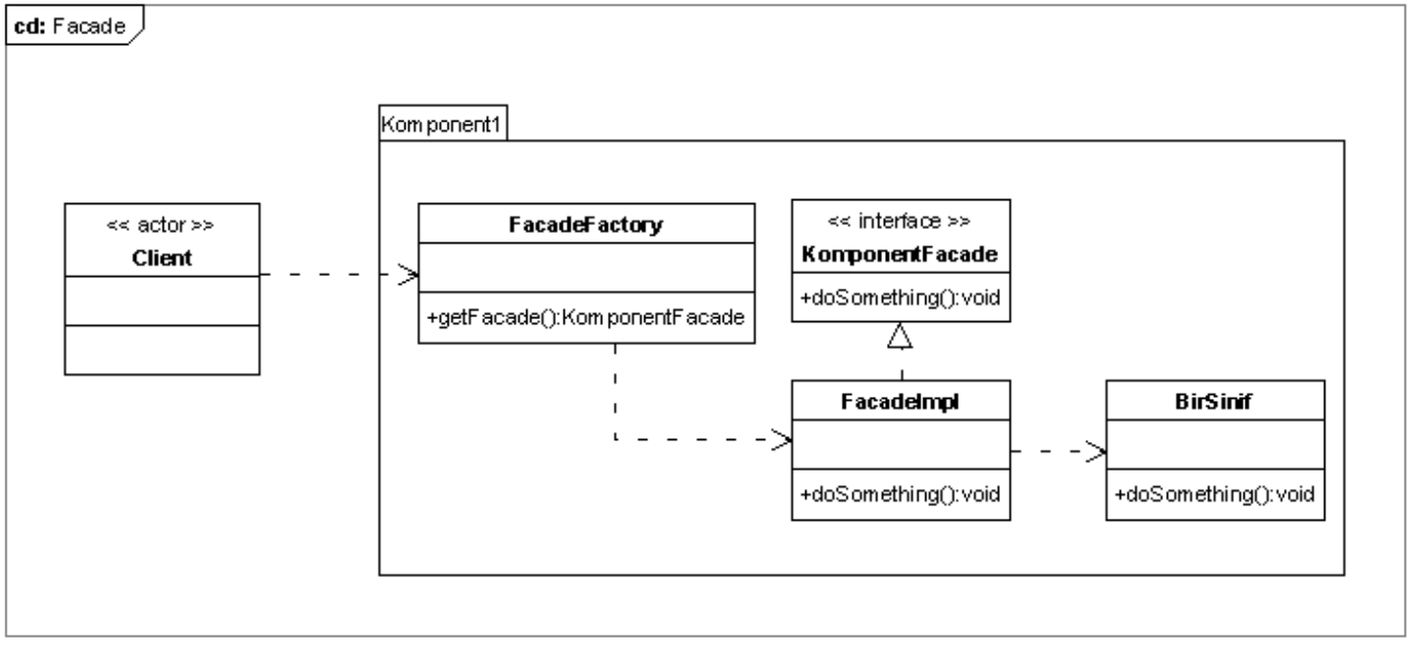
implementasyonların kullanılmasına izin vermektedir.
- Karmaşık sınıf hirarşileri modellemek yerine, köprü tasarım şablonu ile sistem daha esnek ve bakımı kolay hale getirilmek istendiğinde.

İlişkili tasarım şablonları

Abstract Factory köprü oluşumunda kullanılabilir. Adaptör tasarım şablonu ile sistem içinde bulunan sınıflar başka sınıflar tarafından kullanılacak şekilde adapte edilebilir. Adaptör tasarım şablonu, çoğu zaman sistem tasarlandıktan sonra uygulanır. Buna karşın köprü tasarım şablonu, sistemin oluşturulması esnasında, soyut sınıfları ve sahip oldukları implementasyonları ayrıştırmak için kullanılmaktadır.

Cephe (Facade)

Profesyonel yazılım sistemleri birçok komponentin birleşiminden oluşurlar. Yazılım esnasında birçok 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.



Resim 9

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

iletişim noktaları genelde cephe tasarım şablonu kullanılarak programlanır.

Resim 9 da yer alan UML diyagramında görüldüğü gibi Komponent1 isminde bir sistem komponenti dış dünya ile iletişimi KomponentFacade interface sınıfı üzerinden sağlamaktadır. Kullanıcı sınıfın (client) tanıması gereken sınıflar FacadeFactory ve KomponentFacade sınıflarıdır. FacadeFactory ile kullanıcı sınıfın kullanabileceği şekilde bir KomponentFacade nesnesi oluşturulmaktadır. Komponentin sunduğu hizmetlere, KomponentFacade interface sınıfında tanımlanmış metotlar aracılığıyla ulaşılmaktadır. 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 gelmektedir. 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. Örnekte gördüğümüz gibi interface sınıfları kullanarak, sistemin parçaları arasında çok esnek bağlar oluşturabiliyoruz. Esnek ve bakımı kolay sistemler oluşturmak için mutlaka interface sınıfları tercih edilmelidir.

Cephe tasarım şablonunu nasıl implemente edebileceğimizi bir örnekle inceliyelim. Önce KomponentFacade interface sınıfını tanımlıyoruz:

```
// Kod 21

package com.pratikprogramci.designpatterns.bolum6.facade.component;

/**
 * Komponentin sunduğu hizmetlerin kullanılması için oluşturulan metotların
 * bulunduğu facade interface sınıfı.
 *
 */
public interface KomponentFacade {
    /**
     * İmplementasyon sınıfları tarafından implemente edilmesi gereken bir
     * metot.
     */
    public void doSomething();
}
```

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

```
// Kod 22

package com.pratikprogramci.designpatterns.bolum6.facade.component;

/**
 * KomponentFacade oluşturmak için kullanılan singleton factory sınıfı.
 *
 */
public class FacadeFactory {

    /**
     * Singleton tasarım şablonunu kullanmak için bu sınıftan bir değişken
     * tanımlıyoruz.
     */
    private static FacadeFactory instance = new FacadeFactory();

    /**
     * Singleton olabilmesi için sınıf konstruktörünün private olması gerekiyor.
     * Bu durumda başka bir sınıf new FacadeFactory() şeklinde nesne
     * oluşturamaz. Amacımız da bunu engellemek ve bu sınıftan sadece bir
     * nesnenin sistemde bulunmasını sağlamak (singleton tasarım şablonuna
     * bakınız)
     */
    private FacadeFactory() {
    }

    /**
     * Sistemde bulunan tek FacadeFactory nesnesine ulaşmak için instance()
     * metodu kullanılır.
     *
     * @return FacadeFactory singleton nesne
     */
    public static FacadeFactory instance() {
        return instance;
    }

    /**
     * Kullanıcı sınıf tarafından kullanılmak üzere oluşturulan KomponentFacade
     * nesnesi.
     */
    public KomponentFacade getFacade() {
        return new FacadeImpl();
    }
}
```

FacadeFactory sınıfını singleton olarak implemente ediyoruz. Böylece sistem içinde sadece bir tane

FacadeFactory nesnesi bulunmuş oluyor. FacadeFactory sınıfının görevi, komponenti kullanan sınıf için bir KomponentFacade nesnesi oluşturmaktır. Bunu gerçekleştirebilmek için komponent içinde KomponentFacade interface sınıfını implemente eden sınıfların bulunması gerekmektedir. FacadeImpl isminde bir implementasyon sınıfı aşağıdaki yapıya sahiptir:

```
// Kod 23

package com.pratikprogramci.designpatterns.bolum6.facade.component;

/**
 * KomponentFacade interface sınıfının bir implementasyonu.
 *
 */
public class FacadeImpl implements KomponentFacade {

    @Override
    public void doSomething() {
        new BirSinif().doSomething();
    }
}
```

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

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

```
// Kod 24

package com.pratikprogramci.designpatterns.bolum6.facade.component;

/**
 * KomponentFacade interface sınıfının bir implementasyonu.
 *
 */
public class FacadeImpl implements KomponentFacade {

    @Override
    public void doSomething() {
        new BirSinif().doSomething();
    }
```

```
}
```

Test sınıfı FacadeFactory.instance() metodunu kullanarak önce bir FacadeFactory nesnesine sahip olmaktadır. FacadeFactory sınıfı singleton olduğu için sınıf içinde oluşturulmuş ve bilgisayarın hafızasında bulunan FacadeFactory nesnesi Test sınıfına verilmektedir. FacadeFactory.getFacade() metodu üzerinden, komponentin sahip olduğu bir KomponentFacade implementasyon nesnesi oluşturulmaktadır. 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 sınıfı) komponentin sunduğu hizmeti KomponentFacade interface sınıfında tanımlanmış metotlar üzerinden alır. Kullanıcı sınıfın tanıması gereken sadece KomponentFacade 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 metotlardır ve bu metotlar 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 BirSınıf sınıfını kullanıcı sınıf (Test) kesinlikle tanımaz ve new BirSınıf() şeklinde kullanamaz. Eğer Test sınıfı BirSınıf sınıfını doğrudan kullanabilseydi, bu Test sınıfı ve komponent arasında sıkı bir bağ oluşturur ve komponentin yapısı değiştirildiğinde, Test sınıfını da negatif etkilerdi.
- Komponent çeşitli KomponentFacade implemenstasyon sınıfları (örneğin FacadeImpl) sunarak, verdiği hizmetin değişik yöntemlerle ve kullanıcı sınıfın ihtiyaçları doğrultusunda oluşturulmasını sağlıyabilir. Bu gibi değişikliklerden kullanıcı sınıf etkilenmez.

Cephe tasarım şablonu daha sonra detaylı olarak inceliyeceğimiz katmanlı mimarilerde, katmanlar arası izolasyonu gerçekleştirmek içinde kullanılmaktadır. 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, diğer katmanları etkilemez ve böylece katmanlar arası izolasyon sağlanmış olur.

Cephe tasarım şablonu ne zaman kullanılır?

Kullanıcı sınıf ve sistemin parçalarını oluşturan altsistemler (subsystem) ya da komponentler arasındaki bağ, cephe tasarım şablonu ile esnek bir yapıda oluşturulur. Altsistemlerde oluşan değişikliklerden kullanıcı sınıflar etkilenmez. Komponent ve altsistem tasarımlarında cephe tasarım şablonu kullanılmalıdır.

Refactoring yöntemiyle mevcut kod, altsistem ya da komponent olarak yeniden düzenlenmelidir. Bu sayede görevi ve sunduğu hizmet tanımlanmış ve bakımı kolay komponentler oluşur. Bu komponentlerin yapılandırılmalarında cephe tasarım şablonu kullanılmalıdır.

İlişkili tasarım şablonları

Facade yerine abstract factory kullanılarak sistem içinde kullanılan nesnelerin oluşumu ve kullanımı saklı tutulabilir. Facade ile yeni bir interface tanımlanırken, adaptör tasarım şablonu ile mevcut bir interface sınıf kullanılır. Facade sınıfları singleton olarak implemente edilebilirler.

Bileşik (Composite)

Bileşik tasarım şablonu bir sistemin bütünü ve parçaları arasındaki ilişkileri modellemek için kullanılmaktadır. Sistemin bütününü oluşturan parçalar, kendi içlerinde alt parçalardan oluşabilir. Bileşik tasarım şablonu kullanıcı sınıfın, sistem, sistemin parçaları ve alt parçalar arasında ayrım yapmadan nesneleri kullanmasına izin vermektedir. Bu şekilde sistem yazılımı ve kullanımı daha sadeleştirilmektedir.