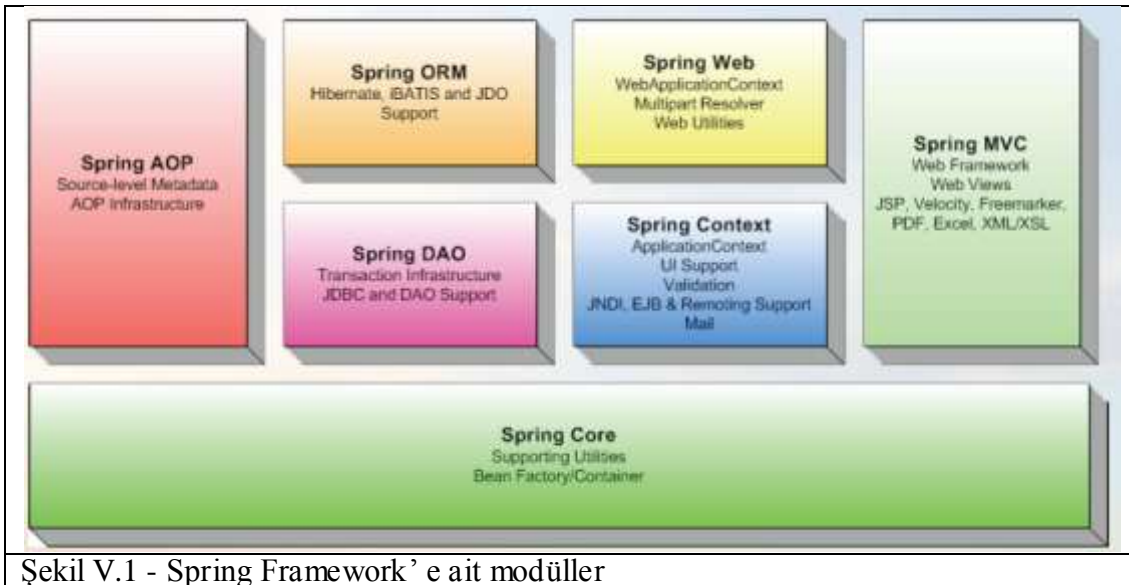


Spring Framework

Spring Framework, Spring Source firmasının geliştirmiş olduğu ve Kurumsal uygulamalara dönük çözümler getiren büyük bir ekosistemdir, altyapıdır. Java platformunun gücü sahip olduğu (*community*) topluluklardan gelmektedir. Spring Source firması bu topluluklardan yalnızca birtanesidir. Spring Framework' ün sunduğu özellikle Inversion of Control (*Kontrolü geliştiriciden Spring'e çevirme*) yaklaşımı ile yazılım geliştiricilere büyük kolaylıklar sağlamaktadır. Dependency Injection (*Bağımlılık zerki*) mekanizması tüm nesnelerin yönetimini kendi üstlenerek, yazılım mühendisliğinde arzu edilen nesneler arası bağlaşımı koparma (*de-coupling*) işlemini esnek bir şekilde karşılamaktadır. Yazılım geliştiricilere sadece, bir XML konfigürasyon dosyasında ya da Notasyon bazlı yapılandırıcılar ile hangi nesnelerin hangi nesne referanslarına bağlanacağını belirtmek kalmaktadır.

Inversion Of Control (*IOC*) ve Dependency Injection (*DI*) kavramları Spring Framework' ün çekirdeğini oluşturmaktadır. Bu çekirdeğin çevresinde kümelenmiş bir çok önemli modül ve mekanizmaları yazılım dünyasına sunmaktadır.



Şekil V.1 - Spring Framework' e ait modüller

Spring Core (*Çekirdek Spring*) modülü haricindeki modülleri kullanmak isteğe bağlıdır. Hiç kullanılmayabilir ya da arzu edilen ve benzer fonksiyonu yerine getiren diğer açık kaynak teknoloji(ler) kullanılabilir.

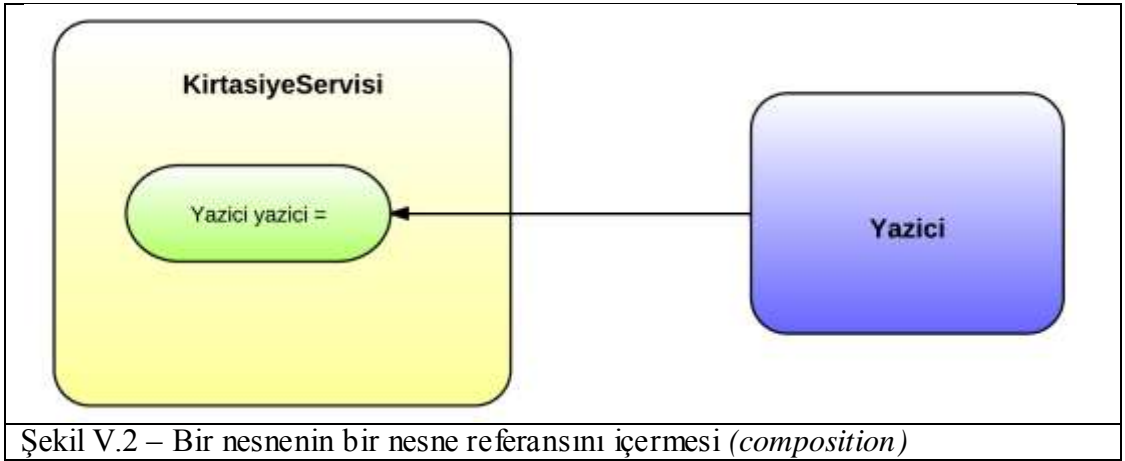
V.1 Inversion Of Control (*IOC*)

Uygulama geliştiriciler bir programlama dili üzerinde uygulama geliştirirlerken, ihtiyaç duydukları nesnelere (*objects*) ait tüm yaşam sürecini kendileri yönetmektedirler. Java programlama dilinde nesneleri oluşturmak ve nesneler arası bağlaşımı/ilişkiyi kurmak yazılım geliştiricinin görevi iken nesneyi sonlandırmak

Çöp Toplayıcı (*Garbage Collector*) ‘ün görevi dahilindedir. Spring Framework kendi konteyner servisleriyle ve IOC (*Inversion Of Control*) mekanizması ile Java nesnelerinin oluşturulması ve yaşam döngüsü tayin etme ve bu yaşam döngüsünü takip etme konusunda tüm sorumluluğu kendi üstüne almaktadır.

V.2 Bağlaşım (*Coupling*) / Bağlaşımı Koparmak (*De-Coupling*)

Nesneye yönelik programlama dillerinde, yazılım geliştiricilerin oluşturduğu nesneler (*objects*), geliştirme zamanında birbirleriyle ilişkilendirilirler. Bu ilişki kalıtım (*inheritance*) ve kompozisyon (*composition*) yöntemlerinden biri ile gerçekleştirilebilmektedir.



Şekil V.2 – Bir nesnenin bir nesne referansını içermesi (*composition*)

Şekil V.2’ de “KirtasiyeServisi” ve “Yazici” adında iki adet Java nesnesi bulunmaktadır. KirtasiyeServisi nesnesi Yazici sınıfı türünde ve yazici adında bir referans bulundurmaktadır. Eğer bu “Yazici” nesnesi “yazici” referansına geliştirici tarafından bağlanacak olursa, kompozisyon yöntemiyle KirtasiyeServisi nesnesine bağlanmış olur.

```
public class Yazici {  
    public void calis() {  
        System.out.println("Yazıcı çalışıyor..");  
    }  
}
```

Şekil V.3 – Yazici sınıfı

Yazici sınıfının içinde bir adet `calis()` yordamı bulunmaktadır. Bu yordam çalıştırıldığında konsol ekranına "Yazıcı çalışıyor.." mesajını ıktılamaktadır.

```
public class KirtasiyeServisi {  
    Yazici yazici;
```

```

public KirtasiyeServisi () {
    yazici=new Yazici ();
}

public void servisYap () {
    yazici.calis ();
}
}

```

Şekil V.4 – YaziciServisi sınıfı

KirtasiyeServisi sınıfı ise, Yazici sınıfını yönetmekten sorumludur.

```

public KirtasiyeServisi () {
    yazici=new Yazici ();
}

```

KirtasiyeServisi sınıfının yapılandırıcısında (*constructor*), **new** Yazici(); ifadesiyle bir adet Yazici nesnesi oluşturulmaktadır. servisYap() yordamı aracılığı ile de KirtasiyeServisi sınıfına dahil edilen Yazici nesnesinin calis() yordamı çağrılmaktadır.

```

public class Main {

    public static void main(String[] args) {

        KirtasiyeServisi servis=new KirtasiyeServisi ();
        servis.servisYap ();

    }
}

```

Şekil V.5 – Kirtasiye servisinin işletilmesi

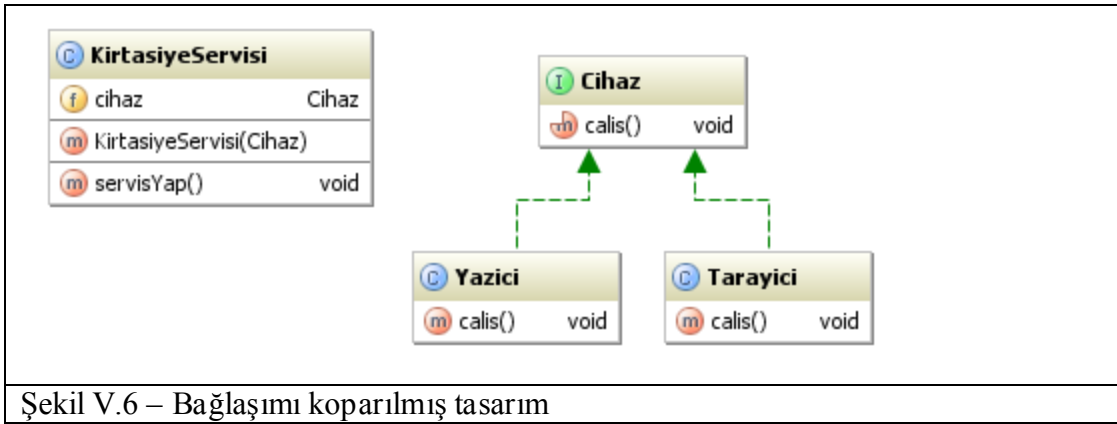
Şekil V.5’ de **new** anahtar kelimesiyle bir KirtasiyeServisi nesnesi oluşturulmaktadır. KirtasiyeServisi nesnesi oluşturulur oluşturmaz da bir Yazici nesnesi oluşturularak, KirtasiyeServisi nesnesinin içine bağlanmaktadır. Bu noktada KirtasiyeServisi nesnesi ile Yazici nesneleri sıkı sıkıya bağlıdır (*tightly-coupled*). KirtasiyeServisi nesnesi oluşturulduğunda Yazici nesnesi oluşturulmakta, KirtasiyeServisi nesnesi öldüğünde Yazici nesnesi de ölmektedir. Dolayısıyla Yazici nesnesinin ömrüne KirtasiyeServisi nesnesi karar vermektedir.

Aynı zamanda bu tasarım, open-closed (*genişlemeye açık – değişime kapalı*) prensibine de aykırıdır. Örneğin geliştirme aşamasında Yazici sınıfı üzerinde calis() yordamının ismi değiştirilmek ya da silinmek ihtiyacı duyulsa, bu değişiklikten YaziciServisi sınıfı da etkilenecektir. Yüzlerce sınıfın birbiriyle ilişkilendirildiği bir yazılım projesi hayal edilirse, yapılacak bu değişiklikler zincir etkisiyle tüm tasarımı tehdit eder şekilde sorunlar doğurabilir.

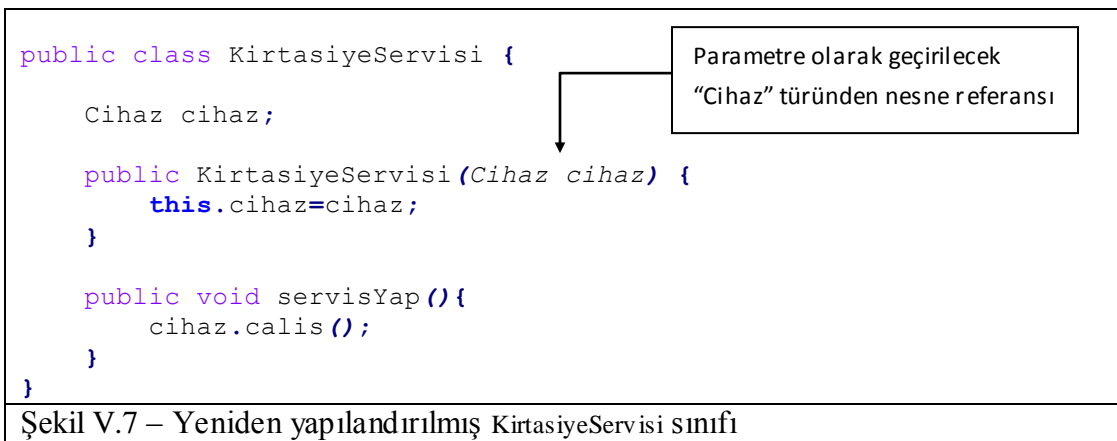
open-closed (*genişlemeye açık – değişime kapalı*) prensibi de aslında tam bu noktaya parmak basmaktadır ; yazılan sınıflar ve sınıflar arası ilişkiler genişlemeye açık olsun ama değişime kapalı olsun demektedir.

Aynı zamanda, bu örnek genişlemeye de kapalıdır. Örnek olarak KirtasiyeServisi uygulaması, Yazici servisi yerine Tarayici servisine hizmet vermek üzere yapılandırılmak istense, yapılacak tek işlem KirtasiyeServisi sınıfına müdahale etmek olacaktır. Bu bakımdan da bu örnekte open-closed (*genişlemeye açık – değişime kapalı*) prensibi çiğnenmiştir.

Gitgide büyüyen ve genişleyen projelerde bağlaşım (*coupling*) kaynaklı sorunlardan kurtulmak için, nesneler arası ilişkiler birbirden bağımsız (*habersiz*) hale getirilmelidir. Bu işleme ise **Bağlaşımı Koparmak (De-Coupling)** denmektedir.



Şekil V.6’ da bağlaşımı koparılmış (*de-coupled*) bir tasarım örneği bulunmaktadır. Dikkat kesilecek olunursa KirtasiyeServisi sınıfı ile diğer sınıflar arasında hiçbir bağ gözükmemektedir.



Bağlaşım (*Coupling*) örneğinde, KirtasiyeServisi sınıfına ait yapılandırıcı metotta Yazici sınıfına ait bir nesne oluşturulmaktaydı. Bağlaşımı koparılmış çözümde ise, KirtasiyeServisi nesnesi içerisine dahil edilecek nesne, KirtasiyeServisi nesnesine ait yapılandırıcıya parametre olarak geçirilmiştir.

Bir önceki örnekte, KirtasiyeServisi sınıfı yalnızca Yazici cihazına hizmet verebilmekte iken yeniden yapılandırılmış örnekte, KirtasiyeServisi dahilinde Yazici cihazı dışında farklı cihazlara dönük hizmetler de verilebilir. (*Örneğin Tarayici, Fotokopi, vs. gibi*).

Yazici yazici; referansına yalnızca Yazici sınıfı türünden nesneler bağlanabileceği için, aynı görevleri farklı şekilde icra eden cihazları tek bir çatı altında toplamak iyi bir fikir olacaktır. Şekil V.6’ da Yazici ve Tarayici sınıflarının her birinde `calis()` yordamı bulunmaktadır ve ikisi de aynı görevi farklı şekillerde icra etmektedirler. Yazici ve Tarayici sınıflarının her biri bir cihaz olduklarından, bu iki sınıfı tek bir çatı altında toplamak üzere Cihaz isimli arayüz (*interface*) tanımlanmaktadır.

```
public KirtasiyeServisi(Cihaz cihaz) {  
    this.cihaz=cihaz;  
}
```

Şekil V.7’ deki yapılandırıcı incelendiğinde, sınıf içerisine dahil edilecek nesnenin Cihaz arayüzü türünde olduğu gözlemlenmektedir. Bu kod kısmına bakıldığında, KirtasiyeServisi, nesnesi hangi cihaz? Sorusuna cevap verememektedir. Yazici nesnesi de olabilir, Tarayici nesnesi de, ihtiyaç olursa Fotokopi nesnesi de.

```
public interface Cihaz {  
  
    public void calis();  
  
}
```

Şekil V.8 – Cihaz arayüzü (*interface*)

Şekil V.8’ de Kirtasiye Servisinde kullanılacak olan cihazları tanımlamak üzere bir arayüz bulunmaktadır. Cihaz isimli arayüzün kullanılmasındaki tek amaç farklı cihazlar arası birleştirici bir çerçeve (*sözleşme*) sunmasıdır. `public void calis();` ifadesiyle gövdesiz bir yordam tanımlanmıştır ve sözleşmeye ait kuralları arayüz içerisinde tanımlı gövdesiz yordamlar oluşturmaktadır.

```
public class Yazici implements Cihaz{  
  
    public void calis() {  
  
        System.out.println("Yazıcı çalışıyor..");  
  
    }  
  
}
```

Şekil V.9 – Cihaz arayüzünü uygulayan (*implementing*) Yazici sınıfı

Şekil V.9’ da Cihaz arayüzüne ait sözleşme kurallarını uygulayan Yazici sınıfı bulunmaktadır. Cihaz arayüzünü uygulayan tüm somut sınıflar, `calis()` yordamını

uygulamak zorundadır. Aksi halde sözleşme kuralları ihlal edilmiş olunur ve böyle bir yapı kurulamaz.

```
public class Tarayici implements Cihaz{  
  
    public void calis () {  
        System.out.println("Tarayıcı çalışıyor..");  
    }  
}
```

Şekil V.10 - Cihaz arayüzünü uygulayan (*implementing*) Tarayici sınıfı

Yazici ve Tarayici sınıfları aynı sözleşmeye uyduklarından dolayı aynı türde (*Cihaz türü*) kabul edilebilirler ve her birine ait nesne Cihaz sınıfı türünden bir referansa bağlanabilir.

```
public class Main {  
  
    public static void main(String[] args) {  
  
        Yazici yazici=new Yazici ();  
        // Tarayici tarayici=new Tarayici ();  
        KirtasiyeServisi servis=new KirtasiyeServisi (yazici);  
        servis.servisYap ();  
    }  
}
```

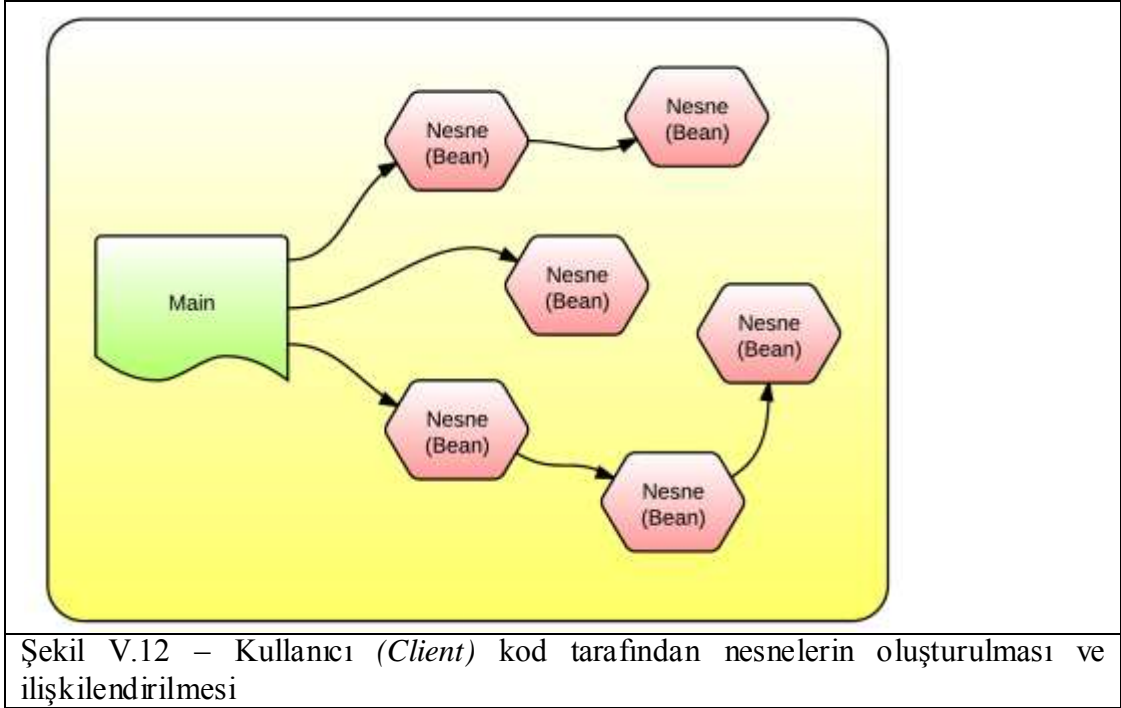
Şekil V.11 – Bağlaşımı Koparılmış (*De-Coupled*) uygulamanın çalıştırılması.

Şekil V.11 – Bağlaşımı Koparılmış (*De-Coupled*) uygulamanın çalıştırılması bulunmaktadır. Birer adet Yazici ve KirtasiyeServisi nesneleri oluşturulmaktadır. Kirtasiye nesnesinin yapılandırıcı metoduna ise Yazici nesnesi bağlanmaktadır.

KirtasiyeServisi, Yazici servisi yerine bir Tarayici servisi vermek istediği takdirde yapılması gereken tek işlem, // *Tarayici tarayici=new Tarayici();* yorum satırını açmak ve *KirtasiyeServisi servis=new KirtasiyeServisi(yazici);* bölümünü *KirtasiyeServisi servis=new KirtasiyeServisi(tarayici);* olarak yapılandırmaktır. Dikkat edilecek olunursa test için kullanılan Main sınıfı haricinde hiçbir sınıf ya da arayüzde değişik gerçekleşmedi ve aynı zamanda open-closed (*genişlemeye açık – değişime kapalı*) prensibi sağlanmış oldu.

V.3 Bağımlılık Zerk (Dependency Injection)

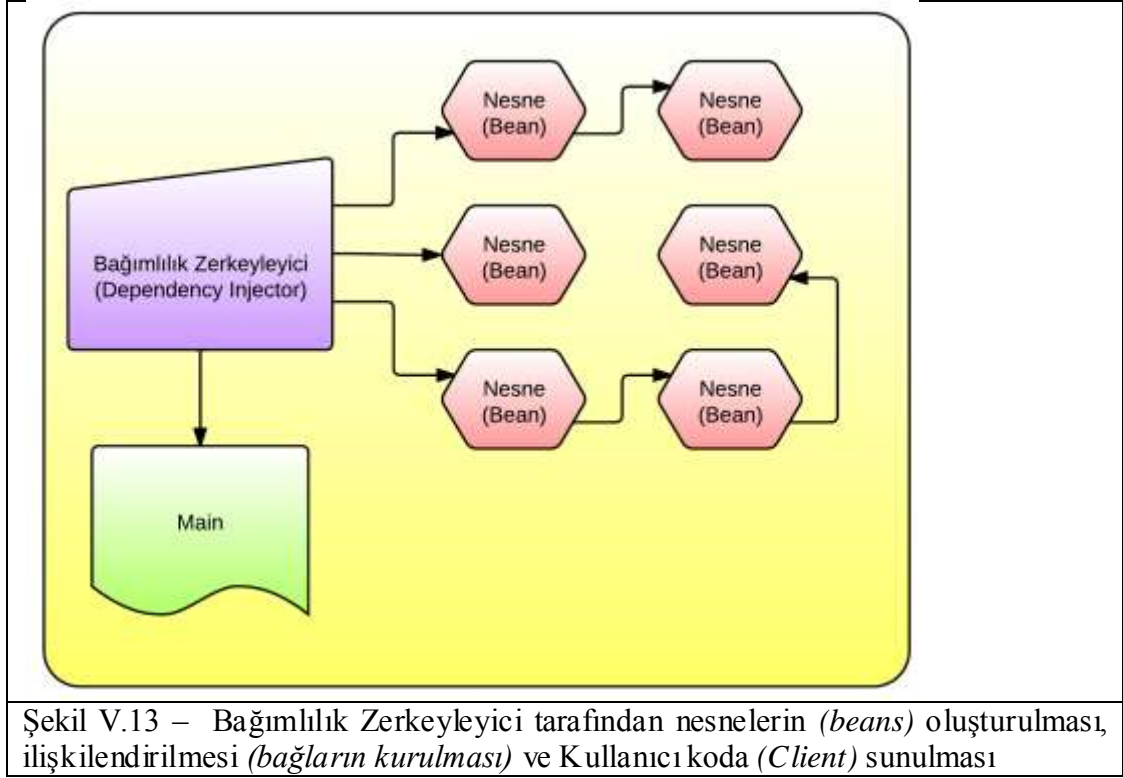
Spring Framework'ün temelini (*core*) Bağımlılık Zerki (*Dependency Injection*) ve IOC (*Inversion Of Control*) kavramları oluşturmaktadır. IOC (*Inversion Of Control*) kavramı ; Nesne oluşturma ve yönetme sorumluluğunu yazılım geliştiriciden kendisine çekerken, Bağımlılık Zerki (*Dependency Injection*) kavramı ise ; Yazılım geliştiricilerin uygulamalarında ihtiyaç duyduğu nesneleri (*beans*), diğer nesneler ile ilişkilendirirken, ilişki kurma işini yazılım geliştiriciden Spring Framework dahilinde bulunan Bağımlılık Zerkeyleyici (*Dependency Injector*) 'e devretmesini tanımlamaktadır.



Şekil V.12 – Kullanıcı (*Client*) kod tarafından nesnelerin oluşturulması ve ilişkilendirilmesi

Bağlaşım (*Coupling*) ve Bağlaşımı Koparma (*De-Coupling*) örneklerinde açıklandığı gibi, Şekil V.12'de de , nesneleri oluşturmak ve ilişkilendirmek , yazılım geliştiricinin görevi durumundadır.

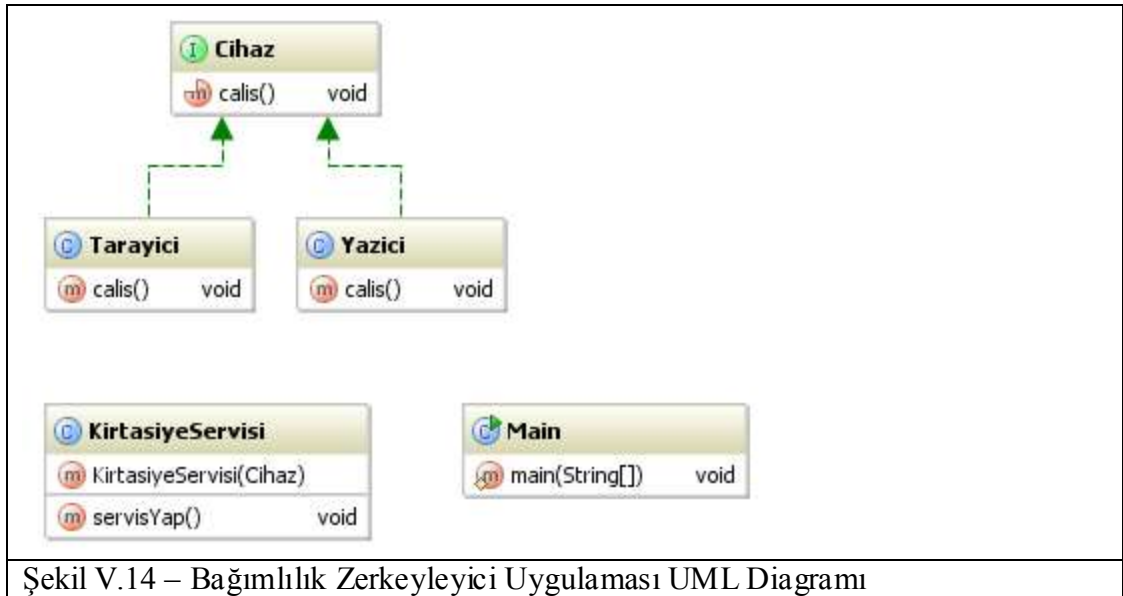
Spring Framework'ün temel kısmının (*core*) görevi içinde bulunan Bağımlılık Zerkeyleyici'nin görevi ise, nesneleri oluşturma ve ilişkilendirme aşamalarını IOC (*Inversion Of Control*) sayesinde kendi inisiyatifine çekmek, ve bu süreçlerden yazılım geliştiricileri soyutmaktır.



Şekil V.13’ deki şekil ile temsil edildiği üzere Bağımlılık Zerkeyleyici, nesnelerin oluşturulması, bağımlılıkların kurulması ve sunulmasından sorumlu bulunmaktadır.

V.4 Bağımlılık Zerkeyleyici (*Dependency Injector*) Uygulaması

Bağımlılık Zerkeyleyici (*Dependency Injector*) Uygulaması, Bağlaşımı Koparma (*De-Coupling*) örneğindeki aynı göreve hizmet etmektedir. Buradaki fark, nesneler (*beans*) ile alakalı tüm gerekliliklerin Spring Framework’e devrini içermektedir.



Spring Framework literatüründe her bir nesne “Bean” adıyla tanımlanmaktadır. Spring Konteyner dahilinde nesnelere ait oluşturma, ilişkilendirme ve yönetme işlemlerini tanımlamak üzere “Metadata” ismi verilen yapılandırıcılar kullanılmaktadır. Java EE 6 ekosistemine benzer bir şekilde, yapılandırma işlemleri XML yapılandırma (*configuration*) dosyaları ile ya da Notasyon (*Annotation*) bazlı tanımlayıcılar ile gerçekleştirilmektedir. Java EE 6 platformunda Notasyon bazlı yapılandırıcılar, XML bazlı yapılandırıcılara göre daha sık tercih edilmektedir. Java EE 6 platformunda Notasyon bazlı tanımlayıcılar daha sık kullanılıyor olmasında karşın, Spring Framework ekosisteminde XML bazlı yapılandırıcıların daha sık kullanıldığı görülebilir. Her ikisi de aynı işi görmesine karşın, hangi yapılandırıcı tipinin kullanılacağı yazılım geliştiricinin tezahüründedir.

V.5 XML Bazlı Spring Framework Yapılandırması

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

  <!-- Nesneler (Spring Beans) -->

</beans>
```

Şekil V.15 – Spring yapılandırıcı XML dosyası

Şekil V.15’de XML bazlı Spring konteyner yapılandırıcısının temel örneği bulunmaktadır. `<!-- Nesneler (Spring Beans) -->` yorum satırı aralığında tanımlı boşluğa ise, yazılım geliştiricilerin Spring Konteyner servisi üzerinde barındıracağı nesneler, nesneler arası ilişkiler (*bağlar*) ve nesnelere ait özellikler tanımlanabilmektedir.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="yazici" class="com.usta.spring.Yazici" />

  <bean id="servis" class="com.usta.spring.KirtasiyeServisi">

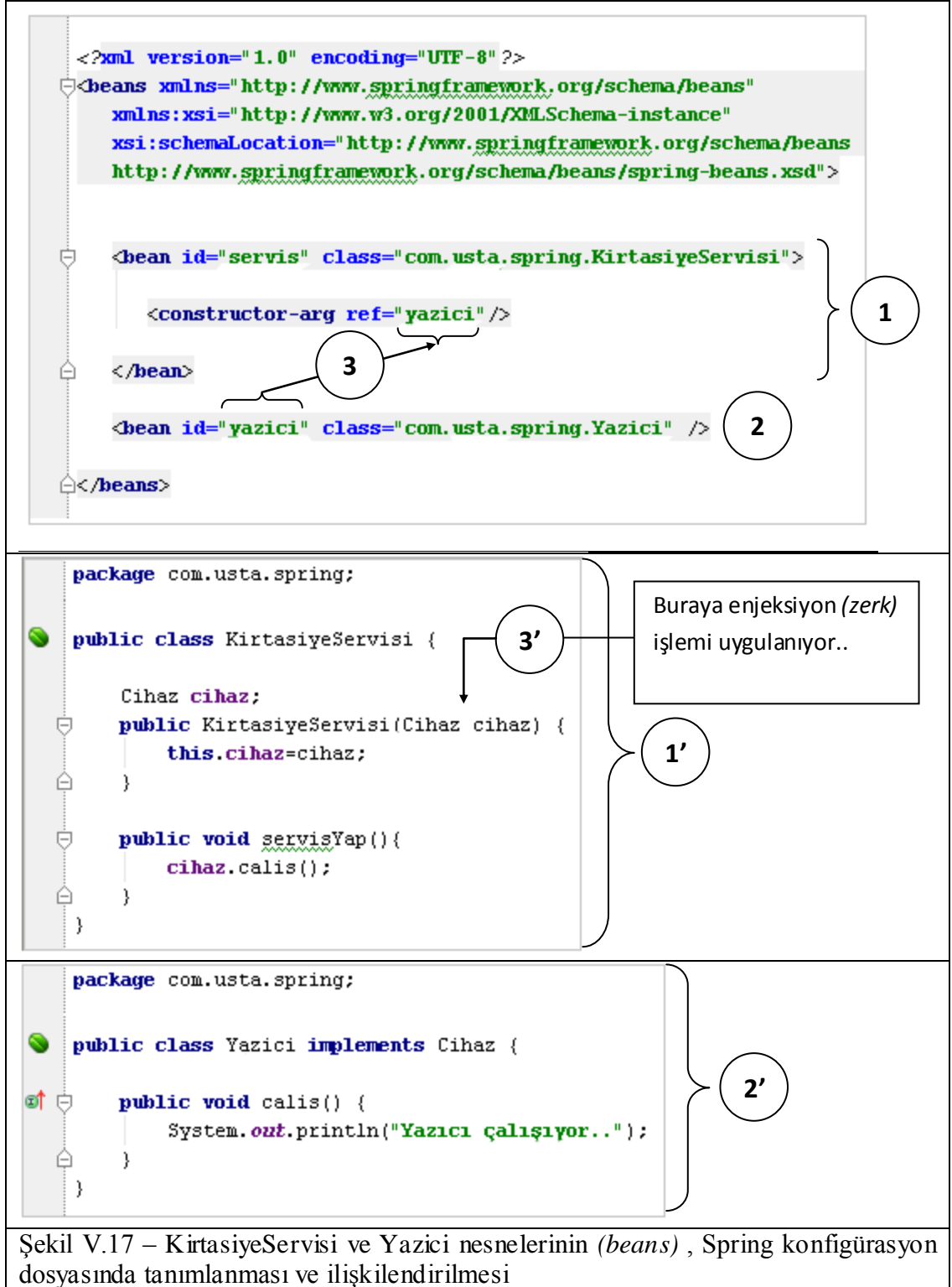
    <constructor-arg ref="yazici"/>

  </bean>

</beans>
```

Şekil V.16 – Yazici ve KirtasiyeServisi nesnelerinin (*beans*), Spring Konteynere tanıtılması

Şekil V.16’ da uygulamada ihtiyaç duyulan nesnelere (*Yazici* ve *KirtasiyeServisi*) ait tanımlamalar bulunmaktadır.



Şekil V.17’de KirtasiyeServisi ve Yazici nesnelerinin (*beans*) , Spring konfigürasyon dosyasında tanımlanması ve ilişkilendirilmesi bulunmaktadır. KirtasiyeServisi ve Yazici sınıflarına dikkat edildiğinde ikisinin de birbirlerinden habersiz olduğu gözükmektedir. Bağımlılıkların koparılmış olması yazılım mühendisliğinde aranan

genişlemeye açık – değişime kapalı prensibine hizmet etmektedir. Spring Framework’ün sağladığı avantaj ile de nesneler arası ilişkileri ve bağımlılığı kurmak tek bir odak noktasından (*spring konfigürasyon dosyası*) gerçekleştirilmektedir.

(1) numaralı kısımda, tam paket adıyla birlikte “servis” id’li KirtasiyeServisi nesnesi (*bean*) tanımlanmaktadır.

(2) numaralı bölümde ise, tam paket yoluyla birlikte, “yazici” id’li Yazici nesnesi (*bean*) tanımlanmaktadır.

(3) numaralı kısımda ise KirtasiyeServisi ve Yazici nesneleri arasındaki bağımlılık (*ilişki*) tanımlaması oluşturulmaktadır. Bu örnekte KirtasiyeServisi nesnesi Yazici nesnesine bağımlıdır. KirtasiyeServisi nesnesi içerisindeki, yapılandırıcı metoda enjekte (*zerk*) edilecek “Cihaz” arayüzü türündeki nesne ise (*bu örnekte Yazici nesnesi*) `<constructor-arg ref=“...”/>` kısmındaki referans (*ref*) özelliği ile KirtasiyeServisi nesnesine zerk edilmektedir.. *ref* özelliğine tanımlanan yazici ifadesi, dikkat edilecek olunursa, enjekte (*zerk*) edilecek nesnenin (*bean*) id’si ile aynıdır.

```
public class Main {  
  
    public static void main(String[] args) {  
  
        ApplicationContext konteyner=  
            new ClassPathXmlApplicationContext("spring-config.xml");  
  
        KirtasiyeServisi servis=  
            konteyner.getBean("servis",KirtasiyeServisi.class);  
  
        servis.servisYap();  
  
    }  
  
}
```

Şekil V.18 – Uygulamayı ve Spring Konteyneri ayağa kaldıran Main sınıf

```
ApplicationContext konteyner=  
    new ClassPathXmlApplicationContext("spring-config.xml");
```

kısımı ile "spring-config.xml" isimli Spring yapılandırıcı dosyadan, Spring nesnelere (*beans*) erişim için kullanılacak ApplicationContext isimli konteyner nesnesi döndürülmektedir. “konteyner” referansına bağlanan ApplicationContext nesnesi aracılığı ile de, Spring Konteyner içine tanımlanan tüm nesnelere erişilebilmektedir.

`konteyner.getBean("servis",KirtasiyeServisi.class);` , kodu ile konteyner dahilinde tanımlanan “servis” id’li nesne (*bean*) talep edilmektedir. İkinci parametredeki `KirtasiyeServisi.class` tanımlaması ile de dönen nesnenin sınıf türü tanımlanmalıdır.

`servis.servisYap ()` ; yordamı çağrılarak da, konsol ekranına "Yazıcı çalışıyor.." mesajı çıktılanmaktadır.

Uygulamaya bakılacak olunursa hiçbir kod bölümünde “new” anahtar kelimesiyle nesne oluşturulmamıştır. Çünkü nesne oluşturmak ve yönetmek bu işlemlerden sonra Spring Framework’ün görevidir.

V.6 Neden Spring?

Spring Framework’ ün en büyük avantajlarından biri ise Uygulama Sunucularına olan bağımlılığı ortadan kaldırmasıdır. (*GlassFish, JBoss, WebLogic gibi.*)

Uygulama Sunucuları (*Application Servers*) konteyner servislerini (*Dependency Injection, IOC, EJB ,AOP gibi.*) uygulama sunucularının çekirdeğinde barındırırken, Spring Framework ise konteyner servislerini sahip olduğu kütüphaneler aracılığı ile gerçekleştirmektedir. Bu sayede Spring Framework ile geliştirilen kurumsal projeler uygulama sunucularına bağımlılıkları olmadan, basit bir Servlet Konteyner’ da koşturulabilmektedir. (*Örneğin : Apache Tomcat, Jetty*). Servlet Konteyner ürünleri genel itibariyle boyut bakımından 5-10 MB bandında olmaktadır, bir uygulama sunucusunu 50-300- MB bandında görmek gayet olağandır. Bu maksatla sistem kaynaklarının sınırlı ve ölçülü olduğu ortamlarda Spring Framework’ den faydalanmak mantıklıdır. Kurumsal ihtiyaçlar ve gereksinimler ölçüsünde ihtiyaç duyulan uygulama geliştirme ortamları

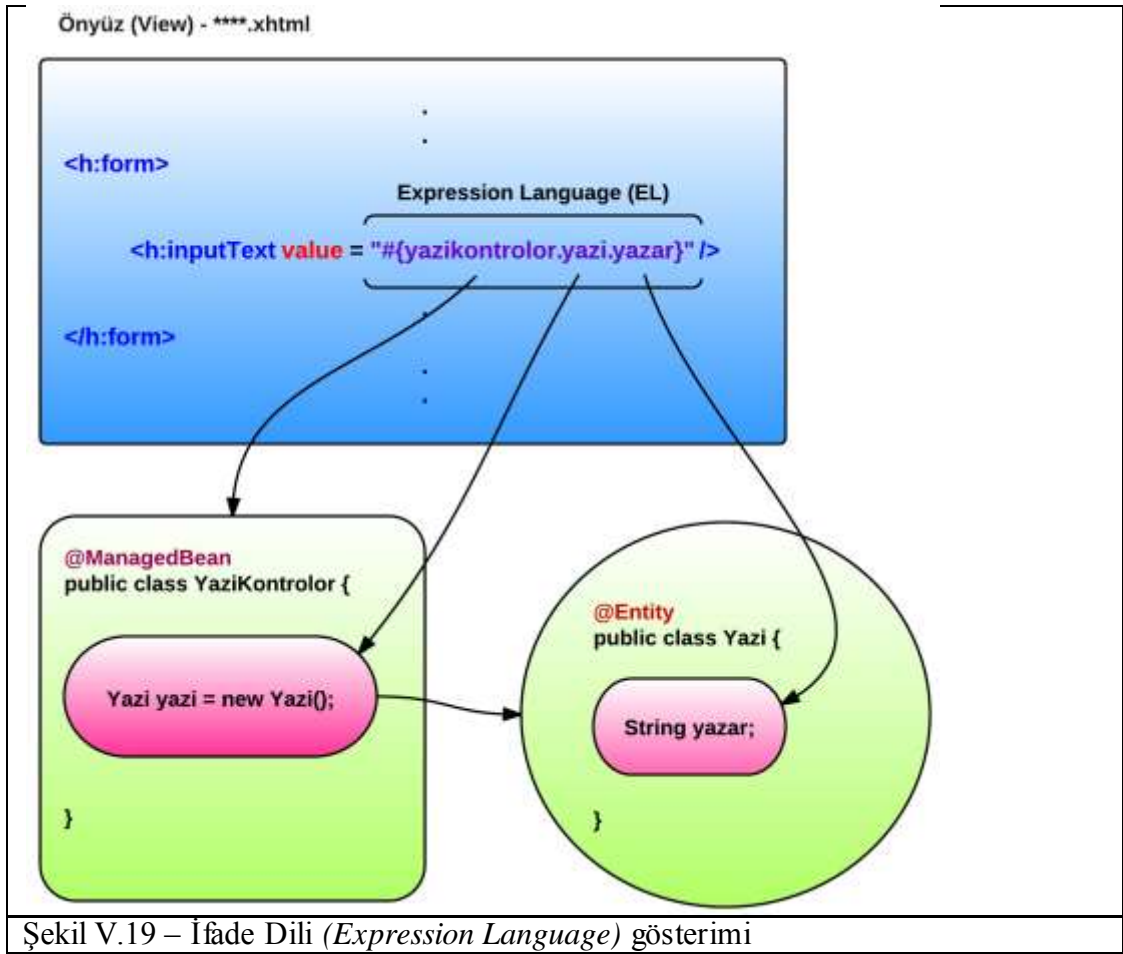
Spring Framework herhangi bir kuruluş ya da vakıf tarafından oluşturulmuş bir standart olmamasına karşın, yazılım pazarında kabul görmüş ve de-facto standart olarak kabul edilen bir üründür. Spring Framework ile kurumsal uygulamaların ihtiyaç duyduğu fonksiyonlar, Java EE spesifikasyon kümesi dahilinde bulunan teknolojilerden bağımsız ya da kısmen bağımlı şekilde karşılanabilmektedir. Kurumsal uygulamalarda hangi teknolojilerden faydalanılacağı, kurumların şartları, durumları ve ihtiyaçları ölçüsünde karar verilmelidir.

V.7 JSF (*JavaServer Faces*) ve Spring Framework Entegrasyonu

Java EE 6 spesifikasyon kümesi dahilinde bulunan JSF (*JavaServer Faces*) teknolojisini, Spring Framework ile birlikte kullanmak için 4 aşamalı bir yol izlenmelidir.

Adım 1 : JSF (*JavaServer Faces*) yapılandırma dosyasına (*faces-config.xml*) , SpringBeanFacesELResolver tanımlamak.

JSF (*JavaServer Faces*) önyüzlerinden , JSF Yönetimli Nesnelere (*Managed Bean*) erişirken, esnek ve kolay kullanım sağlayan JSF deyim dili (*Expression Language*) kullanılmaktadır.



JSF sayfalarındaki deyim dili ifadeleri ile, JSF Yönetimli Nesneler yerine Spring konteyner tarafından yönetimli nesnelere erişim için Şekil V.20’ deki değişiklik uygulanmalıdır.

```

<?xml version='1.0' encoding='UTF-8'?>

<faces-config version="2.0"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd">

    <application>

        <el-resolver>

            org.springframework.web.jsf.el.SpringBeanFacesELResolver

        </el-resolver>

    </application>

</faces-config>

```

Şekil V.20 – Spring – JSF ifade dili çözümleyici tanımlanması (faces-config.xml)

Adım 2 : Spring konyerleri ayağa kaldıracak (*başlatacak*) dinleyici (*listener*) sınıfın web aktarım tanımlayıcısına (*web deployment descriptor – web.xml*) eklenmesi.

```
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

Şekil V.21 – Spring konteyneri başlatan dinleyici (*listener*) sınıfın tanımlanması

Adım 3 : Spring konteynerin, HTTP isteklerini karşılayan FacesServlet isteklerini görebilmesi için dinleyici (*listener*) sınıfın web aktarım tanımlayıcısına (*web deployment descriptor – web.xml*) eklenmesi.

```
<listener>
    <listener-class>
        org.springframework.web.context.request.RequestContextListener
    </listener-class>
</listener>
```

Şekil V.22 – FacesServlet’e gelen HTTP isteklerini Spring konteynerin görmesini sağlayan dinleyici (*listener*) sınıfın tanımlanması

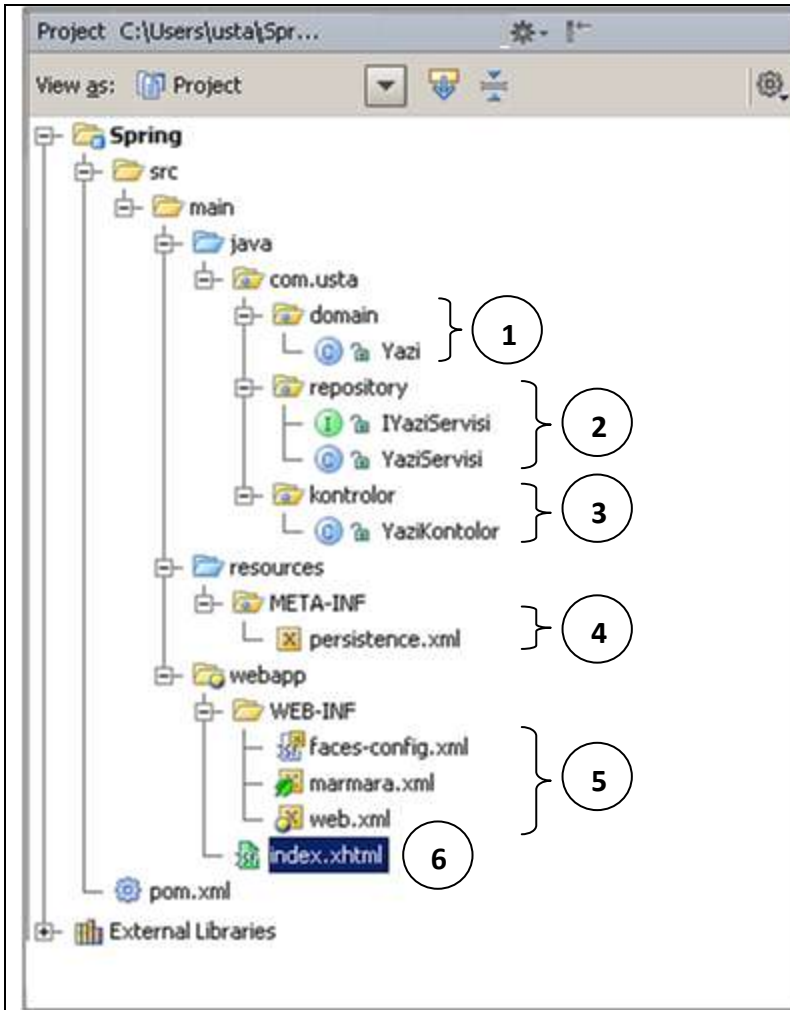
Adım 4 : Spring konteyner dahilindeki nesnelerin (*beans*) tanımlandığı XML konfigürasyon dosyasının, web aktarım tanımlayıcısına (*web deployment descriptor – web.xml*) tanıtılması.

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/marmara.xml</param-value>
</context-param>
```

Şekil V.23 - Spring yapılandırma dosyasının yolunun (*path*) belirtilmesi

Yukarıda listeli 4 adım gerçekleştirildikten sonra JSF (*JavaServer Faces*) ve Spring Framework entegrasyonu tamamlanmış olmaktadır. web.xml, web aktarım tanımlayıcısının son hali için JSF , Spring ve Hibernate uygulaması takip edilebilir.

V.8 JSF , Spring ve Hibernate Uygulaması



Şekil V.24 - JSF , Spring ve JPA Uygulaması proje yapısı

(1) numaralı paket yapısında Yazi domain nesnesi (*Entity*) bulunmaktadır.

```
@Entity(name = "YAZILAR")
@NamedQueries({
    @NamedQuery(name = "tumYazilar", query = "SELECT b FROM YAZILAR b
ORDER BY b.yaziNo DESC"),
    @NamedQuery(name = "suYazar", query = "SELECT b FROM YAZILAR b
WHERE b.yazar LIKE :yazar ")
})
public class Yazi {

    @Column(name = "yaziNo")
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long yaziNo;
    @Column(name = "yazar", nullable = false)
```

```

private String yazar;
@Column(name = "tarih", nullable = true)
private String tarih;
@Column(name = "yazi")
private String yazi;

// Getter, Setter yordamları ve Yapılandırıcılar
}

```

Şekil V.25 – YAZILAR isimli Entity nesnesi

(2) numaralı bölümde Veri Erişim Katmanı ile ilgili operasyonları gerçekleştirecek Veri Erişim Nesneleri (*Data Access Object*) bulunmaktadır.

Spring Framework dahilinde bir DAO nesnesi tanımlamak isteniyorsa mevcut sınıf `@Repository` notasyonu ile sarmalanmalıdır.

```

public interface IYaziServisi {

    public String kaydet(Yazi yazi);

    public List<Yazi> tumYazilar();

    public Yazi suYazar(String yazar);

    public void sil(Yazi yazi);

    public void guncelle(Yazi yazi);

}

```

Şekil V.26 – Veri Erişim Nesnesi (DAO) arayüz (interface) tanımlaması

Şekil V.26’ da bulunan `IYaziServisi` arayüzünün görevi, Veri Erişim Nesneleri (DAO) için bir çerçeve oluşturmaktır. Bu çerçeveye uyan birden fazla implementasyon oluşturulabilir.

```

@Repository
@Transactional(readonly = true)
public class YaziServisi implements IYaziServisi {

    @PersistenceContext
    EntityManager em;

    @Override
    public String kaydet(Yazi yazi) {
        em.persist(yazi);
        em.flush();
        return "/index.xhtml";
    }

    @Override

```



```

public List<Yazi> tumYazilar () {
    Query sorgu = em.createNamedQuery("tumYazilar");

    return sorgu.getResultList();
}

@Override
public Yazi suYazar(String yazar) {
    return em.find(Yazi.class, yazar);
}

@Override
public void sil(Yazi yazi) {
    em.remove(yazi);
}

@Override
public void guncelle(Yazi yazi) {
    em.merge(yazi);
}
}

```

Şekil V.27 – **IYaziServisi** arayüzünü uygulayan DAO nesnesi

(3) numaralı kısımda bulunan YazıKontrolör sınıfı JSF (*JavaServer Faces*) önyüzlerine (*views*) ait arkaplan bilgilerini tutan ve önyüz için çeşitli görevleri işleten Spring nesnesini (*beans*) tanımlamaktadır.

```

@Component(value = "yazikontrolor")
@Scope(value = "request")
public class YaziKontrolor {

    @Autowired
    IYaziServisi servis;
    Yazi yazi = new Yazi();
    List<Yazi> yaziListesi = new ArrayList<>();

    @PostConstruct
    private void yazilariYukle () {
        yaziListesi = servis.tumYazilar();
    }

    public String yaziEkle () {
        servis.kaydet(yazi);
        yaziListesi = servis.tumYazilar();
        return "index.xhtml";
    }
}

```

```

    }

    public Yazi getYazi () {
        return yazi;
    }

    public void setYazi(Yazi yazi) {
        this.yazi = yazi;
    }

    public List<Yazi> getYaziListesi () {
        return yaziListesi;
    }

    public void setYaziListesi(List<Yazi> yaziListesi) {
        this.yaziListesi = yaziListesi;
    }
}

```

Şekil V.28 - JSF (*JavaServer Faces*) önyüzlerine (*views*) ait arkaplan bilgilerini tutan ve önyüz için çeşitli görevleri işleyen *YaziKontrolor* nesnesi

```

@Component(value = "yazikontrolor")
@Scope(value = "request")

```

`@Component` notasyonu, bir Java sınıfından, Spring konteyner servisleri tarafından yönetilebilir bir nesne oluşturabilmek için kullanılmaktadır.

`@Scope` notasyonu ise , o anki nesnenin kapsamını (*yaşam süresini*) tayin etmektedir. Kapsam tipi "request" olan Spring nesnesi (*bean*), kendi içindeki veri alanlarını bir istek/yanıt döngüsü boyunca tutar.

`@Component` ve `@Scope` notasyonları Spring taraflı yönetilebilir basit nesneleri tanımlamak için kullanılmaktadır. `@Component` notasyonu ile JSF (*JavaServer Faces*) bölümünde açıklanan `@ManagedBean` notasyonu aynı amaca hizmet etmektedirler. Spring tarafındaki `@Scope(value = "request")` tanımlamasının karşılığı ise `@RequestScoped` notasyonu denilebilir.

```

@Autowired
IYaziServisi servis;

```

`@Autowired` notasyonu, Spring konteyner tarafında tanımlanan nesneleri (*beans*) enjekte etmek (*zerk*) için kullanılmaktadır. Bu kısımda ise, *YaziServisi* Veri Erişim Nesnesi (*DAO*), "servis" isimli referansa enjekte edilmektedir. Enjekte edilen Veri

Erişim Nesnesi (*Data Access Object*) aracılığı ile de, veritabanı operasyonları gerçekleştirilmektedir.

(4) numaralı bölümde tanımlı persistence.xml konfigürasyon dosyası ile domain (*Entity*) nesnelere dönük yapılandırmalar gerçekleştirilmektedir.

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence version="2.0"
xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

<persistence-unit name="Spring_PU"
transaction-type="RESOURCE_LOCAL">

<class>com.usta.domain.Yazi</class>
</persistence-unit>

</persistence>
```

Şekil V.29 – persistence.xml konfigürasyonu

"Spring_PU" isimli kalıcılık birimi (*persistence unit*) dahilinde , `<class>...</class>` özelliği ile veritabanına eşleştirilecek Yazi entity sınıfı tanımlanmaktadır. Kalıcılık biriminde tanımlı `transaction-type` özelliği JTA ya da `RESOURCE_LOCAL` olabilmektedir. JTA Transaction tipi Uygulama Sunucuları tarafından karşılanmaktadır. Transaction mekanizması olarak JTA yerine 3. Parti kütüphaneler kullanılarak bir çözüm sağlanmak isteniyorsa `RESOURCE_LOCAL` niteliği tanımlanmalıdır.

(5) numaralı kısımda, faces-config.xml JSF(*JavaServer Faces*) yapılandırma dosyası, marmara.xml Spring konfigürasyon dosyası ve web aktarım tanımlayıcısı (*web deployment descriptor –web.xml*) bulunmaktadır.

```
<?xml version='1.0' encoding='UTF-8'?>

<faces-config version="2.0"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd">

<application>

<el-resolver>
```

```

        org.springframework.web.jsf.el.SpringBeanFacesELResolver

    </el-resolver>

</application>

</faces-config>

```

Şekil V.30 – Spring – JSF ifade dili çözümleyici tanımlanması (*faces-config.xml*)

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd">

  <display-name>Spring</display-name>

  <context-param>
    <param-name>javax.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
  </context-param>

  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/marmara.xml</param-value>
  </context-param>

  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>

  <listener>
    <listener-class>
      org.springframework.web.context.request.RequestContextListener
    </listener-class>
  </listener>

  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>

  <session-config>
    <session-timeout>

```

```

</session-timeout>
</session-config>

<welcome-file-list>
<welcome-file>faces/index.xhtml</welcome-file>
</welcome-file-list>

</web-app>

```

Şekil V.31 – Web aktarım tanımlayıcısı (*Web deployment descriptor – web.xml*)

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">

<context:annotation-config/>
<context:component-scan base-package="com.usta"/>
<tx:annotation-driven/>

<bean id="entityManagerFactory"

class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">

    <property name="dataSource" ref="dataSource"/>
    <property name="jpaVendorAdapter" ref="jpaAdapter"/>
    <property name="persistenceUnitName" value="Spring_PU"/>

</bean>

<bean id="dataSource"

class="org.springframework.jdbc.datasource.DriverManagerDataSource">

    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost/spring"/>
    <property name="username" value="root"/>
    <property name="password" value="root"/>

</bean>

<bean id="jpaAdapter"

class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">

    <property name="showSql" value="true"/>
    <property name="generateDdl" value="true"/>
    <property name="database" value="MYSQL"/>

```

```

</bean>

<bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>

<!--<bean id="yazikontrolor" class="com.usta.spring.YaziKontrolor"
scope="request"/> -->

</beans>

```

Şekil V.32 – Spring nesnelerinin (*beans*) ve nesnelere ait özelliklerin tanımlandığı marmara.xml dosyası

`<context:annotation-config/>` tanımlaması, Spring Framework dahilinde Notasyon bazlı yapılandırıcıları kullanmayı aktif etmektedir.

Spring Framework, sınıf isimlerinin başında, sınıf alanlarının başında ve metodların başında tanımlanan Notasyon bazlı yapılandırıcıları tanımak için, tüm sınıfları tarayarak keşfetmektedir. Bu amaçla, tarama yapılacak paket dizininin tanımlanması gerekmektedir. `<context:component-scan base-package="com.usta"/>` , Bu tanımlamada "com.usta" paketi altındaki tüm sınıflarda Notasyon araması yapılmaktadır.

`<tx:annotation-driven/>`, tanımlaması ise Transaction mekanizmasına ait, Notasyon bazlı tanımlayıcıları tespit etmektedir.

```

<bean id="entityManagerFactory"
      class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="jpaVendorAdapter" ref="jpaAdapter"/>
    <property name="persistenceUnitName" value="Spring_PU"/>
</bean>

```

Şekil V.33 – Entity Manager Üreteç nesnesi (*bean*)

`entityManagerFactory` id'li Spring nesnesi (*bean*) Spring Framework tarafından tanımlanan **EntityManager Üreteç** nesnesini tanımlamaktadır. Bu nesne aracılığı

ile veritabanı operasyonlarını gerçekleştirecek **EntityManager** nesneler üretilmektedir. Şekil V.27’de bulunan, **EntityManager em;** referansı bu **EntityManager Üreteci** (*Entity Manager Factory*) ile beslenmektedir.

`<property name="persistenceUnitName" value="Spring_PU"/>`, tanımlaması persitence.xml dosyasında tanımlı kalıcılık birimlerinden hangisinin kullanılacağını belirtmektedir.

`<property name="dataSource" ref="dataSource"/>`, özelliği **"dataSource"** isimli (*name*) özelliğe, **"dataSource"** id’li nesneyi (*bean*) referans almaktadır.

```
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">

    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost/spring"/>
    <property name="username" value="root"/>
    <property name="password" value="root"/>

</bean>
```

Şekil V.34 – Veri Kaynağı (*DataSource*) nesnesi

Şekil V.34’de bulunan dataSource id’li nesne (*bean*), veritabanı erişimi için kullanılacak veri kaynağını oluşturmaktadır.

`<property name="jpaVendorAdapter" ref="jpaAdapter"/>`, tanımlaması ise JSR 317 numaralı Java Persistence API şartnamesini sağlayan kütüphaneleri referans almaktadır. (*Hibernate, EclipseLink, Toplink, JDO* gibi.)

```
<bean id="jpaAdapter"
class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">

    <property name="showSql" value="true"/>
    <property name="generateDdl" value="true"/>
    <property name="database" value="MYSQL"/>

</bean>
```

Şekil V.35 – JPA (*Java Persistence API*) sağlayıcısı

Şekil V.35’ de **jpaAdapter** id’li ve Hibernate kütüphanesinden beslenen nesne (*bean*) tanımlanmaktadır.

`<property name="showSql" value="true"/>`, özelliği uygulama çalıştığında konsol ekranında, JPA uygulayıcı kütüphane tarafından üretilen SQL komutları listelemektedir.

`<property name="generateDdl" value="true"/>` tanımlaması, entity nesneleri veritabanı tablolarına dönüştürülürken, gerekli DDL (*Data Definition Language*) tümcelerini üretme ve uygulama işlemini sağlamaktadır. Bu sayede gerekli tablo yapıları ve ilişkileri oluşturulmaktadır.

`<property name="database" value="MYSQL"/>`, tanımlaması ise kullanılan veritabanı sistemini, JPA (*Java Persistence API*) uygulayıcı kütüphaneye tanıtmaktadır. Bu özellik opsiyonel olmasına karşın, daha iyi bir başarımlı açısından veritabanı türünü belirtmek önem arz etmektedir.

```
<bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">

    <property name="entityManagerFactory" ref="entityManagerFactory"/>

</bean>
```

Şekil V.36 – Transaction Yönetici Nesne (*Bean*)

Şekil V.36’da Transaction yönetiminden sorumlu nesne tanımlanmaktadır.

`<property name="entityManagerFactory" ref="entityManagerFactory"/>`, tanımlaması ise EntityManager Üreticini temsil etmektedir.

Spring Framework için gerekli yapılandırmalar XML ya da Notasyon bazlı tanımlayıcılar ile gerçekleştirilebilmektedir. Örneğin ;

```
<bean id="yazikontrolor"
      class="com.usta.spring.YaziKontrolor" scope="request"/>
```

"yazikontrolor" id’li *YaziKontrolor* nesnesi, Şekil V.28’de bulunan `@Component` ve `@Scope` notasyonlarıyla tamamen aynı işi görmektedir.

(6) numaralı kısımda, kullanıcı arabirimi bileşenlerinin (*UI Components*) bulunduğu index.xhtml sayfası bulunmaktadır.

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ui="http://java.sun.com/jsf/facelets">

<h:head>
    <title>Spring ve Hibernate ve JSF</title>
</h:head>
```



```

<h:body>
    <h:form>

        <h:panelGrid columns="2">
            <h:outputLabel value="Yazar : "/>
            <h:inputText value="#{yazikontrolor.yazi.yazar}"/>
            <h:outputLabel value="Tarih : "/>
            <h:inputText value="#{yazikontrolor.yazi.tarih}"/>
            <h:outputLabel value="Yazi : "/>
            <h:inputTextarea value="#{yazikontrolor.yazi.yazi}"/>
        </h:panelGrid>

        <h:commandButton value="Yazi Ekle" action="#{yazikontrolor.yaziEkle}"/>

    </h:form>

    <br></br>
    <br>Yazı Listesi</br>
    <br></br>

    <h:dataTable value="#{yazikontrolor.yaziListesi}" var="liste"
        border="1">

        <h:column>
            <f:facet name="header">
                <h:outputText value="Yazı No"/>
            </f:facet>
            <h:outputText value="#{liste.yaziNo}"/>
        </h:column>

        <h:column>
            <f:facet name="header">
                <h:outputText value="Yazar"/>
            </f:facet>
            <h:outputText value="#{liste.yazar}"/>
        </h:column>

        <h:column>
            <f:facet name="header">
                <h:outputText value="Tarih"/>
            </f:facet>
            <h:outputText value="#{liste.tarih}"/>
        </h:column>

        <h:column>
            <f:facet name="header">
                <h:outputText value="Yazı içeriği"/>
            </f:facet>
            <h:outputText value="#{liste.yazi}"/>
        </h:column>

    </h:dataTable>

```

```
</h:body>  
</html>
```

Şekil V.37 – Blog yazarları ve yazılarını kaydeden ve listeleyen index.xhtml dosyası

Uygulamaların içeriğine kaynak kodlar bölümünden erişebilirsiniz..

Görüşmek Üzere
Rahman USTA

