Hacettepe Üniversitesi

Mühendislik Fakültesi

Bilgisayar Mühendisliği Bölümü

ARAŞTIRMA ÖDEVİ
HIBERNATE IOC

Ali Gökalp Peker
20221829

İçerik

1.IoC(Inversion of Control)

1.1. IoC nedir ?

Değişen yazılım geliştirme koşulları altında daha büyük ve geniş çaplı projelerin daha hızlı , daha etkili ve daha kolay geliştirilebilmesi için bir çok tasarım örüntüsü geliştirilmiştir.

Bir çok farklı geliştirme ortamlarından oluşan projeleri birbirine entegre etmek gelişen yazılım sektöründe ele alınması gereken problemlerden birisi olmuştur.Günümüzde projelerin büyük çapta olması ve birçok çalışan tarafından geliştirilmesi beraberinde birçok problemi getirmiştir.Projelerin ayrı parçalarının ayrı framework'lerde yapılması veya değişik türdeki parçaların her biri için farklı uygulama geliştirme politikaları kullanılması , projelerin tek bir yapı halinde birleştirilmesini zorlaştırmaktadır.Mesela bir web uygulamasında sayfa tasarımı için ayrı bir ortam kullanılması , iş mantığı için ise ayrı bir ortamın kullanılması bu iki ortamın birbirine entegre olmasını gerektirmektedir.Ayrıca işin içine veritabanı gibi başka etmenlerin de girmesi web uygulamasının tek , bütün bir uygulama olarak ortaya çıkarılmasını zorlaştırmaktadır.

Java tabanlı yazılım geliştirimlerinde ise her zaman farklı projelerdeki bileşenleri birleştirmek için lightweight container'lar kullanılmıştır.Bu container'ların birden çok nesneyi bağlayabilmesi için kullanılan genel tasarım örüntüsü IoC'dir.

IoC kısaca , uygulamalar arasında kurulan iş akışının kurulmasına dayanır.Uygulamanın değişik mimariden gelen birçok parçasının , bu parçaları kendisi etkilenmeden kolaylıkla kurulabilmesini sağlar.Bunu sağlamanın birçok yolu bulunmaktadır.Fakat asıl amac iletişimin veya birleştirme işleminin parçaları etkilemeden yapılabilmesini sağlamaktır.Bu yüzden bu yollar belirtilen kriter göz önünde bulundurularak tespit edilmiştir.

IoC ilk başta , inversion of control yani kontrolün çevrilmesi adıyla ortaya çıkmış olsa da , Martin Fowler bir eğitimci bu terimin anlamını irdelemiştir."kontrolü hangi açıdan çeviriyorlar?" diye bir soru yönelten Fowler daha sonra terimin Dependency Injection olarak değiştirilmesini önermiştir.Tasarım örüntüsünü daha iyi açıklayan bu terim daha sonra birçok çevre tarafından kabul edilmiştir.

IoC şu an birçok framework'de kullanılan ve frameworklerde yeni yeni vurgulanan fakat aslında karakteristik olarak birçok framework'ün içinde çoğu kez kullanılmış olan etkili bir tekniktir.

1.2.IoC'yi derinlemesine inceleme

Bir Örnek

IoC'yi daha derinlemesine incelemek için bu konuda bir örnekten bahsedeceğiz.Bu örnekte bir yönetmen tarafından çekilen filmleri listeleyen bir bileşen yazacağız.Bu bileşen tek bir fonksiyon içeriyor.

```
class MovieLister...

        public Movie[ ] moviesDirectedBy(String arg) {
                List allMovies = finder.findAll();
                for (Iterator it = allMovies.iterator(); it.hasNext();) {
                        Movie movie = (Movie) it.next();
                        if (!movie.getDirector().equals(arg)) it.remove();
                }
                return (Movie[]) allMovies.toArray(new Movie[allMovies.size()]);
        }
```

Bu sınıf basitce şu işlemleri yapmakta ; tüm filmleri al ve içinden bizi ilgilendiren filmleri seç , daha sonra bu filmleri geri döndür.

Bu kod ile anlatılmak istenen nokta , finder sınıfıyla list sınıfının nasıl bağlandığı.Acaba kullandığımız moviesDirectedBy methodunu filmlerin saklanış biçiminden nasıl bağımsızlaştırabilirim.Böylece metodum sadece findAll metoduna başvurur ve sonucunu alır. moviesDirectedBy metodunu findAll metodu hakkında başka hiçbirşey ilgilendirmez.findAll metodu ise sadece nasıl yanıt vereceğini bilir.Bunu basit bir arayüz ile sağlayabiliriz.

```
public interface MovieFinder {
        List findAll();
}
```

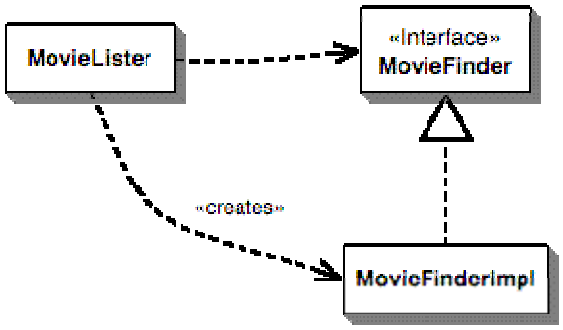
Şu an tüm problem halledilmiş durumda.Fakat aynı zamanda bu arayüzü kullanan somut bir sınıfı başlangıçta atamam gerekmekte.Bu sayede bizim için gerekli arayüze sahip sınıfı atanacak ve bize sadece arama yapma kalacak.

```
class MovieLister...
        private MovieFinder finder;
        public MovieLister() {
        finder = new ColonDelimitedMovieFinder("movies.txt");
        }
```

Burada kullanılan *ColonDelimitedMovieFinder* sınıfı gerekli arayıcı sınıfı bize üretip döndürecektir (Factory örüntüsü).

Şu haliyle bu uygulama gayet iyi bir durumda ve genişletilebilir yapıda.Yazan kişinin kullanımı için mükemmel.Fakat eğer bu sistem başka bir program içinde kullanılacaksa….Peki o zaman ne olur?Film depolama için kullanılan dosya biçimi aynı olduğu zaman da problem yok , çünkü sistem hali hazırda dosya biçimi için de gayet düzgün çalışıyor.Fakat ya dosya biçimi değişecekse?Mesela dosya XML biçiminde saklanacak veya dosya ile hiç uğraşılmayacak filmler veritabanında saklanacak.Bu durumda başka bir sınıf daha yazmam gerekecek.

Tanımladığım arayüz sayesinde *moviesDirectedBy* metodunun değişmesine gerek kalmıyor.Fakat yeni bir *MovieFinder* gerçekleştirimine ihtiyacım var:



Yukarıdaki şekil bu durumda ortaya çıkan bağımlılıkları gösteriyor.Bu durumda **MovieLister** sınıfı diğer iki sınıfa da bağlı.Bu durumda **MovieLister** hem bir arayüze hem de bu arayüzün gerçekleştirimine bağlı.Bu durumda acaba **MovieLister** sınıfının bağımlılığını nasıl sadece arayüze düşürebiliriz?

Bu durumu sağlamak için uygulamanın gerçekleştirimi derleme zamanı sırasında değil de daha sonra eklenebilmesini sağlamalıyız.Fakat bunun için programın gerçekleştirim ayırımını ihmal eden bir yapısı olması gerekmektedir.Yani program arayüze uyan herhangi bir gerçekleştirimi derleme zamanı dışında da kullanabilmelidir.

Ayrıca gerçek hayat uygulamalarında , bunun gibi yüzlerce hatta binlerce sınıf kullanılmaktadır.Acaba bu kadar sınıf birbiriyle düzgün bir biçimde nasıl bir araya gelecekler?Nasıl birleştirilip bir uygulama haline dönüşecekler?İşte , birçok framework bu sorunları IoC örüntüsü çerçevesinde hazırlanmış container'lar ile yanıtlamaktadır.

IoC(Inversion of Control) veya Dependecy Injection

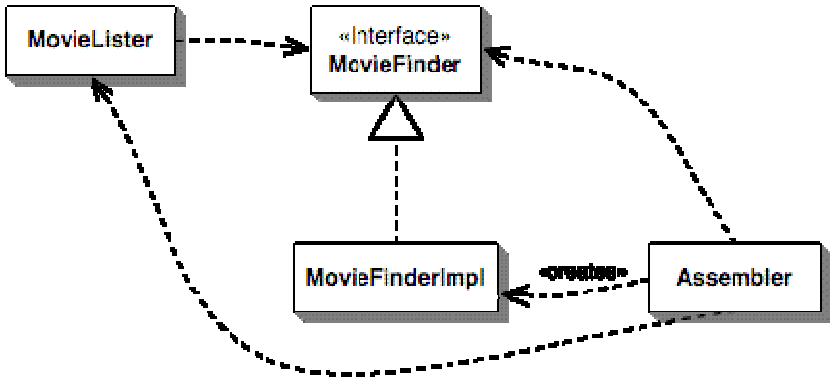
Her ne kadar şu an birçok framework IoC gerçekleştirimi yaptıklarını özellikle vurgulasalar da bu gerçekleştirim her framework'ün ortak özelliğidir.

Acaba IoC tanımı yeterince anlamlı mı.Bu konuya bir de şu açıdan bakalım,eskiden kullanıcı arayüzleri yapılırken , arayüzler program içinde dahil edilirdi.Yani gerekli arayüz için metodlar veya sınıflar yazılırdı.Şimdi ise UI framework'leri ile sadece belirli event'lara göre ve EventHandler'larla bu iş halledilebilmektedir.Uygulama ana döngüyü ise verilerin değerlendirilmesi için çalıştırmaktadır.Bu durumda UI kontrolü uygulamadan UI framework'e geçmiştir.

Fakat yukarıdaki örnekte bahsettiğimiz örnekte , IoC sadece gerçekleştirimin nasıl bulunduğuna dairdir.Örnekte listeleyen sınıf arama yapan sınıfın gerçekleştiriminin direk kullanmak yerinde ortamda bulunan veya verilen gerçekleştirimi kullanmıştır.Bu yaklaşım duruma yönelik gerekli bağlımlılıkların sisteme bir nevi enjekte edilmesini içermektedir.Bu yüzden yöntemin daha çok Dependency Injection (Bağımlılıkların Enjeksiyonu) diye adlandırılması daha mantıklıdır.

Dependency Injection Yöntemleri

Dependency Injection'ın ana mantığı, ayrı bir çevirici sınıfın bulunmasıdır.Bu sınıf gerekli arayüz için gerekli gerçekleştirimi yerleştirir.Sonuç olarak şu şekilde bir bağımlılık ortaya çıkar :



Dependecy Injection'ı sağlamanın 3 yolu vardır :

- Setter Injection
- Constructor Injection
- Interface Injection

Bu yöntemler IoC için ise söyle geçmektedir :

- Type 1 IoC (interface injection)
- Type 2 IoC (setter injection)
- Type 3 IoC (constructor injection).

Setter Injection ya da Type 2 IoC,yaralanılan hizmet ya da bileşenlerin herbiri için bir setter metodun kullanıldığı ve daha sonra gerçekleştirimler için gerekli

konfigürasyon dosyalarının ayarlandığı bir yöntemdir.Konfigürasyon dosyası olarak web tabanlı uygulamalarda sıkça kullanılan XML biçimi kullanılabilir.Bu yöntem Spring Framework ve Pico Container tarafından kullanılan bir IoC yöntemidir.

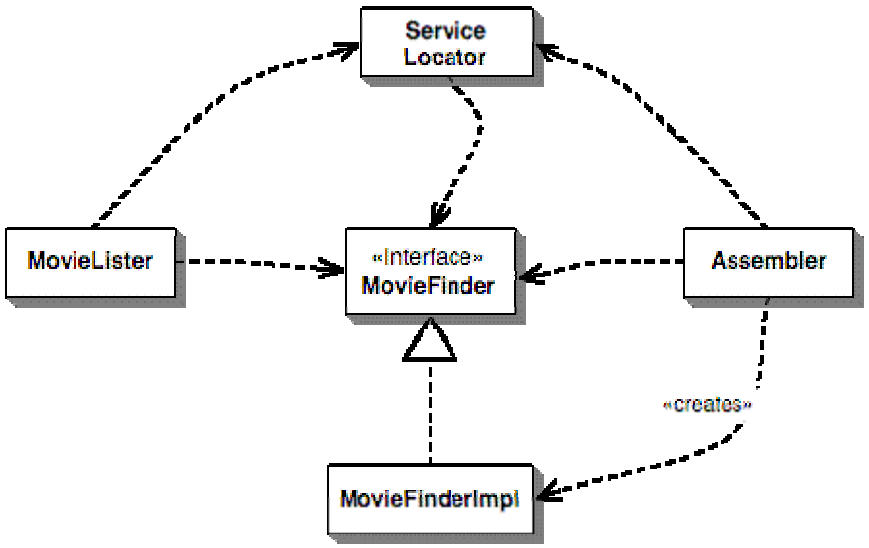
Constructor Injection ya da Type 3 IoC,bir arayüz için gerekli gerçekleştirimin sınıfın constructor'u içinde belirtilmesi yöntemine dayanır. Bu yüzden sınıf constructor'ları gerekli tüm bağımlılıkları içerir.Bahsettiğimiz örnekte ki sonuçları listeleyen sınıf , arayıcı sınıfın gerçekleştirimini ayarlamak için constuctor'u kullanır.

Bu IoC yöntemi PicoContainer ve ayrıca Spring Framework tarafından kullanılan bir yöntemdir. PicoContainer de sınıflar constructor'ların gerekli arayüz gerçekleştirimlerin ayarlanması için kodlar bulundurdukları gibi ayrıca sistem sınıflar arasında ki bağlantıların ayarlanması için konfigürasyon containerları içerirler.

Interface Injection ya da Type 1 IoC , bağımlılıkların enjeksiyonu için arayüzler tanımlayan ve bunları kullanan bir yöntemdir.Bu yöntemde bağımlılıkları tespit etmek için gerekli arayüzleri içeren container'lar kullanılır.Bu yöntemi kullanan frameworklere örnek olarak Avalon verilebilir.

Bu yöntemde , gerekli hizmetler ve bileşenler için enjeksiyon arayüzleri tanımlanır.Daha sonra bu arayüzlerin gerçekleştirimi , kendilerine bağımlı olan sınıflarca sağlanır.Ve daha sonra gerekli konfigurasyon için bir sınıf yazılır.Bu ayarlamalar dosya olarak da ayarlanabilir.

Service Locator(Hizmet Bulucu) , ise bir uygulamanın ihtiyaç duyduğu hizmetleri tespit eden nesnelerin bulunduğu bir yöntemdir.Bu metod IoC'ye alternatif olarak yapılmıştır.Yaptığımız örneği bu bağlamda yeniden ele alırsak , hizmet bulucu uygulamaya gerektiği zaman bir arayıcı gerçekleştirimi sağlayabilmelidir.Bu bağımlılığı başka bir nesne kaydırıyor.Bu durumu şematik olarak ifade edersek , bağımlılıklar :



Burada Service Locator'ı singleton olarak gerçekleştirirsem, bu durumda listeleyici bu sınıf oluştuğu zaman istediği arayıcı gerçekleştirimini ele alabilir:

```
class MovieLister...
        MovieFinder finder = ServiceLocator.movieFinder();
class ServiceLocator...
        public static MovieFinder movieFinder() {
                return soleInstance.movieFinder;
        }
        private static ServiceLocator soleInstance;
        private MovieFinder movieFinder;
```

IoC gibi bu teknikte de hizmet bulucu'yu ayarlamamız gerekmektedir.Bu ayarlama hem kod hem de konfigürasyon dosyasıyla yapılabilir.

```
class Tester...
        private void configure() {
          ServiceLocator.load(new ServiceLocator(new ColonMovieFinder("movies1.txt")));
        }

class ServiceLocator...
        public static void load(ServiceLocator arg) {
                soleInstance = arg;
        }

        public ServiceLocator(MovieFinder movieFinder) {
                this.movieFinder = movieFinder;
        }
```

Ve test kodu olarak da :

```
class Tester...
        public void testSimple() {
                configure();
                MovieLister lister = new MovieLister();
                Movie[] movies = lister.moviesDirectedBy("Sergio Leone");
                assertEquals("Once Upon a Time in the West", movies[0].getTitle());
        }
```

Bu yöntemin eksiklerinden birisi yazılan Service Locator sınıflarının test edilebilirliğinin az olması.Çünkü test için herhangi bir ikinci Service Locator nesnesi yazılamıyor.Bu yüzden ilk başta bunların çok iyi tasarlanması gerekiyor.

Yukaridaki yöntemin en önemli eksiği , **MovieLister** sınıfı sadece bir hizmeti kullanmış olmasına rağmen , Service Locator'e tamamen bağlı durumda.bu durumda ayrılmış hizmet arayüzleri kullanmak bu sorunu halledebilir.Listeleyici işi yapan sınıf ServiceLocator sınıfını tamamen tanımlamak yerine sadece istediği hizmetler için bir kısmını tanımlayabilir.

Bu durumda listeleyici sınıfı sağlayan aynı zamanda bir arayüz de ServiceLocator için sağlamalıdır.

```
public interface MovieFinderLocator {
        public MovieFinder movieFinder();
```

ServiceLocator, arayıcı nesneyi sağlamak için arayüzü gerçekleştirir :

```
MovieFinderLocator locator = ServiceLocator.locator();
MovieFinder finder = locator.movieFinder();

public static ServiceLocator locator() {
        return soleInstance;
}

public MovieFinder movieFinder() {
        return movieFinder;
}

private static ServiceLocator soleInstance;
private MovieFinder movieFinder;
```

Bu örnekte , static bir örnekti.Yani bağımlılıkları karşılamak için metodlar yazdık.Fakat bunu çalışma zamanında yapabilmeliyiz.Bunun için de dinamik Service Locator yapabiliriz.Bu sayede uygulama çalışma zamanında çalıştıracağı hizmete göre gerçekleştirimleri tespit eder.

1.3. Kullanım alanları ?

IoC birçok framework'ün yapısında bulunan bir örüntüdür.Her framework bu örüntüyü kendine has yapısıyla yorumlamıştır.

Günümüzde birçok proje çok büyük ölçekte yapılmaktadır.Büyük ölçekte yapılması bu projenin içine birden fazla framework'ün , çalışma aracının dahil olmasına neden olur.Doğal olarak mimariler bu tür projelerin ihtiyaçlarını karşılayabilecek şekilde yapılandırılır.Mesela Spring ; Spring famework , persistence framework , web uygulamalar ve JDBC gibi öğeler için soyutlama katmanını kendi bünyesinde bulundurmaktadır.

IoC veya diğer adıyla Dependency Injection da framework'lerin büyük projeleri birleştirmeleri için adeta bir gereklilik haline gelmiştir.Bir çok framework IoC'nin belirli tiplerini kullanmaktadır.Mesela PicoContainer , Type 2 ve Type 3 IoC kullanmaktadır.Spring framework'de PicoConainer gibi Type 2 ve Type 3 IoC kullanmaktadır.Fakat her framework kendi mimarisine bağlı olarak belirli tiplere yoğunlaşmıştır.PicoContainer Type 3 IoC'ye yoğunlaştığı gibi Spring daha çok Type2 IoC'ye yönelmiştir.

Daha önce de birçok kez bahsettiğimiz gibi IoC , aslında framework'ün gerçekten framework olması için gerekli bir etmendir.Bu yüzden IoC her türlü framework de bulunan bir örüntü.Fakat bunun şimdi öne çıkmasının nedeni , belki de projelerin gittikçe büyümesi ve IoC'nin giderek önem kazanmasıdır.

Zaten şu an framework'leri öne çıkartan özelliği bir framework'ün büyük projeleri ne kadar sorunsuz ve etkili ele alabileceği.Bu yüzden artık bazı framework'ler yeterli olmadığı için piyasadan kalkıyor.

Sonuç olarak kullanım alanı olarak IoC çok geniş ve bu alan giderek artmaya devam ediyor.Projeleri büyümesi de bu artışı tetikleyen önemli etmenlerden birisi.

1.4. IoC tiplerinin artıları ve eksileri ?

Her IoC tipinin düşünülüş amacı vardır ve her bir tipin duruma özgü olarak avantajları ve dezavantajları vardır.Bu yüzden oluşturulan IoC tipi projeye uygun olarak seçilirken bu tür kriterler göz önünde bulundurulmalıdır.

(Type 2 IoC) Setter ve (Type 3 IoC)Constructor Injection

Constructor Injection , biraz daha fazla kod gerektiriyor.Bağımlılıklar için fazladan sınıflar yazılması gerekiyor.Az miktarda bağımlılık için bu fazla önemli değil fakat eğer çok fazla varsa bu sıkıntı oluşturuyor.

Şu da bir gerçek ki , Seter Injection ile Constructor Injection arasındaki fark aslında nesne yönelimli bir problem.Bu problem , acaba alanları constructor'la mı dolduracağız yoksa setter alanlarla mı?

Constructor oluşturmanın avantajlarından birisi , parameterleri girilen bir constructor uygulamaya nerede nasıl bir nesnenin oluşacağı hakkında kesin bir bilgi verir.Ayrıca constructor ile nesnelerin oluşturulması değişmez nesnelerin setter metodlara göre daha iyi saklar.Bu da projenin daha sağlam bir tabana oturmasını sağlar.Fakat istisnalar da bulunmaktadır.Uzun parametrelerin kullanılması constructor'ların şişmesine neden olur.Çok fazla parametresi olan bir constructor'a sahip olan bir sınıfın aslında birden fazla sınıfa parçalanması gerekir.

Aynı zamanda bir nesneyi oluşturmanın birden fazla yolu varsa bunu constructor'larla belirtmek zordur.Çünkü aynı sınıf için farklı constructor'lar oluşturmak için en fazla parametre tipi veya sayısı değiştirilebilir.Bu durumda Factory metodları kullanılabilir.Fakat bu durumda da bir yanlışlık var , çünkü Factory metodları static metodlar ve bunların arayüzlerde tanımlanması imkansız.Buna karşılık bir Factory sınıfı oluşturulabilir.Bu da tabii ki başka bir hizmet sınıfına karşılık gelir.

Ayrıca çok kalıtımlı sistemlerde bu sistem çok problem çıkarabilir.Her kalıtım sınıfı ayrı bir constructor yazmak , ayrıca kendi parametreleriyle , çok zor olabilir.

Buradan anlaşıldığı üzere 2 sistemin de kendine has özellikleri var ve proje takımları bunlara göre herhangi birini seçmek zorundalar.Bu yüzden frameworkler , mimarilerinde her 2 tekniğe de yer veriyorlar.

Kod veya Konfigürasyon Dosyaları

Yazılım bileşenlerinin bir API vasıtasıyla kod ile veya konfigürasyon dosyalarıyla bağlanması birbirinden çok farklı fakat karıştırılan bir konudur.Yazılımın birçok yere kurulacağı düşünülürse burada mantıklı seçim konfigürasyon dosyaları kullanmaktır.Bu dosya için genel olarak XML dosyası kullanılır.Fakat bazen kod kullanarak birleştirme işlemini yapmanın tercih edildiği durumlar da olabilmektedir.Mesela çok fazla kuruluma ihtiyacı olmayan küçük bir programın parçalarını birleştirmek için kod kullanmak daha kolaydır.

Ayrıca dikkat edilmesi gereken diğer konu da , kodlamanın nerede karmaşıklaşacağı.Belirli koşullarda çok fazla kodun yapabileceği iş birkaç XML satırıyla halledilebilir.

Konfigürasyon işinin kodla yapılmasının getirdiği bir dezavantaj da konfigürasyon ayarlarının programcı olmayan birisi tarafından yapılamaması.Bu da uygulamanın bakımını zorlaştırmaktadır.Fakat her bileşenin kendine has konfigürasyon dosyasının bulunması da çok fazla konfigürasyon dosyasının oluşmasına ve bu nedenle karışıklık oluşmasına neden olacaktır.

Sonuç olarak bir projede yapılabilecek en iyi hareket , proje konfigürasyonunu kod ile hallederek başlamak fakat konfigürasyon dosyasını opsiyonel olarak da içermektir.

Ayarları , Kullanımdan Ayrı Tutmak

Önemli olan bir nokta var ki bu da servislerin konfigürasyonları servislerin kendilerinden ayrı tutmaktır.Bu aslında gerçekleştirimleri arayüzlerden ayırmaya yarayan en önemli örüntülerden birisidir.Bu sistem nesne yönelimli programlama da kullanılan yöntemlerdendir.

Bileşenlerin ve diğer hizmetlerin kullanıldığı bir durumda böyle ayrımların yapılması proje için hayati önem taşır.Konfigürasyonların veya programda kullanılan parçaların birbirinden tamamen ayrık olması programın kolaylıkla farklı parçalarla veya kurulumlarda çalışabilmelerini sağlar.

Bir uygulama hazırlanırken yapılacak en önemli seçimin hizmetleri , ayarlamalarından bağımsız tutmak olduğu unutulmamalıdır.

2. Spring Framework

Spring Framework J2EE mimarisinde geliştirilmiş birçok framework'den birisi olarak düşünülebilir.Fakat Spring birçok özelliği bakımından diğer frameworklerden ayrılmaktadır.Bu özellikler:

- **Diğer birçok framework'ün değinmediği birçok sorunla ilgili çözüm sunar**.Spring , iş nesnelerinin(Business Objects) yönetimine yoğunlaşıyor.
- **Spring hem çok amaçlı hem de modüler bir frameworktür**.Spring katmanlı bit altyapıya sahiptir.Yani geliştirimi yapan kişi istediği herhangi bir katmanı seçip onu kullanabilir.Ayrıca mimarisi kendi içerisinde tutarlı bir yapıdadır.Mesela Spring , JDBC kullanımını basitleştirmek için kullanılabilir.Spring projelerin içine ,artırımsal olarak , kolaylıkla yerleştirilebilir.
- **Spring baştan aşağıya kolaylıkla test edebilecek kodlar yazabilmesi için geliştirilmiştir.**Spring test amacı güdülen projeler için birebirdir.
- **Spring önemi gittikçe artan bir entegrasyon aracıdır.**Spring'in uygulama geliştirme alanındaki rolü birçok yazılım sağlayıcı tarafından anlaşılmıştır.

Spring birçok uygulamanın altyapı ile ilgili birçok konuya değinen bir framework'tür.Spring kuruluş uygulamaları için light weight çözümler sunmaktadır.Spring MVC bir altyapı sunmaktadır, ayrıca çok iyi yapılanmış kural dışı durum hiyerarşisine sahiptir.

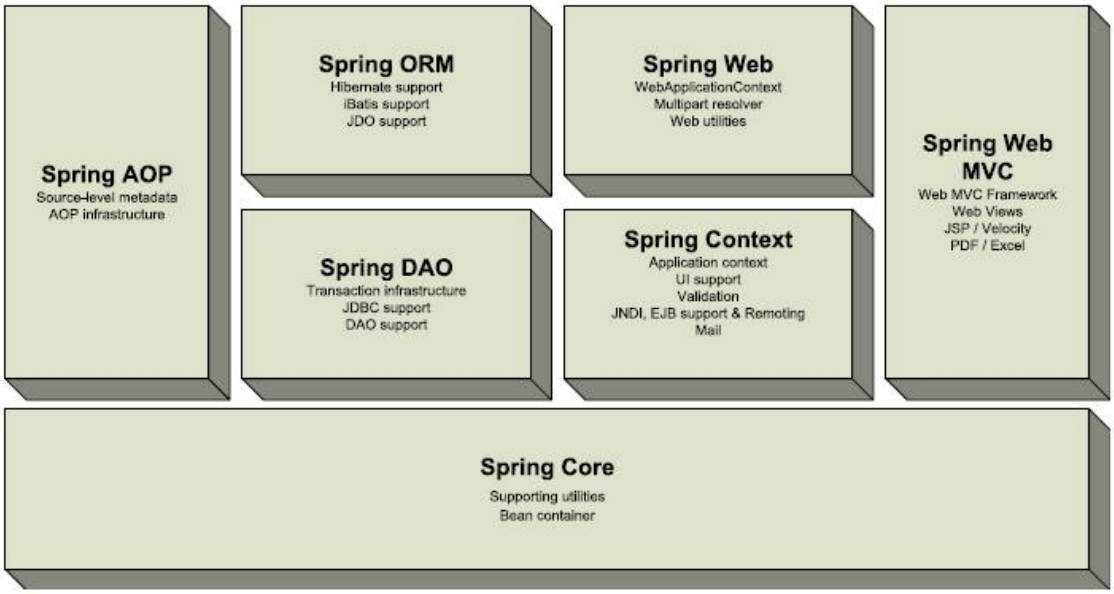
Spring , modüleritesi sayesinde tamamına ihtiyaç olmadan belirli parçalarının kullanılabilmesine izin verebilmektedir.Bu bağlamda bir uygulama geliştirirken Spring , Struts mimarisiyle beraber ya da Hibernate ile entegre , JDBC katmanıyla beraber kullanılabilir.Bu yüzden Spring kullanışsızlığa yer vermeyen,herhangi bir yerde bir parçasının kullanılabileceği , (kullanıldığı yere göre değerlendirirsek) bir framework'tür.

Bu bağlamda Spring şu an geliştirlmiş birçok framework'ün içerdiği özellikleri bünyesinde barındırmakta ayrıca tutarlılık , modularite ve geliştirim açısından bir çok yeni özellik sunmaktadır.Spring mimarisi sayesinde öğrenim konusunda açıklık ve sadelik sunmaktadır.

Spring Şubat 2003 yılında açık kaynaklı bir proje bir proje olarak başlamış ve bu zaman süreci içerisinde çok ilerlemiştir.Mimari olarak Spring 2000 yılı ve öncesinde tanımlanan tasarım konseptlerine dayanmaktadır.Şu an 20 adet geliştirici ile Spring framework'ün ilerleme süreci devam etmektedir.

2.1. Framework'e genel bakış

Spring birçok işlevsellik ve özelliğe sahiptir.Bunlar çok iyi tasarlanmış ve organize edilmiş yedi modülden oluşmaktadır.



Spring Core

Bu modül Spring'in en temel parçasıdır ve **Dependecy Injection** özelliği ile bean container'ın işlevselliğinin yönetilmesine izin verir.Burada bu olayı sağlayan temel konsept ise uygulamanın ihtiyacı olan programmatik singleton'u sağlayan ve bağımlılıkların özelliklerini ve ayarlarını program mantığından ayıran **BeanFactory**'dür.

Context Package

Çekirdek(Core) paketinin üstünde yer alan bu modül JNDI'a benzeyen bir şekilde erişim bean'lerine framework usülü erişimini sağlar.Context paketi özelliklerini Beans paketinden kalıtır ve buna metin iletişimi desteği ekler.

DAO Package

DAO paketi gereksiz JDBC kodlamalarını ve Veritabanı üreticilerine özel hata kodlarını kaldıran JDBC soyulama katmanını sunmaktadır.Ayrıca , bu katman JDBC işlemlerini programlama ve tanımlama yöntemleri ile gerçekleştirme olanağı sunmaktadır.

ORM Package

ORM paketi , JDO,Hibernate ve iBatis gibi nesne ilişkisel eşleme API(ORM API)'leri için entegrasyon sağlamaktadır.Paketin kullanımıyla , bu nesne ilişkisel eşleştiricilerden herhangi biri Spring ile beraber entegre olarak kullanılabilir.

AOP Package

Bu paket , cephe yönelimli programlama(aspect oriented programming) yapmaya olanak verir.Bu işlem , işlevsellik olarak ayrılması gereken kod kısımları için kesici metodlar(interceptor methods) yazmaya olanak vererek sağlanır.Kaynak seviyesinde yardımcı veriler(metadata) kullanılarak davranışsal özelliklerin her tipini kodunuzda içerebilirsiniz.

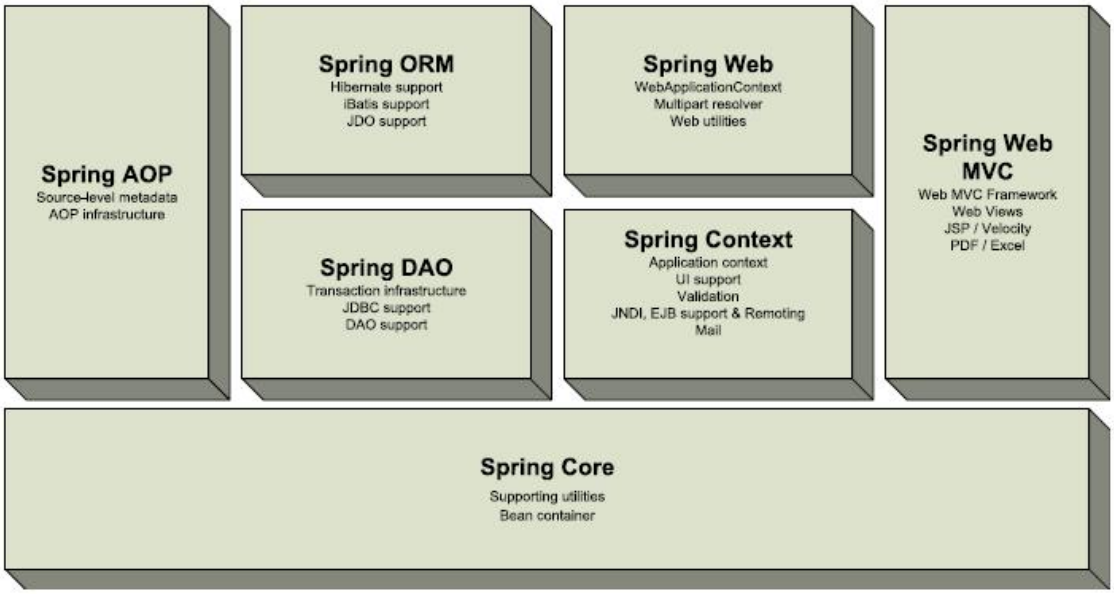
WEB Package

Spring'in Web paketi , Spring'in cok bölümlü işlevsellik , servlet listener kullanma ve web yönelimli uygulama bağlamı gibi temel web entegrasyon özelliklerini içerir.

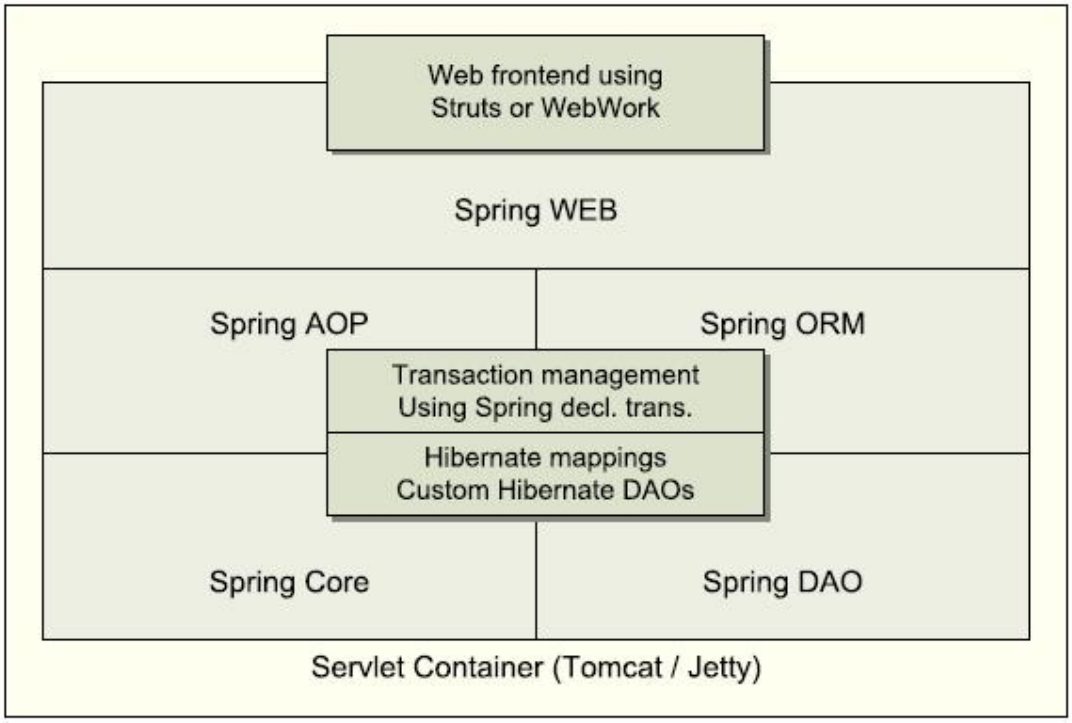
WEB MVC Package

Bu paket web uygulamaları için Model-View-Controller gerçekleştirimini sağlar.Spring'in MVC gerçekleştirimi herhangi bir alelade gerçekleştirim değildir.Bu gerçekleştirim model kodu ile web formlarını birbirinden tamamen ayıran ve Spring diğer özelliklerinin kullanımına izin veren bir gerçekleştirimdir.

Yukarida anlatılan altyapılarla , Spring hemen hemen tüm senaryolarla kullanılabilir, appletlerden tutun da çok karmaşık enterprise uygulamalarına kadar birçok uygulama Spring'in web framework'ü ve işlembilgi yönetimi ile yapılabilir.



Yukarida Spring'in birçok özelliğini kullanan bir web uygulamasının nasıl bir yapıda olacağı görülmektedir. TransactionProxyFactoryBeans kullanımı sayesinde web uygulaması,Enterprise JavaBeans tarafından sağlanan barındırıcı yönetimli bilgiişlem kullandığımız zaman olacağı gibi , tamamen ticarisel(transactional).İş mantığı kısmı Spring'in Dependency Injection barındırıcısının yöneteceği basit POJO'larla kolaylıkla gerçekleştirilebilir.Email yollama ve geçerlilik sınaması gibi web katmanından bağımsız hizmetlerle , geçerlilik sınaması gibi işlemleri nerede yapacağınızı seçebilirsiniz.Spring ORM desteği Hibernate,JDO ve iBatis'i entegredir.HibernateDAOSupport ile , varolan Hibernate eşleştirmelerinizi tekrar kullanabilirsiniz. Form kontrolörler , web katmanıyla alan modelini saydam olarak entegre eder.Bu sayede HTTP parametrelerini aktarmak için ActionForm veya başka sınıflara ihtiyaç kalmaz.



Bazı durumlarda , projeler tamamen farklı bir framework'e geçmeye izi vermez.Spring hiçbir zaman uygulama geliştirenleri , kendinin tümünü birden kullanmaya zorlamaz.Şu an bulunan WebWork , Struts gibi framework'ler Spring ara katmanıyla kolaylıkla proje ile entegre olabilir.Yapılacak tek şey Application Context ile iş mantığını bağlamak ve WebApplicationContext kullanarak Web kullanıcı arayüzünü entegre etmektir.

Spring ayrıca Enterprise Java Beans için soyutlama katmanı desteği sağlamaktadır.Bu şekilde EJB ile yaptığınız çalışmalarınızı , varolan POJO'larınızı Stateless Session Beans ile sararak kolaylıkla tekrar kullanabilirsiniz.

SPRING ÇATISI NASIL ÇALIŞIR

Spring framework'ün modulleri arasında Core kısmı asıl fonksiyonaliteyi taşır. Core modülde IoC uygulanır ve bağımlılık nesneleri kontrol edilir. Bu modülde görev alan en önemli sınıf Bean Factory sınıfıdır.

Bean Factory bütün Java nesnelerini konfigüre eden ve yöneten bir arayüzdür. XMLBeanFactory sınıfı XML dosyalarındaki nesne tanımlarını okuyup, nesneleri yönetirken, ListableBeanFactory sınıfı ise property dosyalarındaki nesneleri okuyup yönetir. Bean Factory oluşturulurken Spring tüm nesnelerin konfigürasyonlarını başlatır. Singleton olan tüm nesneler Bean Factory'in bir nesnesi oluşturulurken beraber oluşturulurlar. Diğer nesneler ise uygulamanın içinde ne zaman kullanılacaklar ise o zaman oluşturulurlar.

"Definition" yani tanımlama durumda nesneler constructor'ları çağrılarak (tüm nesnelerin constructor'ları kod içerisinde oluşturulmalıdır) oluşturulurlar. "Pre-initialized" duruma geçilir ve bu durumda ilk önce bağımlı nesneler birbirine bağlanır, bağımlılıklar kontrol edilir, setter metodları kullanılarak özelliklere değerler atanır, nesnelere bean factory'leri verilir ve son kontrollerle bu durum tamamlanır. Artık uygulamanın kullanımına sunulabilir hale ("Ready" duruma) getirilmiş olur. Uygulama bu nesneleri kullanır ve nesnelerin işi bittiği zaman veya container yok edileceği zaman "destroy()" metodu çağrılarak nesneler yok edilerek nesnelerin hayat çemberi sona erdirilir. Bütün bu işler Spring'in lightweight container'i içerisinde **Inversion of Control** prensibi ile yapılırlar.

Peki nesneler nasıl oluşturuluyor diye düşünürsek, bu sorunun cevabı şu şekilde olur. Yazılan sınıfların bir şekilde Spring'in lightweight container'i ile tanıştırılması gerekmektedir. Bunun için aslında çok farklı yollar bulunmaktadır (XML dosyaları , property dosyaları, LDAP, RDBMS,vb..), Fakat bunlar arasında en çok kullanılanılanı ve en kullanışlı olanı XML dosyaları ile tanımlamadır.

```
<bean id="company" class="ioc.ComputerCompany">

    <property name="engineer">

        <ref bean="engineer"/>

    </property>

</bean>
```

Burada ComputerCompany isimli sınıfımız için bir tanım oluşturuyoruz. İlk başta bu sınıftan oluşacak olan nesne için "company" isimli bir indis (id) veririz, bu indis ile artık bu nesnemize erişiyor olacağız. Bu nesnemizin "engineer" diye bir özelliği olsun ve bu özelliğin dış dünyadaki "engineer" indisli bir nesneye bağımlılığı olsun. İşte böyle bir nesne içi oluşturulacak tanım yukarıda gibi olacaktır. Burada "id" etiketi verilecek olan indisi, "class" etiketi sınıfın ismini, "property" etiketi özelliğin ismini, "ref"ise referanslanacak olan nesnenin indisini gösterir. Bu etiketler ile lightweight Spring container'imizi bir bakıma şekillendirmiş ve konfigüre etmiş olmaktayız.

Bu etiketleri iyi kullanmak Spring uygulamaları için çok önemlidir. Uygulamanın performansını da bu etiketlerle şekillendirebilirsiniz. Bir örnek vermek gerekirse, nesne tanımlarında aksi bildirilmediği takdirde tüm nesneler singleton olarak oluşturulur, eğer bir sınıfın birden fazla nesnesini oluşturacaksınız bu etiketi false yapmanız gerekmektedir. Yine nesneler tanımlanırken aksi söylenmedikçe tüm nesneler bean factory tarafından bean factory ile birlikte yaratılırlar. Eger çok fazla nesneniz var ise bu işlem uzun sürebilir, bu da size performans kaybı olarak geri dönebilir. Bu yüzden kullanıp kullanmayacağınızı bilmediğiniz nesneleri "lazy-init= true" etiketiyle tanımlayıp istendigi zaman yaratabilirsiniz. Spring uygulamalarınızda bu etiketlerin en optimum şekilde kullanılması tavsiye edilir.

2.2. Spring Framework ve IoC container

Spring'in en önemli iki paketi org.springframework.beans ve org.springframework.context paketleridir.Bu paketlerdeki kodlar Spring'in IoC özelliklerini sağlar.**BeanFactory** sınıfı herhangi bir depolama biçimini kullanan herhangi bir bean sınıfını ayarlayabilecek kapasitede gelişmiş bir ayarlama mekanizmasınıdır.**ApplicaionContext** sınıfı ise Beans Factory sınıfına dayanan(alt sınıfı olan) ve Spring AOP ile entegrasyon, mesaj yönetim , olay yayılma(event propagation) gibi başka islevler katan önemli bir sınıftır.Kısacası , BeanFactory ayarlama ortamını ve temel işlevleri sağlarken , ApplicationContext bu sınıfın özelliklerini geliştirir ve J2EE yeteneklerini içine katar.

Spring'in çekirdeği olan org.springframework.beans JavaBeans ile çalışabilecek şekilde tasarlanmıştır.Bu paket kullanıcılar tarafından direk olarak kullanılmaz, daha çok Spring'in işlevselliğini destekler.BeanFactory bu paketin soyutlama katmanıdır.Bu sınıf sınıfların isimleriyle üretilebilmelerini sağlayan ve bu sayede nesneler arasındaki iişkileri yöneten sınıftır.

Bean Factory iki tipte nesneyi destekler :

- **Singleton** : Bu durumda , bir isimdeki nesnenin paylaşılan bir örneği vardır.Bu durum , varsayılan olarak ve belki de en çok kullanılan tipidir.
- **Prototype (non-singleton)** : Bu durumda , her erişim yeni,bağımsız bir nesnenin oluşmasına neden olacaktır.

Spring kapsayıcısı , nesneler arasındaki ilişkileri yönettiği için , POJO'ların saydam olarak ilişkilendirilmesi ve hot-swapping gibi kullanıcılara çalışma zamanı değişikliklerinin yansıtılmadığı hizmetlerde önem kazanır.Dependency Injection'ın sağladığı en büyük güzelliklerden birisi de tüm bu işlerin saydam bir biçimde , hiçbir API işin içine girmeden gerçekleştirilebilmesidir.

org.springframework.beans.factory.BeanFactory , basit bir arayüzdür ve birçok şekilde gerçekleştirilebilir.**BeanDefinitionReader** arayüzü , BeanFactory ile yardımcı bilgileri(metadata) birbirinden ayırır.Böylece Spring tarafıdan sağlanan BeanFactory gerçekleştirimleri her türlü yardımcı veri ile çalışabilir.En çok kullanılan BeanFactory gerçekleştirimleri :

- **XmlBeanFactory** : İsimleri verilen nesnelerin sınıflarını ve özelliklerini tanımlandığı XML yapılarını okuyan ve yorumlayan sınıfdır.
- **DefaultListableBeanFactory** : BeanFactory'nün programmatik olarak oluşturulmasını veya özellik dosyalarından okunarak bean özelliklerinin ayarlanmasını sağlayan sınıftır.

Her tanım bir POJO veya FactoryBean olabilir.FactoryBean arayüzü bir nevi dolaylama sağlar.Genel olarak bu arayüz , AOP veya başka yaklaşımlar kullanarak vekil(Proxy) nesneler oluşturulması için kullanılır.Bu kavramsal olarak EJB interception yöntemine benzer , fakat pratik olarak daha basit ve etkilidir.

BeanFactory kapsayıcısı itibariyle , Spring bir IoC kapsayıcısı olarak nitelendirilir.Fakat Spring kapsayıcısı daha çok Dependency Injection olarak tanımlanmıştır.

IoC , gerçekleşmesi istenen olayların koddan soyutlanarak framework'ün sorumluluğu altına girmesini sağlar. Dependency Injection , kapsayıcı API'ye olan açık bağımlılığı kaldıran bir IoC tipidir. Bir bileşen X sınıfına ihtiyaç duyduğunda , kapsayıcıyı çağırarak x sınıfını talep eder , Dependency Injection ile kapsayıcı bu sınıfın nerede olduğunu tespit eder.Bunu metod imzalarını tespit ederek ve XML ayar dosyalarını kullanarak yapar.

Dependency Injection ın en büyük iki çesidi : Setter Injection ve Constructor Injection'dır.Spring bu iki çeşidi de kullanılabilmesine izin verir.Hatta bu yöntemi beraber kullanabilirsiniz.Dependency Injection birçok yönten avantajlıdır :

- Bileşenler birlikte çalıştıkları diğer bileşenlere çalışma zamanında bakma ihtiyacı duymazlar,böylece yazılmaları ve bakımları çok daha basittir.Spring'de bileşenler bağımlılıklarını JavaBean setter metodları veya constructor'lar vasıtasıyla bildirirler.

- Aynı nedenden ötürü , geliştirilen uygulamaları test etmek çok daha kolaydır.

- İyi bir IoC gerçekleştirimi , türe bağlılığı korur.IoC ile bağımlılıklar kod ile belirtilir ve tür dönüştürümleri için framework sorumludur.Bu uygulama geliştiricinin bu tür yükümlülüklerden kurtulmasını sağlar.

- Bağımlılıklar açıktır.Constructor veya JavaBean özellikleri ile belirtildiği için tüm projeyi incelemeden , ney yapıldığı hakkında bilgi sahibi olabilirsiniz.

Spring BeanFactory sınıfları lightweight nesnelerdir.Appletlerin içine , Swing uygulamalarına başarı ile yerleştirilebilirler.Herhangi bir kurma işlemine ihtiyaç duymaz veya barındırıcıdan dolayı herhangi bir başlatma zamanı gecikmesi olmaz.Bu framework açısından çok önemli bir artıdır.

BeanFactory kavramı Spring'in başından sonuna kadar kullanılmıştır ve Spring'in iç yapı olarak tutarlı olmasının asıl nedenidir.Spring ayrıca IoC'yi tam kapsamlı bir framework olarak baştan aşşagıya kullanmasıyla diğer framework'lerden ayrılır.

ApplicationContext sınıfıda BeanFactory'e şu ek özellikleri sağlayan bir altsınıfıdır :

- Mesaj arama , yerelleştirme ve uluslararasılaştırma
- Olay yönetim mekanizması , uygulama nesnelerinin olay yayınlamasını ve almasını sağlar
- Uygulama özel veya genel bean tanımlarının algılanması
- Taşınabilir dosya ve kaynak erişimi

3.Spring IDE

3.1.Spring IDE nedir ?

Spring IDE , Spring framework projesinin bir alt projesidir ve Bean Factory ayar dosyalarıyla daha kolay çalışılabilmesi için yazılmış bir Eclipse plug-in'dir.Spring IDE bean ayar dosyaları ile daha rahat çalışabilmek için XML editor,geçerlilik denetimcisi gibi birçok araç sunmaktadır.Bu proje ilk olarak SpringUI adıyla çıkmış olup daha sonra Spring IDE olarak ismi değiştirilmiştir.

Şu an en son olarak 1.2.5(11 Ocak 2006 sürümü çıkmış olup,Eclipse 3.1 ile çalışmaktadır.Spring IDE gereksinim olarak ayrıca GEF(Graphical Editing Framework)'e ve WTP 1.0'a ihtiyaç duymaktadır.

3.2.Spring IDE'nin özellikleri ?

Spring IDE aşşağıdaki özellikleri sunmaktadır :

- **Proje doğası(Project Nature) :** Spring bean ayar dosyalarını destekleyen bir yardımcıdır.Herhangi bir proje seçilerek bu projeye kolaylıkla Spring Project Nature eklenebilir.Bu durumda proje için gerekli bean ayar dosyaları ve altyapı ayarlanır.

- **Arttırımsal inşa edici(Incremental Builder) :** Spring projesini artırımsal olarak inşa eden ve bu işlemler esnasında Spring projesinde bulunan tüm bean ayar dosyalarını geçerlilik denetiminden geçiren bir araçtır.

- **Görüntüleyici(View) :** Spring projelerini ve bean ayar dosyalarını agaç biçiminde görüntüleyen bir araçtır.

- **Editor** : Tanımlanmış tüm bean ayar dosyalarını görüntüleyen bir araçtır.Bu gösterim grafiksel olarak yapılır ve bean dosyaları arasındaki bağlantılar da bu grafikte yer alır.Bu görünüm tipleri bir çok kapsamda açılabilir :
  - Görünümde şeçilen bir ayar dosyasında
  - Görünümde seçilen bir ayar grubunda
  - Görünümde seçilen bir bean dosyasıyla

- **Grafiksel Editor** : Spring WebFlow için kullanılan ayar dosyalarını grafiksel olarak düzenlemek için bu araç kullanılır.

- **XML Editor** : Spring Bean ayar dosyalarını düzenlemek ve geçerlilik kontrolü yapmak için kullanılan yararlı bir araçtır.Bu editor Spring IDE içine entegre olduğu için özel içerik yardımcısı ve özel şablon desteği gibi özellikleri taşımaktadır.

4.Örnek

Spring IoC tabanı çekirdek paketlerine dayandığı için örneğimizi bu pakete dayanarak vereceğiz.Bu bağlamda projede IoC'yi sağlamak için BeanFactory sınıfını ve ayar bean'lerinin kullanacağız.

Örnek olarak web sayfalarında sıkça kullanılan , kullanıcı geçerliliğini denetleme modülünü gerçekleştirmeye çalışacağız.Spring'in modüler bir yapısı olduğu için web uygulama arayüzünü veya diğer modüllerini kullanmaya gerek kalmadan , bu projeyi bir java uygulaması olarak gerçekleştirebiliriz.

Senaryo olarak , bir kullanıcı yazdığımız uygulama ile geçerlilik denetiminden geçecektir.Geçerlilik sonucu gerekli sınıflarca geri bildirilecektir.Geçerlilik denetimi yapılırken kullanıcı listesi bir veritabanından veya bir başka dosyadan veya hazırlanmış kullanıcı dizisinden alınabilir.Bu konuda örnekte sadece bir kullanıcıyı örnekledik.Örnek , Spring'in yapısı ve IoC sayesinde kolayca çok kullanıcılı bir sisteme dönüştürülebilir.

İlk olarak iş mantığı sınıflarımızı oluşturuyoruz ,Aşağıda ise şifresi ve kullanıcı ismi olan tipik bir kullanıcı sınıfı ve bunu yönetmek için yazılan arayüz :

(User.java)

```
package userAuthentication;

public class User {
        private String login, password;

        public User(String login, String password) {
          this.login = login;
          this.password = password;
        }

        public String getPassword() {
          return password;
        }


       public String getLogin() {
          return login;
        }

}
```

(UserManager.java)

```
package userAuthentication;

public interface UserManager {
        //Kullanıcının tabanda bulunup bulunmadığını test eder
        public boolean exists(String login);

        //Kullanıcı ismi verilen kullanıcıyı dönderir
        public User get(String login);

}
```

Bu sınıf kullanıcıların yönetimi için gerekli soyutlamayı yapan sınıfımız.Bu sınıf görüldüğü üzere bir arayüz , bu sınıfla ilgili yazacağımız gerçekleştirimler daha sonra framework tarafından otomatik atanacak.Bu arayüzün basit bir gerçekleştirimini yapalım:

(SimpleUserManager.java)

```
package userAuthentication;

public class SimpleUserManager implements UserManager {
        public boolean exists(String login) {
           return "agp".equals(login);
        }

        public User get(String login) {
           return new User(login,login);
        }
}
```

Gördüğünüz gibi , bu kısıma istenirse veritabanı işlemleri , dosya işlemleri veya hafıza işlemleri gibi kullanıcı alım işini yapan kod gömülebilir.Ayrıca birden çok gerçekleştirim yapıp bunları yönetebilirsiniz.Kullanıcı geçerliliğini denetleme işlemini sapan hizmet sınıfı ise :

(AuthenticationService.java)

```
package userAuthentication;

public class AuthenticationService {

        private UserManager manager;

        public AuthenticationService(){}
```

```
        public boolean authenticate(String name, String password)
        {
                if (getManager().exists(name)) {
                        User user = getManager().get(name);
                        return user.getPassword().equals(password);
                }else{
                        return false;
                }
        }

        //Kullanıcı yöneticiyi ata , geri çağırım metodu(callback)
        public void setManager(UserManager manager) {
                this.manager = manager;
        }

}
```

Tüm bu sınıflar için bean ayar dosyası oluşturacağız.Burada tanımladığımız bean'lar her gerçekleştirimimizi isimler halinde tutacak.Gerektiği zaman ise BeanFactory aracılığıyla bu tanımlarla ortamdaki gerçekleştirimler kullanılacak

(bean.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

 <bean id="simpleUserManager"
class="userAuthentication.SimpleUserManager"/>

 <bean id="authenticationService"
class="userAuthentication.AuthenticationService">

   <property name="manager">
        <ref bean="simpleUserManager"/>
   </property>

</bean>

</beans>
```

Çalıştırmak için kullanacağımız ana çalıştırılabilir dosya ise :

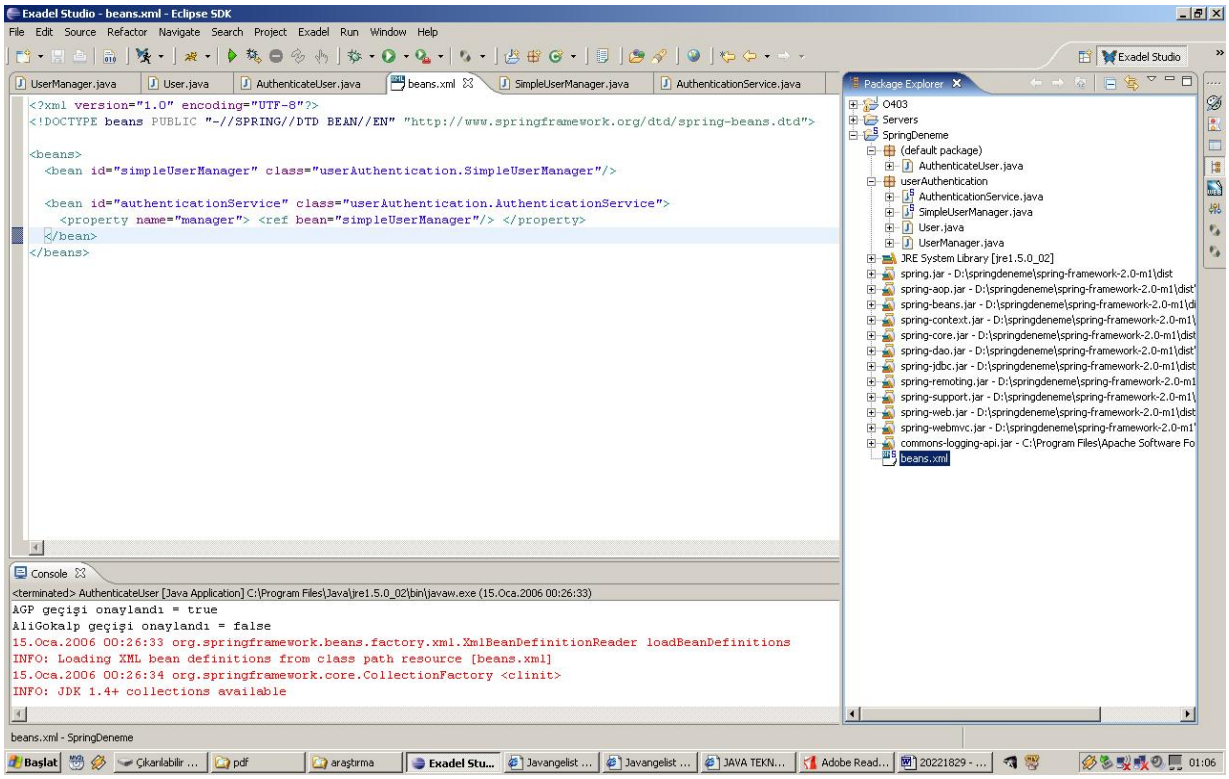
(AuthenticateUser.java)

```
import userAuthentication.AuthenticationService;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
```

```
public class AuthenticateUser {

        public static void main(String[] args) throws Exception {
                //Bean dosyasını al
                ClassPathResource res =
                                        new ClassPathResource("beans.xml");
                XmlBeanFactory beanFactory = new XmlBeanFactory(res);

                //Denetleme için gerekli sınıfı al
                AuthenticationService as = (AuthenticationService)
                beanFactory.getBean("authenticationService");

                //Test yap
                 System.out.println("AGP geçişi onaylandı
                                        as.authenticate("agp", "agp"));

                 System.out.println("AliGokalp geçişi onaylandı = " +
                                        as.authenticate("AliGokalp", "gir"));
        }
}
```

Bu örnekte **Dependency Injection** ya da IoC yöntemi olarak **Setter Injection** kullandık.Dikkat ederseniz **AuthenticationService** sınıfında **UserManager** sınıfını atamak için bir set metodu kullandık.

Eğer Constructor Injection kullanacak olsaydık , **AuthenticationService** sınıfında **UserManager** kısmını constuctor kullanarak atayacaktık.



Kaynaklar

- **Eclipse Plugins – www.eclipse-plugins.info**
- **Spring IDE – www.springide.org**
- **Spring Framework – www.springframework.org**
- **Javangelist Spring Space – javangelist.snipsnap.org**

- **Spring Reference Documentation v1.2.6.**
- **Inversion of Control Containers and the Dependency Injection patter – Martin Fowler**
- **Introduction to Spring Framework – Rod Johnson**