2. Bölüm

Spring İle Tanışalım

Bir Program Nasıl Oluşur?

Spring'in detaylarına girmeden önce, bir programın oluşum hikayesine göz atmamızda fayda var. Bir programın oluşumundaki yön verici en önemli etken, programı kullanacak olan müşterinin iş piyasasındaki gereksinimleridir. Program müşterinin iş hayatını kolaylaştırmak, firma bünyesindeki aktiviteleri organize etmek ve kazanç sağlamak için kullanılır. Yazılım esnasında müşteri tarafından oluşturulan kriterlerin dikkate alınması gerekmektedir, aksi taktirde müşterinin isteklerini karşılayamayan ve günlük iş hayatında kullanılamaz bir program ortaya çıkar.

Yazılım süreci müşteri isteklerinin analizi ile başlar. Analiz safhasında müşterinin gereksinimleri tespit edilir ve yazılım için gerekli taban oluşturulur. Analiz ve bunu takip eden yazılım, müşteri isteklerinin transformasyona uğradığı ve netice olarak bir programın oluşturulduğu karmaşık bir süreçtir.

Resim 2.1

Transformasyon müşteri gereksinimlerinin tespiti ile başlar. Sadece müşteri ne istediğini bilebilir ve programcı olarak bizim görevimiz, müşterinin istekleri doğrultusunda programı şekillendirmektir. Şimdi bu sürecin nasıl işlediğini bir örnek üzerinde birlikte inceleyelim. Hayali bir müşterimiz bizden sahip olduğu araç kiralama servisini yönetmek için bir yazılım sistemi oluşturmamızı istiyor. Kitabı oluşturan bundan sonraki bölümlerin hepsinde araç kiralama servisini bölüm konusuna uygun olacak şekilde Spring çatışını kullanarak geliştireceğiz. Vereceğim tüm örnekler bu uygulamadan alıntıdır ve kodlar her bölüm için hazırladığım Maven projesinde yer almaktadır. Bu projeleri Eclipse altında kullanabilirsiniz.

Araç Kiralama Servisi

Spring'in nasıl kullanıldığını göstermek amacıyla bu bölümden itibaren hayali bir araç kiralama servisi için gerekli yazılım sistemini oluşturmaya başlayacağız.

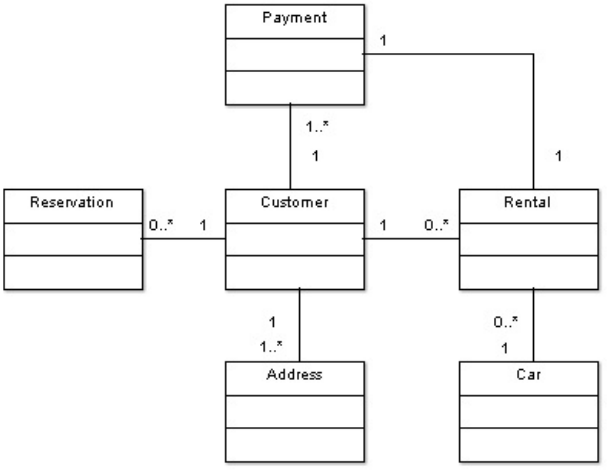
Bu yazılım sistemi aracılığı ile müşteriler internet üzerinden rezervasyon ve kiralama işlemlerini yapabilecekler.

Program Alan Modeli (Domain Model)

Müşteri gereksinimlerini program koduna dönüştürebilmemiz için bir model oluşturmamız gerekiyor. Bu modelden yola çıkarak neyin nasıl programlanması gerektiğini daha iyi anlayabileceğiz ve yazılım sürecine yön vereceğiz. Yazılımda bu tür modellere alan modeli (domain model) ismi verilmektedir. Alan ile, programın kapsadığı çalışma alanını kastedilmektedir. Araç kiralama servisi örneğinde içinde bulunduğumuz alan araç kiralama servisinin çalışma sahasıdır.

Alan modelinde çalışma sahasında kullanılan birimler, bu birimlerin rolleri ve birbirleriyle olan ilişkileri yer alır. Bu birimler çalışma sahasına has kullanılan terminolojiden türetilir. Bir araç kiralama servisinde çalışan personelin aralarında yaptıkları konuşmalara kulak verdiğinizde araba, müşteri, rezervasyon, ödeme, adres, rezervasyon iptali gibi bu çalışma sahasına has terimler duyarsınız. Bu terimler genelde oluşturulan alan modelinde yer alan birimlerdir. Müşterinin sahip olduğu gereksinimleri tespit edebilmek için yazılım ekibi tarafından gerekli soruların sorularak, çalışma sahasında yer alan birimlerin ortaya çıkarılması gerekmektedir.

Yazılım sistemleri için alan modelleri UML (Unified Modeling Language) olarak isimlendirilen bir modelleme dili ile yapılır. Aşağıda araba kiralama servisi için oluşturduğumuz alan modeli yer almaktadır.



Resim 2.2

Resim 2.2 de yer alan alan modelinde birimler arası bağlantılar olduğu görülmektedir. Bu bağlantılar iki birim arasındaki interaksiyonu tanımlamak için kullanılmaktadır. Birimler birbirlerini kullanarak, modelledikleri verilerin kullanım ve işlenme tarzlarını belirlerler.

Sınıflar arası ilişkilerin derecesini belirtmek için rakamlar kullanıyoruz. Kullanılan rakamlar şu şekildedir:

0,1	Sınıf karşı sınıf tarafından ya hiç kullanılmamaktadır ya da en fazla bir kez kullanılmaktadır.
0..*	Sınıf karşı sınıf tarafından ya hiç kullanılmamaktadır ya da birçok kez kullanılmaktadır.
1,1	Sınıf karşı sınıf tarafından ya en az bir kez ya da en fazla bir kere kullanılmaktadır.
1..*	Sınıf karşı sınıf tarafından ya en az bir kez ya da birçok kez kullanılmaktadır.

Şimdi bu şemayı kullanarak araç kiralama servisi alan modelinde yer alan nesne ilişkilerine bir göz atalım.

Bir müşteriyi temsil eden Customer ile bir araç kiralama işlemini temsil eden Rental arasında 1-0,* ilişkisi vardır. Buna göre bir müşteri sıfır ya da birden fazla araç kiralamış olabilir ve her bir araç kiralama işlemi sadece bir müşteri için yapılmıştır.

Rental ve kiralanan aracı temsil edenCar arasında 0,*-1 ilişkisi vardır. Buna göre bir araç kiralama işleminde (Rental) sadece bir araç kiralık olarak verilebilir. Buna karşın bir araç sıfır ya da birden fazla müşteri için kiraya verilmiş olabilir.

Bir müşteri rezervasyonunu temsil eden Reservation ile Customer arasında 0,*-1 ilişkisi mevcuttur. Bu bir rezervasyonun sadece bir müşteriye ait olduğunu, bir müşterinin sıfır ya da birden fazla rezervasyonu olabileceği anlamına gelmektedir.

Müşteri tarafından yapılan ödemeyi temsil eden Payment ile Customer arasında 1,*-1 ilişkisi bulunmaktadır. Bu yapılan bir ödemenin bir müşteriye ait olduğu, bir müşterinin en az bir ya da daha fazla ödemesi bulunduğuna işaret etmektedir.

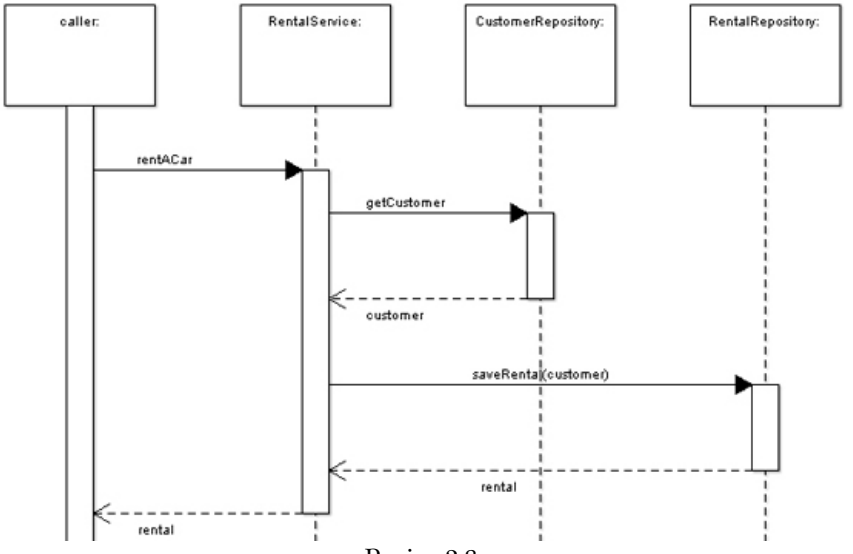
Müşterinin adresini temsil eden Address ile Customer arasında 1-1,* ilişkisi bulunmaktadır. Bu her müşterinin en az bir adresi olması gerektiği, her adresin mutlaka bir müşteriyi ait olduğu anlamına gelmektedir.

Spring ile geliştireceğimiz uygulamada alan modelinde yer alan birimlerin hepsini birer Java sınıfı olarak implemente edeceğiz. Bu tür sınıflara yazılımda entity (öğe) ismi verilmektedir. Bu öğeler aracılığı ile uygulama bünyesinde kullanılan verileri modelleyebiliriz. Bir Customer öğesi örneğin bir müşteriyi temsil eden tüm özellikleri ihtiva eder. Bu öğelerin Java gibi nesneye yönelik bir programlama dilinde sınıf olarak kodlanmaları, yazılımcının öğeler arasındaki ilişkileri daha iyi anlamasını sağlamaktadır.

Bir uygulama sadece alan modelinde yer alan entity nesnelerinden oluşmaz. Alan modeli veri yapılarını modellemek için kullanılır. Asıl uygulama bu verileri üzerinden işlem gerçekleştiren yazılım yapısıdır. Veriler üzerinde yapılan işlemleri ve program akışını daha iyi kavramak için dizge diyagramları oluşturabiliriz. Dizge diyagramları hangi kod birimlerinin hangi işten sorumlu olduğunu gösterirler. Bu şekilde uygulamanın hangi parçalardan oluştuğunu anlamak ve parçaları geliştirmek kolaylaşır.

Dizge Diyagramı (Sequence Diagram)

Kod yazmadan önce program akışını görselleştirmek için kullanabileceğimiz diğer bir araç ta UML dizge diyagramlarıdır. Resim 2.3 de bir araç kiralama işlemi için yapılması gereken işlemler yer almaktadır. RentalService, CustomerRepository ve RentalRespository diyagram bünyesinde kullandığımız Java sınıflarıdır. Bu diyagram bünyesinde hangi sınıfın hangi metodu çağırarak, işlem yaptığı yer almaktadır.



Resim 2.3

İlk Çözüm

Spring kullanmadan araç kirala servisi uygulamasını nasıl geliştirirdik? Bu sorunun cevabını aramaya koyulmadan önce bir birim testi yazalım. Bu birim testi bize uygulamanın geliştirilmesinde yol gösterici nitelikte olacaktır. Yeni bir kiralama işlemini test eden birim testi kod 2.1 de yer almaktadır.

```
Kod 2.1 - RentalServiceTest

package com.kurumsaljava.com.spring;

import static junit.framework.Assert.assertTrue;
```

```
import java.text.SimpleDateFormat;<br/>
import java.util.Date;<br/>
import org.junit.Test;<br/></code>

public class RentalServiceTest {

    @Test
    public void add_new_rental() throws Exception {
        Car car = new Car("Ford Fiesta");
        RentalService service = new RentalService();
        Date rentalBegin = new SimpleDateFormat("dd/MM/yy")
                                    .parse("22/12/2013");
        Date rentalEnd = new SimpleDateFormat("dd/MM/yy")
                                    .parse("29/12/2013");
        Rental rental =
            service.rentACar("Özcan Acar", car, rentalBegin,
                                        rentalEnd);
        assertTrue(rental.isRented());
    }
}
```

Resim 2.3 de yer alan dizge diyagram uygulamanın hangi modüllerden oluştuğuna dair fikir sahibi olmamızı sağlamıştı. Kod 2.1 de yer alan birim testine göz attığımızda, dizge diyagramında yer alan RentalService biriminin Java sınıfı olarak var olduğunu ve sahip olduğu sorumluluk alanına göre kullanıldığını görmekteyiz.

RentalService sınıfının rentACar() metodu ile bir aracın kiralanma işlemi gerçekleştirilmektedir. Bu metot müşteri ismini, kiralanan aracı, kiralama süresinin başlangıç ve bitiş tarihlerini parametre olarak almaktadır. rentACar() metodu bünyesinde yapılan işlemler kod 2.2 de yer almaktadır.

Kod 2.2 - RentalService

```
package com.kurumsaljava.com.spring;

import java.util.Date;

public class RentalService {

    public Rental rentACar(String customerName, Car car,
                                    Date begin, Date end) {

        CustomerRepository customerRepository =
```

```
                                new CustomerRepository();
        Customer customer =
                customerRepository.getCustomerByName(customerName);
        if (customer == null) {
            customer = new Customer(customerName);
            customerRepository.save(customer);
        }
        Rental rental = new Rental();
        rental.setCar(car);
        rental.setCustomer(customer);
        RentalRepository rentRepository = new RentalRepository();
        rentRepository.save(rental);
        return rental;
    }
}
```

CustomerRepository sınıfı müşteri verilerinden sorumlu olan sınıftır. Veri tabanında yer alan bir müşteri kaydını bulmak için getCustomerByName() metodunu kullanıyoruz. Eğer getCustomerByName() metodu aradığımız müşteriyi bulamadıysa, new Customer() ile yeni bir müşteri nesnesi oluşturup, save() metodu ile bu müşteriyi veri tabanına ekleyebiliriz.

Bir aracın kiralama işlemi Rental sınıfı ile sembolize edilmektedir. Bu sınıf bünyesinde hangi aracın hangi müşteri tarafından kiralandığı yer almaktadır. setCar() ve setCustomer() metotları ile kiralanan araç ve aracı kiralayan müşteri oluşturulan rental nesnesine yerleştirilir. Akabinde RentalRepository sınıfının sahip olduğu save() metodu ile rental nesnesi veri tabanına kayıtlanır. Bu metot bünyesinde eğer kayıt esnasında bir hata oluşmadı ise, rental nesnesinin sahip olduğu rented değişkenine true değeri atanır. Bu durumda isRented() metodu kiralama işleminin olumlu şekilde tamamlandığını gösterecektir.

Bağımlılıkların Enjekte Edilmesi (Dependency Injection)

Kod 2.2 de görüldüğü gibi RentalService sınıfı bünyesinde bir kiralama işlemini gerçekleştirebilmek için kullanılan başka sınıflar mevcuttur. Örneğin müşteri verilerine ulaşmak için CustomerRepository ve bir rental nesnesini veri tabanına kayıtlamak için RentalRepository sınıfı kullanılmaktadır. Bu iki sınıf RentalService sınıfının bağımlılıkları (dependency) olarak adlandırılır. Kısaca RentalService sınıfı bağımlı olduğu sınıflar olmadan iş göremez.

RentalService sınıfını yakından incelediğimizde, rentACar() metodu bünyesinde new operatörü kullanılarak bağımlı olunan sınıflardan nesneler oluşturulduğu görülmektedir. Java'da new operatörü ile yeni nesneler oluşturulur. Lakin bu şekilde nesne üretmenin beraberinde getirdiği bir dezavantaj vardır. Kod içinde new operatörü kullandığı taktirde, hangi sınıftan bir nesne oluşturulmak istendiği belirtilmek zorundadır. Bu katı kodlama (hard coding) yapmak anlamına gelmektedir. new her zaman derleme esnasında kullanılan sınıf tipinin sistem tarafından tanınıyor olmasını gerektirir. Ayrıca başka bir sınıf kullanabilmek için kodun değiştirilerek, yeniden derlenmesi gerekmektedir.

Yazılım yaparken karşılaşılan en büyük zorluklardan birisi, bağımlılıkların yönetilmesidir. Kodu değiştirmek zorunda kalmadan, bağımlılıkların yönetimi mümkün olmalıdır. RentalService sınıfında bağımlılıkların katı kodlanması nedeni ile, bağımlılıkların yönetimi çok güçtür. Bağımlılıkların katı kodlanmasına gerek duyulmadan, yönetilebileceği bir mekanizmaya ihtiyaç duyulmaktadır.

new operatörünü kullanmak yerine, RentalService sınıfının ihtiyaç duyduğu nesneleri dışarıdan bu sınıfa verebilseydik nasıl olurdu? Metot parametreleri gibi, sınıfın ihtiyaç duyduğu tüm bağımlılıkları dışarıdan enjekte edebiliriz. Bu yönteme bağımlılıkların enjekte edilmesi ismi verilmektedir. RentalService sınıfı için bağımlılıkların nasıl enjekte edildiği kod 2.3 de yer almaktadır.

Kod 2.3 - RentalService

```
package com.kurumsaljava.com.spring;

import java.util.Date;

public class RentalService {

    private CustomerRepository customerRepository;
    private RentalRepository rentRepository;
    public RentalService(CustomerRepository customerRepository,
                                RentalRepository rentRepository) {
        this.customerRepository = customerRepository;
        this.rentRepository = rentRepository;
    }

    public Rental rentACar(String customerName, Car car,
                                    Date begin, Date end) {
```

```
        Customer customer =
                customerRepository.getCustomerByName(customerName);
        if (customer == null) {
            customer = new Customer(customerName);
            customerRepository.save(customer);
        }

        Rental rental = new Rental();
        rental.setCar(car);
        rental.setCustomer(customer);
        rentRepository.save(rental);
        return rental;
    }
}
```

Son yapılan değişiklik ile rentAcar() metodunda kullanılan new operatörleri kaldırmış ve bunun yerine bağımlı olunan sınıflardan oluşan sınıf değişkenleri tanımlamış olduk. Yapılan bu değişiklik ile RentalService sınıfına ihtiyaç duyduğu bağımlılıklar enjekte edilebilir hale gelmiştir. Görüldüğü gibi sınıf konstrüktörü üzerinden CustomerRepository ve RentalRepository sınıflarından oluşturulan nesneler RentalService sınıfına enjekte edilmektedir.

Bu değişikliğin ardından kod 2.1 de yer alan RentalServiceTest sınıfının aşağıdaki şekilde yeniden yapılandırılması gerekmektedir.

```
Kod 2.4 - RentalServiceTest

package com.kurumsaljava.com.spring;</code>

import static junit.framework.Assert.assertTrue;
import java.text.SimpleDateFormat;<br/>
import java.util.Date;<br/>
import org.junit.Test;<br/></code>

public class RentalServiceTest {

    @Test
    public void add_new_rental() throws Exception {
        Car car = new Car("Ford Fiesta");
        CustomerRepository customerRepository =
                    new CustomerRepository();
        RentalRepository rentalRepository =
                    new RentalRepository();
        RentalService service =
```

```
                    new RentalService(customerRepository,
                                                rentalRepository);
        Date rentalBegin =
                new SimpleDateFormat("dd/MM/yy")
                                                .parse("22/12/2013");
        Date rentalEnd =
                new SimpleDateFormat("dd/MM/yy")
                                                .parse("29/12/2013");
        Rental rental =
                service.rentACar("Özcan Acar", car,
                                            rentalBegin, rentalEnd);
        assertTrue(rental.isRented());
    }
}
```

Bir sınıfa ihtiyaç duyduğu bağımlılıkların dışarıdan enjekte edilebilmesi için doğal olarak bu bağımlılıkların sınıf dışında oluşturulmaları gerekmektedir. Kod 2.4 de görüldüğü gibi RentalService sınıfının ihtiyaç duyduğu iki bağımlılık RentalServiceTest.add*new*rental() metodunda oluşturulmakta ve bu iki bağımlılık RentalService sınıfının konstrüktörü üzerinden bu sınıftan oluşturulan nesneye enjekte edilmektedir.

RentalService sınıfı için gerek duyulan bağımlılıkların dışarıdan enjekte edilebilir hale gelmesi RentalService sınıfını daha esnek hale getirmiştir. Örneğin CustomerRepository sınıfını genişleten yeni bir alt sınıf oluşturarak, RentalService sınıfına enjekte edebiliriz. Bu yeni sınıf örneğin müşteri verilerini veri tabanından değil, başka bir kaynaktan edinmemizi sağlayabilir. RentalService sınıfını değiştirmeden sisteme böylece yeni bir davranış biçimi eklemek mümkündür. **Bağımlılıkların enjekte edilmesi metodunun ana amaçlarınından bir tanesi budur.**

Spring birçok iş için kullanılabilir, ama en güçlü olduğu saha ve tercih edilme sebeplerinin başında bağımlılıkların yönetimini ve enjekte edilmesini sağlaması gelmektedir. Bağımlılıkların enjekte edilmesi yöntemi ile kod birimleri arasında daha esnek bağ oluşturulur. Birbirini kullanan birimler kod değişikliğine gerek kalmadan değişik türde bağımlılıkları kullanabilecek şekilde yeniden yapılandırılabilir. Bu bir nevi değişik kod birimlerini bir araya getirerek, yazılım sistemi konfigüre etmek gibi bir şeydir.

Spring bağımlılıkların enjekte edilmesi yönteminde interface sınıfların kullanımına ağırlık verir. Eğer bir nesne bağımlı olduğu kod birimlerinin sadece

interface sınıflarını tanırsa, Spring bu nesneye ihtiyaç duyduğu interface sınıfların herhangi bir implementasyonunu enjekte edebilir. RentalService sınıfının Spring kullanılarak nasıl yeniden yapılandırılabileceğini bir sonraki bölümde yakından inceleyelim.

Spring İle Bağımlılıkların Enjekte Edilmesi

Bu bölümde araç kiralama servisi uygulamasını Spring kullanarak yeniden yapılandıracağız. Spring bağımlılıkların tanımlanması için bir XML dosyası kullanır. Bu konfigürasyon dosyası Spring sunucusunu oluşturmak için gerekli tüm meta bilgileri ihtiva eder. Spring sunucusu kavramını bir sonraki bölümde inceleyeceğiz. Şimdilik bunu çalışır durumda olan bir Spring uygulaması olarak düşünebilirsiniz.

Spring 3.0 ile XML dosya kullanma zorunluluğu kalkmıştır. @Configuration anotasyonu kullanılarak Java bazlı Spring konfigürasyonu yapılabilir. Yinede Spring'in temellerini daha iyi anlayabilmek için bu bölümde XML bazlı konfigürasyonu inceleyeceğiz.

Bağımlılıkların Spring tarafından yönetilebilmesi için mevcut kod birimlerinin Spring XML dosyasında tanımlanması gerekmektedir. Spring XML dosyasında tanımlanan her kod birimine Spring bean (Spring nesnesi) ismi verilmektedir. Bundan sonraki bölümlerde Spring bean yanı sıra Spring nesnesi tanımlaması ya da bean tanımlaması terimlerini kullanacağım.

Araç kiralama servisi uygulamasında Spring XML dosyasında tanımlamamız gereken Spring nesneleri RentalServiceImpl, CustomerRepositoryImpl ve RentalRepositoryImpl sınıflarıdır.

Kod 2.5 - applicationContext.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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/
                            spring-beans-3.0.xsd">

    <bean id="rentalService"
            class="com.kurumsaljava.spring.RentalServiceImpl">
```

```
        <constructor-arg ref="customerRepository" />
        <constructor-arg ref="rentalRepository" />
    </bean>

    <bean id="customerRepository"
            class="com.kurumsaljava.spring.CustomerRepositoryImpl" />
    <bean id="rentalRepository"
            class="com.kurumsaljava.spring.RentalRepositoryImpl" />
</beans>
```

Herhangi bir Java sınıfı <bean/> XML elementi kullanılarak Spring nesnesi olarak tanımlanabilir. Kod 2.5 de yer alan Spring XML dosyasında rentalService, customerRepository ve rentalRepository isimlerinde üç Spring nesne tanımlaması yer almaktadır.

<bean/> bünyesinde kullanılabilecek element özellikleri (attribute) aşağıda yer almaktadır. Bu element özelliklerinin nasıl kullanıldığını kitabın bu ve diğer bölümlerinde inceleyeceğiz.

- ***class*** - Kullanılan sınıfı tanımlar.
- ***id*** - Oluşturulan Spring nesnesinin tekil tanımlayıcısıdır.
- ***name*** - Bir Spring nesnesine id haricinde birden fazla isim vermek için kullanılır. İsimler arasında virgül kullanılarak birden fazla isim tanımlanabilir. id ve name element özelliklerine sahip olmayan Spring nesne tanımlamaları anonimdir ve doğrudan başka nesnelere enjekte edilemezler.
- ***abstract*** - Nesne tanımlamasını soyutlaştırır. Bu tür Spring nesne tanımlamalarından nesneler oluşturulmaz. Soyut Spring nesne tanımlamaları başka Spring nesnelerin üst tanımlaması olarak kullanılır.
- ***parent*** - Kullanılacak üst soyut Spring nesne tanımlamasını belirlemek için kullanılır.
- ***scope*** - Varsayılan scope singleton yani tekil scopodur. Bu Spring sunucusundan bir nesne talep ettiğimizde singleton olan aynı nesneyi edindiğimiz anlamına gelmektedir. prototype scope kullanıldığında Spring her isteğe yeni bir nesne ile cevap verir.
- ***constructor-arg*** - Sınıf konstrüktörü aracılığı ile bağımlılıklar enjekte edilmek istendiğinde kullanılır.
- ***property*** - Enjeksiyon yapılacak sınıf değişkenini tanımlar.
- ***init-method*** - Nesneler enjekte edildikten sonra koşturulacak sınıf metodunu tanımlar. Bu metodun parametresiz olması gerekmektedir.
- ***destroy-method*** - Spring sunucusu son bulduğunda sunucu bünyesinde

bulunan nesneleri sonlandırmak için kullanılan metotları tanımlar. Bu metodun parametresiz olması gerekmektedir. Sadece Spring sunucusu tarafından kontrol edilen nesneler üzerinde tanımlanabilir.
- ***factory-bean*** - Bu nesneyi oluşturabilecek fabrika sınıfını (Factory Bean) tanımlar. factory-bean kullanıldığı taktirde sınıf değişkenleri üzerinden enjeksiyon yapılmamalıdır.
- ***factory-method*** - Bu nesneyi oluşturmak için kullanılacak fabrika metodunu tanımlar.

Kod 2.3 de yer alan RentalService sınıfına tekrar göz attığımızda, bu sınıfın ve bağımlılık duyduğu diğer sınıfların somut sınıflar olduğunu görmekteyiz. Genel olarak bağımlılıkların soyut sınıflar yönünde olmasında fayda vardır, çünkü sadece bu durumda sınıfa istedigimiz bir implementasyon nesnesini enjekte edebiliriz. Bunun yanı sıra Spring bean olarak tanımladığımız sınıfın da bir interface sınıf implementasyonu olması faydalıdır. Bu şekilde Spring birden fazla implementasyonu yönetebilir. Nitekim kod 2.5 de Spring bean tanımlamalarında kullandığımız sınıflar şu interface sınıfları implemente etmektedirler:

```
public interface RentalService{
    public Rental rentACar(String customerName,
             Car car, Date begin, Date end);
}

public interface RentalRepository{
    public void save(Rental rental);
}

public interface CustomerRepository{
    public Customer getCustomerByName(String name);
    public void save(Customer customer);
}
```

Spring bir nesneye ihtiyaç duyduğu bağımlılıkları sahip olduğu sınıf değişkenleri, set metotları ya da sınıf konstrüktörleri üzerinden enjekte edebilir. Kod 2.5 de yer alan örnekte rentalService ismini taşıyan Spring nesnesi bağımlılıkları constructor-arg kullanılarak sınıf konstrüktörleri üzerinden enjekte edilmektedir. RentalServiceImpl sınıfının sahip olduğu sınıf değişkenleri ve bu değişkenlerin sınıf kontrüktörü üzerinden nasıl enjekte edildiği kod 2.6 da yer almaktadır.

```
Kod 2.6 - RentalServiceImpl
```

```
package com.kurumsaljava.spring;<br/>
import java.util.Date;

public class RentalServiceImpl implements RentalService {

    private CustomerRepository customerRepository;
    private RentalRepository rentalRepository;
    public RentalServiceImpl(CustomerRepository customerRepository,
        RentalRepository rentalRepository) {
        super();
        this.customerRepository = customerRepository;
        this.rentalRepository = rentalRepository;
    }

    @Override
    public Rental rentACar(String customerName, Car car, Date
                                                begin, Date end) {
        Customer customer =
                customerRepository.getCustomerByName(customerName);
        if (customer == null) {
            customer = new Customer(customerName);
            customerRepository.save(customer);
        }
        Rental rental = new Rental();
        rental.setCar(car);
        rental.setCustomer(customer);
        rentalRepository.save(rental);
        return rental;
    }
}
```

Spring'in bir bağımlılığı enjekte edebilmesi için hangi Spring nesnesini nereye enjekte edeceğini bilmesi gerekir. Nereye sorusuna constructor-arg ile cevap verdik. Hangi sorusuna ise constructor-arg elementinde kullanılan ref ile cevap verebiliriz. ref referans anlamına gelmektedir ve enjekte edilecek bağımlılığa işaret eder. rentalService örneğinde customerRepository ve rentalRepository isimlerini taşıyan iki Spring nesnesi sınıf konstrüktörü üzerinden rentalService nesnesine enjekte edilmektedir. Spring'in bu iki nesneyi rentalService nesnesine enjekte edebilmesi için customerRepository ve rentalRepository isimlerini taşıyan Spring nesnelerin XML dosyasında tanımlanmış olması gerekmektedir. Spring tanımlanan Spring nesne isimlerini kullanarak gerekli olan bağımlılıkları enjekte eder.

RentalServiceImpl sınıfının konstrüktöründe kullanılan parametrelerin belli bir

sırası vardır. İlk parametre CustomerRepository sınıfından bir nesne, ikinci parametre RentalRepository sınıfından bir nesnedir. RentalServiceImpl sınıfından sahip olduğu bu iki parametreli sınıf konstrüktörü üzerinden bir nesne oluştururken, parametrelerin hangi sıra ile konstrüktöre verildiği önemlidir. Spring XML dosyasında bu parametre sırası nasıl belirtilir? Kod 2.5 de yer alan XML dosyasında rentalService için böyle bir sıra tanımlaması yapmadık. Spring ilk etapta böyle bir sıralamaya ihtiyaç duymamaktadır. Spring tanımlanan Spring nesnelerin hangi Java sınıfından oluşturulduğunu (class aracılığı ile) bildiği için bağımlılıkları kontrüktöre enjekte ederken doğru nesneyi doğru sırada enjekte eder. Bir konstrüktör aynı Java sınıfından olan birden fazla parametre almadığı sürece bir sorun oluşmaz. Eğer sınıf konstrüktörü aynı Java sınıfından olan birden fazla parametreye sahip ise, Spring'in bağımlılıkları enjekte ederken kafası karışabilir. Bunu engellemek için aşağıdaki şekilde parametre sırası belirtilebilir.

```
<bean id="rentalService"
            class="com.kurumsaljava.spring.RentalServiceImpl">
    <constructor-arg ref="customerRepository" index="0"/>
    <constructor-arg ref="rentalRepository" index="1"/>
</bean>
```

Spring 3.0 dan itibaren konstrüktör parametresinin ismini kullanarak enjeksiyon yapmak mümkündür. Spring'in Java byte kodunda parametre isimlerini bulabilmesi için byte kodun debug flag kullanılarak derlenmiş olması gerekmektedir. Aksi taktirde değişken isimleri byte kod içinde olmayacağından, bu yöntem kullanılarak bağımlılıklar enjekte edilemez.

```
<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl">
    <constructor-arg name="customerRepository"
                                ref="customerRepository"/>
    <constructor-arg name="rentalRepository"
                                ref="rentalRepository"/>
</bean>
```

JDK bünyesinde yer alan @ConstructorProperties anotasyonu ile konstrüktör parametrelerini kod içinde şu şekilde işaretleyebiliriz:

```
@ConstructorProperties({"rentalRepository", "custromerRepository"})
public RentalService(RentalRepository rentalRepository,
                CustomerRepository customerRepository ) {
```

```
    this.rentalRepository = rentalRepository;
    this.customerRepository = customerRepository;
}
```

Spring ile oluşturduğumuz bu yeni yapıyı nasıl kullanabiliriz? Bunun cevabı kod 2.7 de yer almaktadır.

```
Kod 2.7 - Main

package com.kurumsaljava.spring;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.
                    ClassPathXmlApplicationContext;
public class Main {

    public static void main(String[] args) throws Exception {

        ApplicationContext ctx =
            new ClassPathXmlApplicationContext("
                                applicationContext.xml");
        RentalService rentalService =
                (RentalService) ctx.getBean("rentalService");
        Rental rental = rentalService.
                rentACar("Özcan Acar", new Car("Fiesta"),
                        getRentalBegin(), getRentalEnd());
        System.out.println("Rental status: "
                            + rental.isRented());
    }

    private static Date getRentalEnd()
                throws ParseException {
        return new SimpleDateFormat("dd/MM/yy").
                                parse("29/12/2013");
    }

    private static Date getRentalBegin()
                    throws ParseException {
        return new SimpleDateFormat("dd/MM/yy").
                                parse("22/12/2013");
    }
}
```

Idref Kullanımı

Idref ile konfigürasyon dosyasında tanımlı olan herhangi bir Spring nesnesinin ismini herhangi bir sınıf değişkenine enjekte edebiliriz.

```
<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="targetName">
        <idref bean="customerRepository" />
    </property>
</bean>
```

Bu örnekte targetName isimli değişkene bir Spring nesnesinin ismi olan customerRepository enjekte edilmemektedir. Burada targetName isimli değişkene atanan bir String değerdir. Bu değer customerRepository String nesnesidir. Bu tanimlamayı şu şekilde de yapabilirdik:

```
<bean id="rentalService"
            class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="targetName" value="customerRepository"/>
</bean>
```

idref kullanıldığı taktirde Spring uygulama çalışmaya başlamadan önce customerRepository ismini taşıyan bir Spring nesnesi tanımlaması olup, olmadığı kontrol eder. Eğer böyle bir tanımlama yoksa, Spring uygulamayı durdurur ve hata bildiriminde bulunur. Verdiğim ikinci örnekte targetName değişkenine doğrudan customerRepository değeri atandığında, Spring bu değeri kontrol etmez. Eğer yanlış bir değer kullanıldıysa, bu hata uygulama çalışırken gün ışığına çıkacaktır. Bu sebepten dolayı mevcut Spring nesne isimleri idref kullanılarak değişkenlere enjekte edilmelidir, çünkü sadece bu şekilde bu isimlerin hatalı olup, olmadıkları hemen anlaşılabilir.

idref elementinin local element özelliği kullanıldığı taktirde, Spring bu ismi taşıyan Spring nesnesinin idref'in kullanıldığı konfigürasyon dosyasında tanımlanıp, tanımlanmadığını kontrol eder. Spring böyle bir nesne tanımlaması bulamaması durumunda uygulamayı durdurur ve hata bildiriminde bulunur.

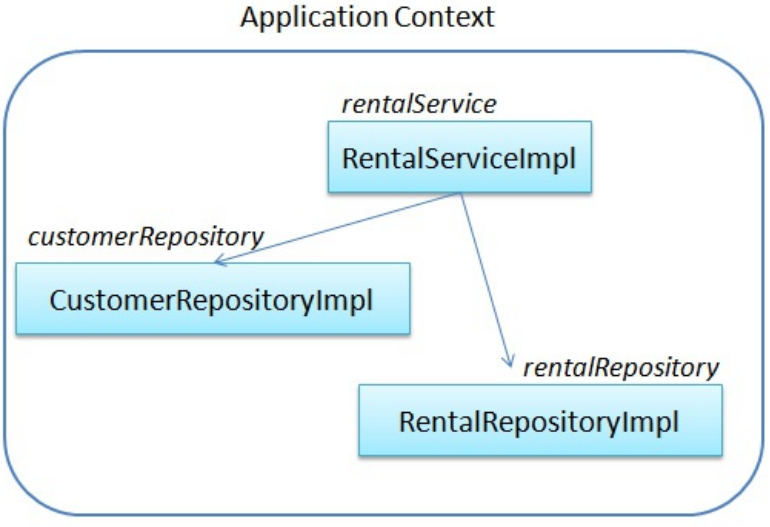
```
<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="targetName">
```

```
        <idref local="customerRepository" />
    </property>
</bean>
```

Spring Sunucusu ve BeanFactory

Spring uygulamasının çalışabilir hale gelmesi için Spring konfigürasyonunun yer aldığı XML dosyasının hafızaya yüklenmesi gerekmektedir. Bu amaçla org.springframework.context.ApplicationContext sınıfı ve türevleri kullanılır. ApplicationContext org.springframework.beans.factory.BeanFactory interface sınıfını genişletir. Spring sunucusunu temsil eden en üst sınıf BeanFactory sınıfıdır. Bu sınıf Spring sunucusunda yer alan nesnelerin konfigürasyonunu ve bu nesnelere erişimi tanımlar. BeanFactory Spring'in temel konfigürasyon yapısıdır.

ApplicationContext türevlerinden bir tanesi ClassPathXmlApplicationContext sınıfıdır. ClassPathXmlApplicationConect ile Java classpath içinde bulunan bir XML dosyası yüklenerek, Spring uygulaması çalışır hale getirilir.



Resim 2.5

Kod 2.7.1 de yer alan satır Spring sunucusunu oluşturur. Sunucuyu oluşturmak için gerekli tüm meta bilgiler applicationContext.xml ismini taşıyan XML

dosyasında yer almaktadır.

```
Kod 2.7.1 - Main

ApplicationContext ctx =
        new ClassPathXmlApplicationContext("
                        applicationContext.xml");
```

ClassPathXmlApplicationContext sınıfı kullanılarak Spring XML dosyasının yüklenmesi hafızada Application Context ismi verilen bir yapıyı oluşturur. Bunu Spring uygulamasının hafızadaki yüklenmiş hali olarak düşünebiliriz. ApplicationContext bu yapıyı temsil eden interface sınıftır. Bu yapıya Spring Container, yani Spring sunucusu ismi verilmektedir. Spring, XML dosyasında tanımlanmış tüm Spring nesneleri tarayarak, önce hiç bağımlılığı olmayan Spring nesne tanımlamalarından nesneler oluşturur. Daha sonra bu nesneleri bağımlılık duyan diğer nesnelere enjekte eder. Bu işlem sonunda her Spring nesne tanımlaması için bir nesne oluşturularak, hafızada yer alan Application Context bünyesinde konuşlandırılmış olur.

Hafızada yer alan Application Context bünyesindeki rentalService isimli Spring nesnesine erişmek için ctx.getBean("rentalService") şeklinde bir çağrı yapmamız yeterli olacaktır. getBean() metodu bize istediğimiz Spring nesnesini sanki new yapmışcasına geri verir. Artık new operatörünü kullanan biz değil, Spring'dir. Spring Java'nın Reflection özelliğinden faydalanarak nesneleri oluşturur.

Spring 3 ile oluşturulan yeni getBean() metodu ile cast yapmak zorunda kalmadan şu şekilde bir rentalService nesnesi edinebiliriz:

```
RentalService rentalService =
            ctx.getBean("rentalService", RentalService.class);
```

Eğer kullandığımız interface sınıfın sadece bir implementasyon sınıfını Spring nesnesi olarak tanımladıysak, aşağıdaki şekilde Spring nesnesinin ismini kullanmadan bir rentalService nesnesi edinebiliriz:

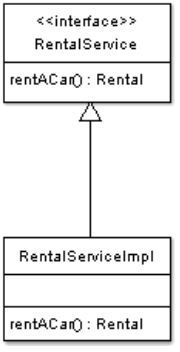
```
RentalService rentalService =
            ctx.getBean(RentalService.class);
```

Kod 2.5 de yer alan XML konfigürasyon dosyasına tekrar göz attığımızda rentalService isimli Spring nesneyi tanımlamak için RentalServiceImpl sınıfını

kullandığımız görülmekte. Nasıl oluyor da kod 2.7 de yer alan main() metodunda

```
RentalService rentalService =
            (RentalService) ctx.getBean("rentalService");
```

şeklinde bir tanımlama ile rentalService nesnesini kullanıyoruz? RentalService tanımladığımız interface sınıftır.



Resim 2.6

```
public interface RentalService {
    public Rental rentACar(String customerName, Car car,
                                    Date begin, Date end);
}
```

RentalService soyut bir interface sınıfıdır. Kod içinde RentalService interface sınıfını kullanarak somut sınıflara olan bağımlılığımızı ortadan kaldırmış oluyoruz. RentalService interface sınıfını implemente eden yeni bir sınıf oluşturarak, kod 2.7 de yer alan Main sınıfını değiştirmek zorundan kalmadan, bu oluşturduğumuz yeni implementasyon sınıfını kullanabiliriz. Bunun için yapmamız gereken tek şey, Spring XML dosyasında rentalService ismini taşıyan Spring nesnesinin kullandığı sınıfı değiştirmek olacaktır.

Tekrar kod 2.2 ye göz attığımızda, Spring'in bizim için üstlendiği görevi net olarak görmek mümkündür. Spring öncesi her bağımlılığı new ile kendimiz oluştururken, Spring ile bunu yapma zorunluluğu ortadan kalkmıştır. Ayrıca oluşturduğumuz yeni uygulamanın bağımlılıkların tersine çevrilmesi tasarım prensibine DIP - Dependency Inversion Principle uygun hale geldiğini görmekteyiz. Bu tasarım prensibi uygulamanın meydana gelebilecek değişikliklere karşı daha dayanıklı hale gelmesini sağlamaktadır. Spring ayrıca

interface sınıfları kullanmayı teşvik ederek, uygulamayı değiştirmeden uygulamaya yeni davranış biçimleri eklemeyi mümkün kılmaktadır. Daha fazlası can sağlığı demeyin. Spring'in uygulama geliştirirken bizim için üstlenebileceği daha birçok şey var. İlerleyen bölümlerde Spring'in bizim için yapabileceklerini daha yakından inceleyeceğiz. Kısaca Spring'in var olma nedenini tanımlamak istersek, "Spring yazılımcının hayatini kolaylaştırmak için vardır" dememiz yanlış olmaz.

Spring Nesne İsimlendirme

Application Context bünyesinde yer alan her Spring nesnesinin bir ya da birden fazla ismi vardır. Spring sunucusu bünyesinde bu isimlerin tekil olması gerekmektedir. Aynı ismi iki değişik Spring nesnesi paylaşamaz.

Şimdiye kadar kullandığımız örneklerde id element özelliği aracılığı ile oluşturduğumuz Spring nesne tanımlamalarına isimler verdik. Tanımladığımız bu isimler aracılığı ile Spring nesnelerine erişmemiz mümkün oldu.

id element özelliği bünyesinde sadece bir isim barındırabilir, yani her nesne tanımlamasına id element özelliği ile sadece bir isim atanabilir. Bunun yanı sıra Spring 3.1 öncesi id element özelliği bünyesinde / ve : gibi işaretler kullanmak mümkün değildi. Spring 3.1 ile aşağıdaki şekilde bir nesne tanımlaması mümkün olmuştur.

```
<bean id="clio/myclio" .../>

Car clio = ctx.getBean("clio/myclio", Car.class);
```

Bu değişiklik ne yazık ki bir Spring nesne tanımlamasına birden fazla ismi atayamama sorununu çözmemektedir. Bu sorunu çözmek için name element özelliği kullanılabilir. Örneğin yukarıda tanımlamış olduğumuz clio nesnesine aynı zamanda myclio ya da yourclio isimlerini atamak isteseydik, o zaman şu şekilde bir bean tanımlaması yapardık:

```
<bean id="clio" name="myclio, yourclio" .../>
```

Bir Spring nesnesi tanımlarken id ya da name element özelliklerini kullanma zorunluluğu bulunmamaktadır. Bu element özellikleri ile Spring nesnesine bir isim atanmadığı taktirde, Spring sunucusu bu nesneye otomatik olarak bir isim

atar. İsmi olmayan Spring nesneleri anonimdir ve isimleri olmadığı için ref element özelliği kullanılarak diğer nesnelere enjekte edilemezler.

Mevcut Spring nesnelerine yeni isimler atamak için kullanılabilecek diğer bir element <alias/> elementidir. Örneğin

```
<alias name="birIsim" alias="baskaBirIsim"/>
```

ile birIsim ismini taşıyan Spring nesnesine baskaBirIsim ismini kullanarak ta erişmek mümkün olmaktadır.

Bağımlılıkları Enjekte Etme Türleri

RentalServiceImpl sınıfının ihtiyaç duyduğu bağımlılıkları sınıf konstrüktörü üzerinden constructor-arg elementi ile enjekte etmiştik. Bu bağımlılıkları enjekte etmek için kullanabileceğimiz yöntemlerden bir tanesidir. Bunun yanı sıra bağımlılıklar property elementi ile de enjekte edilebilir.

```
<bean id="rentalService"
            class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="customerRepository" ref="customerRepository" />
    <property name="rentalRepository" ref="rentalRepository" />
</bean>
```

property elementi sınıf bünyesindeki bir sınıf değişkenine işaret etmektedir. Hangi sınıf değişkenine bağımlılığın enjekte edilmesi gerektiğini name element özelliği ile tanımlıyoruz. Kllandığımız örnekte (bakınız kod 2.6) sınıf değişkeninin ismi customerRepository ya da rentalRepository'dir. Eğer sınıf değişkenleri bu isim ile tanımlanmadı ise, Spring istenilen bağımlılıkları enjekte edemez. Hangi bağımlılığın sınıf değişkenine enjekte edilmesi gerektiğini ref element özelliği ile tanımlıyoruz. Kod 2.5 e tekrar göz attığımızda customerRepository ve rentalRepository ismini taşıyan iki Spring nesnesi tanımlamasının mevcut olduğunu görmekteyiz. Bu örnekte sınıf değişkenleri ve Spring nesneleri aynı ismi taşımaktadır.

Bağımlılıkların property elementi kullanılarak enjekte edilebilmesi için sınıf değişkenlerinin set() metotlarına sahip olması gerekmektedir. Bunun yanı sıra sınıf parametresiz bir sınıf konstrüktörüne ihtiyaç duymaktadır.

Kod 2.8 - RentalServiceImpl

```
package com.kurumsaljava.spring;
import java.util.Date;
public class RentalServiceImpl implements RentalService {

    private CustomerRepository customerRepository;
    private RentalRepository rentalRepository;

    public RentalServiceImpl() {

    }

    public RentalServiceImpl(
        CustomerRepository customerRepository,
        RentalRepository rentalRepository) {
        super();
        this.customerRepository = customerRepository;
        this.rentalRepository = rentalRepository;
    }

    @Override
    public Rental rentACar(String customerName, Car car,
                               Date begin, Date end) {
        Customer customer =
            customerRepository.getCustomerByName(customerName);
        if (customer == null) {
            customer = new Customer(customerName);
            customerRepository.save(customer);
        }

        Rental rental = new Rental();
        rental.setCar(car);
        rental.setCustomer(customer);
        rentalRepository.save(rental);
        return rental;
    }

    public CustomerRepository getCustomerRepository() {
        return customerRepository;
    }

    public void setCustomerRepository(CustomerRepository
                                      customerRepository) {
        this.customerRepository = customerRepository;
    }
```

```
    public RentalRepository getRentalRepository() {
        return rentalRepository;
    }

    public void setRentalRepository(RentalRepository
                                rentalRepository) {
        this.rentalRepository = rentalRepository;
    }
}
```

Eğer bağımlılıkların enjekte edilmesi için property elementini kullandıysak, Spring bağımlılıkların enjekte edileceği sınıftan parametresiz sınıf konstrüktörünü kullanarak yeni bir nesne oluşturur. Akabinde Spring setCustomerRepository() ve setRentalRepository() metotlarını kullanarak tanımlanmış olan bağımlılıkları enjekte eder. Spring'in bir RentalServiceImpl nesnesini nasıl oluşturduğunu anlamak için bir sonraki kod bloğuna göz atalım.

```
RentalServiceImpl service = new RentalServiceImpl();
CustomerRepositoryImpl customerRepository =
                    new CustomerRepositoryImpl();
RentalRepositoryImpl rentalRepository =
                    new RentalRepositoryImpl();
service.setCustomerRepository(customerRepository);
service.setRentalRepository(rentalRepository);
```

Eğer kendimiz bir RentalServiceImpl nesnesi oluşturmak isteseydik, bir önceki kod bloğundaki gibi işlem yapmamız gerekirdi. Oysaki Spring aracılığı ile bir RentalServiceImpl nesnesi edinmek için yapmamız gereken tek işlem

```
RentalService rentalService =
        (RentalService) ctx.getBean("rentalService");
```

şeklindedir.

Bağımlılıkların enjekte edilmesi için hangi yöntemi kullanmalıyım sorusu aklınıza gelmiş olabilir. Seçiminizi sadece bir yönde yapmak zorunda değilsiniz. Her iki yöntemi de birlikte kullanabilirsiniz. Bir sonraki kod bloğunda hem constructor-arg hem de property elementleri beraber kullanılmaktadır.

```
<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl">
    <constructor-arg ref="customerRepository"/>
```

```
    <property name="rentalRepository" ref="rentalRepository" />
</bean>
```

Spring her iki enjekte metodu birlikte kullanıldığında, bir sonraki kod bloğundaki işlemleri gerçekleştirir.

```
CustomerRepositoryImpl customerRepository =
                new CustomerRepositoryImpl();
RentalServiceImpl service =
                new RentalServiceImpl(customerRepository);
RentalRepositoryImpl rentalRepository =
                new RentalRepositoryImpl();
service.setRentalRepository(rentalRepository);
```

Eğer sonradan değiştirilemeyen, sabit (immutable) nesneler oluşturmak istiyorsanız, seçiminiz constructor-arg yönünde olmalıdır. Bir sınıftan sabit bir nesne oluşturmak için set metotlarının kaldırılması ve sadece bağımlıkları parametre olarak alan bir sınıf konstrüktörü kullanılması gerekir. Örneğin RentalServiceImpl sınıfından sabit bir nesne oluşturmak için kod 2.9 da yer alan implementasyon kullanılabilir.

```
Kod 2.9 - RentalServiceImpl

package com.kurumsaljava.spring;
import java.util.Date;
public class RentalServiceImpl implements RentalService {

    private final CustomerRepository customerRepository;
    private final RentalRepository rentalRepository;

    public RentalServiceImpl(
            CustomerRepository customerRepository,
            RentalRepository rentalRepository) {
        super();
        this.customerRepository = customerRepository;
        this.rentalRepository = rentalRepository;
    }

    @Override
    public Rental rentACar(String customerName, Car car,
                           Date begin, Date end) {
        Customer customer =
            customerRepository.getCustomerByName(customerName);
        if (customer == null) {
```

```
            customer = new Customer(customerName);
            customerRepository.save(customer);
        }

        Rental rental = new Rental();
        rental.setCar(car);
        rental.setCustomer(customer);
        rentalRepository.save(rental);
        return rental;
    }
}
```

Listelerin Enjekte Edilmesi

Spring nesneler yanı sıra konfigürasyonu tanımlanan listeleri de enjekte edebilir. Örneğin RentalServiceImpl sınıfına kiralanabilecek araç listesini enjekte etmek isteseydik, bir sonraki kod bloğundaki gibi araç listesini oluştururduk.

```
bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl">
    <constructor-arg ref="customerRepository" />
    <constructor-arg ref="rentalRepository" />
    <property name="carList">
        <list>
            <ref bean="fiesta" />
            <ref bean="clio" />
        </list>
    </property>
</bean>
<bean id="fiesta" class="com.kurumsaljava.spring.Car">
    <constructor-arg name="brand" value="ford" />
    <constructor-arg name="model" value="fiesta" />
</bean>
<bean id="clio" class="com.kurumsaljava.spring.Car">
    <constructor-arg name="brand" value="renault" />
    <constructor-arg name="model" value="clio" />
</bean>
```

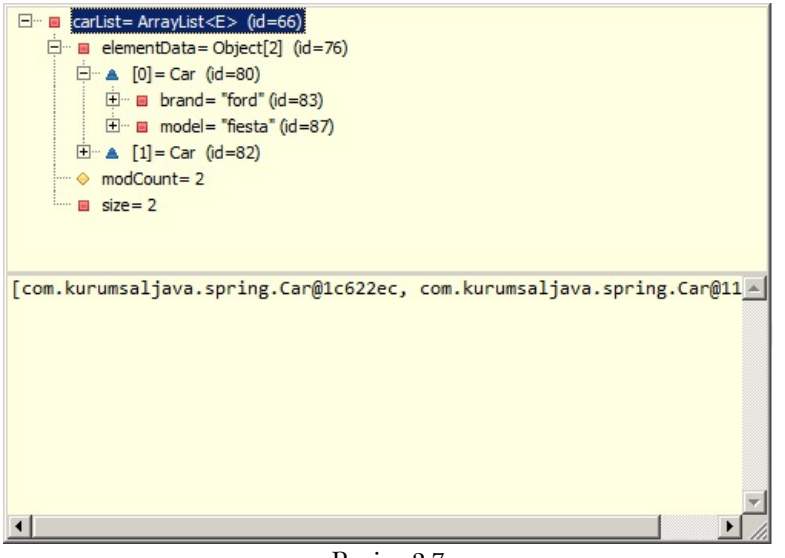
Araç listesinin RentalServiceImpl sınıfından oluşturulan bir nesneye enjekte edilebilmesi için RentalService bünyesinde bir sonraki kod bloğunda görüldüğü gibi List veri tipinde bir listenin tanımlanması gerekmektedir.

```
private List<Car> carList;
```

```
public List<Car> getCarList() {
    return carList;
}

public void setCarList(List<Car> carList) {
    this.carList = carList;
}
```

Main sınıfını Eclipse Debugger ile koşturduğumuzda, Spring'in XML dosyasında tanımlandığı şekilde iki Car nesnesini oluşturarak, RentalServiceImpl sınıfında yer alan carList listesini eklediğini görmekteyiz.



Resim 2.7

Arka planda Spring'in araç listeni oluşturmak için yaptığı işlemler bir sonraki kod bloğunda yer almaktadır.

```
CustomerRepositoryImpl customerRepository =
                new CustomerRepositoryImpl();
RentalRepositoryImpl rentalRepository =
                new RentalRepositoryImpl();
RentalServiceImpl service =
            new RentalServiceImpl(customerRepository,
                                  rentalRepository);
Car fiesta = new Car("ford", "fiesta");
Car clio = new Car("renault", "clio");
List<Car> carList = new ArrayList<Car>();
```

```
carList.add(fiesta);
carList.add(clio);
service.setCarList(carList);
```

Map tipi bir koleksiyon aşağıdaki şekilde enjekte edilebilir.

```
<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="carMap">
        <map>
            <entry key="fiesta">
                <ref bean="fiesta" />
            </entry>
            <entry key="clio">
                <ref bean="clio" />
            </entry>
       </map>
    </property>
</bean>
```

Set tipi bir koleksiyon aşağıdaki şekilde enjekte edilebilir.

```
<bean id="rentalService"
            class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="carSet">
        <set>
            <ref bean="fiesta" />
            <ref bean="clio" />
        </set>
    </property>
</bean>
```

java.util.Properties tarzı bir özellik listesi aşağıdaki şekilde enjekte edilebilir.

```
<bean id="rentalService"
            class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="carProperty">
        <props>
            <prop key="ford"> fiesta </prop>
            <prop key="renault"> clio </prop>
        </props>
    </property>
</bean>
```

Basit Değerlerin Enjekte Edilmesi

Enjekte edilmek istenen değerler int ya da String gibi basit değerler olabilir. property elementi kullanılarak bu tür enjeksiyonları gerçekleştirmek mümkündür. Örneğin CustomerRepositoryImpl sınıfında database ismini taşıyan bir String değişken olsaydı, bu değişkene aşağıdaki şekilde bir değer enjeksiyonu yapabilirdik.

```
<bean id="customerRepository"
        class="com.kurumsaljava.spring.CustomerRepositoryImpl">
    <property name="database" value="oracle" />
</bean>
```

Yukarıda yer alan örnekte CustomerRepositoryImpl sınıfının database isimli sınıf değişkenine oracle değeri enjekte edilmektedir. Bunun sağlanabilmesi için CustomerRepository isimli sınıfın database isimli bir değişkene ve bu değişkene değer atanmasında kullanılacak setDatabase() metoduna sahip olması gerekmektedir. Spring'in customerRepository isimli bir nesneyi oluşturmak için yaptığı işlem aşağıda yer almaktadır.

```
CustomerRepositoryImpl customerRepository =
                    new CustomerRepositoryImpl();
customerRepository.setDatabase("oracle");
```

value elementini kullanarak ta deüişkenlere değerler enjekte edilebilir:

```
<property name="database"><value>oracle</value></property>
```

ya da

```
<constructor-arg type=int><value>10000</value></constructor-arg>
```

Null ya da Boş String Değerinin Enjekte Edilmesi

property elementinin value element özelliği boş bırakıldığı taktirde, bu değişkene sıfır uzunluğunda bir String nesnesinin enjekte edildiği anlamana gelmektedir.

```
<bean id="customerRepository"
           class="com.kurumsaljava.spring.CustomerRepositoryImpl">
```

```
    <property name="database" value="" />
</bean>
```

Eger bir değişkene null degerini atamak istiyorsak, <null/> elementini kullanabiliriz.

```
<bean id="customerRepository"
        class="com.kurumsaljava.spring.CustomerRepositoryImpl">
    <property name="database"><null/></property>
</bean>
```

Nesne Oluşturma Sırasının Tanımlanması

Normal şartlar altında iki nesne arasındaki bağımlılık ref elementi kullanıldığı taktirde, nesnelerin oluşturulma sırası tanımlanmış olur. Spring bu durumda ilk önce enjekte edilecek nesneyi oluşturur, daha sonra bu nesnenin enjekte edileceği diğer nesneyi. Bunun yanı sıra depends-on elementi ile de nesne oluşturma sırası tayin edilebilir. Aşağıda yer alan örnekte Spring customerRepository nesneyi oluşturmadan önce, dbManager isimli nesneyi oluşturur. Bu genel olarak bir nesnenin sebep olduğu yan etkilere bağımlı olunduğu durumlar kullanılan bir yöntemdir.

```
<bean id="customerRepository"
        class="com.kurumsaljava.spring.CustomerRepositoryImpl"
                depends-on="dbManager"/>
<bean id="dbManager"
        class="com.kurumsaljava.spring.DbManager" />
```

depends-on bünyesinde birden fazla bean ismi tanımlaması şu şekilde yapılabilir:

```
<bean id="customerRepository"
        class="com.kurumsaljava.spring.CustomerRepositoryImpl"
    depends-on="dbManager, txManager, myBean"/>
```

Uygulama son bulduğunda da tanımlanan bu sıranın tam tersine göre nesneler imha edilir.

Otomatik Veri Tipi Dönüşümü

Basit değerlerin enjeksiyonu için property elementinin value özelliği ile nasıl beraber kullanıldığını bir önceki bölümde gördük. value özelliği içinde tanımlanan değer yapı itibari ile bir String nesnesidir. Eğer bir int veri tipine sahip bir değişkene değer enjekte etmek istersek, value özelliğinde yer alan bu değer Spring tarafından otomatik olarak bir int değerine dönüştürülür. Aşağıda yer alan kod örneğinde int veri tipinde olan port değişkenine 1234 değeri enjekte edilmektedir. Bu değer otomatik olarak Spring tarafından 1234 rakamına dönüştürülerek, port isimli int değişkenine enjekte edilmektedir.

```
<bean id="customerRepository"
        class="com.kurumsaljava.spring.CustomerRepositoryImpl">
    <property name="port" value="1234" />
</bean>
```

float, double gibi diğer basit verit tipleri için de Spring tarafından otomatik veri tipi dönüşümü sağlanır.

Tekil Nesneler ve Bean Scope

Çoğu yazılımcı tarafından tasarım şablonu olarak görülmese de tekillik singleton tasarım şablonu ismini taşıyan bir tasarım şablonu mevcuttur. Tekillik tasarım şablonu ile bir sınıftan sadece bir nesne oluşturulması amaçlanır.

Spring için varsayılan nesne oluşturma ayarı tekil nesnedir. Bunun ispatı için bir sonraki kod bloğunu inceleyelim.

```
System.out.println((RentalService) ctx.getBean("rentalService"));
System.out.println((RentalService) ctx.getBean("rentalService"));
```

Ekra çıktısı:

```
com.kurumsaljava.spring.RentalServiceImpl@18c908b
com.kurumsaljava.spring.RentalServiceImpl@18c908b
```

getBean() metodu ile rentalService nesnesini edindiğimizde, sahip oldukları adres alanlarının (@18c908b) aynı olduğunu görmekteyiz. Bu Spring'in rentalService ismini taşıyan nesneyi hafızada sadece bir kez tuttuğunun kanıtıdır. Eğer getBean() metodu üzerinden her defasında yeni bir rentalService nesnesi edinmek istiyorsak, rentalService'in bean tanımlamasını bir sonraki kodda yer

aldığı şekilde değiştirmemiz gerekmektedir.

```
CustomerRepositoryImpl customerRepository =
                new CustomerRepositoryImpl();
customerRepository.setDatabase("oracle");

<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl"
                        scope="prototype">
    <constructor-arg ref="customerRepository" />
    <constructor-arg ref="rentalRepository" />
</bean>
```

Tekil nesne oluşturmayı önleyen scope="prototype" tanımlamasıdır. scope ile nesnenin ömrü tanımlanır. scope bünyesinde kullanılabilecek değerler şunlardır:

- ***singleton***: Tekil nesne tanımlamak için kullanılır. Spring tarafından varsayılan scope singletondır.
- ***prototype***: Spring sunucusundan her talep için yeni bir nesne oluşturulur.
- ***request***: Web uygulamalarında bir istek (http request) boyunca geçerli olan nesne oluşturulmak için kullanılır. Sadece web uyumlu bir Application Context (örneğin Spring MVC uygulaması) oluşturulduğunda kullanılabilir.
- ***session***: Web uygulamalarında kullanıcı oturumu (http session) boyunca geçerli olan nesne oluşturmak için kullanılır. Sadece web uyumlu bir Application Context (örneğin Spring MVC uygulaması) oluşturulduğunda kullanılabilir.
- ***globalSession***: Portlet uygulamalarında tüm uygulama parçaları için geçerli olan kullanıcı oturumunda (global http session) kullanılmak üzere nesne oluşturmak için kullanılır. Sadece web uyumlu bir Application Context (örneğin Spring MVC uygulaması) oluşturulduğunda kullanılabilir.
- ***custom***: Yazılımcı tarafından oluşturulur Spring sunucusu yazılımcının tayin ettiği şekilde nesne oluşturma sürecini yönlendirir.

Scope Bazlı Nesnelerin Enjeksiyonu

Scope bazlı nesnelerin bazı şartlar altında singleton ya da prototype olan nesnelere enjeksiyonu sorun oluşturabilir. Aşağıda yer alan örnekte session scope olan userDetails nesnesi singleton scope olan rentalService nesnesine enjekte edilmektedir. Burada nasıl bir sorun mevcuttur?

```
<bean id="userDetails"
    class="com.kurumsaljava.spring.UserDetails"
        scope="session"/>

<bean id="rentalService"
    class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="userDetails" ref="userDetails"/>
</bean>
```

Singleton olan nesneler Application Context oluştuğunda Spring sunucusu tarafından sadece bir kere yapılandırılırlar. Yukarıda yer alan örnekte singleton olan rentalService nesnesine session scope sahibi userDetails enjekte edilmektedir. rentalService nesnesi oluşturulduğu esnada Application Context bünyesinde userDetails isminde bir nesne olamaz, çünkü böyle bir nesne kullanıcının bir web oturumu (HttpSession) açmasıyla var olabilir. Bu yüzden singleton olan rentalService nesnesine session scope olan bir nesne enjekte edilmesi mümkün değildir. Kısaca burada uygulamanın ilerleyen bir safhasında oluşturulacak bir nesne var oluşunun çok öncesinde mevcut bir nesneye enjekte edilmeye çalışılmaktadır ki bu mümkün değildir.

Bu sorunu userDetails nesnesine vekillik edecek bir nesne oluşturup, bu vekil nesneyi rentalService nesnesine enjekte ederek çözebiliriz. Vekil nesne rentalService tarafından kullanıldıgında, vekil olduğu nesneyi içinde buludugu scopedan alarak, rentalService nesnesine verecektir.

Bu gibi durumlarda kullanılmak üzere vekil nesne oluşturmak için aop isim alanında yer alan scoped-proxy elementi kullanılabilir.

```
<bean id="userDetails"
    class="com.kurumsaljava.spring.UserDetails"
            scope="session">
    <aop:scoped-proxy/>
</bean>
```

aop:scoped-proxy elementi kullanıldığında Spring sunucusu CGLIB kütüphanesi yardımıyla bir vekil nesne oluşturur. userDetails nesnesinin enjekte edildigi her nesneye bu vekil nesne enjekte edilmiş olur, çünkü vekil nesne userDetails nesnesinin sahip olduğu sınıfı genişletmektedir. Bu vekil nesne userDetails nesnesine yöneltilen tüm istekleri karşılar. Kullanılan scope türüne göre vekil nesne vekil olduğu nesneyi ait oldugu scopedan alarak, kullanıcı nesnenin

geçerli nesne üzerinde işlem yapmış olmasını sağlar.

scoped-proxy aşağıdaki şekilde tanımlandığında Spring sunucusu CGLIB yerine JDK interface bazlı dinamik vekil oluşturma mekanizmasını kullanır. Bu durumda mutlaka vekil olunan nesnenin bir interface sınıfı implemente etmiş olması gerekmektedir. Sadece bu durumda vekil nesneler oluşturulabilir.

```
<aop:scoped-proxy proxy-target-class="false"/>
```

Tanımlanabilir (Custom) Bean Scope

Spring tarafından kullanılan bean scope mekanizması genişletilebilir yapıdadır. singleton ve prototype bean scopelar harici mevcut bean scopeları değiştirebilir ya da kendi bean scopelarımızı oluşturabiliriz. Yeni bir bean scope oluşturmak için org.springframework.beans.factory.config. Scope interface sınıfını implemente eden bir alt sınıf oluşturmamız gerekmektedir. Yeni bir scope oluşturma işlemini bir örnek üzerinde yakından inceleyelim.

Kod 2.9.1 de ThreadScope ismini taşıyan yeni bir scope sınıfı yer almaktadır. Bean tanımlaması yaparken scope=thread kullanılarak oluşturulan nesnelerin ömürlerini bir threadin ömrü ile sınırlı tutmak istiyoruz. Bu yeni scope tanımlaması kullanıldığı taktirde oluşturulan nesneler kullanılan thread hayatta olduğu sürece kullanımda olacaktır. Threadin son bulmasıyla sahip olduğu nesneler yok edilir.

```
Kod 2.9.1 - ThreadScope

package com.kurumsaljava.spring.scope;

import java.util.HashMap;
import java.util.Map;

import org.springframework.beans.factory.ObjectFactory;
import org.springframework.beans.factory.config.Scope;
import org.springframework.core.NamedThreadLocal;

public class ThreadScope implements Scope {

    private final ThreadLocal<Map<String, Object>> threadScope =
            new NamedThreadLocal<Map<String, Object>>(
            "ThreadScope") {
```

```
        @Override
        protected Map<String, Object> initialValue() {
            return new HashMap<String, Object>();
        }
    };

    @Override
    public Object get(String name, ObjectFactory objectFactory) {
        Map<String, Object> scope = threadScope.get();
        Object object = scope.get(name);
        if (object == null) {
            object = objectFactory.getObject();
            scope.put(name, object);
        }
        return object;
    }

    @Override
    public Object remove(String name) {
        Map<String, Object> scope = threadScope.get();
        return scope.remove(name);
    }

    @Override
    public void registerDestructionCallback(String name,
                        Runnable callback) {

    }

    @Override
    public Object resolveContextualObject(String key) {
        return null;
    }

    @Override
    public String getConversationId() {
        return Thread.currentThread().getName();
    }

}
```

Oluşturduğumuz yeni scope sınıfını Spring'e tanıtmamız gerekiyor. Bu amaçla kod 2.9.2 de yer aldığı gibi CustomScopeConfigurer sınıfının sahip olduğu scopes isimli listeye thread ismi altında ThreadScope sınıfını ekliyoruz.

```
Kod 2.9.2 - applicationContext.xml
```

```
<bean
    class="org.springframework.beans.factory.config.
                                CustomScopeConfigurer">
    <property name="scopes">
        <map>
            <entry key="thread">
                <bean
                  class="com.kurumsaljava.spring.scope.
                                    ThreadScope"/>
            </entry>
        </map>
    </property>
</bean>

<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl"
        scope="thread">
        <property name="customerRepository"
                        ref="customerRepository" />
        <property name="rentalRepository"
                        ref="rentalRepository" />
</bean>
```

Spring scope-thread ile karşılaştığı yerlerde ThreadScope sınıfının get() metodu üzerinden talep edilen nesneyi oluşturup bir ThreadLocal içindeki listeye (Map) ekleyecektir. Bu liste içinde yer alan nesneler tekrar oluşturulmadan kullanıcıya geri verilir.

Metot Enjeksiyonu

Spring'in varsayılan nesne oluşturma ayarının singleton yani tekil nesne olduğunu gördük. Tekil nesneler Spring tarafından bir kez oluşturulur ve kullanıma sunulur. Bu sebepten dolayı tekil olan bir nesneye prototype tipinde olan bir nesneyi devamlı enjekte etmek mümkün değildir, çünkü bağımlılığın enjekte edilmesi işlemi nesne oluşturulurken bir kez yapılan bir işlemdir. Prototype tipinde olan nesneler Spring tarafından devamlı yeniden oluşturulur, ama böyle bir nesneye bağımlılık duyan bir tekil nesne sürekli yeni prototype nesneyi enjeksiyon yöntemiyle edinemez, çünkü Spring tekil nesnelere oluşturulma süreçlerinden sonra bağımlılıkların enjeksiyonu konusunda ilgi göstermez.

Kod 2.9.3 de yer alan örnekte Singleton nesnesine prototype nesnesi konstrüktör üzerinde enjekte edilmektedir. Singleton sınıfının Spring konfigürasyon dosyasında tekil, Prototype sınıfının prototype nesne olarak tanımlandığını düşünelim. Application Context oluştuktan sonra Spring bir defaya mahsus olmak üzere Singleton sınıfından olan tekil nesneye prototype değişkenini enjekte edecektir. Bu işlemin ardından Singleton sınıfından olan tekil nesnenin yeni bir prototype nesnesi edinmesi imkansızdır. Bu sorunu çözmek için metot enjeksiyonu yöntemini kullanabiliriz. Spring konstrüktör ve set() metotları bazlı enjeksiyon yanı sıra, mevcut bir sınıf metodunu değiştirerek, sınıfa yeni bir davranış biçimi kazandırmak için kullanılabilecek metot enjeksiyonu yöntemini sunmaktadır. Kod 2.9.3 - Singleton

```
public class Singleton {

    private Prototype prototype;

    public Singleton(Prototype prototype) {
        this.prototype = prototype;
    }

    public void doSomething() {
        prototype.foo();
    }
}
```

Tekil bir nesneye ihtiyaç duyduğu prototype tipinde bir nesneyi verebilmek için prototype nesne edinme işlemini soyut bir metot olarak tanımlayabiliriz. Kod 2.9.4 de yer alan örnekte Singleton sınıfına prototype tipindeki nesneye konstrüktör aracılığı ile enjekte etmek yerine, createPrototype() isminde soyut bir metot tanımlıyoruz. createPrototype() metodun soyut metot olarak tanımlanabilmesi için Singleton sınıfının da soyut olarak tanımlanması gerekmektedir. Bu şartlar altında Spring'in Singleton sınıfından yeni bir nesne oluşturamayabileceğini düşünebilirsiniz. Metot enjeksiyon yöntemiyle bu mümkün.

```
Kod 2.9.4 - Singleton

public abstract class Singleton {

    private Prototype prototype;
```

```
    public Singleton() {
    }

    public void doSomething() {
        createPrototype().foo();
    }

    protected abstract Prototype createPrototype();
}
```

Metot enjeksiyonu yöntemi kullanıldığında Spring CGLIB kütüphanesini kullanarak Singleton sınıfından olma yeni bir altsınıf oluşturur. Dinamik olarak oluşturulan bu altsınıf soyut olan createPrototype() metodunu implemente eder. Bu implementasyon tekil nesneye yeni bir prototype nesneyi kullanma fırsatı tanır. Bu şekilde tekil nesneye dolaylı olarak bir prototype nesne enjekte edilmiş olur. Bu işlemi gerçekleştirmek için kullanılan yöntem metot enjeksiyonudur. Kod 2.9.5 de metot enjeksiyonu yöntemini kullanmak için gerekli Spring konfigürasyonu yer almaktadır. lookup-method elementi ile değiştirilmesi yani dinamik olarak implemente edilmesi gereken metot ismi tanımlanmaktadır.

```
Kod 2.9.5 - app-config.xml

<bean id="prototype"
        class="com.kurumsaljava.spring.Prototype" scope="prototype"/>

<bean id="singleton"
        class="com.kurumsaljava.spring.Singleton">
    <lookup-method name="createPrototype" bean="prototype"/>
</bean>
```

Dinamik olarak bir altsınıfın oluşturulabilmesi için tekil nesnenin ait olduğu sınıfın ve implemente edilen metodun final olmaması gerekmektedir. Bunun yanı sıra tekil nesnenin ait olduğu sınıf soyut olarak tanımlandığı için, bu sınıfı test edebilmek için bir altsınıfın oluşturulması gerekmektedir. Ayrıca metot enjeksiyon yöntemiyle oluşturulan altsınıflar üzerinde serialization işlemi yapılamaz. Spring 3.2 ile CGLIB kütüphanesinin classpath içinde olma zorunluluğu bulunmamaktadır, çünkü bu kütüphanede yer alan sınıflar org.springframework paketine alınmıştır.

Metot Değiştirme Yöntemi

replaced-method konfigürasyon elementini kullanarak bir nesnenin sahip olduğu

metot yerine başka bir nesnenin sahip olduğu metodu koşturabiliriz. Bu yönteme method overriding ismi verilmektedir. Nasıl altsınıflar üstsınıflarda yer alan metotları @Override anotasyonunu kullanarak yeniden yapılandırabiliyorlarsa, Spring'de mevcut bir metodu başka bir metotla değiştirebilmekte ve nesneye yeni bir davranış biçimi katabilmektedir. Bu metot değiştirme mekanizması da Spring tarafından kullanılan metot enjeksiyon yöntemiyle sağlanmaktadır.

Kod 2.9.5 de yer alan örnekte koşturmak istediğimiz metot doA() metodudur.

```
Kod 2.9.5 - AClass

public class AClass {

    public String doA(String a) {
        System.out.println(a);
        return a;
    }
}
```

Kullanılacak yeni metodu tanımlamak için org.springframework.beans.factory.support.MethodReplacer sınıfını implemente eden yeni bir sınıf oluşturmamız gerekmektedir. Spring'in doA() yerine koşturacağı metot kod 2.9.6 da ye alan reimplement() metodudur.

```
Kod 2.9.6 - ReplacerClass

public class ReplacerClass implements MethodReplacer {

    public Object reimplement(Object o, Method m, Object[] args)
                            throws Throwable {
        // args[0] doA() metodunda yer alan parametredir.
        String input = (String) args[0];
        ...
        return ...;
    }
}
```

Gerekli konfigürasyon kod 2.9.7 de yer almaktadır. replaced-method elementi ile orijinal metot ve bu metodun yerine geçecek yeni metot tanımlanmaktadır. arg-type elementi ile yeni metot bünyesinde kullanılacak orijinal metot parametreleri tanımlanmaktadır.

```
Kod 2.9.7 - app-config.xml

<bean id="aClass" class="com.kurumsaljava.spring.AClass">
    <replaced-method name="doA" replacer="replacer">
        <arg-type>String</arg-type>
    </replaced-method>
</bean>

<bean id="replacer" class="com.kurumsaljava.spring.ReplacerClass"/>
```

Fabrika Metodu

Her sınıfın kullanıma açık sınıf konstrüktörü olmayabilir. Bu durum genelde tekil (singleton) nesneler için geçerlidir. Bir sınıftan birden fazla nesne üretimini engellemek için sınıf konstrüktörleri private olarak işaretlenir. Bu tür sınıflardan new kullanılarak nesne üretilmesi mümkün değildir. bean elementinin factory-method özelliği aracılığı ile tekil bir nesnenin Spring ile kullanımı mümkündür.

```
Kod 2.10 - DBSingleton

package com.kurumsaljava.spring;
import java.sql.Connection;
public class DBSingleton {

    private static final DBSingleton instance = new DBSingleton();

    private DBSingleton() {

    }

    public static final DBSingleton getInstance() {
        return instance;
    }

    public Connection getConnection() {
        return null;
    }
}
```

Normal şartlar altında kod 2.10 da yer alan DBSingleton sınıfının kullanımı

```
DBSingleton.getInstance().getConnection();
```

şeklinde olacaktır. Bu sınıfı bir Spring nesnesi olarak tanımlayıp, kullanmak istersek, factory-method özelliği bu sınıfın tekil nesnesini kullanmamızı mümkün kılacaktır.

```
<bean id="dbSingleton"
        class="com.kurumsaljava.spring.DBSingleton"
    factory-method="getInstance" />
```

factory-method kullanılarak tanımlanan sınıfın static olması gerekmektedir. Bu tanımlamanın ardından

```
DBSingleton dbSingletion = (DBSingleton) ctx.getBean("dbSingleton");
Connection connection = dbSingletion.getConnection();
```

şeklinde DBSingleton'ın ihtiva ettiği tekil nesneyi kullanabiliriz.

Fabrika Sınıfları

Bir Spring nesnesi tanımlaması yaparken class element özelliğinden faydalandık. Bu Spring'e hangi sınıfı kullanarak bir nesne oluşturması gerektiği bilgisini vermektedir. class ve factory-method element özellikleri harici factory-bean element özelliği kullanılarak ta bir Spring nesnesi tanımlaması yapılabilir. factory-bean kullanıldığı taktirde, bir POJO (Plain Old java Object) sınıfı nesneleri üreten fabrika (factory) olarak kullanılır.

```
<bean id="customerRepositoryFactory"
        class="com.kurumsaljava.spring.CustomerRepositoryFactory" />

<bean id="customerRepository"
                factory-bean="customerRepositoryFactory"
    factory-method="getNewInstance">
</bean>
```

Bir önceki kod bloğunda görüldüğü gibi customerRepository Spring nesnesi tanımlamasında class yerine factory-bean ve factory-method element özellikleri kullanılmıştır. Böyle bir konfigürasyonda Spring factory-bean ile tanımlanan fabrika sınıfının factory-method ile tanımlanan metodu ile bir customerRepository nesnesi oluşturur. Kullanılan fabrika sınıfı CustomerRePositoryFactory aşağıda yer almaktadır. Burada nesneyi oluşturma sorumluluğu tamamen factory-bean ile tanımlanan sınıfa aittir.

```
package com.kurumsaljava.spring;

public class CustomerRepositoryFactory {

    public CustomerRepository getNewInstance() {
        return new CustomerRepositoryImpl();
    }
}
```

FactoryBean Interface Sınıfı

Spring'in nesne oluşturma ve enjekte etme metotları sadece fabrika metodu ve fabrika sınıfları ile sınırlı değildir. Spring'in bir parçası olan FactoryBean interface sınıfı kullanılarak da nesne oluşturma ve enjekte etme işlemleri yapılabilir.

```
Kod 2.10.1 CustomerRepositoryFactoryBean

package com.kurumsaljava.spring;
import org.springframework.beans.factory.FactoryBean;
public class CustomerRepositoryFactoryBean implements
        FactoryBean<CustomerRepository> {

    public CustomerRepository getNewInstance() {
        return new CustomerRepositoryImpl();
    }

    @Override
    public CustomerRepository getObject() throws Exception {
        return new CustomerRepositoryImpl();
    }

    @Override
    public Class<?> getObjectType() {
        return CustomerRepository.class;
    }

    @Override
    public boolean isSingleton() {
        return true;
    }
}
```

FactoryBean interface sınıfında implemente edilmesi gereken üç metot bulunmaktadır. getObject() metodu tanımlanan sınıftan bir nesne oluşturur. getObjectType() metodu hangi sınıfın nesneyi oluşturmak için kullanılacağını tayin eder. isSingleton() metodu true değerini geri verirse, Spring tarafından tekil bir nesne oluşturulur. Bu değer false ise, Spring her istekte yeni bir nesne oluşturur.

RentalServiceImpl sınıfından bir nesneye CustomerRepositoryImpl sınıfından bir nesneyi enjekte etmek için CustomerRepositoryFactoryBean (kod 2.10.1) isimli, FactoryBean interface sınıfını implemente eden bir fabrika sınıfı oluşturmuş olduk. Bu fabrika sınıfını aşağıdaki şekilde Spring XML dosyasında bir customerRepository nesnesi oluşturmak ve rentalService nesnesine enjekte etmek için kullanabiliriz.

```
<bean id="rentalService"
        class="com.kurumsaljava.spring.RentalServiceImpl">
    <property name="customerRepository"
                    ref="customerRepository" />
    <property name="rentalRepository"
                    ref="rentalRepository" />
</bean>

<bean id="customerRepository"
    class="com.kurumsaljava.spring.CustomerRepositoryFactoryBean" />
```

Bu örnekte görüldüğü gibi customerRepository isimli Spring nesnesi tanımlamasında class olarak FactoryBean interface sınıfını implemente eden CustomerRepositoryFactoryBean sınıfını kullandık. Böyle bir konfigürasyonda Spring CustomerRepositoryFactoryBean sınıfının getObject() metodunu kullanarak yeni bir CustomerRepositoryImpl nesnesi oluşturacaktır. Eğer Spring isSingleton() metodundan true değerini geri alırsa, bu durumda sadece bir defaya mahsus getObject() metodunu kullanarak bir nesne oluşturur. Oluşturduğu nesneyi hafızada tutarak, tekil bir nesne olmasını sağlar ve bu nesneyi her defasında kullanıma sunar. isSingletion() metodunun false değerini geri vermesi durumunda Spring her defasında getObject() metodu üzerinden yeni bir nesne oluşturur.

@Component anotasyonu kullanılmadığı taktirde enjekte edilmek istenen nesnelerin XML konfigürasyon dosyasında Spring nesnesi olarak tanımlanması zorunludur. Karmaşık yapıdaki nesnelerin XML konfigürasyon dosyasında

Spring nesnesi olarak tanımlanmaları zor olabilir. Bu gibi durumlarda FactoryBean interface sınıfını implemente eden bir fabrika sınıfı oluşturulabilir. Karmaşık nesne oluşturma işlemi bu şekilde getObject() metoduna taşınarak bir metot bünyesinde gerçekleştirilmiş olur. Bu işlemin bir Java metodunda yapılması, bu kod biriminin tekrar kullanılabilirlik oranını ve şansını artırır.FactoryBean interface sınıfını implemente eden sınıflar Spring tarafından nesne üretici fabrika sınıfları olarak otomatik olarak keşfedilir ve kullanılır.

CustomerRepositoryFactoryBean tarafından oluşturulan nesneye

```
context.getBean("customerRepository");
```

şeklinde ulaşabiliriz. CustomerRepositoryFactoryBean sınıfından olan, yani fabrika nesnesine erişmek için & işaretinin şu şekilde kullanılması gerekmektedir.

```
context.getBean("&customerRepository");
```

Bu bize doğrudan fabrika nesnesini verir.

Sirküler Bağımlılıklar

Aşağıda yer alan örnekte service1 ve service2 arasında sirküler (cyclic) bağımlılık bulunmaktadır. server1 nesnesini oluşturmak için service2, service2 nesnesini oluşturmak için service1 nesnesine ihtiyaç duyulmaktadır.

```
<bean id="server1"
        class="com.kurumsaljava.spring.service.MyService">
    <constructor-arg name="service1" ref="server2"/>
</bean>

<bean id="server2"
        class="com.kurumsaljava.spring.service.MyService">
    <constructor-arg name="service1" ref="server1"/>
</bean>
```

Spring böyle bir sirküler bağımlılık keşfettiğinde BeanCurrentlyInCreationException hatasını fırlatır. Bu sorunu aşmanın bir yöntemi bağımlılıkları set() metotları aracılığı ile enjekte etmektir. Genel olarak sirküler bağımlılıklar bakımı zor olduğundan, bu yapıların çözülmelerinde fayda

vardır.

Bağımlılıkların Enjekte Edilmesi Yönteminin Avantajları

Spring çatısının sunduğu bağımlılıkların enjekte edilmesi yönetiminin yazılımcı ve uygulama için sağladığı avantajlar şunlardır:

- İnterface sınıflarının kullanımını teşvik eder. Bu yazılımcının uygulamayı bağımlılıkların tersine çevrilmesi (DIP - Dependency Inversion Principle) prensine uygun geliştirmesini sağlar.
- Uygulamanın hazır, konfigüre edilmiş nesneleri kullanmasını sağlar. Bağımlılıkların yazılımcı tarafından programlanmasını engeller.
- Kodu basitleştirir ve bakımını ve geliştirilmesini kolaylaştırır.
- Uygulamanın test edilmesini kolaylaştırır. Bağımlılıkları oluşturan nesneler yerine test amaçlı sahte nesne implementasyonları (stub ya da mock object) kullanımını mümkün kılar.
- Nesnelerin yaşam döngüsünü merkezi bir yerden kontrol etmeyi sağlar.

Bağımlılıkları Enjekte Ederken Hangi Yöntem Kullanılmalı?

Spring çatısında bağımlılıkları konstrüktör ya da set() metotları aracılığı ile enjekte edebileceğimizi gördük. Hangi yöntemi kullanmalıyız sorusu aklınıza gelmiş olabilir. Bu soruya açıklık getirmek isterim.

Her iki yöntemi de birlikte kullanmak mümkündür. Konstrüktör parametreleri zaman içinde artabileceği için konstrüktör aracılığı ile yapılan enjeksiyon karmaşık hale gelebilir. Bu yüzden benim tavsiyem set() metotlarını kullanarak bu işlemi gerçekleştirmektir. @Required anotasyonu kullanılarak, set() metotları aracılığı ile bağımlılıkların enjekte edilmesi zorunlu hale getirilebilir. Bunun yanı sıra altsınıflar otomatik olarak public ve protected olan set() metotları miras alırlar ve altsınıflara herhangi yeni bir metot tanımlama zorunluluğu olmadan nesneler enjekte edilebilir.

Yazılımcının kontrolü dışındaki sınıfların set() metotları olmayabilir. Bu durumda geriye sadece konstrüktör aracılığı ile enjeksiyon seçeneği kalmaktadır.

Bunun yanı sıra Konstrüktör bazlı enjeksiyonda bir nesne için tüm bağımlılıkları nesne oluşturma aşamasında doğrudan enjekte edildiğinden, kullanıcıya verilen nesne tam teşekküllü yapıda olacaktır. Final olan değişkenlere konstrüktör aracılığı ile enjeksiyon yapılabildiği için bu tür oluşturulan nesneler değiştirilemez (immutable) yapıdadır. Bunun gerekli olduğu durumlarda seçim konstrüktör bazlı enjeksiyon olmalıdır.

Spring'in Motoru Nasıl Çalıştırılır?

Spring'in iç organları sahip olduğu sınıflar ise, nefes alıp, vermesini sağlayan XML konfigürasyon dosyasıdır. Böyle bir XML dosyası Spring uygulamasının nasıl yapılandırılması gerektiğini ihtiva eder. Kullandığımız Araç kiralama servisi örneğinde applicationContext.xml isminde böyle bir Spring XML konfigürasyon dosyası oluşturmuştuk. Bu dosyanın kullanılışını ve Spring uygulamasının nasıl ayağa kaldırıldığını kod 2.7 de incelemiştik.

Bir Spring uygulaması değişik ortamlarda çalışır hale getirilebilir. Bunlar:

- JUnit testleri
- Web uygulamaları
- Kurumsal Java uygulamalar (EJB)
- Tek başına (standalone) çalışan uygulamalar

Spring'in motorunun çalıştırılabilmesi için XML konfigürasyon dosyasının uygulama tarafından yüklenebilmesi gerekir. XML konfigürasyon dosyası değişik lokasyonlardan yüklenebilir. Bunlar:

- classpath
- sistemin herhangi bir dizini
- ortama ilişkin bir dizin (environment relavite path, örneğin web uygulamasının bir dizini)

Motorun çalışmasıyla birlikte Spring hafızada Application Context (resim 2.5) ismini taşıyan bir sunucu (container) oluşturur. Bu sunucuyu XML konfigürasyon dosyasının hafızadaki hali olarak düşünebiliriz. Singleton (tekil) olarak tanımlanmış tüm nesneler Spring tarafından oluşturularak sunucu içine konuşlandırılır. Singleton olmayan nesneler örneğin getBean() metodu aracılığı ile talep edildiklerinde, Spring tarafından gerekli bağımlılıklar enjekte edilerek

kullanıcıya verilir.

XML Konfigürasyon Dosyasının Yüklenmesi

Bir Spring uygulamasının çalışır hale gelebilmesi için Spring nesnelerinin tanımlandığı XML dosyasının bulunması ve yüklenmesi gerekir demiştik. Kod 2.7 de kullandığımız Main sınıfı ClassPathXmlApplicationContext aracılığı ile classpath için bulunan applicationContext.xml dosyasını bularak, yüklemiş ve bir Application Context oluşturmuştu. XML konfigürasyonun classpath içinde bulunmadığı durumlarda FileSystemXmlApplicationContext kullanılabilir.

Kod 2.11 - FileSystemXmlApplicationContext Kullanımı

```
package com.kurumsaljava.spring;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.
                        FileSystemXmlApplicationContext;

public class Main {

    public static void main(String[] args) throws Exception {
        ApplicationContext ctx =
            new FileSystemXmlApplicationContext(
                "C:/resources/applicationContext.xml");
        RentalService rentalService =
                    (RentalService) ctx.getBean("rentalService");
        Rental rental =
                rentalService.rentACar("Özcan Acar",
                                new Car("ford", "fiesta"),
                getRentalBegin(), getRentalEnd());
        System.out.println("Rental status: " + rental.isRented());
        System.out.println((CustomerRepository)
                    ctx.getBean("customerRepository"));

    }

    private static Date getRentalEnd()
                        throws ParseException {
        return new SimpleDateFormat("dd/MM/yy").
```

```
                                    parse("29/12/2013");
    }

    private static Date getRentalBegin()
                        throws ParseException {
        return new SimpleDateFormat("dd/MM/yy").
                                    parse("22/12/2013");
    }

}
```

Kod 2.11 de bulunan Main sınıfı FileSystemXmlApplicationContext sınıfını kullanarak C:\resource\applicationContext.xml lokasyonunda bulunan applicationContext.xml dosyasını yüklemektedir.

Web uygulamalarında XML konfigürasyon dosyalarını yüklemek için XmlWebApplicationContext sınıfı kullanılabilir. Bu sınıfın nasıl kullanıldığını ilerleyen bölümlerde birlikte inceleyeceğiz.

classpath içinde bulunan birden fazla konfigürasyon dosyasını yüklemek için joker işareti olarak bilinen kullanılabilir.

```
ApplicationContext ctx =
        new ClassPathXmlApplicationContext("conf/*-config.xml");
```

Yukarıda yer alan tanımlama ile conf dizininde bulunan ve -config terimini ihtiva eden tüm konfigürasyon dosyaları Application Context'i oluşturmak için yüklenir.

Konfigürasyon dosyalarının hangi kaynaklardan yüklendiğini ifade etmek için classpath, file ve http gibi kısaltmalar kullanılabilir. Örneğin biri classpath içinde bulunan ve bir diğeri c:\resource dizinindeki iki konfigürasyon dosyası aşağıda yer alan kod satırında olduğu gibi yüklenebilir.

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
            "classpath:applicationContext.xml",
        "file:C:/resources/test-datasource.xml");
```

ClassPathXmlApplicationContext classpath içinde yer alan konfigürasyon dosyalarını yüklemek için kullanılırken, file: kısaltması herhangi bir dizinde yer alan konfigürasyon dosyasını adreslemewk için kullanılabilmektedir.

Eğer kullanmak istediğimiz konfigürasyon dosyası classpath ya da sistemdeki bir dizin içinde değilse, http: kısaltması kullanılarak bir konfigürasyon dosyasını bir uygulama sunucusundan edinebiliriz.

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
            "http://ip:port/applicationContext.xml");
```

Çoklu XML Konfigürasyonu

Birden fazla XML dosyası kullanılarak bir Application Context oluşturulabilir. Bu şekilde Spring nesnelerini mantıksal guruplara ayırmak mümkündür. Çoğu zaman bu mekanizma uygulamanın servis katmanında kullanılan Spring nesneleri ile altyapı katmanında kullanılan Spring nesnelerini birbirlerinden bağımsız olarak, ayrı XML dosyalarında tanımlamak için kullanılır. Her ortama göre kullanılan altyapı komponentleri değişik olacağından, böyle bir ayrıştırma servis katmanı konfigürasyonunun değişik altyapı konfigürasyonları ile kombine edilerek kullanılmasını mümkün kılar.

Araç kiralama servisi uygulaması için böyle bir ayrıma gitseydik, konfigürasyonumuz nasıl olurdu? Bu sorunun cevabı bir sonraki kod bloklarında yer almaktadır.

```
Kod 2.12 - applicationContext.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"
    xsi:schemaLocation="http://www.springframework.org/schema/
                        beans
    http://www.springframework.org/schema/beans/
                        spring-beans-3.0.xsd">

    <bean id="rentalService"
            class="com.kurumsaljava.spring.RentalServiceImpl"
        scope="prototype">
        <property name="customerRepository"
                        ref="customerRepository" />
        <property name="rentalRepository"
                        ref="rentalRepository" />
    </bean>
```

```
    <bean id="rentalRepository"
            class="com.kurumsaljava.spring.
                        RentalRepositoryImpl">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <bean id="customerRepository"
            class="com.kurumsaljava.spring.
                        CustomerRepositoryImpl">
        <property name="dataSource"
                    ref="dataSource" />
    </bean>

</beans>
```

Kod 2.13 - test-datasource.xml

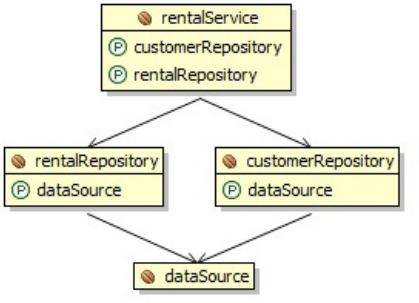
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/
                       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

    <bean id="dataSource"
        class="com.kurumsaljava.spring.DummyDataSourceImpl"/>
</beans>
```

Kod 2.14 - Main

```
public static void main(String[] args) throws Exception {
    ApplicationContext ctx =
        new ClassPathXmlApplicationContext("applicationContext.xml",
            "test-datasource.xml");
    RentalService rentalService =
            (RentalService) ctx.getBean("rentalService");
    Rental rental =
            rentalService.rentACar("Özcan Acar",
                new Car("ford", "fiesta"),
            getRentalBegin(), getRentalEnd());
    System.out.println("Rental status: " + rental.isRented());
    System.out.println((CustomerRepository)
                    ctx.getBean("customerRepository"));
}
```

Servis katmanında yer alan rentalService, customerRepository ve rentalRepository sınıfları için applicationContext.xml isminde bir konfigürasyon dosyası (kod 2.12) oluşturduk. Şimdiye kadar kullandığımız RentalRepositoryImpl ve CustomerRepositoryImpl sınıfları herhangi bir veri tabanı işlemi yapma özelliğini sahip değiller. Bu sınıfların veri tabanı işlemlerini gerçekleştirebilmeleri için DataSource ismini taşıyan bir interface sınıf tanımlıyoruz. Kod 2.12 de yer alan Spring nesne tanımlamaları ile rentalRepository ve customerRepository nesnelerine datasource ismini taşıyan bir nesne enjekte edilmektedir. Nesneler arasındaki oluşturmak istediğimiz bağlantı resim 2.8 de yer almaktadır.



Resim 2.8

Uygulamanın geliştirilmesi için kullanılan veri tabanı sistemi ile, uygulama sunucuları üzerinde çalışırken kullanılan veri tabanı sistemi değişik türde olabilir. Bu tür değişiklikleri uygulamadan saklamak amacıyla DataSource tarzı interface sınıflar kullanılır. Bu interface sınıfın değişik ortamlara göre implemente edilmesi gerekmektedir. Örneğin oluşturduğumuz birim testlerini koşturmak için DummyDataSourceImpl isminde gerçek bir veri tabanı sistemini kullanmayan bir implementasyon oluşturabiliriz. Bu implementasyon birim testlerini koşturmak için gerekli verileri veri tabanından edinmişcesine birim testine verebilir. Uygulamanın gerçek sunucularda bir Oracle veri tabanı sistemi ile çalışmasını sağlamak amacıyla OracleDataSourceImpl implementasyonu oluşturulabilir.

Eğer dataSource tanımlamasını applicationContext.xml dosyasında yapmış olsaydık, uygulamanın çalıştığı ortama göre kullanılan DataSource implementasyonunu bu dosya üzerinde değiştirmek zorunda kalırdık. Bunu önlemek amacıyla kod 2.13 de yer aldığı gibi test-datasource.xml ismini taşıyan

yeni bir XML konfigürasyon dosyası oluşturuyoruz. applicationContext.xml ve test-datasource.xml dosyaları kod 2.14 de yer aldığı gibi ClassPathXmlApplicationContext aracılığı ile kombine edilebilmektedir. Bu uygulamayı geliştirmek için kullanabileceğimiz bir Application Context oluşturur. Uygulamayı müşterimiz için çalışır hale getirmek istediğimizde datasource.xml isminde, OracleDataSourceImpl sınıfını kullanan yeni bir konfigürasyon dosyası oluşturabilir ve ClassPathXmlApplicationContext üzerinden applicationContext.xml dosyasını bu konfigürasyon dosyası ile kombine ederek, uygulamayı bir Oracle veri tabanını kullanacak şekilde konfigüre edebiliriz.

Çoklu XML konfigürasyon imkanı uygulamayı tanımlanmış parçalara bölerek, bu parçaları değişik ortamlarda kombine etmemizi ve ortama uygun bir Spring uygulaması oluşturmamızı mümkün kılmaktadır. Uygulamayı oluşturan bu konfigürasyon dosyaları aynı kod birimlerinde olduğu gibi tekrar kullanımı teşvik etmekte ve uygulamanın modüler bir yapıda olmasını sağlamaktadır.