Cephe Yönelimli Programlama(Aspect Oriented Programming(AOP))

Temel Kavramlar

Cephe Yönelimli Programlama'yı anlamak için, öncelikle **cross cutting concerns** kavramını anlamak gerekmektedir. Her projede, belli miktarda kod parçası, birden çok sınıf içinde veya modüller içerisinde kullanılmaktadır. Örneğin, **Log** tutmak için gereken kodlar, neredeyse tüm sınıflarda **aynen** kullanılmaktadır.

Bu tarz kodlardan dolayı, yazılan kodların **re-usability(yeniden kullanılabilirlik)** , **maintainability(bakımı)** ve **scalability(ölçeklenebilirlik)** gibi özellikleri **azalmaktadır**. Örneğin, bir sınıf yarattıktan sonra o sınıfı, loglamanın ihtiyaç duyulmadığı yerde kullanmak istediğimizde, bu sınıfı değiştirmemiz gerekecektir. Benzer şekilde, loglama ile ilgili kodlar değiştiği zaman, bu kodları hangi sınıflarda kullandıysak, hepsini değiştirmemiz gerekecektir.

Bir sınıfın, sadece ana fonksiyonları gerçekleştirmesi gerekir. **Loglama, validation, transaction** gibi işlemler, sınıfın fonksiyonlarından **değildir!** Aspect Oriented Programlama **loglama, validation, transaction** gibi bir çok sınıf tarafından ortak kullanılan **cross cutting concerns** özelliğe sahip işlemlerin, yazmış olduğumuz sınıflara eklenmeden kullanılmasını sağlamak için yol, yöntem sağlar.

Cephe Yönelimli Programlama dilinde kullanılan **anahtar kelimeler şunlardır:**

JoinPoint: Uygulamamız **AOP** yapısıyla entegre olsun ya da olmasın, uygulamamızdaki herhangi bir metodu temsil eder. Yani **AOP'tan** bağımsız metodlardır.

Advice: JoinPoint metodu çağrıldığı zaman, çalışan kodtur. Bu kod **JoinPoint** metodunun içine eklenmez. Bunun yerine ayrı bir metod yazılır ve onun içine eklenir. Daha sonra bu metod, **JoinPoint** metodundan önce veya sonra çalışması sağlanır.

Aspect: Advice kodunu tutan sınıftır.

PointCut: JoinPoint metodlarından **önce, sonra veya önce-sonra** hangi **Advice** kodunun çalışması gerektiğini belirleyen bir **ifadedir**. Bu ifadeyi **SQL** cümlesi veya **Regular expression**'a benzetebiliriz.

Örnek:

```
1  import org.aspectj.lang.annotation.Aspect;
2  import org.aspectj.lang.annotation.Before;
3
4  @Aspect
5  public class EmployeeAspect {
6
7      @Before("execution(public String getName())")
8      public void getNameAdvice(){
9          System.out.println("Executing Advice on getName()");
10     }
11 }
```

PointCut: `execution(public String getName())` ifadesidir.

Advice: `getNameAdvice()` metodudur.

JoinPoint: `public String getName()` metodudur.

Aspect: EmployeeAspect sınıfıdır.

Not: @Before annotasyonu ile `public String getName()` metodundan **önce getNameAdvice()** metodunun çalışması sağlanmıştır.

Cephe Yönelimli Programlamanın Java Dili İle Uygulanması

Aspect Oriented Programlamayı anlamak için **Proxy Tasarım Desenini** (http://www.softwarevol.com/tutorial/Vekil-Proxy-Tasarim-Deseni) bilmek gereklidir. Çünkü **AOP** temelde **Proxy** yapısını kullanır. İki tür proxy vardır:

Static Proxy: Her sınıf için bir proxy nesnesi yaratılır. Kullanışlı olmadığı için **AOP** yapısında kullanılmaz.

Dynamic Proxy: Reflection (http://stackoverflow.com/questions/37628/what-is-reflection-and-why-is-it-useful) kullanarak proxy sınıfları dinamik olarak yaratılır. Bu özelliklik Java **JDK 1.3** ile gelmiştir.

Not: Dinamik proxy, **Spring AOP** yapısının temelini oluşturur.

Örnek:

Bir metodun çalışma süresinin hesaplanmasını 3 farklı yolla yapabiliriz: Klasik yöntem, Java JDK'nın dinamik proxy yöntemi ve CGLib kütüphanesinin proxy yöntemi.

Yöntem 1: **method1()** isimli metodun kaç saniye çalıştığını öğrenmek için şu şekilde kod yazabiliriz:

`BasicFunction` isminde bir interface:

```
1  public interface BasicFunction {
2      void method1();
3  }
```

`BasicFunction interface` implement eden bir sınıf:

```
1  public class Example1 implements BasicFunction {
2      @Override
3      public void method1() {
4          long startTime=System.currentTimeMillis();
5          System.out.println("Function1's method1 method runs...");
6          long endTime=System.currentTimeMillis();
7          System.out.println("Elapsed time in millis: "+endTime-startTime);
8      }
9  }
```

Eğer sadece **method1()** değilde yüzlerce metodun kaç saniye çalıştığını yazdırmak isteseydik, `System.currentTimeMillis()` kodunu her metod içerisinde yazmamız gerekecekti. Sonuçta yüzlerce kez aynı kod tekrarlanmış olacaktı.

Yöntem 2: **Proxy** nesnesi yaratarak gereksiz kod yazmayı engelleyebiliriz. **Proxy** nesnesi yaratmak için sırasıyla öncellikle **InvocationHandler** interface'ini implement eden bir sınıf yaratırız. Daha sonra `Proxy.newProxyInstance()` metodunu kullanarak **Proxy** nesnesini yaratırız.

Adım 1: **InvocationHandler interface** implement eden bir sınıf yaratılır:

```
1  public class MyInvocationHandler implements InvocationHandler {
2
3      private Object target;
4
5      public MyInvocationHandler(Object target) {
6          this.target = target;
7      }
8
9      public Object getTarget() {
10         return target;
11     }
12
13     public void setTarget(Object target) {
14         this.target = target;
15     }
16
17     @Override
18     public Object invoke(Object proxy, Method method, Object[] params)
19             throws Throwable {
20         long a = System.currentTimeMillis();
21         Object result = method.invoke(target, params);
22         System.out.println("total time taken  " +
23                 (System.currentTimeMillis() - a));
24         return result;
25     }
26
27 }
```

Not: **Dinamik proxy** nesnesine yapılan tüm metod çağırma işlemleri **invoke()** metodundan geçer. **invoke()** metodunun ilk parametresi olan **proxy** parametresinin değeri dinamik olarak yaratılan **Proxy** sınıfının nesnesidir, yani aşağıda tanımlanan sınıftaki **proxied** isimli nesnedir. Genelde bu nesne **invoke()** metodu içerisinde kullanılmaz. **invoke()** metodunun üçüncü parametresi ise, **BasicFunction interface** içerisinde tanımlanan metodun parametre(lerini) temsil eder. Eğer bu parametreler primitive tip(**int,** double, **char** vs) ise bu tiplerin sınıf karşılıkları(**int** tipi için Integer, **double** için Double vs) parametre olarak geçirilir. **MyInvocationHandler** sınıfının parametreli constructor'u parametre olarak, aşağıda tanımlanan **MainClass** sınıfında gösterildiği gibi **BasicFunction** sınıfını implement eden sınıftan yaratılan nesneyi alır.

Adım 2: **Proxy** nesnesi yaratılır:

```java
public class MainClass {
    public static void main(String[] args) {

        Example1 ex = new Example1();
        //BasicFunction Proxy
        BasicFunction proxied =(BasicFunction)Proxy
                .newProxyInstance(MainClass.class.getClassLoader(),
                        ex.getClass().getInterfaces() ,new MyInvocationHandler(ex));
        proxied.method1();
    }
}
```

getClassLoader() metodu **ClassLoader** nesnesi dönderir. Bu nesne dinamik olarak yaratılacak **Proxy** sınıfını, JVM'ye yükler. **newProxyInstance()** metodunun ikinci parametresi **interface** array'i alır. Bu parametre değerine göre, dinamik olarak yaratılacak **Proxy** sınıfı **interface** array'inde bulunan tüm interface'leri implement eder. Son parametre ise **InvocationHandler** interface'ini implement eden bir sınıf nesnesi alır.

Not: ex nesnesinin yaratıldığı sınıf, hangi interface'leri implement etmişse, o interface'leri **newProxyInstance()** metodunun ikinci parametresi olarak vermemiz gereklidir. Eğer bu sınıf herhangi bir interface'i implement etmemiş olsaydı, **ClassCastException** hatası meydana gelirdi.

Yöntem 3: CGLib kütüphanesini kullanarak, **Yöntem 2**'de belirtilen **interface** kullanım zorunluluğunu ortadan kaldırabiliriz. **Spring AOP,** hem Java'nın **default** olarak sunmuş olduğu dinamik proxy yöntemini(**Yöntem 2**) kullanarak hem de **CGLib** kütüphanesini kullanarak proxy nesnelerini üretir.

CGLib Kütüphanesi Örneği:

Kendisinden dinamik proxy yaratılacak sınıf:

```java
public class Algorithm {
    public void runAlgorithm() {
        System.out.println("running the algorithm");
        try {
            // do something  here -
            // simulate some real time-consuming operation here
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Yöntem 2'de bahsedilen **InvocationHandler** interface'i yerine **CGLib** kütüphanesinde kullanılan **MethodInterceptor** interface'ini implement eden sınıf:

```java
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;

import java.lang.reflect.Method;

public class MyInterceptor implements MethodInterceptor {
    // the real object
    private Object realObj;

    // constructor - the supplied parameter is an
    // object whose proxy we would like to create
    public MyInterceptor(Object obj) {
        this.realObj = obj;
    }

    // this method will be called each time
    // when the object proxy calls any of its methods
    public Object intercept(Object o,
                            Method method,
                            Object[] objects,
                            MethodProxy methodProxy) throws Throwable {
        // just print that we're about to execute the method
        System.out.println("Before");
        // measure the current time
        long time1 = System.currentTimeMillis();
        // invoke the method on the real object with the given params
        Object res = method.invoke(realObj, objects);
        // print that the method is finished
        System.out.println("After");
        // print how long it took to execute the method on the proxified object
        System.out.println("Took: " + (System.currentTimeMillis() - time1) + " ms");
        // return the result
        return res;
```

```
    }
}
```

Main Sınıfı:

```
import net.sf.cglib.proxy.Enhancer;
public class Main {
    public static void main(String[] args) {
        // 1. create the 'real' object
        Algorithm alg = new Algorithm();
        // 2. create the proxy
        Algorithm  proxifiedAlgorithm = createProxy(alg);
        // 3. execute the proxy - as we see it has the same API as the real object
        proxifiedAlgorithm.runAlgorithm();
    }
    // given the obj, creates its proxy
    // the method is generified - just to avoid downcasting...
    public static <T> T createProxy(T obj) {
        // this is the main cglib api entry-point
        // this object will 'enhance' (in terms of CGLIB) with new capabilities
        // one can treat this class as a 'Builder' for the dynamic proxy
        Enhancer e = new Enhancer();

        // the class will extend from the real class
        e.setSuperclass(obj.getClass());
        // we have to declare the interceptor  - the class whose 'intercept'
        // will be called when any method of the proxified object is called.
        e.setCallback(new MyInterceptor(obj));
        // now the enhancer is configured and we'll create the proxified object
        T proxifiedObj = (T) e.create();
        // the object is ready to be used - return it
        return proxifiedObj;
    }
}
```

Bu kodları çalıştırdığımızda ekran çıktısı:

```
Before
running the algorithm
After
Took: 500 ms
```

Not: CGLib kütüphanesini kullandığımız zaman, sınıfların herhangi bir interface'i implement etmelerine **gerek kalmamaktadır**. Bundan dolayı **Spring AOP**'ta **CGLib** kütüphanesi kullanılarak, interface'i olmayan sınıflardan da dinamik proxy üretilmesi sağlanmıştır.

Dinamik Proxy'nin Kullanıldığı Yerler

1. Veritabanı bağlantısı ve Transaction Management
2. Unit testing için dinamik Mock nesneleri
3. Dependency Injection yapısında kullanılan Custom Factory Interface
4. AOP-like Method Interception

Sonuç: Dinamik proxy kullanarak **cross cutting concerns** probleminin nasıl üstesinden gelindiğini açıkladık. **Aspect Oriented Programlama**, **kurumsal** projelerin olmazsa olmazlarından bir tanesidir. Bundan dolayı bu programlama yapısını anlamak son derece önemlidir.