 있다.

이런 상황에서 어떤 특정 문제에 딱 들어맞는 패턴을 24가지에서 찾기보다는 수천 가지 패턴에서 찾는 것이 훨씬 유리하지 않을까? 물론 몇 천 가지 패턴을 언제 습득하느냐고 반박할지도 모르지만, 틈틈이 공부해둔다면 시행착오를 줄이는 데 큰 도움이 될 것이다.

선무당이 사람 잡는다

만약 메모리의 제약이 있으면서 빠른 응답속도를 요구하는 임베디드 시스템(Embedded System)에 변화와 확장에는 유연하지만 응답속도가 더딘 패턴(Component Configurator, Pipe&Filter)들을 연동해 적용시키면 어떻게 될까? 흔히 패턴의 남용은 대부분 이러한 상황에 놓여 있다. 자신이 알고 있는 패턴이 이 문제에 적합해 보인다는 것만 고려한 나머지, 정작 시스템이 추구하는 QoS(Quality of Service)를 전혀 고려하지 않고 패턴을 적용하는 것이다. 실제 대부분의 패턴이 추구하는 것은 유연성과 확장성 그리고 유지보수성이지만, 시간적 데드라인을 요구하는 리얼타임 시스템이나 메모리 및 CPU의 제약을 받는 임베디드 시스템 등에서 사용하기에는 다소 무리가 따르는 패턴이 존재함을 잊어서는 안 된다.

모든 개발자가 패턴에 익숙하지 않다

또 하나의 문제는 패턴을 이용해 충분한 협업이 가능할 만큼 모든 개발자들이 패턴에 익숙하지 않다는 점이다. 우리 주위에 비일비재하게 일어나는 한 가지 예를 들겠다. GoF 패턴을 잘 아는 개발자 A가 변화에 유연하면서 확장이 자유로운 프로그램을 만들었다고 하자. 하지만 어느 날 개발자 A가 다른 파트로 자리를 옮기게 되면서 개발자 B가 A의 업무를 대신 맡게 되었다. 하지만 고급 패턴을 더 잘 알고 있는 B는 지금의 상황에 그 구조가 적합했음에도 불구하고, 향후 재활용성을 높이기 위해 A가 만든 모듈을 기반으로 프레임워크를 만들었다. 그러던 중에 갑작스레 B가 회사를 그만두고 떠났다. 그 후 패턴에 익숙하지 못한 개발자 C가 '울며 겨자 먹기'로 특별한 인수인계 없이 프로그램 모두를 떠맡게 된다. 하지만 C는 회사에서 정해 놓은 데드라인을 지키기 위해 프레임워크에 대한 이해 없이 자신만의 코드를 짜게 되고, 심지어 당장의 결과를 보기 위해 프레임워크 내부에 자신이 원하는 코드를 삽입하게 된다. 이런 상황에서 여러분이 이 프로그램의 개발 업무를 인수 받았다고 생각해 보자. 생각만 해도 끔찍할 것이다. 여러분이 패턴에 익숙한 개발자라고 할지라도 프레임워크 안에 뒤죽박죽된 코드 탓에 상당히 많은 시간 동안 리팩토링 작업을 해야 할 것이고, 또한 말도 안 되는 데드라인이 눈앞에 있다면 리팩토링은 고사하고 당장의 결과를 보기위해 더욱 더 꼬인 스파게티와 같은 프로그램을 만들게 될 것이다. 결국 패턴을 잘 안다고 하더라도 그에 대한 지식을 조직과 팀 내에서 모든 구성원이 충분히 이해하고 공유하지 않는다면, 모든 것이 무용지물이 될 가능성이 크다.

패턴을 바라보는 올바른 자세

비록 부족한 경험이지만, 좀 더 실무에서 디자인 패턴을 잘 사용할 수 있도록 몇 가지 조언을 제시해 본다.

나무보다는 숲을 보라(패턴을 만드는 5가지 법칙 파악)

문제를 해결하기 위해 특정 패턴을 하나하나 생각하면서 매치하는 것보다는 패턴을 만드는 5가지 법칙에 의거해 생각하길 권한다(짧은 페이지에 패턴을 만드는 5가지 법칙을 설명하는 것은 매우 어려운 일이므로, 필자의 온라인 동영상 참고자료를 활용하길 바란다). 여러분이 Observer, Model-View-Controller, Publisher-Subscriber, Event Channel 패턴을 각각 별개의 패턴으로 생각한다면, 무엇보다 패턴을 만드는 5가지 법칙을 살펴볼 필요가 있다. 이 패턴들은 사용되는 도메인은 모두 다르지만, DIP(Dependency Inversion Principle, 의존 관계 역전의 법칙)에 의거하면 모두 '수다쟁이 객체'를 만날 때 적용됨을 알 수 있다.

패턴간의 연관성을 살펴보라

패턴은 결코 독단적으로 사용되지 않는다. 함께 사용되는 패턴들을 주의 깊게 살펴보길 바란다. Component Configurator, Reflection, Template Method(Strategy)는 항상 하나의 묶음으로, Chain of Responsibility도 역시 Iterator 패턴과 함께 연동되어 사용된다.

패턴은 크게 두 가지 원칙에 의거한다.

어떤 관점에서 보면 패턴은 크게 두 가지 원칙에 의거하고 있다. GoF 서두에 나온 이 두 원칙을 과소평가하는 개발자들이 생

각보다 많이 존재한다. 패턴과 관련된 책 한 권을 읽은 개발자라면 이 두 가지 원칙을 다시 한 번 살펴보길 바란다. 다시금 전율을 느끼게 되는 순간을 맞이하게 될것이다.

● 1원칙 : Program to an interface, not to an implementation

'인터페이스(계약) 기반의 프로그래밍을 해라.' 지금 한번 디자인 패턴 서적을 펼쳐보길 바란다. 모든 패턴들이 추상 클래스(Abstract Class)나 인터페이스(Interface)로부터 상속을 받고 있다. 이것은 다형성을 유지하기 위한 방법으로 동일한 인터페이스(Interface)를 상속받은 객체라면 런타임(Runtime)시에 언제든지 쉽게 바꿔 사용할 수 있다. 여러분의 프로그램에서 정렬 알고리즘으로 Quick Sorting을 사용했는데, 누군가 더 빠른 Super Quick Sorting 알고리즘을 개발했다고 하자.

이 두 정렬 알고리즘이 동일한 인터페이스만 가지고 있다면 시스템의 중지 없이 교체할 수 있을 것이다. 인터페이스의 상속(계약 기반)이 어떻게 보면 각 객체가 지닌 표현의 자유를 억압하는 것으로 생각할 수 있지만, 이런 약간의 억압으로 인해 자유롭게 교체할 수 있는 장점을 얻은 것은 분명하다. 사실 모든 패턴이 Strategy나 Template Method 패턴을 약간 변형했다고 해도 과언은 아닐 것이다.

● 2원칙 : Favor Object Composition over Class Inheritance

'클래스 상속(Class Inheritance)보다는 객체 조합을 선호해라.' 이 두 번째 원칙은 현재 많은 논란을 지니고 있다. 심지어 국내외의 몇 가지 서적들은 이 두 번째 원칙의 의미를 제대로 분석하지 않은 채, '상속보다는 조합을 선호하라'는 극단적인 표현을 사용하기도 한다. GoF에 나오는 모든 패턴이 상속을 사용하는데 상속을 사용하지 말라는 의미를 어떻게 받아들여야 할까? GoF 디자인 패턴이 등장한 시대적 배경을 고려하면 이는 의외로 간단해진다. 당시 이 원칙은 C++를 기준으로 쓰였던 터라 인터페이스라는 명시적인 개념이 잡혀 있지 않아 의미의 차이를 발생시킨 것이다. 클래스 상속을 현대적인 의미로 풀이하자면 구현 상속에 가깝다. 이미 상속된 클래스를 구현 받아 사용하는 것보다는 객체 조합(Component 방식)을 선호하라는 의미인 것이다. 중요한 것은 객체 조합이 내부적으로 인터페이스 상속(지금의 용어로는 Interface Implementation)을 사용한다는 것이다. 이와 관련해 『GoF Design Pattern』의 저자인 Eric Gamma의 인터뷰를 참고하면 보다 자세한 내용을 확인할 수 있다.

시스템 중지가 없는 유연한 프로그램 만들기

필자는 몇 가지 패턴의 조합을 예로 들어 시스템의 중지 없이 변화에 유연한 프로그램을 만드는 방법을 소개하고자 한다. 요즘 많은 플랫폼이나 개발자들 사이에서 메타데이터(Metadata) 지향적인 프로그래밍이 대두되고 있다. 변화가 자주 발생하는 부분(보안 프로토콜 교체, DB 연결 문자열 변경, 네트워크 프로토콜)들을 메타데이터에 기술함으로써, 프로그램의 수정 없이 변화에 빠르게 대처하기 위해 많이 사용하고 있다. 하지만 이런 과정에서는 또 다른 문제점이 발생한다. 파일(file)에 설정 정보가 있다 보니 자주 읽어 들이는 부분의 경우에는 파일 접근을 위해 지나치게 빈번한 I/O가 발생한다는 것이다. 이는 빠른 응답속도나 많은 클라이언트를 처리해야 하는 프로그램에서는 성능 저하의 주된 원인이 된다. 따라서 설정 파일(App.Config 또는 Web.Config)의 정보를 고스란히 들고 있는 Configuration Handler를 〈리스트 1〉처럼 만들고, 프로그램을 로딩할 때 Handler에서 설정 파일의 정보를 메모리에 로드하게 된다. 그러면 클라이언트(Client)들이 Handler를 통해 정보를 얻어오게 되므로, 파일 I/O에 대한 성능 저하 문제를 해결할 수 있다.

〈리스트 1〉 설정 파일의 정보를 읽어오는 Handler의 예 – .NET 용

```
using System.Collections.Specialized;
using Microsoft.ApplicationBlocks.ExceptionManagement;

public class WebAppConfig
{
    private static string mSiteName;
    private static string mConnectString;
    static WebAppConfig()
    {
        NameValueCollection nvc = new
NameValueCollection();
        try {
        // GetConfig 메소드를 호출하면 web.config에 등록된 클래스 내의
        // Create method가 호출되어 NameValueCollection 객체를
        // 되돌린다.
nvc =
(NameValueCollection)System.Configuration.ConfigurationSet
tings.GetConfig("AppConfig");
            if ((nvc != null))
            {
                mConnectString = nvc("ConnectString");
                mSiteName = nvc("SiteName");
            }
        }
        catch (Exception exp)
        {
            ExceptionManager.Publish(exp);
        }

    }
    public static string SiteName
```

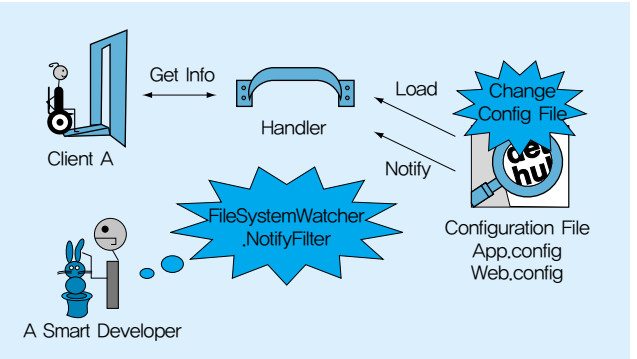
```
    {       get { return mSiteName; } }

    public static string ConnectString
    {       get { return mConnectString; }      }

}

using System.Collections.Specialized;
using Microsoft.ApplicationBlocks.ExceptionManagement;
public class WebAppConfigHandler :
IConfigurationSectionHandler
{
    public object
IConfigurationSectionHandler.Create(object parent, object
configContext, Xml.XmlNode input)
    {
        // 1. xpath를 통한 처리
        // 2. xml elements enumeration
        // 3. NameValueSectionHandler의 이용
        NameValueCollection nvc;
        NameValueSectionHandler handler;
        try {
            handler = new NameValueSectionHandler();
            nvc =
(NameValueCollection)handler.Create(parent, configContext,
input);
        }
        catch (Exception exp) {
            ExceptionManager.Publish(exp);
            throw exp;
        }
        return nvc;
    }
}
```

하지만 우리는 부작용이라 할 만한 새로운 문제를 만나게 된다. 바로 파일 정보와 Handler간의 데이터 일치성을 보장할 수 없다는 점이다. 프로그램 로딩시에만 해당 정보를 로드하기 때문에 설정 파일의 정보를 변경하더라도 메모리에 올라 있는 Handler의 정보가 변경되지 않는 것이다. 결국 시스템을 잠시 정지시키고 재 시작함으로써 설정 파일 정보의 변경된 내용을 읽어오는 방법 밖에는 없다. 다시 말하자면, 파일 I/O를 줄이다 보니 Handler와 설정 파일간의 데이터 불일치가 발생해 시스템의 중지 없이, 런타임(Runtime)시에 객체의 속성을 자유롭게 변경하지 못하게 되었다. 그러나 설정 파일의 내용이 변경될 때 Handler가 다시 설정 파일로부터 정보를 읽어오는 로직을 추가한다면 Handler와 파일간의 데이터 불일치 문제를 해결할 수 있다. 단 주기적으로 설정 파일을 체크하는 로직을 넣는다는 것은 성능상의 저하를 가져올 수 있으므로 다른 방법을 강구해야 한다. 이러한 문제를 해결하기 위해 DIP를 적용한 Publisher-Subscriber 패턴을 적용할 필요가 있다.



〈그림 1〉 Publisher-Subscriber와 Configuration Handler의 만남

〈그림 1〉과 같이 설정 파일을 감시하는 하나의 Monitor(Watcher)를 하나의 Publisher로 만들고 Handler들을 Subscriber로 만들면 될 것이다. 중요한 것은 설정 파일의 변경 내용을 감지하는 감시자(Watcher)를 어떻게 구현하느냐다. 다행스럽게도 .NET 2.0 SDK부터는 System.IO 네임스페이스에서 FileSystemWatcher.NotifyFileter를, 자바에서도 개발자에 의해 만들어진 유사한 NotifyFilter를 제공하고 있다(구현 모듈은 참고 자료에 소개).

이달의 디스켓에 수록된 〈리스트 2〉의 Event Handler 부분에 화면 출력 대신 설정 정보를 가진 Handler로 변경 내용을 통보하는 로직만 추가하면 런타임시에도 유연한 변화를 나타내는 프로그램을 만들 수 있다. 여기서는 페이지 제약 탓에 완벽한 소스 코드를 보여주지 못해 아쉽지만, 큰 힌트는 모두 주었으므로 나머지의 구현은 독자들의 몫으로 남기고자 한다.

이달의 디스켓 : Pattern.zip

참고 자료

1. Design Principles from Design Patterns(Eric Gamma의 인터뷰), http://www.artima.com/lejava/articles/designprinciplesP.html
2. Windows File System Watcher for Java, http://www2.hawaii.edu/~qzhang/FileSystemWatcher/index.html
3. Eric Gamma et al, 『Design Patterns : Elements of Reusable Object Oriented Software』, Addison Wesley
4. FileSystemWatcher.NotifyFilter, http://msdn2.microsoft.com/en-us/library/system.io.filesystemwatcher.notifyfilter(VS.80).aspx
5. Frank Buschmann et al, 『Pattern-Oriented Software Architecture : A System of Patterns』, Volume 1, Wiley
6. Douglas Schmidt et al, 『Pattern-Oriented Software Architecture』, Volume 2, Wiley
7. 패턴의 5가지 법칙, 손영수, 백범호, http://www.devpia.com/DevStudy/Lecture/LectureDetail.aspx?nSemiID=1171