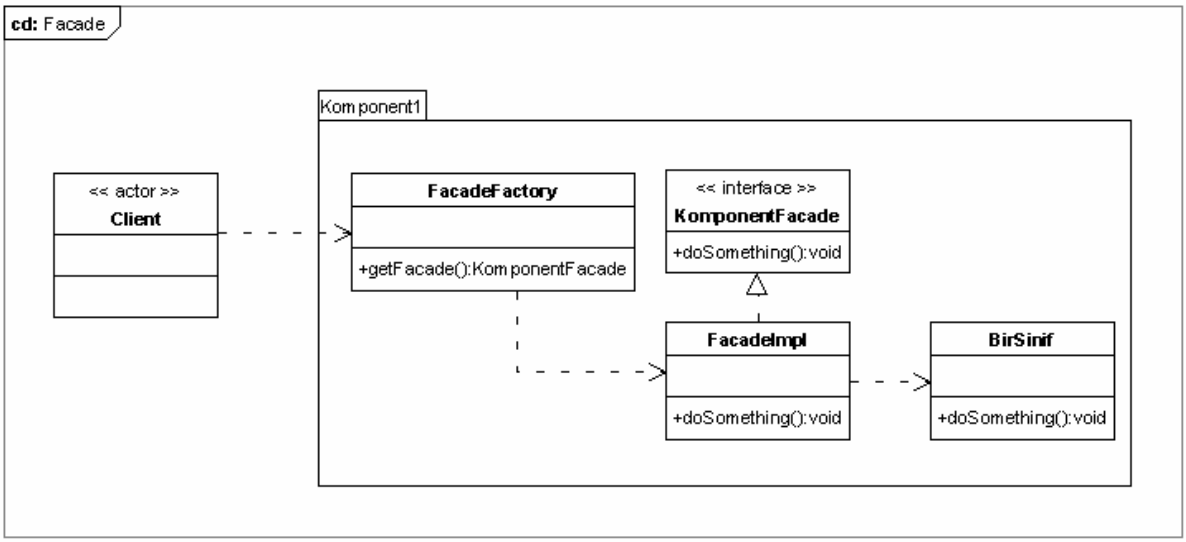
Facade (Cephe) Tasarım Şablonu

KurumsalJava.com

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

Profesyonel yazılım sistemleri birçok komponentin birleşiminden oluşur. Yazılım esnasında bir çok ekip birbirinden bağımsız, sistemin bütününü oluşturan değişik komponentler üzerinde çalışırlar. Bir komponent, belirli bir işlevi yerine getirmek için hazırlanmış bir ya da birden fazla Java sınıfından oluşmaktadır.



Bir komponentin sunmuş olduğu hizmetten yararlanabilmek için, komponentin dış dünya için tanımlamış olduğu giriş/çıkış noktaları (input/output interface) kullanılır. Komponent sadece bu giriş/çıkış noktaları üzerinden dış dünya ile iletişim kurar ve iç dünyasını tamamen gizler. Bu komunikasyon noktaları genelde Facade tasarım şablonu kullanılarak programlanır.

Uml diagramında görüldüğü gibi **Komponent1** isminde bir sistem komponenti, dış dünya ile komunikasyonu **KomponentFacade** interface sınıfı üzerinden sağlıyor. Kullanıcı sınıfın (client) tanıması gereken sınıflar **FacadeFactory** ve **KomponentFacade** sınıflarıdır. **FacadeFactory** ile, kullanıcı sınıfın kullanabileceği şekilde bir **KomponentFacade** nesnesi oluşturulur. Komponentin sunduğu hizmetlere, **KomponentFacade** interface sınıfında tanımlanmış metodlar aracılığıyla ulaşılır. Komponenti kullanmak için tanımlanan giriş noktası **KomponentFacade**.*doSomething()* metodudur.

KomponentFacade sınıfını bir interface olarak tanımlıyoruz. Bu sayede komponentin kullanıldığı yere göre sunduğu hizmet, kullanıcı sınıf (client) etkilenmeden değiştirilebilir hale gelir. Bu amaçla komponent içinde değişik **KomponentFacade** implementasyon sınıfları programlanır. Kullanıcı sınıfın gereksinimleri doğrultusunda **FacadeFactory** tarafından gerekli görülen interface implementasyon nesnesi oluşturulur ve kullanılmak üzere kullanıcı sınıfa verilir. Örneğimizde gördüğümüz gibi, interface sınıfları kullanarak, sistemin parçaları arasında çok esnek bağlar oluşturabiliyoruz. Esnek ve bakımı kolay sistemler oluşturmak için interface sınıfları tercih edilmelidir.

Java dilinde Facade tasarım şablonunu nasıl implemente edebileceğimizi bir örnekle inceliyelim. Önce **KomponentFacade** interface sınıfını tanımlıyoruz:

```
package org.javatasarim.pattern.facade.komponent1;

/**
 * Komponentin sundugu hizmetlerin
 * kullanilmasi için olusturulan
```

```
 * metodlarin bulundugu facade interface
 * sınıfi.
 *
 * @author Oezcan Acar
 *
 */
public interface KomponentFacade
{
	/**
	 * Implementasyon siniflari tarafindan
	 * implemente edilmesi gereken bir
	 * metod.
	 */
	public void doSomething();
}
```

Komponentin sunduğu hizmetlerden yararlanabilmek için öncelikle bir **KomponentFacade** nesnesinin oluşturulması gerekiyor. Bu görevi **FacadeFactory** sınıfı üstleniyor:

```
package org.javatasarim.pattern.facade.komponent1;

/**
 * KomponentFacade olusturmak
 * için kullanilan singleton
 * factory sinifi.
 *
 * @author Oezcan Acar
 *
 */
public class FacadeFactory
{
	/*
	 * Singleton tasarım şablonunu
	 * kullanmak için bu sınıftan bir
	 * degisken tanimliyoruz.
	 */
	private static FacadeFactory instance = new FacadeFactory();

	/**
	 * Singleton olabilmesi için
	 * sınıf konstruktörünün
	 * private olmasi gerekiyor. Bu
	 * durumda baska bir sınıf new FacadeFactory()
	 * seklinde nesne olusturamaz. Amacimizda
	 * bunu engellemek ve bu sınıftan sadece bir
	 * nesnenin sistemde bulunmasini saglamak
	 * (singleton tasarım şablonuna bakiniz)
	 */
	private FacadeFactory()
	{
	}

	/**
	 * Sistemde bulunan tek FacadeFactory
	 * nesnesine ulasmak için instance()
	 * metodu kullanilir.
	 *
	 * @return FacadeFactory singleton nesne
	 */
```

```
	public static FacadeFactory instance()
	{
		return instance;
	}

	/**
	 * Kullanici sinif tarafindan kullanilmak
	 * üzere olusturulan KomponentFacade
	 * implementasyonu.
	 *
	 * @return FacadeImpl facade implementasyon sınıfi
	 */
	public KomponentFacade getFacade()
	{
		return new FacadeImpl();
	}
}
```

FacadeFactory sınıfını singleton olarak implemente ediyoruz. Böylece sistem içinde sadece bir tane **FacadeFactory** nesnesi olmuş oluyor. **FacadeFactory** sınıfının görevi, komponenti kullanan sınıf için bir **KomponentFacade** nesnesi oluşturmaktır. Bunu gerçekleştirebilmek için, komponent içinde **KomponentFacad**e interface sınıfını implemente eden sınıfların bulunması gerekmektedir. **FacadeImp**l isminde bir implementasyon sınıfı aşağıdaki yapıya sahiptir:

```
package org.javatasarim.pattern.facade.komponent1;

/**
 * KomponentFacade interface sinifinin
 * bir implementasyonu.
 *
 * @author Oezcan Acar
 *
 */
public class FacadeImpl implements KomponentFacade
{
	public void doSomething()
	{
		new BirSinif().doSomething();
	}
}
```

FacadeImpl sınıfı **KomponentFacade** interface sınıfında bulunan *doSomething()* metodunu implemente etmek zorundadır. **FacadeImpl** sınıfı implemente ettigi *doSomething()* metodu içinde **BirSinif** sınıfından bir nesne oluşturarak, bu nesnenin **doSomething()** metodunu kullanıyor. Bu sayade, komponenti kullanan sınıfın (Test sınıfi) **KomponentFacade** interface sınıfında bulunan *doSomething()* metodunu kullanmış olması, **BirSinif**.*doSomething()* metoduna kadar iletilmiş oluyor.

Komponenti ve sunduğu hizmeti test etmek için aşağıda yer alan Test sınıfı kullanilabilir:

```
package org.javatasarim.pattern.facade;

import org.javatasarim.pattern.facade.komponent1.FacadeFactory;

/**
 * Test programi.
```

```
 *
 * @author Oezcan Acar
 *
 */
public class Test
{
	public static void main(String[] args)
	{
		FacadeFactory.instance().getFacade().doSomething();
	}
}
```

Test sınıfı, **FacadeFactory**.*instance()* metodunu kullanarak, önce bir **FacadeFactory** nesnesine sahip olur. **FacadeFactory** sınıfı singleton olduğu için, sınıf içinde oluşturulmuş ve bilgisayarın hafızasında bulunan FacadeFactory nesnesi Test sınıfına verilir. **FacadeFactory**.*getFacade()* metodu üzerinden, komponentin sahip olduğu bir **KomponentFacade** implementasyon nesnesi oluşturulur. *getFacade()* metodu içinde new **FacadeImpl()** ile **KomponentFacade** interface sınıfını implemente eden bir nesne oluşturulur. Akabinde **FacadeImpl**.*doSomething()* metodu kullanılarak, komponentin **KomponentFacade**.*doSomething()* metodunda tanımlanmış olan servis, Test sınıfı tarafından kullanılmış olur.

Bu şekilde modellenmiş ve implemente edilmiş bir komponentin bir çok avantajı bulunmaktadır. Bunlar:

- Komponenti kullanan sınıf (Test sınıfi) komponentin sunduğu hizmeti **KomponentFacade** interface sınıfında tanımlanmış metodlar üzerinden sağlar. Kullanıcı sınıfın tanıması gereken sadece **KomponentFacad**e ve **FacadeFactory** sınıflarıdır.
- Komponent, bünyesinde barındırdığı sınıfları, kullanıcı sınıf kodunun tekrar derlenmesine gerek kalmadan değiştirebilir. Komponent ile kullanıcı sınıf arasındaki tek bağ, **KomponentFacade** interface sınıfından tanımlanmış olan metodlardır ve bu metodlar değişmediği sürece, kullanıcı sınıf, komponent içinde yapılan değişikliklerden etkinlenmez.
- Komponent, bünyesinde barındırdığı ve sunduğu hizmeti implemente eden sınıfları dış dünyadan saklar. Örneğin komponent içinde yer alan **BirSinif** sınıfını kullanıcı sınıf (Test) kesinlikle tanımaz ve **new BirSinif()** şeklinde kullanamaz. Eğer Test sınıfı **BirSinif** sınıfını direk kullanabilseydi, bu Test sınıfı ve komponent arasında sıkı bir bağ oluşturur ve komponentin yapısı değiştirildiğinde, Test sınıfınıda negatif etkilerdi.
- Komponent çesitli **KomponentFacade** implemenstasyon sınıfları (örneğin **FacadeImpl**) sunarak, verdiği hizmetin değişik yöntemlerle ve kullanıcı sınıfın ihtiyaçları doğrultusundan oluşturulmasını sağlıyabilir. Bu gibi değişikliklerden kullanıcı sınıf etkilenmez.

Facade tasarım şablonu, daha sonra detaylı olarak inceliyeceğimiz katmanlı mimarilerde, katmanlar arası izolasyonu gerçekleştirmek içinde kullanılır. Her katmanın belirlenmiş bir görevi vardır ve üst katmanlar kendi görevlerini yerine getirmek için bir alt katmanın sunduğu hizmetlerden faydalanırlar. İki katman arasındaki bağı esnek tutmak için, üst katman, alt katmanın sunduğu hizmete, alt katmanda tanımlanmış olan **KatmanFacade** interface sınıflarından erişir. Bir katman içinde meydana gelen değişiklikler, diger katmanları etkilemez ve böylece katmanlar arası izolasyon sağlanmış olur.

Facade tasarım şablonu ne zaman kullanılır?

- Kullanıcı sınıf ve sistemin parçalarını oluşturan alt sistemler (subsystem) ya da komponenter arasındaki bağ, Facade tasarım şablonu ile esnek bir yapıda oluşturulur. Alt sistemlerde oluşan değişikliklerden kullanıcı sınıflar etkilenmez. Komponent ve alt sistem tasarımlarında Facade tasarım şablonu kullanılmalıdır.
- Refactoring[1] yöntemiyle mevcut kod, alt sistem ya da komponent olarak yeniden düzenlenmelidir. Bu sayede görevi ve sunduğu hizmet tanımlanmış ve bakımı kolay komponentler oluşur. Bu komponentlerin yapılandırılmalarında Facade tasarım şablonu kullanılmalıdır.

İlişkili tasarım şablonları:

- Facade yerine Abstract Factory kullanılarak sistem içinde kullanılan nesnelerin oluşumu ve kullanımı saklı tutulabilir.

EOF (End Of Fun)
Özcan Acar

[1] Mevcut bir sınıf yada kodun, yeniden yapılandırılıp, bakımı ve kullanımı daha kolay bir hale getilrilmesi işlemine refactoring adı verilir. Yazılım sistemleri, genelde uygun tasarım şablonları kullanılarak yeniden yapılandırılır (refactor edilir).