Tasarım Örüntüleri ile Spring Eğitimi 10

harezmi bilişim çözümleri

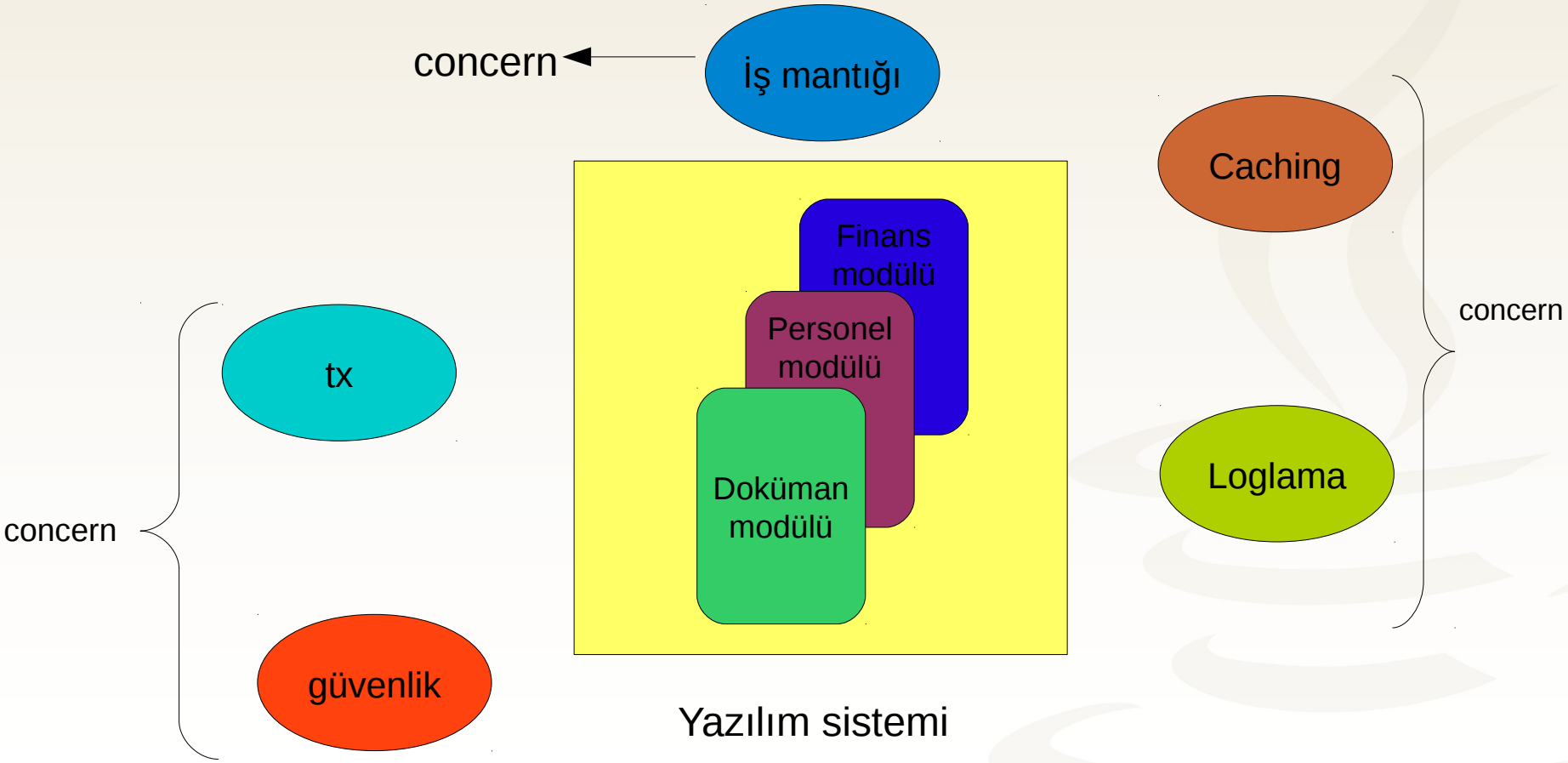
Spring AOP ve AspectJ ile Aspect Oriented Programlama

Aspect Oriented Programlama Nedir?

- AOP değişik tip ve nesnelere dağılmış **ortak bir özellik 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 metot 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.PROPAGATION_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 metot içerisinde de normal iş mantığının yanında loglama ve tx yönetim işlemleri de yapılıyor olsun.

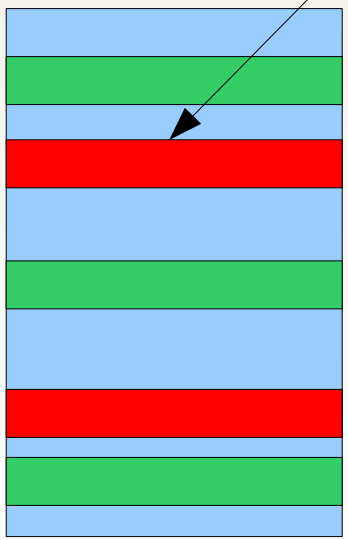
```
public void updateDocument(Document doc) {
    logger.debug("updateDocument started");

    TransactionStatus txStatus = transactionManager.getTransaction(
    new DefaultTransactionDefinition(
        TransactionDefinition.PROPAGATION_REQUIRES_NEW));
    try {
        //doc update ile ilgili is mantigi burada yer alir

        transactionManager.commit(txStatus);
    } catch(Exception ex) {
        transactionManager.rollback(txStatus);

        throw new RuntimeException(ex);
    } finally {
        logger.debug("updateDocument finished");
    }
}
```

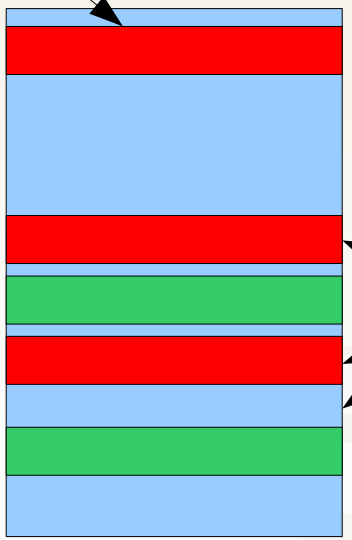
Code Scattering & Tangling

: loglama

: tx yönetimi

Scattered code : Bir fonksiyonalitenin (concern) değişik modüllere dağılmasıdır





Tangled code : Değişik fonksiyonların birbiri içine geçmesidir

UserService

DocumentService

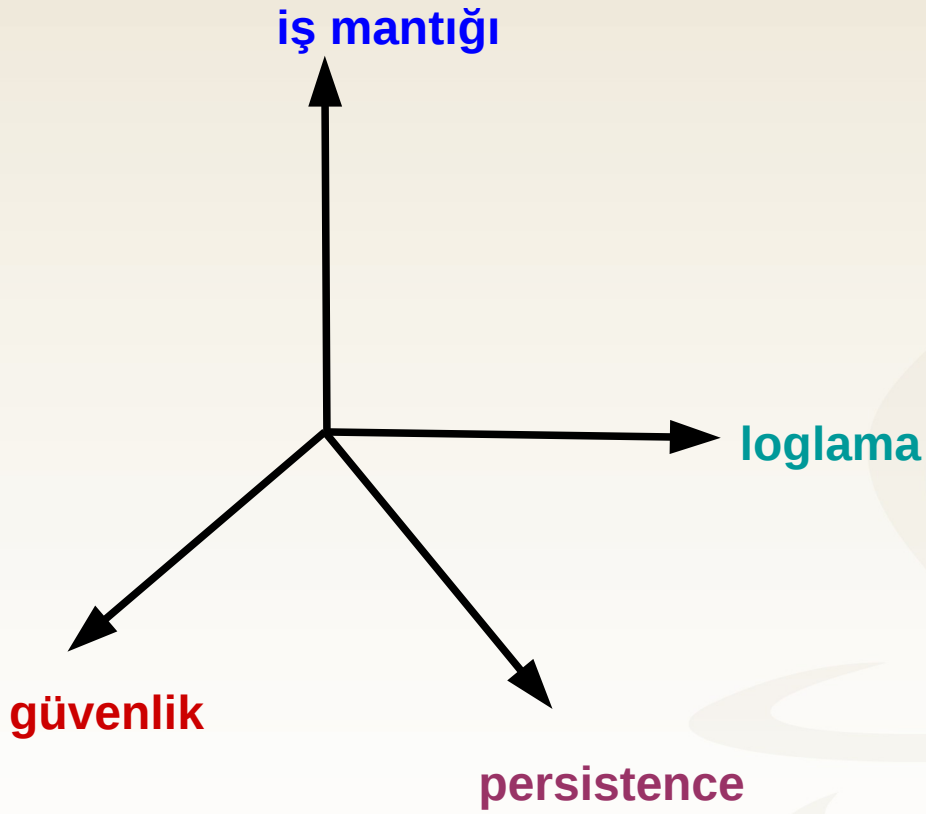
Code Scattering & Tangling

```
public void createUser(User user) {
    //user yaratilmasi ile ilgili is mantigi burada yer alir
}
```

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

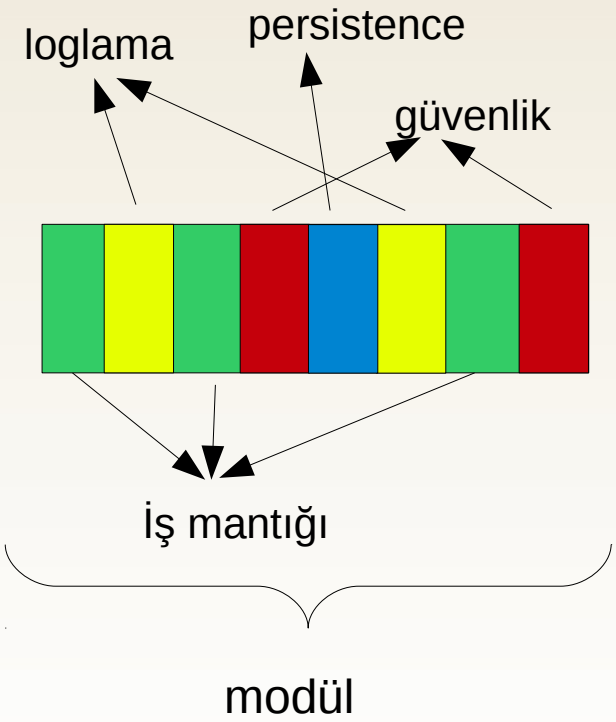
TX yönetimi, loglama, güvenlik, monitoring, auditing, caching gibi pek çok **altyapısal ihtiyaçlar** en iyi AOP ile çözülür

Analizden Gerçekleştirime Geçişte Ortaya Çıkan Problem



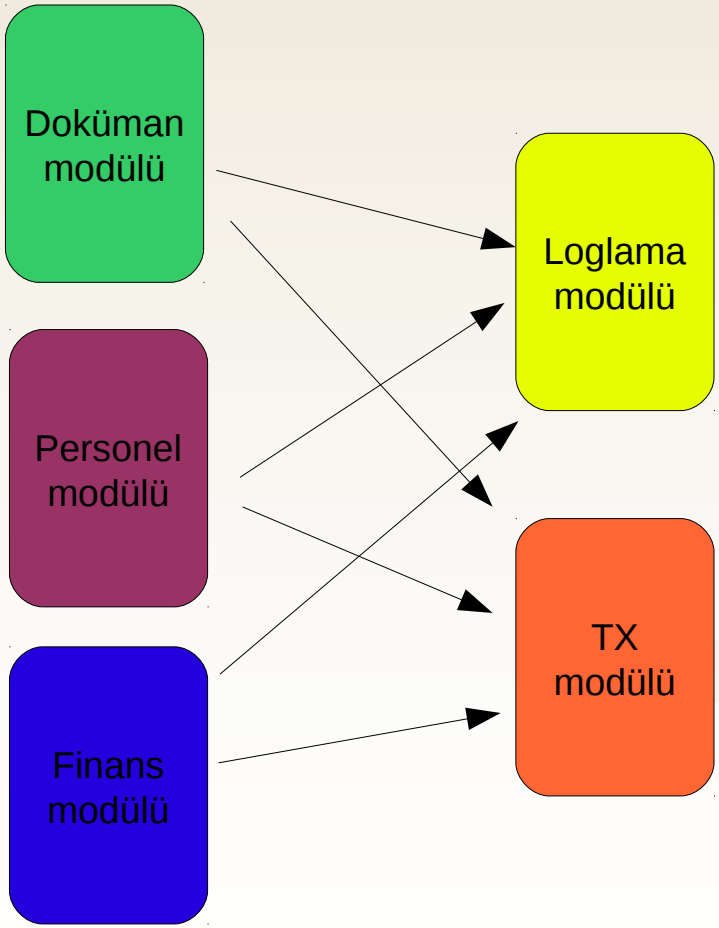
Analiz sürecinde bütün concern'ler birbirinden ayrı modüller biçimde ele alınabilir

Analizden Gerçekleştirime Geçişte Ortaya Çıkan Problem

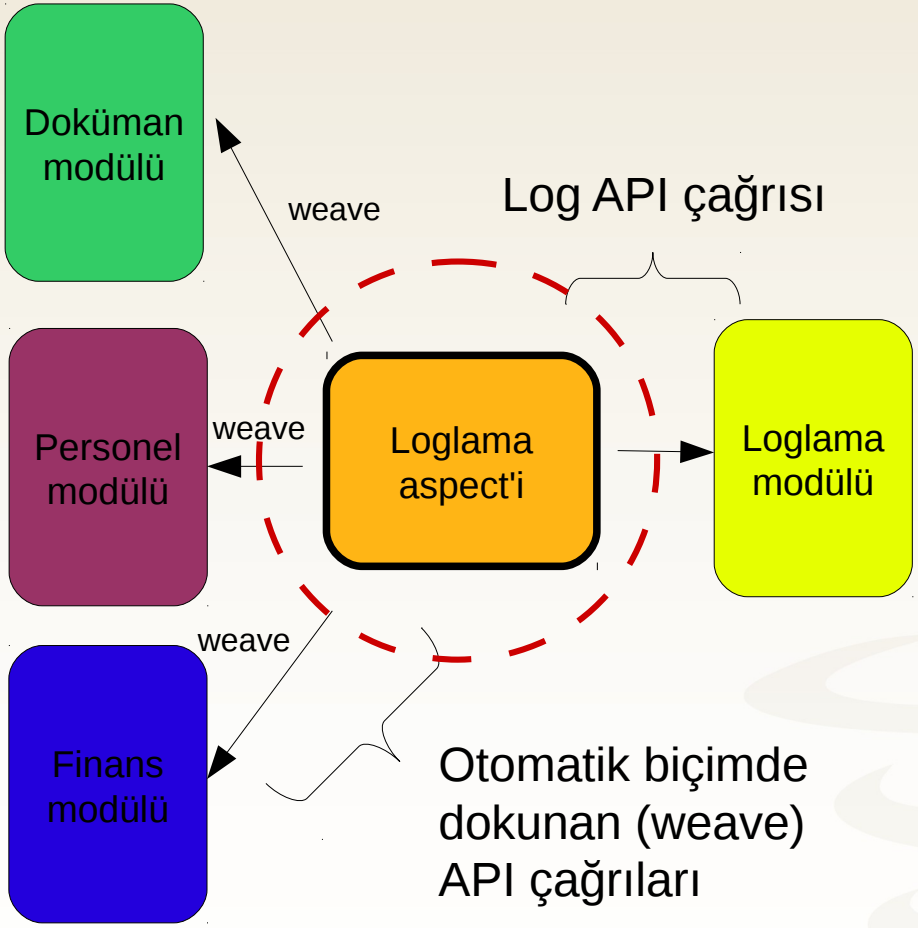


Gerçekleştirim aşamasında ise **code scattering** & **code tangling** problemleri ortaya çıkar

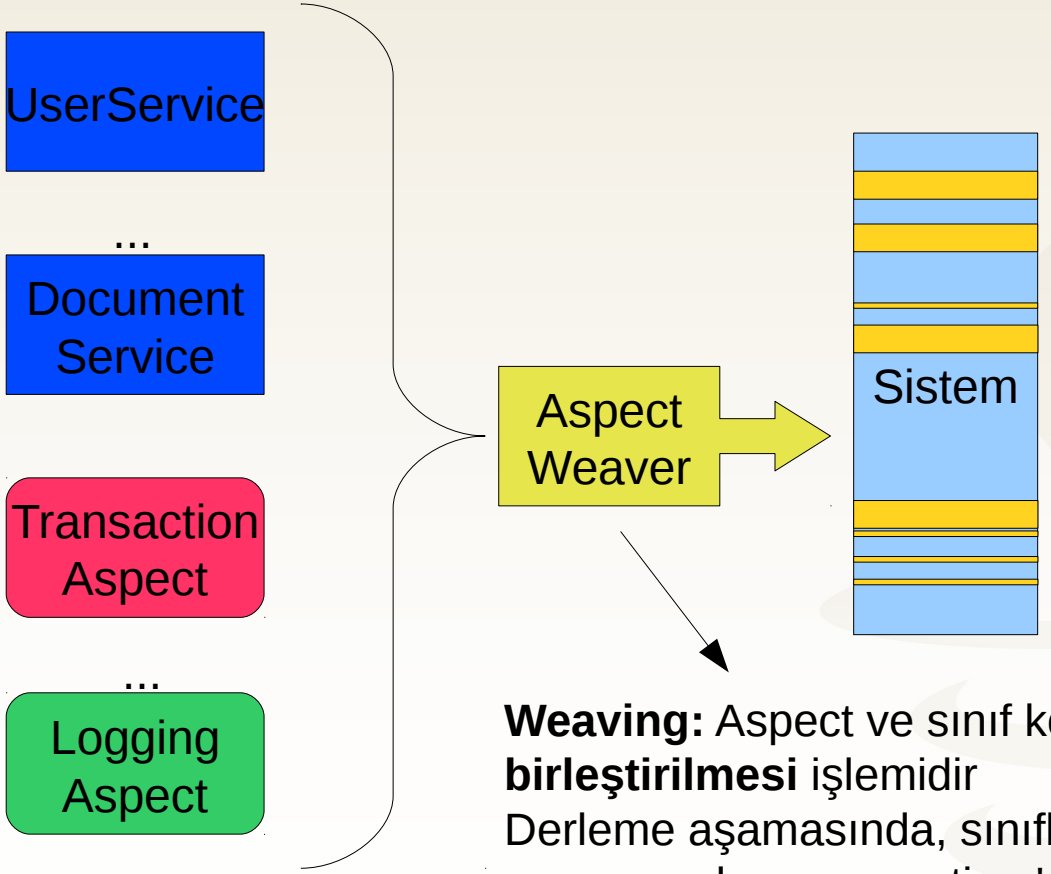
Aspect Bakış Açısı ile Sistem Ayrıştırması



Aspectual Ayrıştırma/Birleştirme

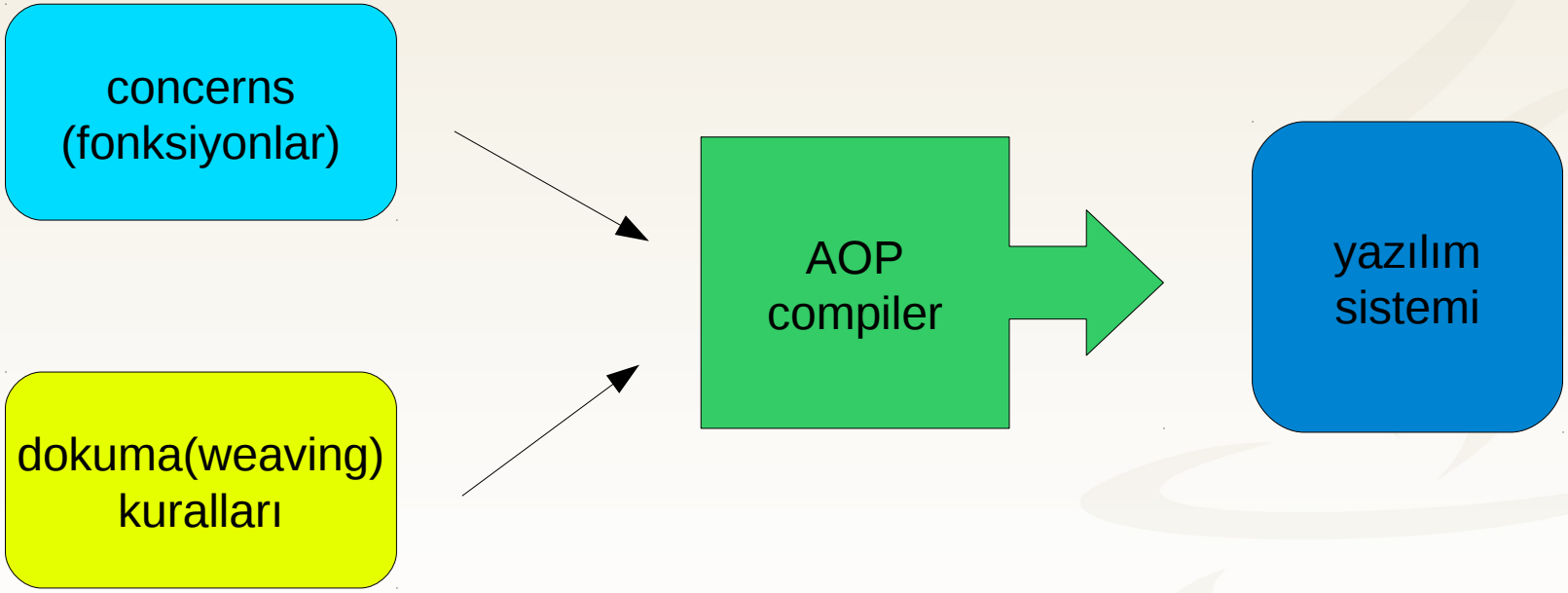


Aspectual Ayrıştırma/Birleştirme

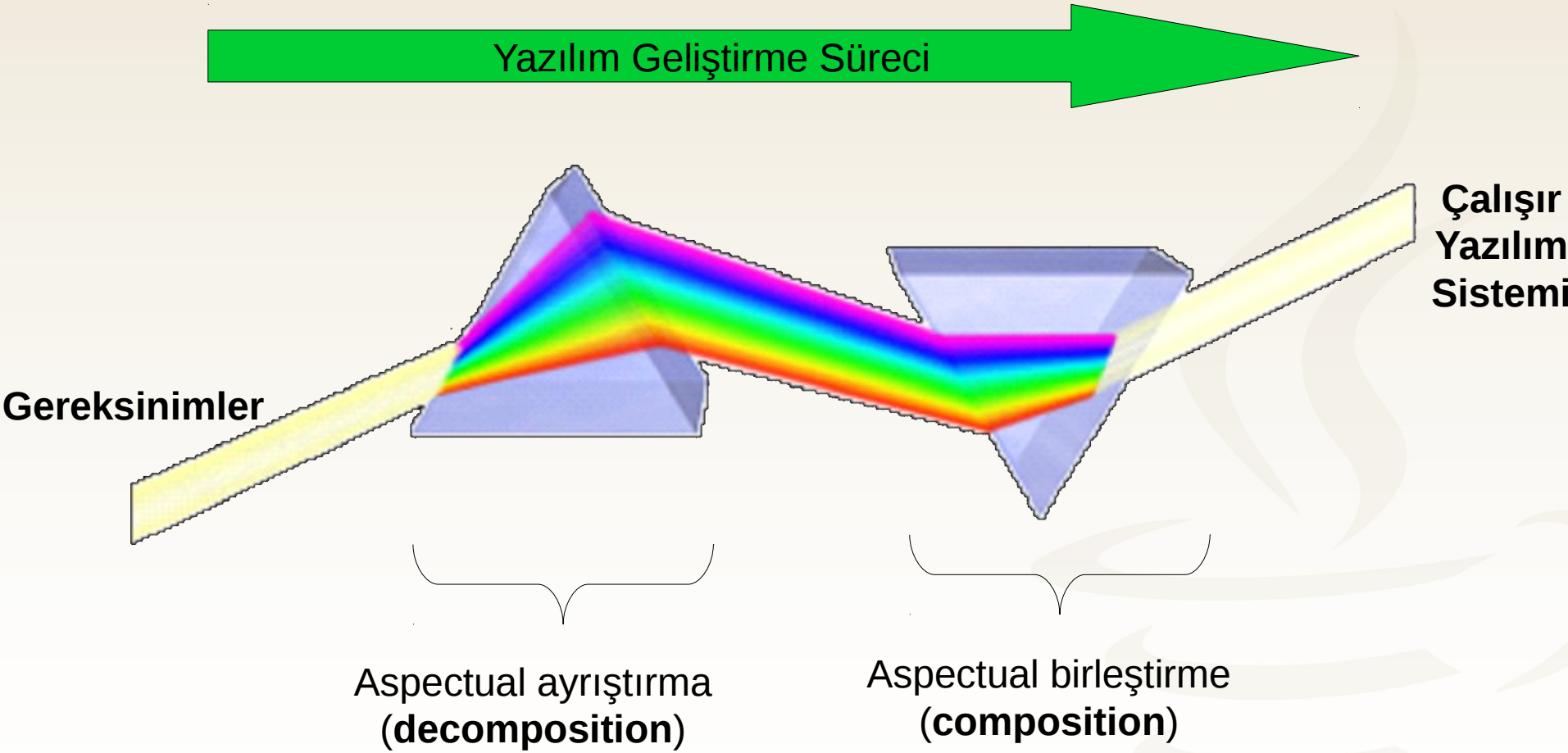


Weaving: Aspect ve sınıf kodlarının tekrardan **birleştirilmesi** işlemidir
Derleme aşamasında, sınıfların yüklenmesi aşamasında, veya runtime'da yapılabilir

Aspectual Ayrıştırma/Birleştirme



Aspectual Ayrıştırma/Birleştirme



AspectJ Nedir?

- AspectJ bir **AOP dili**dir, Java üzerine kuruludur
- OOP'da yapı taşları **sınıf ve nesne**lerdir
- 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)

JAVA Eğitimleri

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

```
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");
        }
    }
}
```

aspect

advice

Değişik nesnelere dağılmış fonksiyonun **tek bir noktada toplanmış hali**dir

Join point: programın **çalışma esnasındaki bir an**ıdır
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 **aksiyon**dur

Bir Aspect'in Anatomisi (Annotasyonlar ile)

JAVA Eğitimleri

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

```
@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;
    }
}
```

aspect

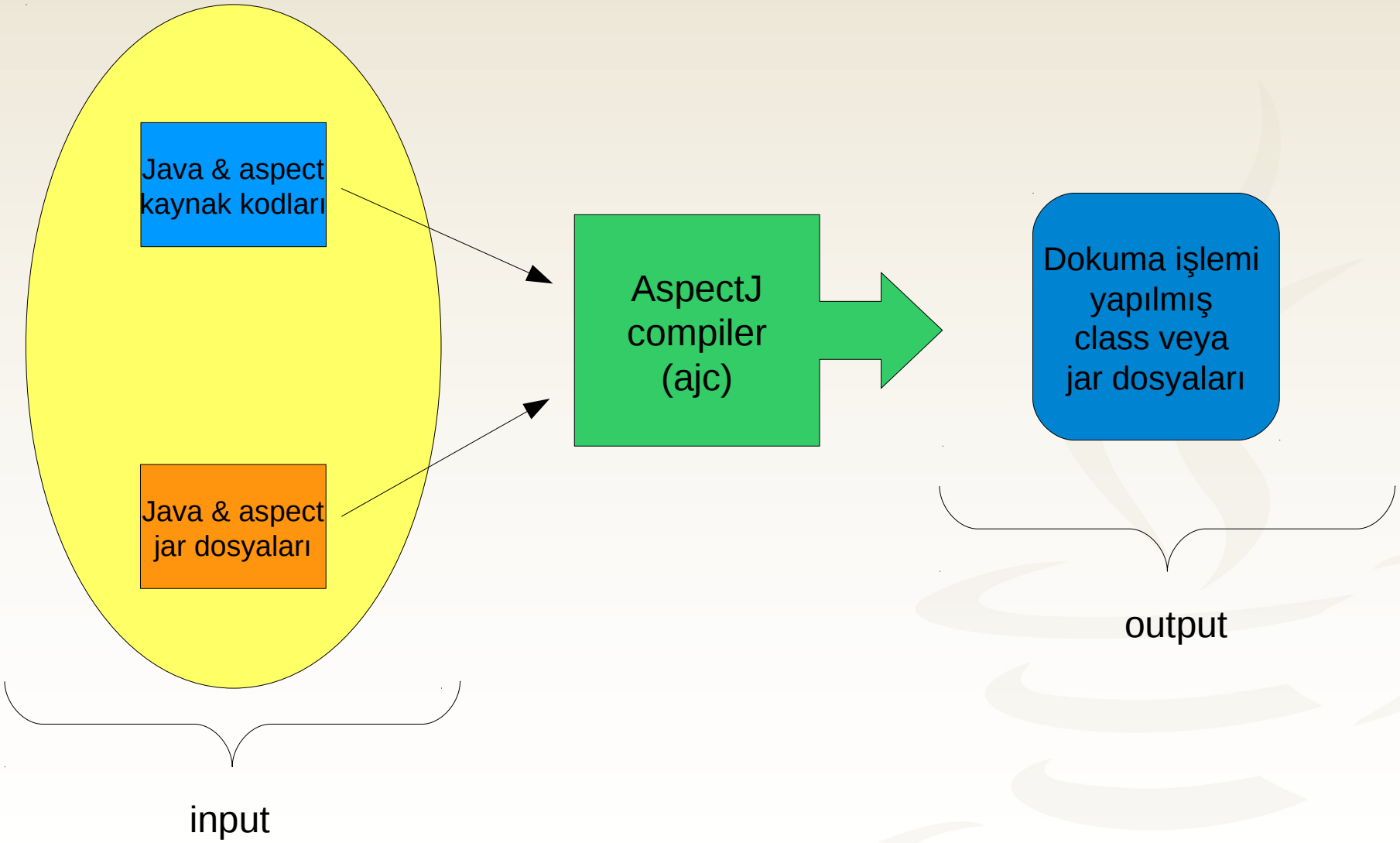
advice

Değişik nesnelere dağılmış fonksiyonun **tek bir noktada toplanmış hali**dir

Join point: programın **çalışma esnasındaki bir anı**dır
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 **aksiyon**dur

AspectJ Derleme Süreci



Spring AOP Nedir?

- Spring AOP, Spring'in en **temel bileşen**lerinden birisidir
- Spring uygulamaları içerisinde **AOP yapmayı** sağlar
- Aynı zamanda Spring **kendi kabiliyetleri**ni hayata geçirmek için de bu modülü kullanır
- TX, güvenlik, caching, validasyon, scoping, remoting gibi pek çok kabiliyet **Spring AOP üzerine kurulu**dur

Spring AOP'un Özellikleri

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

Spring AOP'un Özellikleri

- Spring AOP konfigürasyonu **XML** veya **aspectj annotasyonları** 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 ilgili kütüphanesi**ni 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'lar**da devreye girerler
- **Spring tarafından yönetilmeyen nesneler**de devreye girmezler

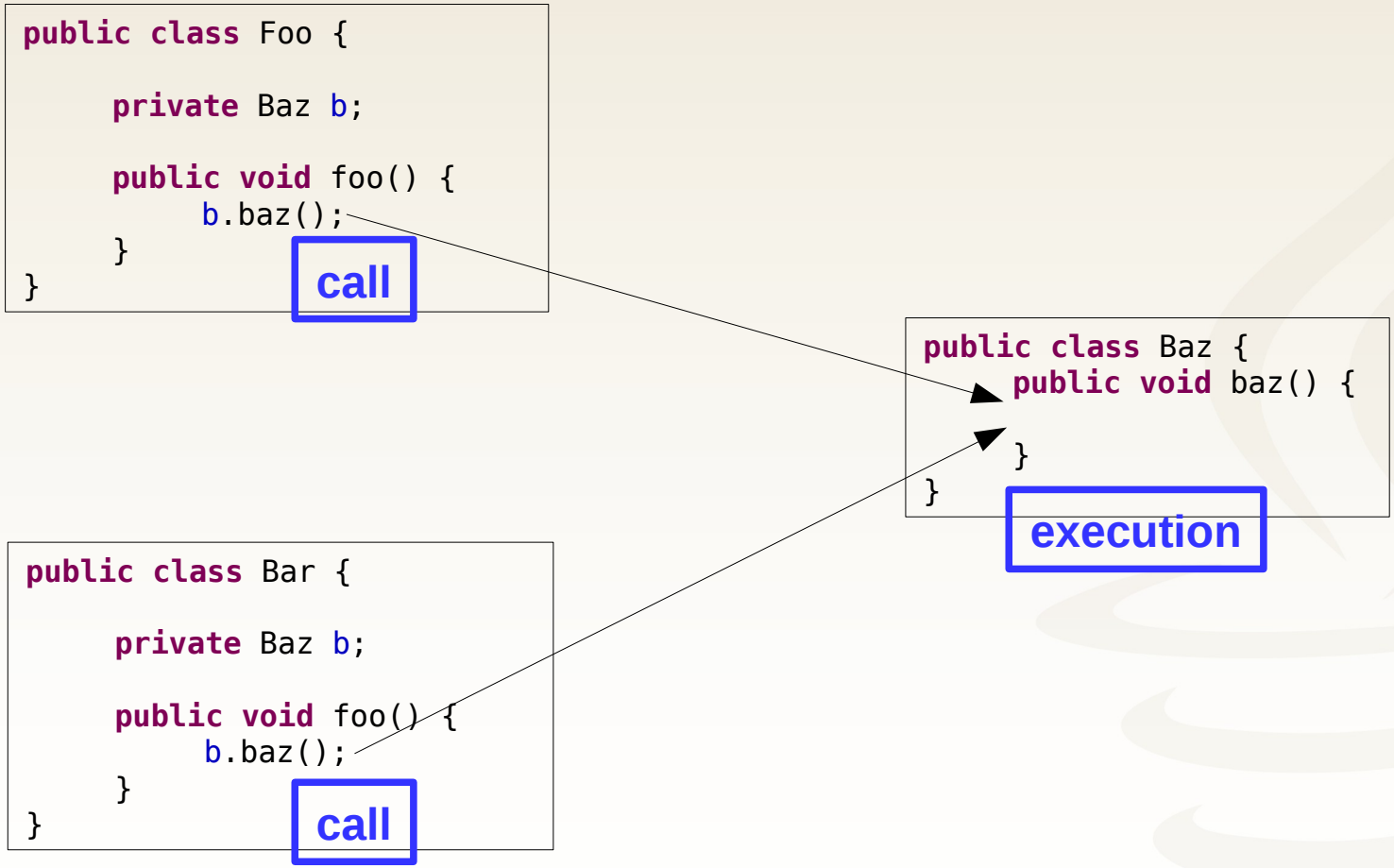
Join Point Nedir?

- Cross-cutting concern'lerin sistem içerisinde **uygulanacakları yerler**dir
  - 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/fieldın çağrıldığı nesne, metot parametreleri vb.

Metot Join Point

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

Metot Join Point: Call vs Execution



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 point**leri 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 **tipte 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 annotasyon**un 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 annotasyon**un 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 değil**dir
  - bean(petClinicService)
  - bean(*Service)

Pointcut'ların Paylaşılması

- Poincut tanımları başka aspectler içerisinden 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 çağrıldıktan sonra...
    return retVal;
  }
}
```

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

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

Before Advice

```
@Aspect
public class BeforeExample {
  @Before("execution(* x.y.service.*.*(..))")
  public void doAccessCheck() {
    // ...
  }
}
```

Eğer advice içinde **exception meydana** gelirse yakalanan joint point execute edilmeyecektir

```
public aspect BeforeExample {

    before() : execution(* x.y.service.*.*(..)) {
    //...
    }

}
```

After Returning Advice

```
@Aspect
public class AfterReturningExample {
  @AfterReturning(pointcut="execution(* x.y.service.*.*(..))",
returning="retVal")
  public void doAccessCheck(Object retVal) {
    // ...
  }
}
```

Dönen değeri advice içerisine parametre olarak geçmek mümkündür

```
public aspect AfterReturningExample {

    after() returning(Object retVal) :execution(* x.y.service.*.*(..)) {
    //...
    }

}
```

After Throwing Advice

```
@Aspect
public class AfterThrowingExample {

  @AfterThrowing(pointcut="execution(* x.y.service.*.*(..))",
throwing="ex")
  public void doRecoveryActions(DataAccessException ex) {
    // ...
  }
}
```

Fırlatılan exception'ı da advice içerisine Parametre olarak geçmek mümkündür

```
public aspect AfterThrowingExample {

    after() throwing(DataAccessException ex) : execution(*
x.y.service.*.*(..)) {
        //...
    }

}
```

After (finally) Advice

```
@Aspect
public class AfterFinallyExample {
@After(pointcut="execution(* x.y.service.*.*(..))")
  public void doActionsAlways() {
    // ...
  }
}
```

```
public aspect AfterFinallyExample {
    after() : execution(* x.y.service.*.*(..)) {
        //...
    }
}
```

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 metot parametresindeki değişken ismi eşleşmelidir. Böylece metot input argümanı hem joinpoint eşleşmesinde kullanılmakta hem de advice metoduna değer olarak geçilebilmektedir

Advice Parametreleri

```
public aspect ArgsExample {

        List around(String name) : execution(List find*(..)) && args(name) {
        String newPattern = preProcess(name);

        return proceed(newPattern);
    }

    private String preProcess(String name) {
        return name;
    }
}
```

Spring AOP ile Aspect Tanımı ve Konfigürasyonu

JAVA Eğitimleri

@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** annotasyonu 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)
                    appContext.getBean("petClinicService");

usageTracked.getUsageCount();
```

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

AspectJ Aspect'lerinin Spring İçerisinde Kullanımı

- AspectJ ile yazılmış ve derlenmiş aspect'leri **Spring Container** içerisinde konfigüre etmek ve kullanmak da mümkündür
- **Runtime**'da bu aspect'ler **AspectJ** tarafından yaratılırlar

```
<beans...>

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

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

AspectJ Aspect'lerinin Spring İçerisinde Kullanımı

- AspectJ aspect'leri çalışma zamanında **LTW yöntemi** ile devreye girerler
- LTW'nin gerçekleşmesi için ilgili aspect'lerin **/META-INF/aop.xml** dosyasında tanımlı olması gerekir

```
<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

AspectJ Aspect'lerinin Spring İçerisinde Kullanımı

- LTW’nin gerçekleşmesi için ayrıca JVM’in -**javaagent**:/path/to/aspectjweaver.jar parametresi ile çalıştırılması gerekir

Sıradan Java Nesnelerine Dependency Injection

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

Sıradan Java Nesnelerine Dependency Injection


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

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

Sıradan Java Nesnelerine Dependency Injection

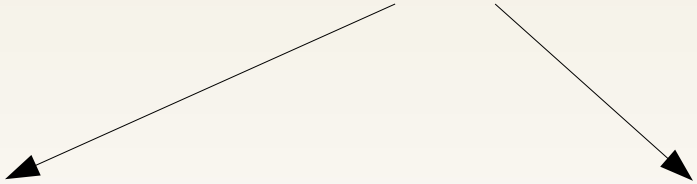
- **@Configurable** anotasyonunun aktive olması için `<context:spring-configured/>` elemanı XML'de tanımlanmalıdır
- Java tabanlı konfigürasyonda **@EnableSpringConfigured** kullanılmalıdır
- DI normalde **constructor'lardan sonra** gerçekleşir
- **@Configurable(preConstruction=true)** ile constructor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

Sıradan Java Nesnelerine Dependency Injection

- Sıradan Java nesnelerine DI işleminin gerçekleşebilmesi için ilgili Java sınıflarının **AspectJ aspectleri** ile **weave edilmesi** gerekir
- LTW işlemi JVM **sınıfları yüklerken** gerçekleştirilir
- LTW'nin gerçekleşebilmesi için Spring Container içerisinde `<context:load-time-weaver/>` elemanı tanımlı olmalıdır

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