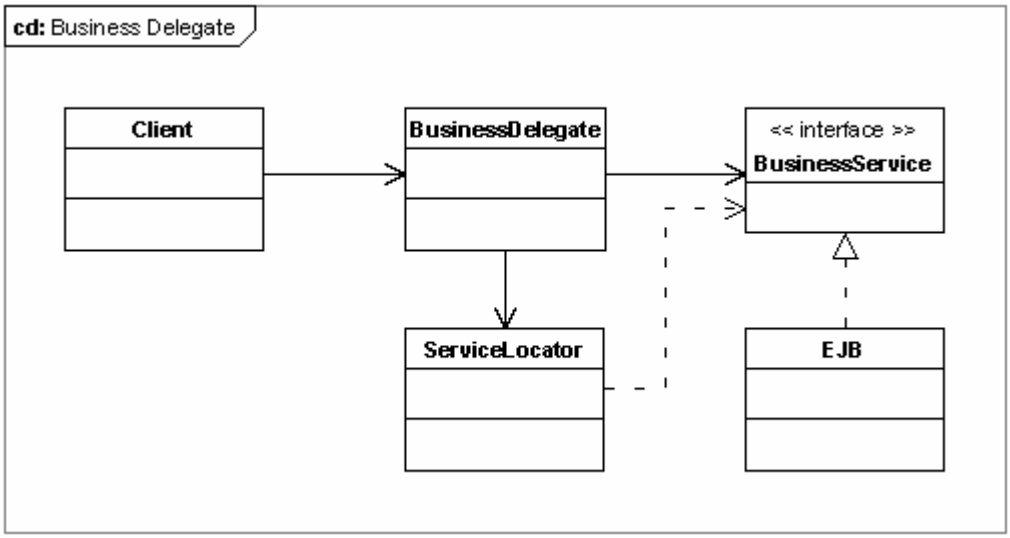
Business Delegate Tasarım Şablonu

KurumsalJava.com

Özcan Acar
Bilgisayar Mühendisi
http://www.ozcanacar.com

Modern yazılım sistemleri birden fazla katmandan oluşur. Bu katmanlar her zaman aynı server üzerinde mevcut olmayabilir. Bu durumda bir katmandan diger katmana ulaşmak için remote call olarak isimlendirilen RMI[1] operasyonları yapılır. Örneğin EJB teknolojisi ile hazırlanan komponentler birden fazla server üzerinde hizmet sunabilir. Bu komponentlere bağlanıp, işlem yapabilmek için RMI kullanılır.

EJB sistemlerinde bazı işlemler bilgisayar ağı üzerinden erişim gerektirebileceği için, zamanla sistem performansı negatif etkilenebilir. Bunun yanısıra gösterim katmanında bulunan sınıflar direk EJB komponentler ile interaksiyona girdikleri taktirde, gösterim katmanı ile EJB'lerden oluşan İşletme (business) katmanı arasında sıkı bir bağ oluşur. EJB komponentler üzerinde yapılan değişiklikler gösterim katmanını etkiler. Bu bağı azaltmak ve RMI performansını artırmak için Business Delegate tasarım şablonu kullanılır.



Business Delegate tasarım şablonu ile, kompleks yapıda olabilecek işletme katmanı ile gösterim katmanı arasına, gösterim katmanının isteklerini işletme katmanına delege edecek BusinessDelegate isminde bir sınıf yerleştirilir. Bu sınıfın öncelikli görevi, işletme katmanında yer alan EJB komponentlerin lokalizasyonu için gerekli lookup işlemlerini gösterim katmanı için transparen hale getirmektir. Bu işlem için BusinessDelegate, Service Locator tasarım şablonundan faydalanır. Bunun yanısıra BusinessDelegate İşletme katmanında oluşan tüm hataları (exception handling) yakalayarak, gösterim katmanı için daha anlaşılır bir hale getirir. Örneğin EJB komponentleri ile çalışırken java.rmi.RemoteException oluşabilir. Gösterim katmanının catch(RemoteException) şeklinde oluşan remote hataları yakalamaya çalışması, gösterim katmanını EJB teknolojisine bağımlı kılar. BusinessDelegate kullanıldığı taktirde, servis katmanı ile gösterim katmanı arasında "loose coupling" olarak tabir edilen esnek bir bağ oluşur.

Business Delegate tasarım şablonunun beraberinde getirdiği diğer bir avantaj ise, bir caching (hızlı önbellek) mekanızması kullanarak ağ (network) üzerinden yapılan işlemlerin neticelerini saklamak ve tekrar RMI operasyonuna gerek kalmadan gösterim katmanı tarafından kullanılmasını sağlıyor olmasıdır. Gerekli verilerin RMI üzerinden başka bir

[1] Remote Method Invocation. Bakınız:
http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp

serverden edinilmesi yerine BusinessDelegate sınıfının sahip olduğu caching sisteminden alınması, sistem performansını büyük ölçüde artırır.

Uml diagramında da görüldüğü gibi BusinessDelegate sınıfı Service Locator tasarım şablonunu kullanarak, işletme katmanında bulunan EJB ve diğer servis veren sınıfların lokalize eder.

Business Delegate tasarım şablonunun nasıl kullanılabileceğini bir EJB örneğinde görelim. İlk işlem olarak EJB komponentini oluşturuyoruz.

```
package org.javatasarim.pattern.businessdelegate.ejb;

import javax.ejb.Stateless;

/**
 * Stateless EJB komponenti.
 *
 * @author Oezcan Acar
 *
 */
@Stateless
public class ServiceBean implements ServiceBeanRemote
{
	public String getValue()
	{
		return "value";
	}
}
```

Servis sunucusu olan ServiceBean stateless tipi bir EJB komponentidir. @Stateless annotasyonu kullanılarak herhangi bir java sınıfı stateless komponent olarak tanımlanabilir. ServiceBean sınıfı bünyesinde bulunan getValue() metodu, gösterim katmanı tarafından kullanılacak olan metoddur.

Bir EJB komponente RMI üzerinden ulaşabilmek için bir Remote Interface sınıfının tanımlanması gerekmektedir:

```
package org.javatasarim.pattern.businessdelegate.ejb;

import java.io.Serializable;

import javax.ejb.Remote;

/**
 * Service için remote interface sinifi.
 *
 * @author Oezcan Acar
 *
 */
@Remote
public interface ServiceBeanRemote extends Serializable
{
	public String getValue();

}
```

EJB komponenti kullanmak isteyen bir sınıf, ServiceBeanRemote interface sınıfını implemente eden bir stub nesne edinmek zorundadır. Bu stub nesne, EJB komponentin deploy

edildiği Container tarafından (Jboss, Weblogic vs...) otomatik olarak oluşturulur. Bu stub nesnesini proxy [2] olarak düşünebilirsiniz. Container tarafından oluşturulan bu proxy otomatik olarak bizim sunduğumuz ServiceBeanRemote interface sınıfını implemente eder. Kullanıcı sınıf bu proxy nesnesinin getValue() metodunu kullandığı zaman, proxy bu isteği gerçek EJB komponentine yönlendirir. @Remote annotasyonunu kullanarak, herhangi bir interface sınıfını EJB Remote Interface haline getirebiliriz.

Şimdi Business Delegate tasarım şablonun uygulandığı BusinessDelegate sınıfını inceliyelim:

```
package org.javatasarim.pattern.businessdelegate;

import org.javatasarim.pattern.businessdelegate.ejb.
            ServiceBeanRemote;

/**
 * Business Delegate Sinifi
 *
 * @author Oezcan Acar
 *
 */
public class BusinessDelegate
{
      /**
       * Kullanılan ejb komponenti
       */
      private ServiceBeanRemote service;

      /**
       * Konstruktör bünyesinde
       * ServiceLocator sinifi kullanilarak
       * ejb komponenti olusturuluyor.
       */
      public BusinessDelegate()
      {
            service = ServiceLocator.instance().
                  getService("ServiceBean/remote");
      }

      /**
       * getValue() metodu ejb komponentine
       * delege ediliyor.
       *
       * @return String value
       */
      public String getValue()
      {
            return service.getValue();
      }
}
```

BusinessDelegate basit bir POJO[3] sınıftır. Gösterim katmanı tarafından gelen istekleri, EJB sınıfına delege edebilmesi için bünyesinde ServiceBeanRemote tipinde bir değişken barındırır. getValue() metodunda görüldüğü gibi, EJB komponentin getValue() metodunu kullanarak, gösterim katmanından gelen isteği EJB komponentine delege eder. Konstruktör içinde ServiceLocator kullanılarak, EJB remote interface nesnesinin nasıl oluşturulduğunu

[2] Proxy tasarım sablonuna bakınız.
[3] POJO - Plain Old Java Object: Basit bir Java sınıfı anlamında.

görmekteyiz. BusinessDelegate sınıfı, ServiceLocator aracılığıyla kullanmak istediği EJB komponenti lokalize eder.

```
package org.javatasarim.pattern.businessdelegate;

import java.util.HashMap;
import java.util.Map;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import org.javatasarim.pattern.businessdelegate.ejb.
            ServiceBeanRemote;

/**
 * Bir servisi lokalize etmek için kullanılan
 * ServiceLocator komponenti
 *
 * @author Oezcan Acar
 *
 */
public class ServiceLocator
{
      /**
       * Olusturulan ejb remote interface
       * nesnei cache içinde tutulur. Böylece
       * her defasinda lookup işlemi yapilmak
       * zorunda kalinmaz.
       */
      private Map<String, ServiceBeanRemote> cache;

      private InitialContext ctx;

      /**
       * Ssingleton tasarım Şablonunu kullanarak
       * sistemde tek bir ServiceLocator
       * olacak sekilde olusturuyoruz.
       */
      private static ServiceLocator locator =
            new ServiceLocator();

      private ServiceLocator()
      {
            try
            {
                  /**
                   * Lookup için gerekli ctx
                   * ve remote interface
                   * nesnelerin tutuldugu cache
                   * nesnesini olusturuyoruz.
                   */
                  ctx = new InitialContext();
                  cache = new HashMap<String,
                                    ServiceBeanRemote>();
            }
            catch (NamingException e)
            {
                  throw new RuntimeException(e);
            }
            catch (Exception e)
            {
                  throw new RuntimeException(e);
            }
```

```
	}

	public static ServiceLocator instance()
	{
		return locator;
	}

	/**
	 * Lookup ismi verilen bir EJB komponentini
	 * geri verir.
	 * @param lookup jndiname
	 * @return Service service
	 */
	public ServiceBeanRemote getService(String lookup)
	{
		ServiceBeanRemote remote = null;
		if(cache.containsKey(lookup))
		{
			remote = (ServiceBeanRemote)
					cache.get(lookup);
		}
		else
		{
			try
			{
				remote = (ServiceBeanRemote)
						ctx.lookup(lookup);
				cache.put(lookup, remote);
			}
			catch (NamingException e)
			{
				throw new RuntimeException(e);
			}
		}
	    return remote;
	}
}
```

Bir EJB komponenti kullanabilmek için InitialContext sınıfı yardımıyla lookup işlemi yapılmak zorundadır. Bu bir nevi mahalleye gelen bir yabancının “Ahmet beyin evini arıyorum, 35 nolu evde oturuyormuş” söylemine eşittir. Lookup işlemi ile, ismi belli olan bir EJB komponent lokalize edilirek EJB’nin sahip olduğu remote interface nesnesi elde edilir. ServiceLocator sınıfı EJB komponentini lokalize etmek için kullanılır. ServiceLocator lookup için gerekli tüm metodları ihtiva ettigi için, BusinessDelegate sınıfının spesifik lookup operasyonlar ile ugraşma zorunluğuğu ortadan kalkmaktadır.

ServiceLocator bünyesinde bir cache oluşturarak, her defasından lookup işleminin yapılmasını engellemiş oluyoruz. Her lookup, RMI üzerinden başka bir servere bağlantı yapmak anlamına geldiği için, cache içinde bulunan bir remote interface nesnesini direk cacheden alıp kullanmak, performansı artıracaktır.

ServiceLocator sınıfını Singleton Tasarım şablonunu kullanarak implemente ediyoruz, çünkü tüm sistem bünyesinde sadece bir ServiceLocator ve bununla bağlantılı olarak sadece bir cache’in olması gerekmektedir.

```java
package org.javatasarim.pattern.businessdelegate;

/**
 * Test sinifi
 *
 * @author Oezcan Acar
 *
 */
public class Test
{
	/**
	 *
	 * Bu test sinifi sadece örnek olarak
	 * programlanmistir. EJB 3 komponent
	 * kullanildigi için, test sinifinin
	 * calismasi imkansizdir. Örneğin
	 * calistirilabilmesi için EJB komponentin
	 * JBOSS 4 gibi bir application server
	 * üzerinde deploy edilmesi gerekmektedir.
	 */
	public static void main(String[] args)
	{
		BusinessDelegate delegate = new BusinessDelegate();
		System.out.println(delegate.getValue());
	}
}
```

Test.main() bünyesinde bir BusinessDelegate nesnesi oluşturarak, getValue() metodunu kullanıyoruz. Gösterim katmanını simule eden Test sınıfı, EJB komponentin varlığından bile haberdar olmadan, servis katmanında yer alan bir EJB'nin sunduğu servisi kullanabilmektedir.

Eger Business Delegate tasarım şablonun kullanmasaydık, Test sınıfı aşağıdakı yapıda olurdu:

```java
package org.javatasarim.pattern.businessdelegate;

import javax.naming.InitialContext;
import org.javatasarim.pattern.businessdelegate.ejb.
		ServiceBeanRemote;

public class Test
{
	public static void main(String[] args)
	{
		InitialContext ctx = new InitialContext();
		ServiceBeanRemote remote = (ServiceBeanRemote)
			ctx.lookup("ServiceBean/remote");
		System.out.println(remote.getValue());
	}
}
```

Böyle bir implementasyonun beraberinde getirdigi sorunlar şöyle olacaktır:

- Gösterim katmanı EJB teknolojine bağımlı olacaktır, çünkü RMI operasyonlarını kullanarak EJB komponenti lokalize etmek zorunda.
- Gösterim katmanı oluşacak javax.rmi.RemoteException ile başetmek zorundadır. Servere baglantı koptuğunda, tekrar baglantı kurmak zorundadır.

- Gösterim katmanı caching uygulamadığı için, her getValue() metodunu kullandığında, bu RMI üzerinden EJB komponentin deploy edildigi servere bağlantı kurmak anlamına geldigi için, yapılmak istenen işlemler uzun sürecek ve performans iyi olmayacaktır.

Görüldüğü gibi Business Delegate tasarım şablonu ile hem servis sağlayan katman gösterim katmanı için daha transparen bir hale dönüştürülüyor, hemde RMI operasyonlarında performans artırılabiliyor.

Business Delegate tasarım şablonu ne zaman kullanılır?

- EJB gibi teknolojiler kullandığında, gösterim katmanı ile işletme katmanı arasındaki bağlantı için Business Delegate kullanılır.
- Servis katmanında uluşan hataları, gösterim katmanı için daha anlaşılır bir hale dönüştürmek için Business Delegate kullanılır.
- RMI operasyonlarında performans çok önemli bir unsurdur. Business Delegate caching mekanizmaları implemente ederek, performansı artırır.

İlişkili tasarım şanlonları:

- Business Delegate, servis katmanında bulunan komponentleri lokalize etmek için Service Locator tasarım şablonunu kullanır.
- Business Delegate bir nevi proxy nesnesi oluşturarak, gösterim katmanını, servis katmanı ile bir araya getirir.

EOF (End Of Fun)
Özcan Acar