

Spring AOP ile Aspect Oriented Programlama

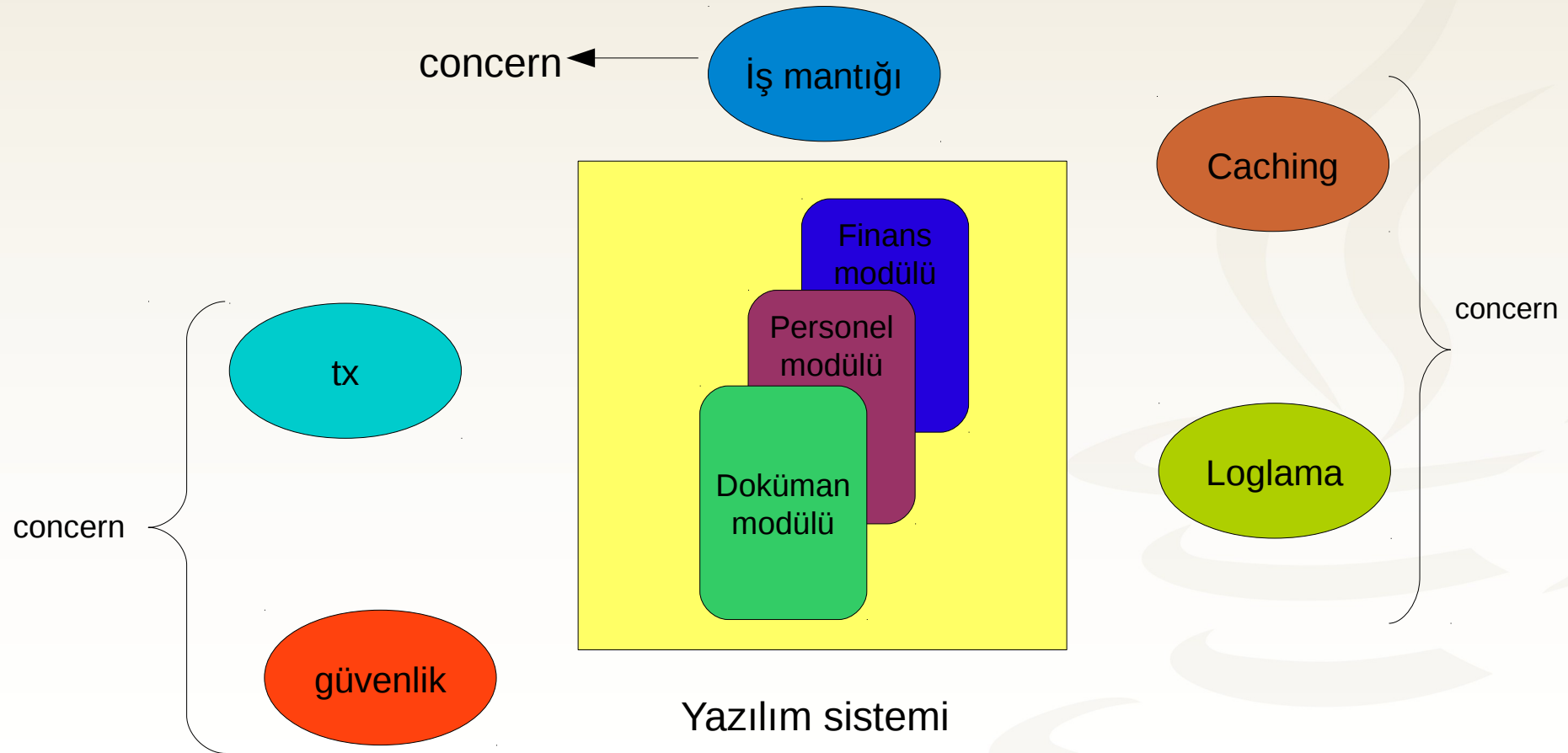


Aspect Oriented Programlama Nedir?

- AOP değişik tip ve nesnelere dağılmış ortak bir **özelliik** veya **davranışın** (**concern**) tek bir yerde ele alınmasını sağlayan programlama modelidir
- OOP'u **tamamlayan** bir yaklaşım sunar, dolayısı ile OOP ile birlikte kullanılır
- Sadece OOP ile çözülemeyen **code scattering** ve **code tangling** gibi problemler en iyi OOP+AOP ile çözülebilir

Concern Nedir?

Herhangi bir yazılım sistemi core(business) ve cross-cutting concern'lerin bileşiminden oluşur



Code Scattering & Tangling

UserService isimli bir sınıfımız olsun ve createUser isimli bir metodunu implement ediyor olalım. Bu metod içerisinde normal iş mantığının yanında loglama ve tx yönetim işlemleri de yapılıyor olsun.

```
public void createUser(User user) {  
    logger.debug("createUser started");  
  
    TransactionStatus txStatus = transactionManager.getTransaction(  
        new DefaultTransactionDefinition(  
            TransactionDefinition.PROPGATION_REQUIRED));  
  
    try {  
        //user yaratilmasi ile ilgili is mantigi burada yer alir  
  
        transactionManager.commit(txStatus);  
    } catch (Exception ex) {  
        transactionManager.rollback(txStatus);  
  
        throw new RuntimeException(ex);  
    } finally {  
        logger.debug("createUser finished");  
    }  
}
```

Code Scattering & Tangling

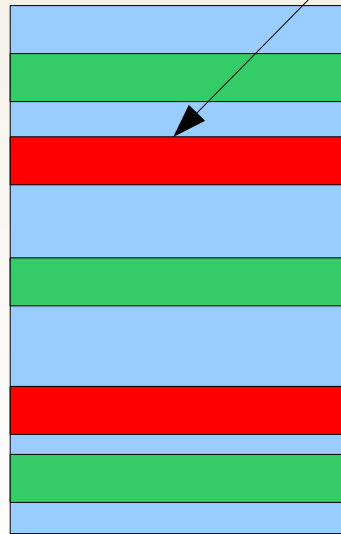
Şimdi de DocumentService isimli bir sınıfımız olsun ve bununda updateDocument isimli bir metodunu implement ediyor olalım. Bu metod içerisinde de normal iş mantığının yanında loglama ve tx yönetim işlemleri de yapılıyor olsun.

```
public void updateDocument(Document doc) {  
    logger.debug("updateDocument started");  
  
    TransactionStatus txStatus = transactionManager.getTransaction(  
        new DefaultTransactionDefinition(  
            TransactionDefinition.PROPROPAGATION_REQUIRES_NEW));  
    try {  
        //doc update ile ilgili is mantigi burada yer alır  
  
        transactionManager.commit(txStatus);  
    } catch (Exception ex) {  
        transactionManager.rollback(txStatus);  
  
        throw new RuntimeException(ex);  
    } finally {  
        logger.debug("updateDocument finished");  
    }  
}
```

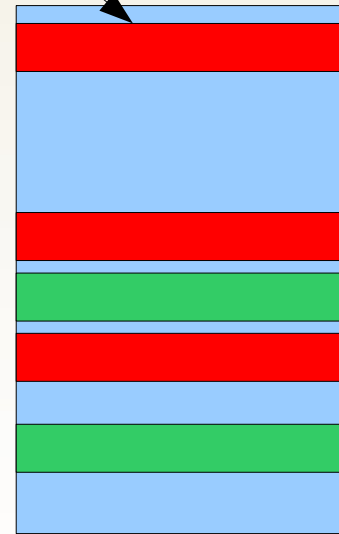
Code Scattering & Tangling

■ : loglama
■ : tx yönetimi

Scattered code : Bir fonksiyonalitenin (concern) değişik modüllere dağılmasıdır



UserService



DocumentService

Tangled code :
Değişik
fonksiyonların
birbiri içine
geçmesidir

Code Scattering & Tangling

```
public void createUser(User user) {
```

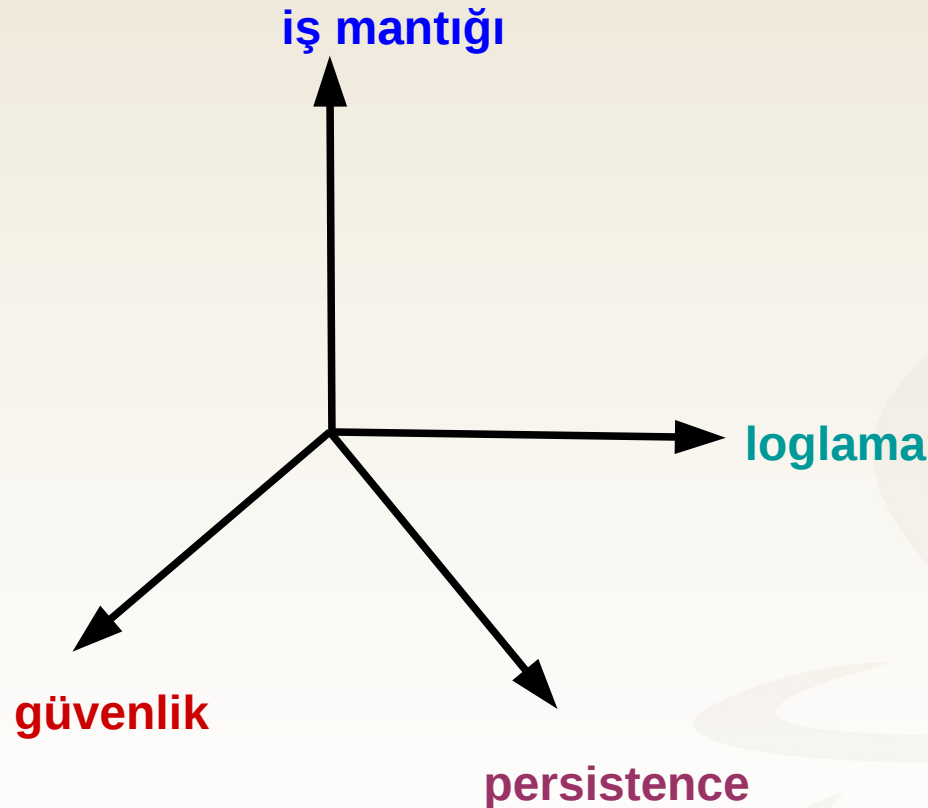
```
//user yaratilmasi ile ilgili is mantigi burada yer alır
```

```
}
```

İş mantığı sadece bu kısımdan oluşmaktadır
Diğer kısımlar iş mantığından tamamen bağımsızdır

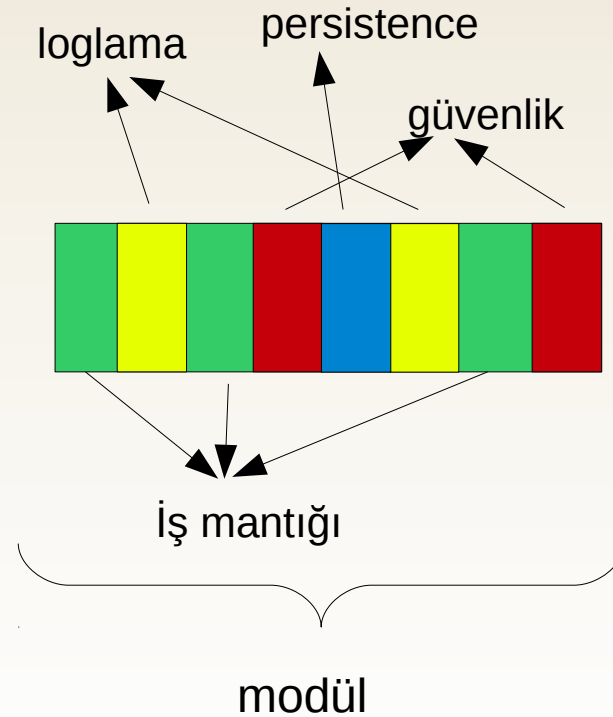
TX yönetimi, loglama, güvenlik, monitoring, auditing, caching gibi pek çok **altyapısal ihtiyaçlar** en iyi AOP ile çözülür

Analizden Gerçekleştirime Geçişte Ortaya Çıkan Problem



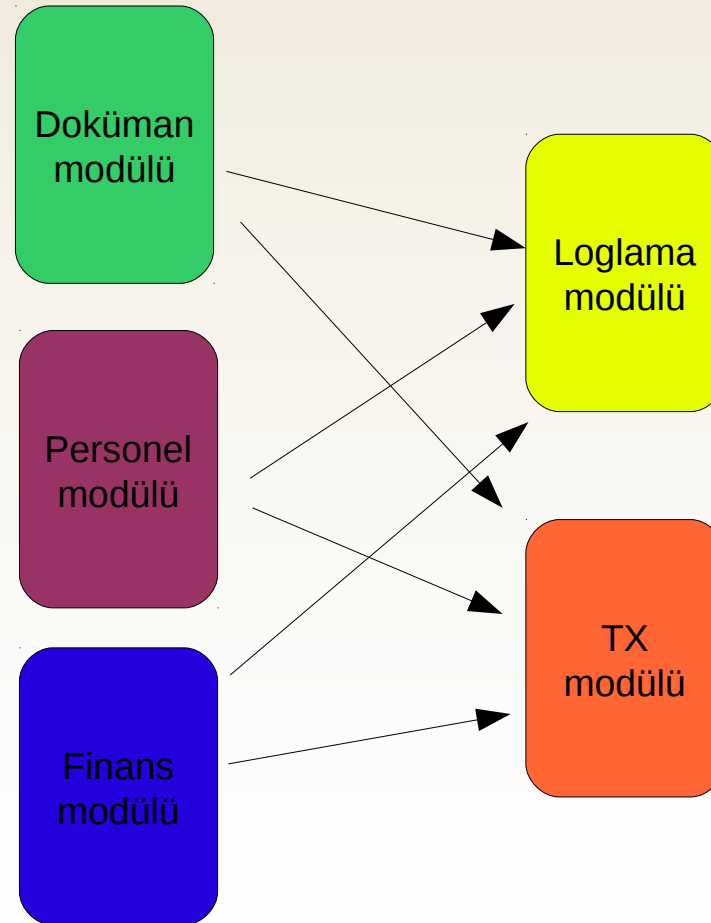
Analiz sürecinde bütün concern'ler birbirinden ayrı modüller biçimde ele alınabilir

Analizden Gerçekleştirmeye Geçişte Ortaya Çıkan Problem

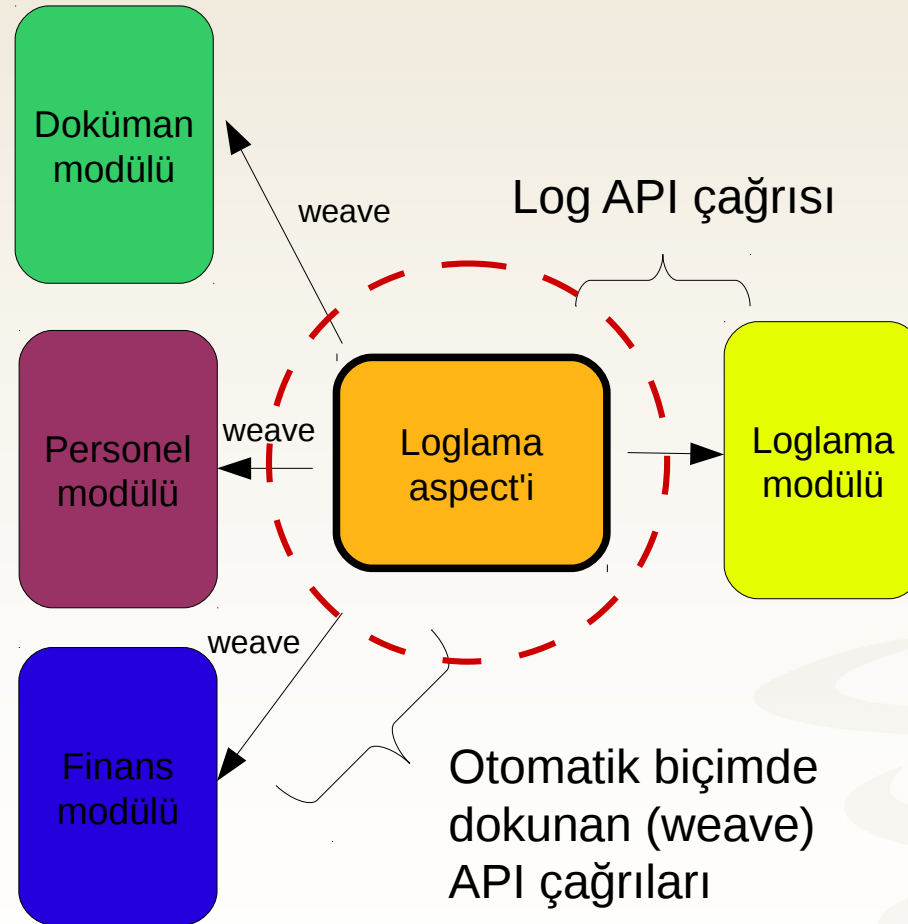


Gerçekleştirim aşamasında ise **code scattering** & **code tangling** problemleri ortaya çıkar

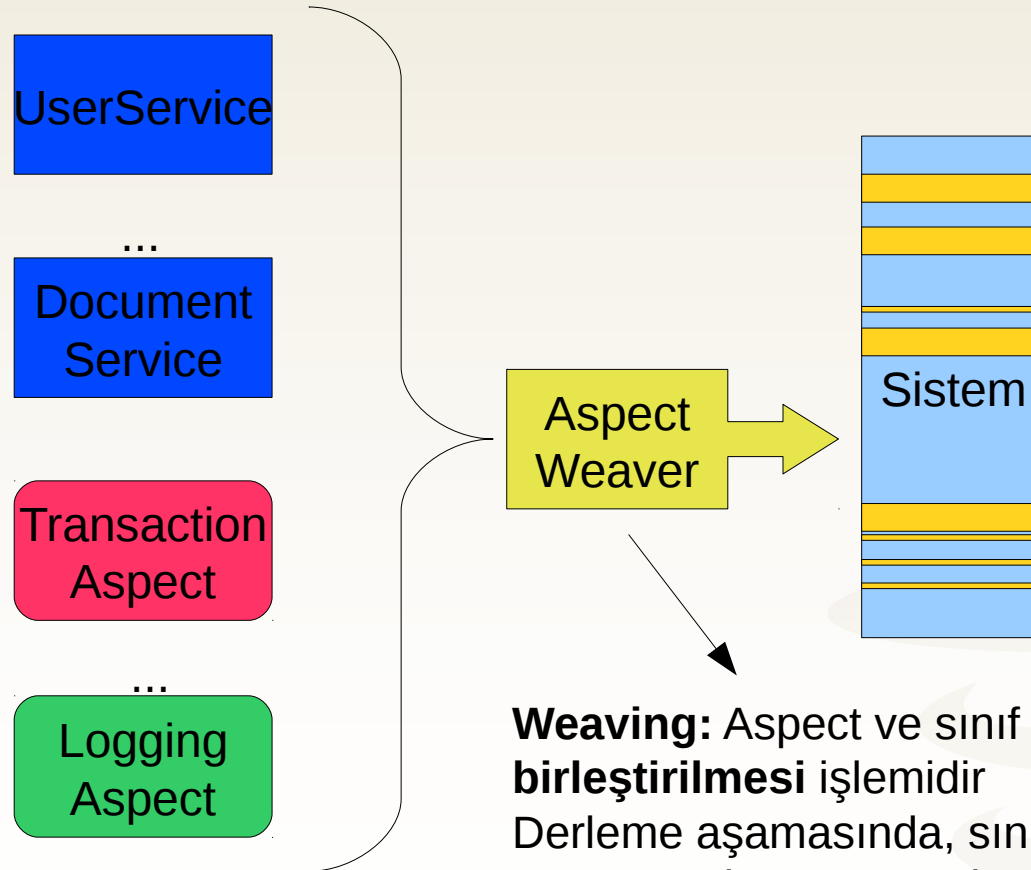
Aspect Bakış Açısı ile Sistem Ayırıştırması



Aspectual Ayrıştırma/Birleştirme

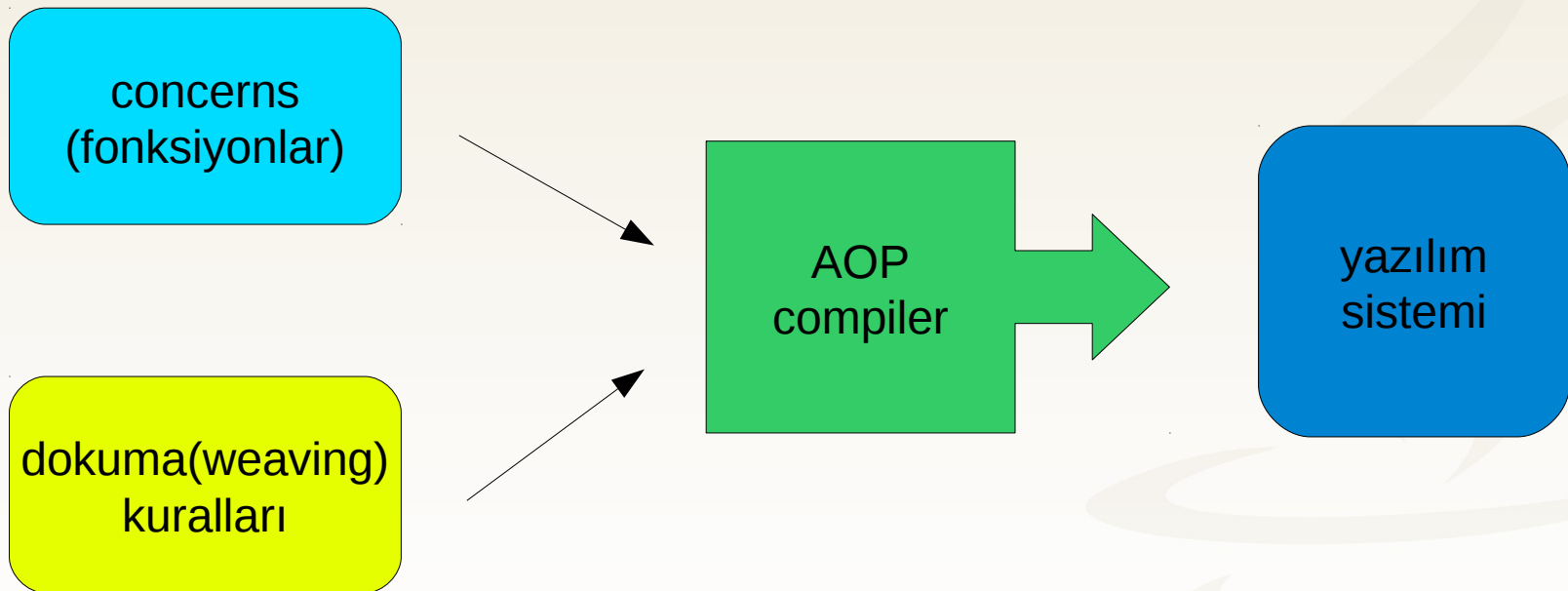


Aspectual Ayrıştırma/Birleştirme

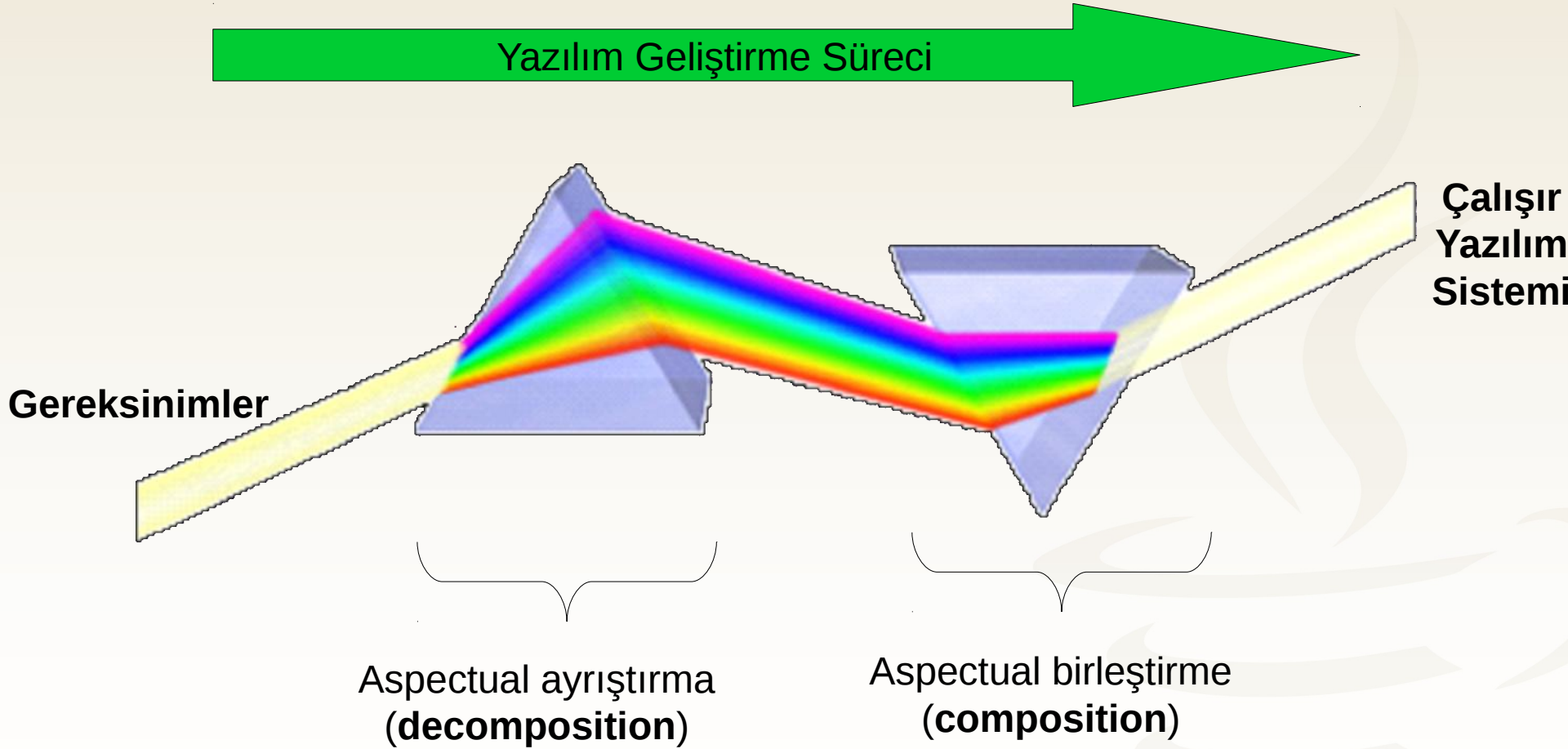


Weaving: Aspect ve sınıf kodlarının tekrardan **birleştirilmesi** işlemidir
Derleme aşamasında, sınıfların yüklenmesi aşamasında, veya runtime'da yapılabilir

Aspectual Ayrıştırma/Birleştirme



Aspectual Ayrıştırma/Birleştirme



AspectJ Nedir?

- AspectJ bir **AOP dilidir**, Java üzerine kuruludur
- OOP'da yapı taşları **sınıf ve nesnelerdir**
- AOP'da ise yapıtaşları **aspect'lerdir**
- Dilin **kendine ait ifade biçimleri** vardır
- Bunun yanında Java5 ile birlikte **annotasyon tabanlı** programlama yapmak da mümkün hale gelmiştir
- Bu sayede **Java dilinin ifade biçimi üzerinden** AOP programlama yapılabilir

Bir Aspect'in Anatomisi (AspectJ İfadeleri ile)

Pointcut expression, ilgili join point(ler)i kod içinde yakalayan **regular expression benzeri** ifadedir

```
public aspect LoggingAspect {  
    private Logger logger =  
        Logger.getLogger(LoggingAspect.class.getName());
```

```
    pointcut allMethodCalls() : execution(* *(..));
```

```
    Object around() : allMethodCalls() {
```

```
        try {
```

```
            logger.info(thisJoinPoint.getSignature().getName() + "  
entered");
```

```
            return proceed();
```

```
        } finally {
```

```
            logger.info(thisJoinPoint.getSignature().getName() + "  
exited");
```

```
        }
```

```
    }
```

```
}
```

Değişik nesnelere dağılmış
fonksiyonun **tek bir noktada**
toplanmış halidir

Join point: programın **çalışma**
esnasındaki bir anıdır

Metoda girmeden önceki an
metot çıkışı

Bir exception fırlatılması anı, bir field'a
erişim anı

advice

Belirli bir join point'de
herhangi bir aspect
tarafından
gerçekleştirilen
aksiyondur

Bir Aspect'in Anatomisi (Annotasyonlar ile)

`@Aspect`

```
public class LoggingAspect {
```

```
    private Logger logger = Logger.getLogger(getClass().getName());
```

```
    @Pointcut("execution(* *(..))")
```

```
    public void allMethodCalls() {  
    }
```

```
    @Around("allMethodCalls()")
```

```
    public Object log(ProceedingJoinPoint pjp) throws Throwable {
```

```
        logger.info(pjp.getSignature().getName() + " started");
```

```
        Object result = pjp.proceed();
```

```
        logger.info(pjp.getSignature().getName() + " finished");
```

```
        return result;  
    }
```

```
}
```

Pointcut expression, ilgili join point(ler)i kod içinde yakalayan regular expression benzeri ifadedir

aspect

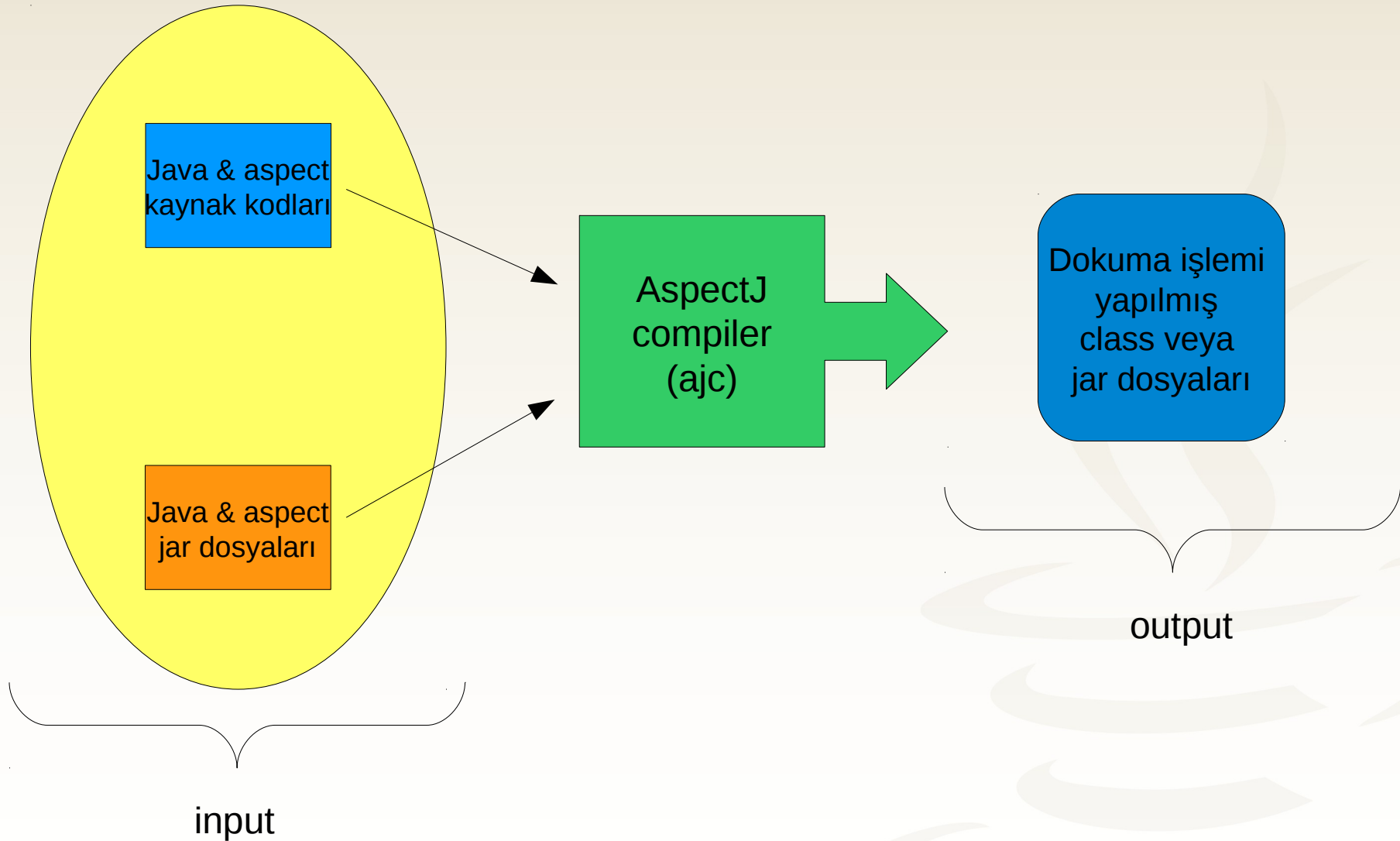
advice

Değişik nesnelere dağılmış fonksiyonun **tek bir noktada toplanmış** halidir

Join point: programın çalışma esnasındaki bir anıdır
Metoda girmeden önceki an
metot çıkışı
Bir exception fırlatılması anı
Bir field'a erişim anı

Belirli bir join point'de herhangi bir aspect tarafından gerçekleştirilen **aksiyondur**

AspectJ Derleme Süreci



Spring AOP Nedir?

- Spring AOP, Spring'in en **temel bileşenlerinden** birisidir
- Spring uygulamaları içerisinde **AOP yapmayı** sağlar
- Aynı zamanda Spring **kendi kabiliyetlerini** hayata geçirmek için de bu modülü kullanır
- TX, güvenlik, caching, validasyon, scoping, remoting gibi pek çok kabiliyet **Spring AOP üzerine kuruludur**

Spring AOP'un Özellikleri

- Spring AOP, AspectJ'ye kıyasla **tam bir AOP çözümü** olma iddiasında **değildir**
- AOP altyapısı ve Spring IoC Container arasında sağlam bir **entegrasyon** kurmayı hedeflemiştir
- Sadece **Spring tarafından yönetilen bean'larda** AOP yapmayı sağlar
- **Proxy tabanlıdır**
- Dolayısı ile sadece **metot execution join point'leri** destekler

Spring AOP'un Özellikleri

- Spring AOP konfigürasyonu **XML** veya **aspectj** anotasyonları vasıtası ile yapılabilir
- Bu sayede, yazılan aspect'lerin daha sonra **AspectJ projesinde de** kullanılması mümkün hale gelir
- Spring AOP sadece pointcut parse ve match işlemleri için **aspectj'nin kütüphanesini** kullanmaktadır

Spring AOP'un Özellikleri

- AOP runtime tamamen Spring AOP proxy'lerinden oluşmaktadır
- **AspectJ compiler** ve **weaver** devreye girmemektedir
- AspectJ runtime'da da **kesinlikle** devre dışıdır

Annotasyon Tabanlı Spring AOP

- Annotasyonlu kullanımı devreye almak için application context XML dosyasında `<aop:aspectj-autoproxy/>` elemanı eklenmesi gerekir
- Java tabanlı konfigürasyonda ise **@EnableAspectJAutoProxy** tanımlı olmalıdır

Annotasyon Tabanlı Spring AOP

- **@Aspect** annotasyonuna sahip bütün **spring managed** bean'lar Spring AOP tarafından tespit edilir
- Bu aspect'ler sadece **diğer Spring managed bean'larda** devreye girerler
- **Spring tarafından yönetilmeyen nesnelerde** devreye girmezler

Join Point Nedir?

- Cross-cutting concern'lerin sistem içerisinde **uygulanacakları yerlerdir**
 - Metot çalışmadan önceki an, çalıştıktan sonraki an
 - Exception fırlatılma anı
 - Field'a erişim anı, field'a değer set etme anı
- Bütün join point'lerin bir de “**context bilgisi**” vardır
 - Metodu/field'ı çağıran, metodun/field'ın çağrıldığı nesne, metot parametreleri vb.

Metot Join Point

- İki türüdür
 - **Execution**: metodun içeriğini kapsar
 - **Call**: metodun çağrıldığı yerleri kapsar
- **Constructor** çağrıları için de aynıdır
 - **Execution**: constructor'ın içeriğini kapsar
 - **Call**: nesnenin yaratıldığı yerleri kapsar
- Spring AOP **sadece execution'ı** destekler
ve **sadece public metot**
invokasyonlarında devreye girebilir

Metot Join Point: Call vs Execution

```
public class Foo {  
    private Baz b;  
    public void foo() {  
        b.baz();  
    }  
}
```

call

```
public class Baz {  
    public void baz() {  
    }  
}
```

execution

```
public class Bar {  
    private Baz b;  
    public void foo() {  
        b.baz();  
    }  
}
```

call

Execution/Call Pointcut Tanımının Yapısı

- **execution(modifiers-pattern? return-type-pattern declaring-type-pattern?.name-pattern(param-pattern) throws-pattern?)**

Sadece AspectJ için anlamlıdır, Spring AOP için anlamı yoktur

Spring AOP proxy tabanlı bir framework olduğu için sadece public metotlarda devreye girebilir

Turuncu renkteki kısımlar opsiyoneldir

Metot Execution Pointcut Tanımları

- **execution/call**

- En yaygın kullanılan pointcut ifadesidir
- **metot join pointleri yakalar**
- `execution(* *(..))`
- `execution(* set*(..))`
- `execution(* com.xyz.service.*.*(..))`
- `execution(* com.xyz.service..*.*(..))`
- `call(* com.xyz.service.AccountService.*(..))`

@annotation Pointcut Tanımı

- **@annotation**
 - Join point'in eşleneceği **metodun belirtilen annotasyona sahip olması** beklenir
 - `@annotation(org.springframework.transaction.annotation.Transactional)`

Yapısal Pointcut Tanımları

- **within**
 - `within(com.xyz.service.*)`
 - `within(com.xyz.service..*)`
- **@within**
 - Join pointlerin tanımlı oldukları **tip**te **belirtilen annotasyon**un olması istenir
 - `@within(org.springframework.transaction.annotation.Transactional)`

Execution Object Pointcut Tanımları

- **this**
 - Join point eşleşmesi için current nesnenin belirli bir tipte olmasını ister
 - `this(com.xyz.service.AccountService)`
- **target**
 - Join point eşleşmesi için hedef nesnenin belirli bir tipte olmasını ister
 - `target(com.xyz.service.AccountService)`

Execution Object Pointcut Tanımları

- **@this**
 - Join point eşleşmesi için current nesnenin sınıfında belirtilen annotasyonun olması gerekir
 - `@this(org.springframework.transaction.annotation.Transactional)`
- **@target**
 - Join point eşlemesi için **hedef nesnenin sınıfında belirtilen annotasyonun** olması gerekir
 - `@target(org.springframework.transaction.annotation.Transactional)`

Argüman Pointcut Tanımları

- **args**
 - `args(java.io.Serializable)`
 - *Metot parametresinin runtime daki tipine bakar*
 - `execution(* *(java.io.Serializable))`
 - *Metot singature'undaki parametre tipine bakar*
- **@args**
 - **Argümanların runtime tiplerinde belirtilen annotasyonun olması gerekir**
 - `@args(com.xyz.security.Classified)`

Spring'e Özel Pointcut Tanımı: bean

- **bean**
 - Spring managed bean isimlerine göre eşleme yapılır
 - `bean(beanIdveyaName)`
 - * wildcard kullanılabilir
 - **AspectJ'de mevcut değildir**
 - `bean(petClinicService)`
 - `bean(*Service)`

Pointcut'ların Paylaşılması

- Pointcut tanımları başka aspectler içerisinde isimleri ile erişilebilir ve yeniden kullanılabilir

```
@Aspect
public class SystemArchitecture {
    @Pointcut("within(x.y.web..*)")
    public void inWebLayer() {}

    @Pointcut("within(x.y.service..*)")
    public void inServiceLayer() {}

    @Pointcut("within(x.y.dao..*)")
    public void inDataAccessLayer() {}

    @Pointcut("execution(* x.y.service.*(..))")
    public void businessService() {}

    @Pointcut("execution(* x.y.dao.*(..))")
    public void dataAccessOperation() {}
}
```

Pointcut'ların Paylaşılması

```
@Aspect
public class SecurityAspect {

    @Around( "x.y.SystemArchitecture.dataAccessOperation()" )
        public Object
doAccessCheck(ProceedingJoinPoint pjp) throws
Throwable {
        // ...
    }
}
```

Bileşke Pointcut Tanımları

```
@Pointcut("execution(public * *(..))")  
private void anyPublicOperation() {}
```

```
@Pointcut("within(x.y.service..*)")  
private void inService() {}
```

```
@Pointcut("anyPublicOperation() && inService()")  
public void publicServiceOperations() {}
```

Pointcut tanımlarına isimleri ile erişmek mümkündür, bu diğer aspectler içinden erişim için de geçerlidir

Pointcut ifadeleri &&, || ile **birleştirilebilir**
Daha kompleks pointcut tanımlarını, basit pointcut'ları bir araya getirerek oluşturmayı sağlar

Around Advice

- **En yaygın** kullanılan advice tipidir
- Join point'i **wrap** eder, bütün diğer advice tiplerini kapsar
- Join point execution'ını **istenildiği gibi yönetmek** mümkündür
- İstenirse asıl metot çalıştırılmayabilir, input parametreleri değiştirilerek çağrılabilir ya da farklı bir return değeri dönülebilir

Around Advice

```
@Aspect
public class AroundExample {
    @Around("execution(* x.y.service.*(..))")
    public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
        //asıl metot çağrılmadan önce...
        Object retVal = pjp.proceed();
        //asıl metot çağırıldıktan sonra...
        return retVal;
    }
}

public aspect AroundExample {
    Object around() : execution(* x.y.service.*(..)) {
        //asıl metot çağrılmadan önce..
        Object retVal = proceed();
        //asıl metot çağırıldıktan sonra...
        return retVal;
    }
}
```

→ **Around** advice metotlarında ilk parametre **ProceedingJoinPoint** olmak **zorundadır**

Advice Parametreleri

- args() pointcut tanımı ile
 - Hem **eşlenecek join point'leri** sınırlanabilir
 - Hem de **metot parametreleri advice'a** input argüman olarak geçilebilir
- this, target, args, @within, @target, @args, @annotation hepsi **metot parametrelerini advice'a geçmek** için kullanılabilir

Advice Parametreleri

```
@Around ("execution(List<Account> find*(..)) && args(accountHolderName)")
public Object preProcessQuery(ProceedingJoinPoint pjp,
    String accountHolderName) throws Throwable {

    String newName = "act_" + accountHolderName + "_user";

    return pjp.proceed(new Object[] { newName });

}
```

args() pointcut'daki değişken ismi ile metod parametresindeki değişken ismi eşleşmelidir. Böylece metod input argümanı hem joinpoint eşleşmesinde kullanılmakta hem de advice metoduna değer olarak geçilebilmektedir

Spring AOP ile Aspect Tanımı ve Konfigürasyonu

@Aspect notasyonunu devreye sokar

```
<aop:aspectj-autoproxy/>
```

Aspect'in devreye girebilmesi için bean olarak tanımlanması şarttır

```
<bean class="com.javaegitimleri.petclinic.aop.LoggingAspect"/>
```

Sadece spring tarafından yönetilen nesnelere etki eder

```
<bean id="petClinicDao"  
class="com.javaegitimleri.petclinic.dao.PetClinicDaoImpl"/>
```

```
<bean id="petClinicService"  
class="com.javaegitimleri.petclinic.service.PetClinicServiceImpl">  
    <property name="petClinicDao" ref="petClinicDao"/>  
</bean>
```

Spring AOP ve Introductions

- AspectJ terminolojisinde **inter-type decleration** olarak bilinir
- Bu tür yapılara **mixin** adı da verilmektedir
- Belirli tipteki nesnelerin runtime'da **yeni bir interface'e daha sahip olmaları** sağlanır
- Aspect tanımı içerisinde **@DeclareParents** anotasyonu ile gerçekleştirilir

Spring AOP ve Introductions

AspectJ type pattern'ı ile eşleşen nesneler UsageTracked arayüzüne de sahip olurlar

```
@Aspect
public class UsageTrackedAspect {
    @DeclareParents(value="com.javaegitimleri.service.*+",
                    defaultImpl=DefaultUsageTracked.class)
    private UsageTracked mixin;
}
```

UsageTracked arayüzüne sahip olacak bean'lere bu arayüz üzerinden gerçekleştirecekleri davranış da **defaultImpl** attribute'undaki sınıf ile sağlanır

Spring AOP ve Introductions

@Aspect

```
public class UsageTrackingAspect {
```

```
    @Before("execution(* com.javaegitimleri.service.*.*(..)) && this(usageTracked)")
```

```
    public void recordUsage(UsageTracked usageTracked) {  
        usageTracked.incrementUsageCount();  
    }
```

```
}
```

```
}
```

Herhangi başka bir aspect içerisinde UsageTracked arayüzüne sahip proxy nesneler yakalanıp UsageTracked arayüzü üzerinden işlem yapılabilir

Spring AOP ve Introductions

```
UsageTracked usageTracked = (UsageTracked)  
    applicationContext.getBean("petClinicService");  
  
usageTracked.getUsageCount();
```

Çalışma zamanında ilave arayüz eklenen bean'lere erişilerek
bu bean'ler yeni arayüze downcast edilerek de kullanılabilir

AspectJ Aspect'lerine IoC Desteği

- AspectJ ile yazılmış ve derlenmiş aspect'leri **Spring Container** içerisinde konfigüre etmek ve kullanmak da mümkündür
- AspectJ aspectlerin konfigürasyonları için Spring **bean tanımları** yapılabilir
- AspectJ aspectleri iki türdür
 - **Singleton** ve **non-singleton** aspect'ler
- Spring **sadece singleton** aspect'leri destekler
- **Runtime**'da bu aspect'ler **AspectJ** tarafından yaratılırlar

AspectJ Aspect'lerine IoC Desteği

```
<beans...>
```

```
<bean id="loggingAspect" class="x.y.LoggingAspect" factory-method="aspectOf">  
  <property name="enabled" value="true"/>  
</bean>
```

```
</beans>
```

Spring, singleton aspectj aspect'inin **aspectOf** metodunu çağırarak singleton instance'a erişir ve bunu bir Spring managed bean olarak ele alır

```
<aspectj>  
<!-- available options: -verbose -showWeaveInfo -debug -->  
  <weaver options="">  
    <!-- only weave classes in our application-specific  
packages -->  
    <!-- <include within="com.javaegitimleri.*"/> -->  
  </weaver>  
  <aspects>  
    <!-- weave in following aspect(s) -->  
    <aspect name="x.y.LoggingAspect"/>  
  </aspects>  
</aspectj>
```

META-INF/aop.xml içeriği

LTW için
LoggingAspect burada da
tanımlı olmalıdır

ayrıca

JVM
-javaagent:aspectjweaver.jar
ile başlatılmalıdır

Domain Nesnelerine Dependency Injection

- Spring ekibi tarafından yazılmış hazır **AspectJ aspect'leri** yardımı ile Spring tarafından yönetilmeyen **domain nesnelerine dependency injection** yapılabilir
- Bu aspect'ler **spring-aspects.jar** içerisinde
- DI yapılacak sınıflar **@Configurable** ile işaretlenmelidir

Domain Nesnelerine Dependency Injection

@Configurable

```
public class Account {  
    @Autowired  
    private AccountService accountService;  
    // ...  
}
```

- Bağımlılıkları enjekte etmek için domain sınıfında **@Autowired** anotasyonu kullanılabilir

Domain Nesnelerine Dependency Injection

- **@Configurable** anotasyonunun aktive olması için `<context:spring-configured/>` elemanı XML'de tanımlanmalıdır
- Java tabanlı konfigürasyonda **@EnableSpringConfigured** kullanılmalıdır
- DI normalde **constructor'lardan sonra** gerçekleşir
- **@Configurable(preConstruction=true)** ile constructor öncesi DI yapması sağlanabilir

LTW ve Spring

- Domain nesnelere DI işleminin gerçekleşebilmesi için ilgili domain sınıflarının **AspectJ aspectleri** ile **weave edilmesi** gerekir
- Bunun için ya compile time build taskları , ya da **load time weaver** kullanılabilir
- LTW işlemi JVM **sınıfları yüklerken** gerçekleştirilir

LTW ve Spring

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

```
<context:load-time-weaver/>
```

```
</beans>
```

aspectj-weaving attribute ile
spring-aspects.jar'daki AspectJ
aspect'lerinin dokunması aktive edilir

on: aspectj weaving aktiftir

off: aspectj weaving devre dışıdır

autodetect: (default)

eğer classpath'de bir **META-INF/aop.xml**
dosya tespit edilirse aktif olur

Tomcat, Resin gibi sunucularda,
standalone uygulamalarda ve
entegrasyon testlerinde ayrıca JRE'de
**-javaagent:path/to/spring-
instrument.jar** belirtilmesi gerekir

İletişim

- **Harezmi** Bilişim Çözümleri
- Kurumsal Java Eğitimleri
- <http://www.java-egitimleri.com>
- info@java-egitimleri.com

