동적으로 자바 어플리케이션 확장하기

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

Document Information

Document title:	**동적으로 자바 어플리케이션 확장하기**
Document file name:	Dynamically_Extend_박준용.doc
Revision number:	<1.0>
Issued by:	<JSTORM>
Issue Date:	<2001/08/21 >
Status:	Final

Content Information

Audience	**자바 초, 중급**
Abstract	**자바애플리케이션을 동적으로 확장하는 방법에 대해 설명한다.**
Reference	http://www.javaworld.com/javaworld/jw-08-2001/jw-0810-extend.html **번역**
Benchmark information	

Table of Contents

동적으로 자바 어플리케이션 확장하기

기능 추가를 위해 인터페이스와 동적 클래스 로딩을 사용하자

요약

소스 코드 변경 없이 확장할 수 있는 프로그램을 작성하고 싶은가? 이 기사에서 설명하는 기술은 상위 확장 시스템(Highly extensions Systems)을 생성하는 인터페이스와 동적 클래스 로딩을 사용하는 방법을 보여준다. 이러한 확장 시스템을 생성하려고 할 때 소스 코드가 필요 없이 다른 프로그래머와 사용자가 가능하도록 하는 방법도 배운다. 동적 로딩과 인터페이스를 사용하지 않는 간단한 예제로 시작해서 파일과 데이터베이스에서 읽어들여 동적으로 클래스를 로딩하는 예제로 발전시켜나가자.

By Bill W.Davis

프로그램의 생명주기 동안 새로운 기능을 필요로 하는 프로그램을 개발해본적이 있는가?

당신의 프로그램은 이러한 새로운 기능들을 다루거나 소스코드 변경없이 사용자가 소프트웨어를 최적화할 수 있도록 해야 한다.

이미 작성된 코드를 변경하거나 새로운 기능 추가에 대한 테스트를 피할 방법이 있는가?

재컴파일하지 않고 프로그램에 새로운 클래스를 추가할 수 있는가? 이미 예상했듯이 이 대답은 인터페이스와 동적 클래스 로딩에 기초해서 둘다 그렇다(Yes) 이다.

이 기사에서 사용되는 대부분의 클래스와 프로그램 구조는 전문적인 프로그래밍에서 사용되는것보다 간소화되어 있다. 전체 코드는 동적 확장 프로그램에 인터페이스를 사용하는 방법을 설명한다. 실제 시스템에서 사용되기를 의도하지는 않는다.

인터페이스란 무엇인가?

인터페이스는 객체가 호출되는 방법을 간단히 묘사한다. 인터페이스를 정의할 때 다른 객체들이 특정한 객체를 어떻게 사용할까와 연관지어 생각해야 한다.

자바를 아는 대부분의 사람들은 이미 인터페이스를 이해할 것이다. 인터페이스에 대한 지식 없이 자바를 사용하기는 어려운 일이다. 그러나 잘 알지 못하는 사람들을 위해 기초부터 시작해서 좀 더 복잡한 예제들을 만들어 갈 것이다. 이미 인터페이스에 대해 알고 있다면 클래스이름을 나타내는 스트링 사용하기 절까지 넘어갈 수 있다.

인터페이스의 강력함

첫번째 예제는 인터페이스의 강력함을 설명한다. 고객이 중개업을 한다고 가정하고 그들은 당신이 거래시스템을 구축해주기를 원한다. 그들은 온갖 종류의 유가증권(주식 , 채권등)들을 거래한다. 각 고객마다 거래에 따라 다른 요금을 매기기를 원한다. 요금은 클라이언트들이 요금표라 부르는 것에 의해 정의된다.

첫번째로 클래스들을 설계해야 한다. 메인 클래스와 클라이언트에 의해 정의된 프로퍼티들은 다음과 같을 것이다

- 고객 : 이름 , 주소 , 전화번호 , 요금표
- 거래 : 거래종류(주식 , 채권) , 거래물품(주식 이름) , 거래량 , 가격 , 수수료
- 요금표 : 거래 수수료를 계산하는 호출 프로시저 제공

인터페이스 없이 코딩하기

인터페이스를 사용하지 않고 코딩할수 있고 그 후 코드들을 강화할 수도 있다. 현재 클라이언트는 다음에 정의된 두가지 요금 정책을 가지고 있다.

- 정책 1 : 일반 고객을 위해 거래당 20%
- 정책 2 : 한달에 처음 10번 거래는 거래당 15$ 그 이후는 거래당 10$

Trade(거래) 객체는 고객에게 수수료를 청구하기 위해 PricingPlan(요금표) 객체를 사용한다. 각각의 요금 정책마다 PricingPlan 클래스를 생성한다. 요금정책1을 위한 클래스는 PricingPlan20 이라 이름짓고 요금정책2는 PricingPlan1510이라 이름짓는다. 양쪽 클래스는 CalcCommission() 프로시저 호출을 사용해 수수료를 계산한다. 코드는 다음과 같다.

Class Name: PricingPlan20

```
public double calculateCommission( Tradetrade )
{
    return 20.0;
}
```

Class Name: PricingPlan1510

```
public double calculateCommission( Tradetrade )
{
    double commission = 0.0;

    if( trade.getCustomer().getNumberOfTradesThisMonth() <= 10 )
        commission = 15.0;
    else
        commission = 10.0;

    return commission;
}
```

Trade 에서 수수료를 얻어 오는 코드

```
public double getCommissionPrice()
{
    double commissionPrice = 0.0;

    if( getCustomer().getPlanId() == 1 )
    {
        PricingPlan20 plan1 = new PricingPlan20();
        commissionPrice = plan1.calculateCommission( this, getCustomer() );
        plan1 = null;
    }
    else
    {
        PricingPlan1510 plan2 = new PricingPlan1510();
        commissionPrice = plan2.calculateCommission( this, getCustomer() );
        plan2 = null;
    }

    return commissionPrice;
}
```

Hardcoded 인터페이스

이전 예제에서 인터페이스를 이용해 더 쉽게 할 수 있는 방법이 있을까? 당신은 PricingPlan 클래스를 구현하는 PricingPlan 인터페이스를 생성할 수 있다.

Interface Name: IPricingPlan

```
public interface IPricingPlan {
     public double calculateCommission( Tradetrade );
}
```

인터페이스를 정의할 때 calculateCommissin() 메소드 바디를 정의하지는 않는다. 실제 PricingPlan 클래스는 메소드를 바디를 채울것이다. PricingPlan 클래스에 대한 첫번째 수정은 이전에 정의한 인터페이스를 구현할 것이라고 선언하는 것이다. PricingPlan 정의의 윗부분에 다음 코드를 추가함으로써 이러한 일을 한다.

```
public class PricingPlan20 extends Object implements IPricingPlan {
```

자바에서 인터페이스를 구현하기 위해 선언할 때 인터페이스가 정의한 모든 메소드 바디를 제공해야 한다. (abstract 클래스를 생성한다면 관계없다.) 그러므로 , IPricingPlan 인터페이스를 구현한 어떠한 클래스라도 calculateCommission() 이라 불리는 메소드를 정의해야 한다. 메소드 시그너쳐는 인터페이스에서 정의한 것과 정확히 같아야 하므로 Trade 객체를 받을 수 있다. 두개의 PricingPlan 클래스에서 calculateCommi ssion() 메소드를 이미 가지고 있으므로 더 이상 수정하지 않아도 된다. 만약 새로운 PricingPlan 클래스를 생성한다면 IPricingPlan 인터페이스와 calculateCommission() 메소드를 구현해야 한다.

다음은 아래와 같은 인터페이스를 사용해서 Trade 클래스의 getCommissionPrice() 메소드를 변경할 수 있다.

Class Name: Trade

```
public double getCommissionPrice()
{
    double commissionPrice = 0.0;

    IPricingPlan plan;

    if( getCustomer().getPlanId() == 1 )
    {
        plan = new PricingPlan20();
    }
    else
    {
        plan = new PricingPlan1510();
    }

    commissionPrice = plan.calculateCommission( this );
    return commissionPrice;
}
```

IpricingPlan 인터페이스로 PricingPlan 변수를 정의한 방법에 주목하라. 실제로 생성한 인스턴스는 고객의 요금체계에 달려있다. 두개의 PricingPlan 클래스들은 IPricingPlan 인터페이스를 구현하므로 어느 하나의 새로운 인스턴스를 동일한 변수에 대입할 수 있다. 일반적으로 자바는 인터페이스를 구현한 실제 객체에는 관심이 없고 인터페이스 그 자체에만 관심이 있다.

클래스 이름을 나타내는 스트링 사용하기

당신의 상사가 와서 두가지 새로운 요금체계가 생긴다고 말한다. 새로운 요금 체계는 거래 당 8$와 10$이다. 당신은 두개의 새로운 PricingPlan 클래스(PricingPlan8 과 PricingPlan10)을 만들기로 결정했다.

이런 경우 당신은 이러한 새로운 PricingPlan 을 포함하도록 Trade 클래스를 변경해야 한다. 간단하게 if/then/else 문을 추가할 수 있지만 요금체계의 숫자가 늘어날수록 덩치가 커지게 된다. 다른 방법은 PricingPlan 인스턴스를 생성할 때 new 대신 Class.forName() 메소드를 사용하는 것이다. Class.forName()은 프로그램에서 이름을 하드코딩하는 대신 클래스 이름을 나타내는 스트링을 이용해서 인스턴스를 생성하도록 한다. Trade 클래스의 이전 코드에서 이러한 방법을 사용하는 예제가 다음에 나온다.

Class Name: Trade

```
public double getCommissionPrice()
{
    double commissionPrice = 0.0;

    IPricingPlan plan;
    Class commissionClass;

    try
    {
        if( getCustomer().getPlanId() == 1 )
        {
            commissionClass = Class.forName( "string_interfaces.PricingPlan20" );
        }
        else
        {
            commissionClass = Class.forName( "string_interfaces.PricingPlan1510" );
        }

        plan = (IPricingPlan) commissionClass.newInstance();

        commissionPrice = plan.calculateCommission( this );
    }
    // ClassNotFoundException, InstantiationException, IllegalAccessException
    catch( Exception e )
    {
        System.out.println( "Exception occurred: " + e.getMessage() );
        e.printStackTrace();
    }

    return commissionPrice;
}
```

이 코드는 이전 코드에 비해 많은 장점을 제공하지는 않는다. 실제로는 예외처리 구문을 포함해야 하므로 더욱 길어졌다. Trade 클래스에서 PricingPlan 클래스 이름의 배열을 생성한다면 어떤 일이 벌어질지 살펴보자.

Class Name: Trade

```
public class Trade extends Object {

    private Customer customer;

    private static final String[]
        pricingPlans = { "string_interfaces.PricingPlan20",
                         "string_interfaces.PricingPlan1510",
                         "string_interfaces.PricingPlan8",
                         "string_interfaces.PricingPlan10"
        };
```

당신은 getCommissionPrice() 메소드를 다음처럼 변경할 수 있다.

Class Name: Trade

```
public double getCommissionPrice()
{
    double commissionPrice = 0.0;

    IPricingPlan plan;
    Class commissionClass;

    try
    {
        commissionClass =
        Class.forName( pricingPlans[ getCustomer().getPlanId() - 1 ] );

        plan = (IPricingPlan) commissionClass.newInstance();

        commissionPrice = plan.calculateCommission( this );
    }
    // ClassNotFoundException, InstantiationException, IllegalAccessException
    catch( Exception e )
    {
        System.out.println( "Exception occurred: " + e.getMessage() );
        e.printStackTrace();
    }
    return commissionPrice;
}
```

예외 처리를 신경쓰지 않는다면 이 코드는 이전에 본 코드보다 훨씬 간결하다. 또한 새로운 요금체계를 추가하는 것도 상대적으로 쉽다. 당신은 Trade 클래스의 배열에 요금체계를 간단히 생성하고 추가한다.

나는 당신이 동적 클래스 로딩을 이용한 프로그래밍의 강력함을 알게되길 원한다. 이제 기어를 바꾸고 깨끗한 코드를 위해 설계를 새로할 것이다(리팩토링에 대한 정보를 원한다면 리소스 절을 보라)

새로운 요금체계를 훨씬 쉽게 추가하도록 설계를 향상시킬수 있다. 새로운 요금체계가 추가될때마다 Trade 클래스를 포함한 소스 파일들을 여전히 재컴파일 해야한다.

데이터베이스/XML 기반 클래스 이름들

클래스 이름을 데이터베이스나 XML 파일 , 일반 텍스트 파일에 저장해야 한다면 어떤 일이 일어날지를 상상해보라. 프로그램이 찾을 수 있는 위치에 존재한다면 간단히 새로운 클래스를 생성함으로써 요금체계를 추가하고 데이터베이스 테이블이나 파일에 기록을 추가할 수 있다. 그리고 새로운 요금체계가 생길때마다 Trade 클래스를 변경시킬 필요가 없다. 가장 쉬운 코드와 간단한 예제를 위해 여기서는 일반 텍스트 파일을 사용할 것이다. 실제 시스템에서는 확장성을 위해 데이터베이스나 XML파일을 이용하기를 권장한다. 텍스트 파일은 다음과 같다.

File Name: PricingPlans.txt

```
1,string_interfaces.PricingPlan20
2,string_interfaces.PricingPlan1510
3,string_interfaces.PricingPlan8
4,string_interfaces.PricingPlan10
```

이제 넘겨진 PlanId 와 일치하는 IPricingPlan 인스턴스를 리턴하는 PricingPlanFactory 클래스를 생성해야 한다. 이 클래스는 텍스트 파일을 읽어서 Map 으로 파싱하므로 PlanId 로 훨씬 쉽게 찾을 수 있다. 그러나 클래스 외부 변경 없이 데이터베이스나 XML 파일을 사용하려면 PricingPlanFactory 클래스를 변경해야 한다.

PlanId 대신 IPricingPlan 인스턴스를 리턴하도록 Customer 클래스를 재구성해야 한다. 이러한 설계는 다른 클래스들은 PricingPlanFactory() 메소드에게 PlanId 를 보내야 한다는 것을 알지 못하므로 PlanId 를 리턴하는 Customer 클래스보다 훨씬 좋다. 이 클래스들은 필요에 따라

간단히 IPricingPlan 인스턴스를 사용할 수 있으므로 PricingPlanFactory 에 대해 아는게 아무것도 없을 수 있다. (나의 요점을 설명하기에 쉽다고 느꼈기 때문에 이전에는 형편없는 설계를 사용했다.)

이러한 변경점은 이 기사의 소스코드의 pricing_plan_factory 패키지에 있다. 다운로드 하려면 리소스를 체크하라.

Take note

이 기사의 DynamicJavaSource.zip 소스코드는 각 패키지안에 Test 클래스를 포함한다. 다음 표는 다음 패키지 안에서 찾을 수 있는 것을 설명한다.

패키지	설명
no_interfaces	인터페이스를 사용하지 않은 예제
hard_coded_interfaces	클래스 이름을 직접 소스코드에 코딩했지만 인터페이스를 사용한 예제
string_interfaces	인터페이스를 사용하고 클래스 이름을 소스코드내에 스트링으로 사용한 예제
pricing_plan_factory	클래스 이름을 얻는데 텍스트 파일을 사용한 예제

표 1. 소스코드 안에 포함되어 있는 패키지들

클래스 로더시에 발생할 수 있는 문제점이 하나 있다. 때때로 클래스 로더는 원하는 대로 동작하지 않는다. 예를 들어 forName() 메소드를 호출하는 클래스가 확장된다면 CLASSPATH 가 동적으로 로딩할 클래스를 찾지 못할 수도 있다. 이 문제에 대해 좀 더 깊이 논의해보고 싶거나 ClassNotFoundExeptions 가 발생한다면 리소스의 Class.forName()에 대한 정보를 살펴보라.

어플리케이션에 탄력을 주자

지금 현재 당신은 유연성 있는 프로그램을 만들수 있는 충분한 지식(인터페이스 사용하는 법과 동적 클래스 로딩)을 가지고 있다. 예를 들어 , 인터페이스를 사용하지 않고 텍스트 파일을 사용해서 코드에 새로운 기능을 추가하는 방법을 보여줄 수 있다. 코드로 실험해 보고 확장하는

방법을 생각해보라. 여기서 당신은 당신의 소스코드 없이 사용자들(다른 프로그래머 , 다른 부서 , 외부 사용자)이 기능을 추가하는 프로그램을 만들 수 있다.

Resources

?? 소스코드 다운로드
http://www.javaworld.com/jw-08-2001/extend/DynamicJavaSource.zip

?? 리팩토링에 대한 마틴폴러 홈페이지

http://www.refactoring.com/

?? 확장에 대한 썬사 정보
http://java.sun.com/products/jdk/1.3/docs/guide/extensions/index.html

?? 클래스 로딩에 대한 더 많은 팁을 원한다면 Jason Marshall's "Java Tip 95: Load Classes Behind the Scenes"을 읽어보라
http://www.javaworld.com/javatips/jw-javatip95.html

?? In "Make Bad Code Good," (*JavaWorld,* March 2001), John Farrell suggests practical steps for revising, refactoring, and rebuilding bad code:
http://www.javaworld.com/jw-03-2001/jw-0323-badcode.html

?? Browse the *JavaWorld* Topical Index:
http://www.javaworld.com/javaworld/topicalindex/jw-ti-index.html

?? Subscribe to the *JavaWorld This Week* weekly email newsletter to find out what's new on *JavaWorld*:
http://www.idg.net/jw-subscribe

?? You'll find a wealth of IT-related articles from our sister publications at IDG.net