

Tasarım Örüntüleri ile Spring Eğitimi 3

Annotasyon Tabanlı Spring Container Konfigürasyonu

Annotasyon Tabanlı Container Konfigürasyonu

- Spring Container'da **bean tanımları** ve **bağımlılıkların belirtilmesi** işlemleri **Java anotasyonları** kullanılarak da yapılabilir
- Anotasyon tabanlı konfigürasyon **iki kısımda** incelenebilir
 - Bean tanımları XML'de yapılmaya devam ederken, bağımlılıkların enjekte edilmesi anotasyonlar ile gerçekleştirilebilir
 - Ya da hem bean tanımları, hem de bağımlılıkların enjeksiyonu tamamen anotasyonlar ile gerçekleştirilir

Annotasyon Tabanlı Container Konfigürasyonu

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

<context:annotation-config/>

</beans>

→
@PostConstruct, @PreDestroy, @Required, @Autowired gibi
annotasyonları devreye sokar

**Bu aşamada henüz bean tanımları XML konfigürasyon
dosyalarında yapılmaya devam etmektedir!**

@Autowired

- Bean tanımı yine **XML tarafında** yapılır
- **@Autowired** anotasyonu ile **sınıf içerisinde hangi property'lere** bağımlılık enjeksiyonu yapılacağı belirtilir
- Spring Container mevcut bean'lardan uygun olanlarını **@Autowired anotasyonu ile işaretli property'lere** enjekte eder
- XML bean tanımında **bağımlılık tanımlarını yapmaya gerek yoktur**

@Autowired

```
public class Foo {
```

```
    private Bar bar;
```

ApplicationContext'de tanımlı Bar tipin'deki bean'ı enjekte eder

```
    @Autowired
```

```
    public void setBar(Bar bar) {
```

```
        this.bar = bar;
```

```
    }
```

```
    // ...
```

```
}
```

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

```
    <bean id="bar" class="x.y.Bar"/>
```

```
    <bean id="foo" class="x.y.Foo"/>
```

```
</beans>
```

@Autowired

```
public class Foo {  
  
    @Autowired  
    private Bar bar;  
  
    private Baz baz;  
  
    @Autowired  
    public Foo(Baz baz) {  
        this.baz = baz;  
    }  
  
    // ...  
}
```

Field, setter ve constructor'a uygulanabilir

Default olarak **byType** modunda çalışır

Birden fazla aynı tipte bean olması ve bu bean'lardan herhangi birisinin ismi property ile eşleşmediği durumda hata verir

Qualifier ile Aday Bean'ların Sınırlandırılması

```
public class Foo {
```

```
    @Autowired  
    @Qualifier("myBar")  
    private Bar bar;
```

```
    // ...
```

```
}
```

```
<bean id="foo" class="x.y.Foo"/>
```


```
<bean id="bar1" class="x.y.Bar">  
</bean>
```

```
<bean id="bar2" class="x.y.Bar">  
    <qualifier value="myBar"/>  
</bean>
```

Eğer tanımlanmaz ise **default qualifier** değeri olarak bean ismi kabul edilir

@Autowired

```
public class Foo {  
  
    private Bar bar;  
  
    @Autowired(required=false)  
    public void setBar(Bar bar) {  
        this.bar = bar;  
    }  
  
    // ...  
}
```



Default **required** attribute değeri **true**'dür
required=true durumunda Spring Container
belirtilen tipte bean bulamadığında
hata verir. Property'nin NULL kalması için
required=false olarak belirtilmelidir.

ApplicationContext'deki Belirli Tipteki Bütün Bean'ları Enjekte Etmek

```
public class Foo {
```

```
    @Autowired  
    private Bar[] bars;
```

```
    // ...
```

```
}
```

→ Bu sayede container'da tanımlı **belirli bir tipteki bütün beanları** bir array'e autowire etmek de mümkündür

Eğer ApplicationContext'de Bar tipinde hiç bean yoksa hata verir


ApplicationContext'deki Belirli Tipteki Bütün Bean'ları Enjekte Etmek

```
public class Foo {  
  
    private Set<Bar> bars;  
  
    @Autowired  
    public void setBars(Set<Bar> bars) {  
        this.bars = bars;  
    }  
  
    // ...  
}
```

Enjekte edilecek bean'lerin tipini java generics'den tespit edebilir

ApplicationContext'i Enjekte Etmek

```
public class Foo {  
  
    @Autowired  
    private ApplicationContext context;  
  
    // ...  
}
```



ApplicationContext kendisini bu bean'a enjekte eder. **ApplicationContextAware** arayüzünü implement etme gereksinimini ortadan kaldırır.

@Autowired ve XML

- Bağımlılık tanımları **hem XML property elemanı ile hem de @Autowired ile** belirtilmiş olabilir
- Ancak **annotation injection XML injection'dan önce** yapılır
- Bu sayede **XML injection** annotation injection ile yapılanları **override edebilir**

Annotasyon Tabanlı Bean Tanımları

- Bean tanımları **sınıf düzeyinde annotasyon kullanarak** da yapılabilir
- Spring Container classpath'i tarayarak belirli anotasyonlarla işaretlenmiş **Java sınıflarını tespit eder** ve bu sınıflardan **birer bean yaratır**
- Tarama ve bean oluşturma işlemi için **<context:component-scan>** elemanı kullanılır

Component Scan İşlemi

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

```
    <context:component-scan  
        base-package ="com.javaegitimleri"/>
```

```
</beans>
```



<context:annotation-config/> elemanını da otomatik olarak aktive eder. Dolayısı ile bu elemanı tanımlaya artık gerek yoktur.

Component Scan İşlemi

- Default olarak aşağıdaki **built-in anotasyonlar** scan edilir
 - @Component
 - @Repository
 - @Service
 - @Controller ve @RestController
 - @ControllerAdvice
 - @Configuration

Annotasyon Tabanlı Bean Tanımlamaya Örnek

```
@Service("securityService")
public class SecurityServiceImpl implements
SecurityService {

    private SecurityDao securityDao;

    @Autowired
    public SecurityServiceImpl(SecurityDao securityDao) {
        this.securityDao = securityDao;
    }
}

@Repository("securityDao")
public class SecurityDaoImpl implements SecurityDao {
    // ...
}
```

@Component ve Diğer Stereo Tipler

- **@Service**
 - Spring için özel bir anlamı yoktur
 - Servis katmanındaki bean'lar için eklenmiştir
- **@Repository**
 - DAO beanlarını tanımlar
 - Exception'ların otomatik çevrimini de tetikler
- **@Controller**
 - MVC controller bean'ları tanımlanır

@Component ve Diğer Stereo Tipler

- **@ControllerAdvice**
 - MVC controller'lar için global error handling metotlarının yazıldığı bean'ları tanımlar
- **@Configuration**
 - Java tabanlı konfigürasyon bean tanımı yapar
 - Bu sınıfların içerisinde diğer bean'ları yaratan factory metotlar yer alır
- Hepsi **@Component**'ten türer

Component Scan

İşlemi: Include/Exclude

- İstenirse farklı anotasyonların kullanıldığı veya hiç **anotasyona sahip olmayan sınıflardan** da bean oluşturulması sağlanabilir
- Ya da default olarak taranan built-in anotasyonlar veya bazı spesifik sınıflar **göz ardı** ettirilebilir

Component Scan İşlemi: Include/Exclude

```
<beans...>

  <context:component-scan base-package="com.javaegitimleri">

    <context:include-filter type="annotation"
      expression="org.aspectj.lang.annotation.Aspect"/>

    <context:include-filter type="assignable"
      expression="com.javaegitimleri.petclinic.dao.BaseDao"/>

    <context:exclude-filter type="annotation"
      expression="org.springframework.stereotype.Controller"/>
  </context:component-scan>

</beans>
```

Component İçinde Bean Tanımları

- Component'ler business metotları dışında **bean factory metotları** da barındırabilir

@Component

```
public class FooFactory {
```

Metot düzeyinde tanımlama **@Bean** annotasyonu ile gerçekleştirilir

```
@Bean @Qualifier("myFoo")
```

```
public Foo foo() {  
    return new Foo();  
}
```

Metot ismi bean ismi olur,
Ayrıca Qualifier'da tanımlanabilir

```
public void doWork() {  
    // ...  
}
```

Component normal bir bean instance'ıdır ve bean yaratma dışında normal işlemlere de sahip olabilir

```
}
```

@Value

- Built-in Java tipli **property değerlerini** enjekte etmek için @Value anotasyonu kullanılır
- İçerisinde property **placeholder** da kullanılabilir

```
@Component
public class Foo {

    @Value("bar-value")
    private String bar;

    @Value("${foo.baz}")
    private String baz;

    // ...
}
```

@Primary

- Birden fazla **aday bean** olduğunda aralarından hangisine **öncelik** verileceğini belirler

@Primary

@Repository

```
public class SecurityDaoHibernateImpl implements SecurityDao {
    // ...
}
```

@Repository

```
public class SecurityDaoJdbcImpl implements SecurityDao {
    // ...
}
```

@Service

```
public class SecurityServiceImpl implements SecurityService {
    @Autowired
    private SecurityDao securityDao;
    // ...
}
```


@Scope

@Scope anotasyonu bean tanımında metot veya sınıf düzeyinde kullanılabilir

```
@Scope("prototype")
@Component
public class CommandImpl implements Command {
    // ...
}
```

```
@Component
public class CommandFactory {

    @Bean @Scope("prototype")
    public Command createCommand() {
        return new CommandImpl();
    }
}
```

@Lazy

```
@Component
@Lazy
public class FooFactory {

    @Bean @Lazy
    private Foo foo() {
        return new Foo();
    }
}
```

Sınıf veya metot düzeyinde **@Lazy** anotasyonu kullanılarak bean'lerin sadece gerektiği anda yaratılmaları sağlanabilir

Java Tabanlı Spring Container Konfigürasyonu

Java Tabanlı Container Konfigürasyonu

- Spring **bean tanımlarının** Java sınıflarında yapılmasını sağlar
- **Bire bir** XML tabanlı konfigürasyona **karşılık** gelmektedir
- Avantajı “**type safety**” dir
- Konfigürasyon metadata'nın yazıldığı Java sınıfları **@Configuration** anotasyonu ile işaretlenmelidir

XML vs Java Konfigürasyonları

appContextConfig.xml

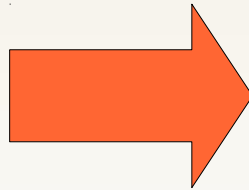
```
<beans...>
```

```
<bean id="foo" class="x.y.Foo">
  <property name="bar">
    <ref bean="bar"/>
  </property>
  <property name="baz"
ref="baz"/>
</bean>
```

```
<bean id="bar"
class="x.y.Bar"/>
```

```
<bean id="baz"
class="x.y.Baz"/>
```

```
</beans>
```



@Configuration

```
public class AppContextConfig {
```

@Bean

```
public Foo foo() {
    Foo foo = new Foo();
    foo.setBar(bar());
    foo.setBaz(baz());
    return foo;
}
```

@Bean

```
public Bar bar() {
    return new Bar();
}
```

@Bean

```
public Baz baz() {
    return new Baz();
}
```

```
}
```

Bağımlılıkların Enjekte Edilmesi

```
@Configuration
public class AConfig {

    @Autowired
    private Bar bar;

    @Bean
    public Foo foo() {
        Foo foo = new Foo();
        foo.setBar(bar);
        return foo;
    }
}
```

```
@Configuration
public class BConfig {

    @Bean
    public Bar bar() {
        return new Bar();
    }
}
```

Bağımlılıkların Enjekte Edilmesi

```
@Configuration
public class AConfig {

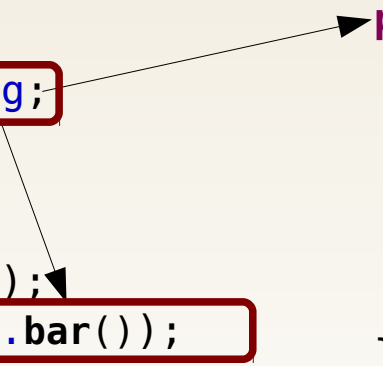
    @Autowired
    private BConfig bConfig;

    @Bean
    public Foo foo() {
        Foo foo = new Foo();
        foo.setBar(bConfig.bar());
        return foo;
    }
}
```

→

```
@Configuration
public class BConfig {

    @Bean
    public Bar bar() {
        return new Bar();
    }
}
```



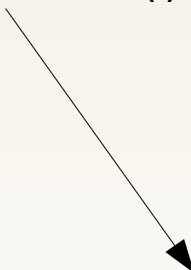
Bağımlılıkların Enjekte Edilmesi

```
@Configuration
public class AConfig {

    @Bean
    public Foo foo() {
        return new Foo();
    }
}
```

```
@Configuration
public class BConfig {

    @Bean
    public Bar bar() {
        return new Bar();
    }
}
```



```
public class Foo {
    @Autowired
    private Bar bar;

    //...
}
```


Bağımlılıkların Enjekte Edilmesi

```
@Configuration
public class AConfig {

    @Bean
    public Foo foo(Bar bar) {
        Foo foo = new Foo();
        foo.setBar(bar);
        return foo;
    }
}
```

```
@Configuration
public class BConfig {

    @Bean
    public Bar bar() {
        return new Bar();
    }
}
```

@ImportResource ve @Import

@ImportResource herhangi bir XML
spring bean definition dosyasının
yüklenmesini sağlar

```
@Configuration
@ImportResource("classpath:/appcontext/beans-config.xml")
public class AConfig {

}
```

```
@Configuration
@Import(AConfig.class)
public class BConfig {

}
```

@Import herhangi bir başka configuration
sınıfının diğer bir configuration sınıfı
tarafından yüklenmesini sağlar

@ComponentScan

@Component ve türevi
anotasyonların scan edileceği
paketleri tanımlar

```
@Configuration
@ComponentScan("com.javaegitimleri.petclinic")
public class AConfig {

}
```

ApplicationContext Yaratılması

```
AnnotationConfigApplicationContext applicationContext =  
new AnnotationConfigApplicationContext();
```

```
applicationContext.register(AConfig.class,BConfig.class);
```

```
applicationContext.refresh();
```

```
Foo foo = applicationContext.getBean(Foo.class);
```

Java Tabanlı Konfigürasyon ve Bean Profile Kabiliyeti

Konfigürasyon sınıflarının hangi profillerde yükleneceği sınıf düzeyinde `@Profile` anotasyonu ile tanımlanır

```
@Configuration
@Profile("dev")
public class DevConfig {
    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("classpath:/schema.sql")
            .addScript("classpath:/data.sql")
            .build();
    }
}
```

```
@Configuration
@Profile("prod")
public class ProdConfig {
    @Bean
    public DataSource dataSource() throws
        Exception {
        Context ctx = new InitialContext();
        return (DataSource)
            ctx.lookup("java:comp/env/jdbc/DS");
    }
}
```

```
AnnotationConfigApplicationContext applicationContext = new
AnnotationConfigApplicationContext();
```

```
applicationContext.getEnvironment().setActiveProfiles("dev");
applicationContext.scan("com.javaegitimleri");
applicationContext.refresh();
```

ApplicationContext yaratılırken aktif profillerin hangileri olacağı belirtilir
Bu işlem -D jvm parametresi veya web.xml'de context param ile de yapılabilir

@PropertySource

@PropertySource anotasyonu ile belirtilen resource'lar Environment'a PropertySource olarak eklenirler

@Configuration

@PropertySource("classpath:/application.properties")

public class AppConfig {

@Autowired

Environment env;

@Bean

public Foo foo() {

Foo foo = **new** Foo();

foo.setName(env.getProperty("foo.name"));

return foo;

}

}

@PropertySource

targetPlatform değişkeni halihazırda bu aşamaya kadar Environment'e register olmuş PropertySource nesneleri arasından resolve edilmeye çalışılır.

```
@Configuration
@PropertySource("classpath:/application_${targetPlatform}.properties")
public class AppConfig {

    @Autowired
    private Environment env;

    @Bean
    public Foo foo() {
        Foo foo = new Foo();
        foo.setName(env.getProperty("foo.name"));
        return foo;
    }
}
```

Tasarım Örüntülerine Devam...

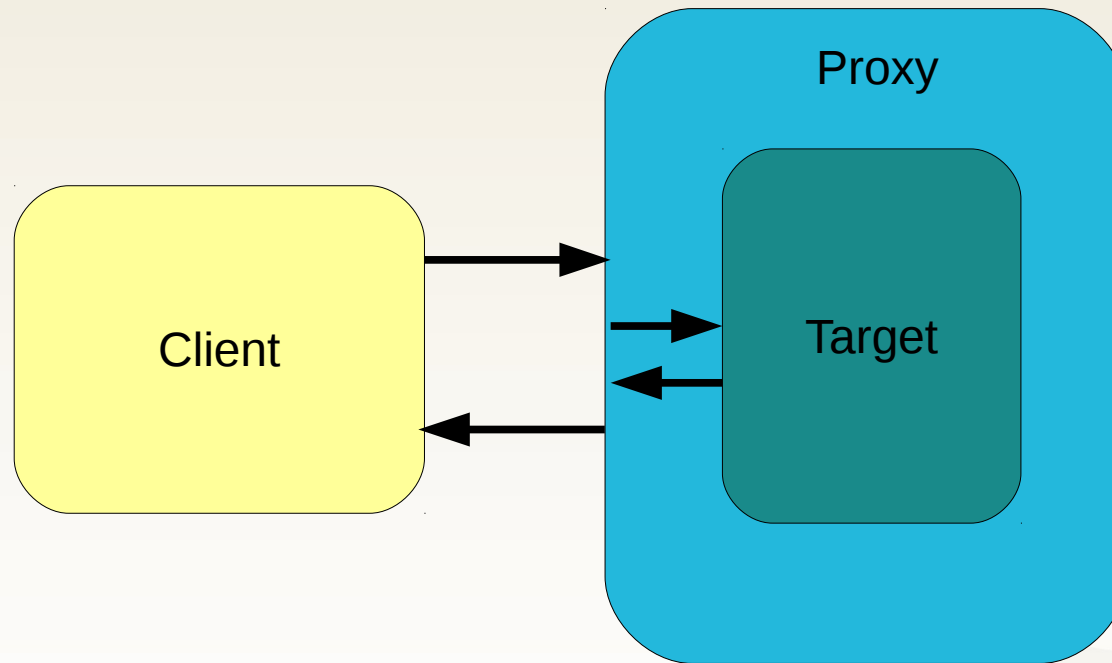
Proxy

- Bazı durumlarda **nesnelerin hemen yaratılması maliyetli** olabilir, yada o anda yaratılmaları **uygun olmayabilir**
- Ya da bazı nesnelere erişmeden evvel veya erişimden sonra **ilave bazı işlemlerin** yapılması gerekebilir
- Asıl nesnenin yaratılmasını ihtiyaç anına kadar erteleyen, asıl nesneden önce veya sonra devreye giren, **asıl nesne yerine kullanılabilen bir nesne** yaratılır
- Vekil nesne asıl nesneye **erişimi dolaylı** hale getirir

Proxy

Proxy, target nesne ile aynı tipte olup, client ile target nesnenin arasına girer

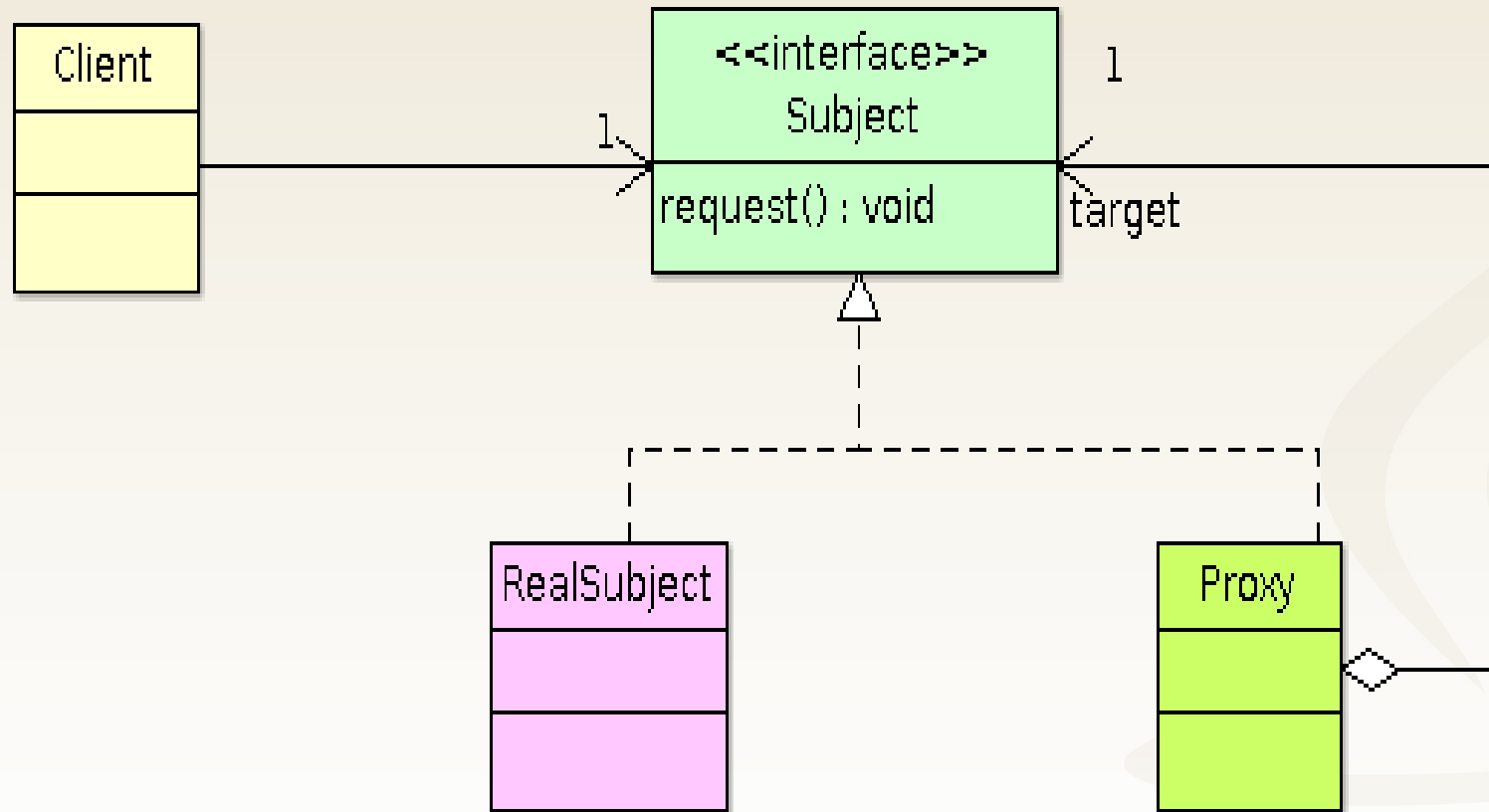
Client proxy nesne ile konuştuğunun farkında değildir



Client'ın target nesne üzerindeki metot çağrıları öncelikle proxy nesneye erişir

Proxy metot çağrısından önce veya sonra bir takım işlemler gerçekleştirebilir

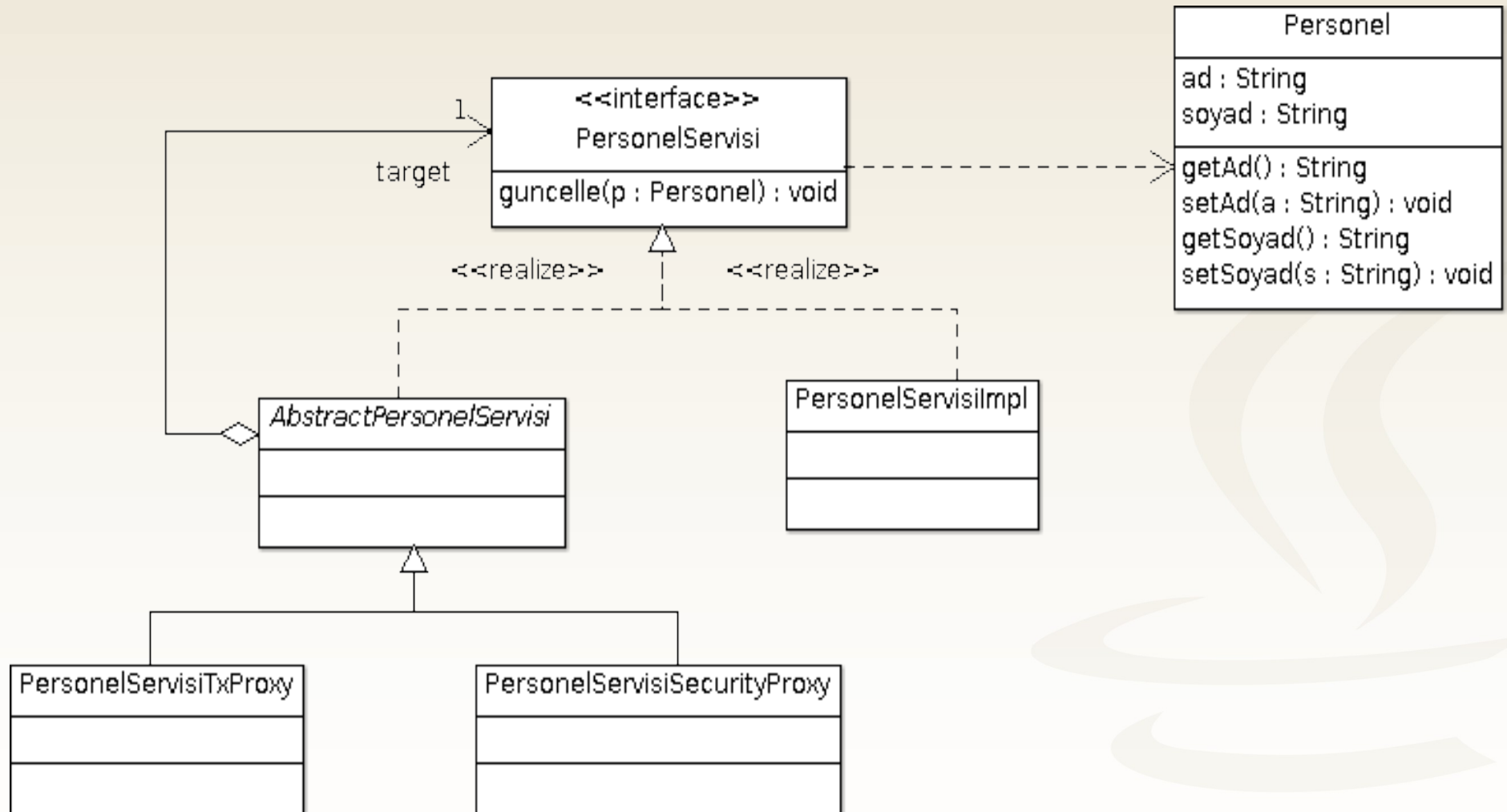
Proxy Sınıf Diagramı



Örnek Problem: Proxy

- Personel bilgilerini güncelleyen PersonelServisi isimli bir sınıf vardır
- Bu sınıf içerisinde personel güncellemesi yapılırken TX yönetiminin de yapılması istenmektedir
- Ayrıca personelin sadece kendi bilgilerini güncellemesi için de yetki kontrolü yapılmalıdır
- Transaction yönetimi ve yetkilendirme işlemlerinin istemci kodu tarafından bilinmesi istenmemektedir
- Bu davranışlar personel güncelleme davranışı üzerine sonradan konfigüratif ve uygulama geliştiricilerin isteğine bağlı biçimde eklenebilmelidir

Örnek Problem: Proxy



PersonelServisiTxProxy

```
public class PersonelServisiTransactionProxy
    extends AbstractPersonelServisi {

    public PersonelServisiTransactionProxy(PersonelServisi target) {
        super(target);
    }

    @Override
    public void guncelle(Personel personel) {
        try {
            System.out.println("begin transaction here");

            target.guncelle(personel);

            System.out.println("commit transaction");
        } catch (Exception ex) {
            System.out.println("rollback transaction");
            throw ex;
        }
    }
}
```

PersonelServisiSecProxy

```
public class PersonelServisiSecurityProxy
    extends AbstractPersonelServisi {

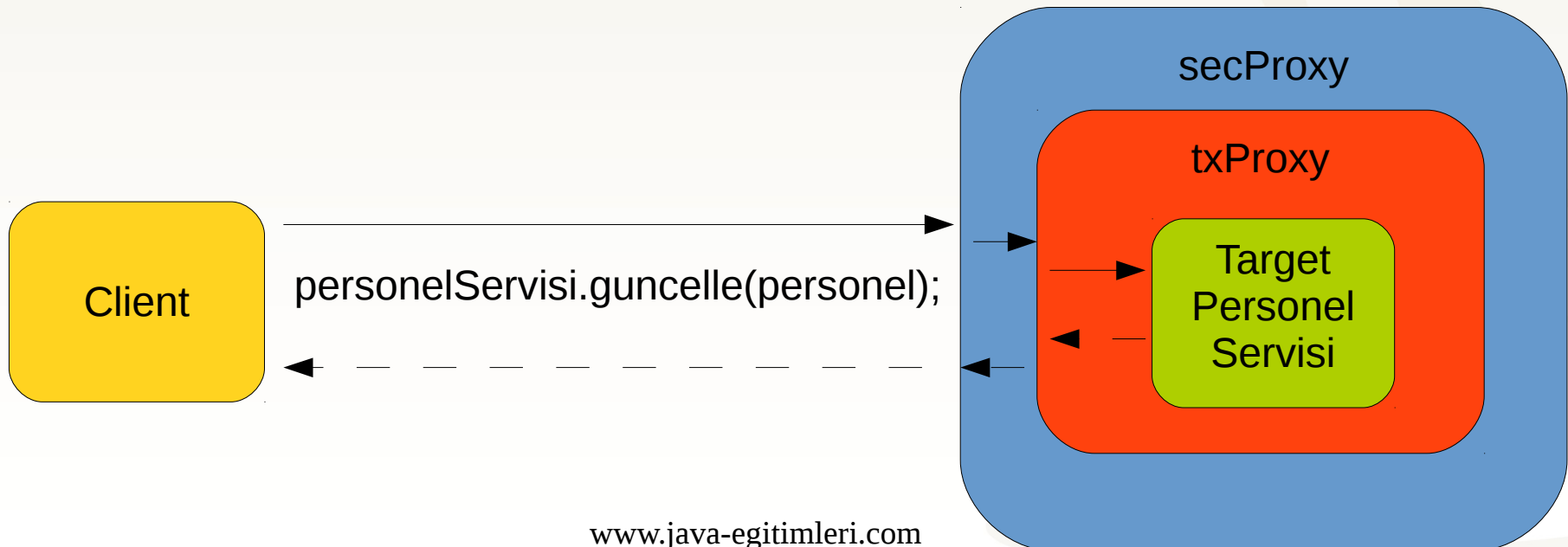
    public PersonelServisiSecurityProxy(PersonelServisi target) {
        super(target);
    }

    @Override
    public void guncelle(Personel personel) {
        System.out.println("perform security check, "
            + "then allow target method execution");

        target.guncelle(personel);
    }
}
```

Client'in Proxy Nesneler ile Etkileşimi

```
public class Client {  
    public static void main(String[] args) {  
  
        PersonelServisi ps = new PersonelServisiSecurityProxy(  
                                new PersonelServisiTransactionProxy(  
                                    new PersonelServisiImpl()));  
  
        Personel personel = new Personel();  
        ps.guncelle(personel);  
    }  
}
```



Dinamik Proxy Oluşturma Yöntemleri

- Spring, Hibernate gibi framework'ler proxy nesneler oluşturmak için **dinamik proxy sınıfları** üretirler
- Dinamik proxy sınıfı üretmek için **iki yol** vardır
 - Interface proxy
 - Class proxy

Dinamik Proxy Oluşturma Yöntemleri

- **Interface Proxy**

- Proxy sınıf üretmek için hedef nesnenin sahip olduğu arayüzlerden birisi kullanılır
- JDK proxy olarak da bilinir, JDK API'sinde mevcuttur

- **Class Proxy**

- Proxy sınıf hedef nesnenin ait olduğu sınıf extend edilerek yaratılır
- CGLIB proxy olarak da bilinir, CGLIB, Javassist gibi kütüphaneler kullanılarak gerçekleştirilir

Java ve Proxy

- JDK API'sinde arayüz tabanlı **dinamik proxy sınıf üretme kabiliyeti** mevcuttur
- Hedef nesnenin sahip olduğu **arayüz veya arayüzler** dinamik olarak üretilen bir **proxy sınıf tarafından implement** edilir
- Proxy sınıf çalışma zamanında **JDK Proxy API'si** tarafından üretilir
- Bu proxy sınıftan bir nesne de yine JDK Proxy API'Si üzerinden elde edilir

Java ve Proxy

java.lang.reflect paketindeki InvocationHandler arayüzü implement edilerek, proxy nesne içerisinde yürütülecek olan işlem invoke metodu içerisinde kodlanır



```
public class TxInvocationHandler implements InvocationHandler {  
  
    private Object target;  
  
    public TxInvocationHandler(Object target) {  
        this.target = target;  
    }  
  
    @Override  
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
        try {  
            System.out.println("tx begin");  
            Object result = method.invoke(target, args);  
            System.out.println("tx commit");  
            return result;  
        } catch (Exception ex) {  
            System.out.println("tx rollback");  
            throw new RuntimeException(ex);  
        }  
    }  
}
```

Java ve Proxy

```
public class SecInvocationHandler implements InvocationHandler {  
  
    private Object target;  
  
    public SecInvocationHandler(Object target) {  
        this.target = target;  
    }  
  
    @Override  
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
        System.out.println("security check");  
        Object result = method.invoke(target, args);  
        return result;  
    }  
}
```

Java ve Proxy

```
Class<?>[] interfaces = new Class[] { PersonelServisi.class };
```

```
ClassLoader classLoader = PersonelServisi.class.getClassLoader();
```

Proxy sınıfın implement edeceği arayüz veya arayüzler, proxy sınıfı yükleyecek ClassLoader belirlenir. Bu ClassLoader genellikle arayüzleri yükleyen sınıf olabilir.

```
PersonelServisi target = new PersonelServisiImpl();
```

Asıl işin delege edileceği hedef nesne yaratılır

```
InvocationHandler txInvocationHandler = new TxInvocationHandler(target);
```

```
Object txProxy = Proxy.newProxyInstance(classLoader, interfaces, txInvocationHandler);
```

Belirlenen arayüz(ler), classLoader ve proxy nesnenin içerisinde yürütülecek olan kod (invocationHandler) ile java.lang.reflect paketindeki Proxy sınıfı kullanılarak Proxy instance yaratılır. Bu işlem sırasında proxy sınıf da dinamik olarak yaratılmaktadır.

Java ve Proxy

```
InvocationHandler secInvocationHandler = new SecInvocationHandler(txProxy);  
  
Object secProxy = Proxy.newProxyInstance(classLoader, interfaces, secInvocationHandler);
```

İstenilen derinlikte proxy zinciri oluşturulabilir.

```
PersonelServisi personelServisi = (PersonelServisi) secProxy;  
personelServisi.guncelle(new Personel());
```

Zincirde oluşturulan en son proxy nesne hangi arayüz ile çalışılacak ise ona downcast edilerek kullanılabilir.

- İlave kabiliyetlerin veya her durumda işletilmesi uygun olmayan **davranışların tek bir sınıf içerisinde birikmesinin önüne geçilir**
- İlave davranışlar farklı tipte nesnelerle beraber de kullanılabilir
- Böylece bu **davranışların yeniden kullanılabilirliği** mümkün hale gelir

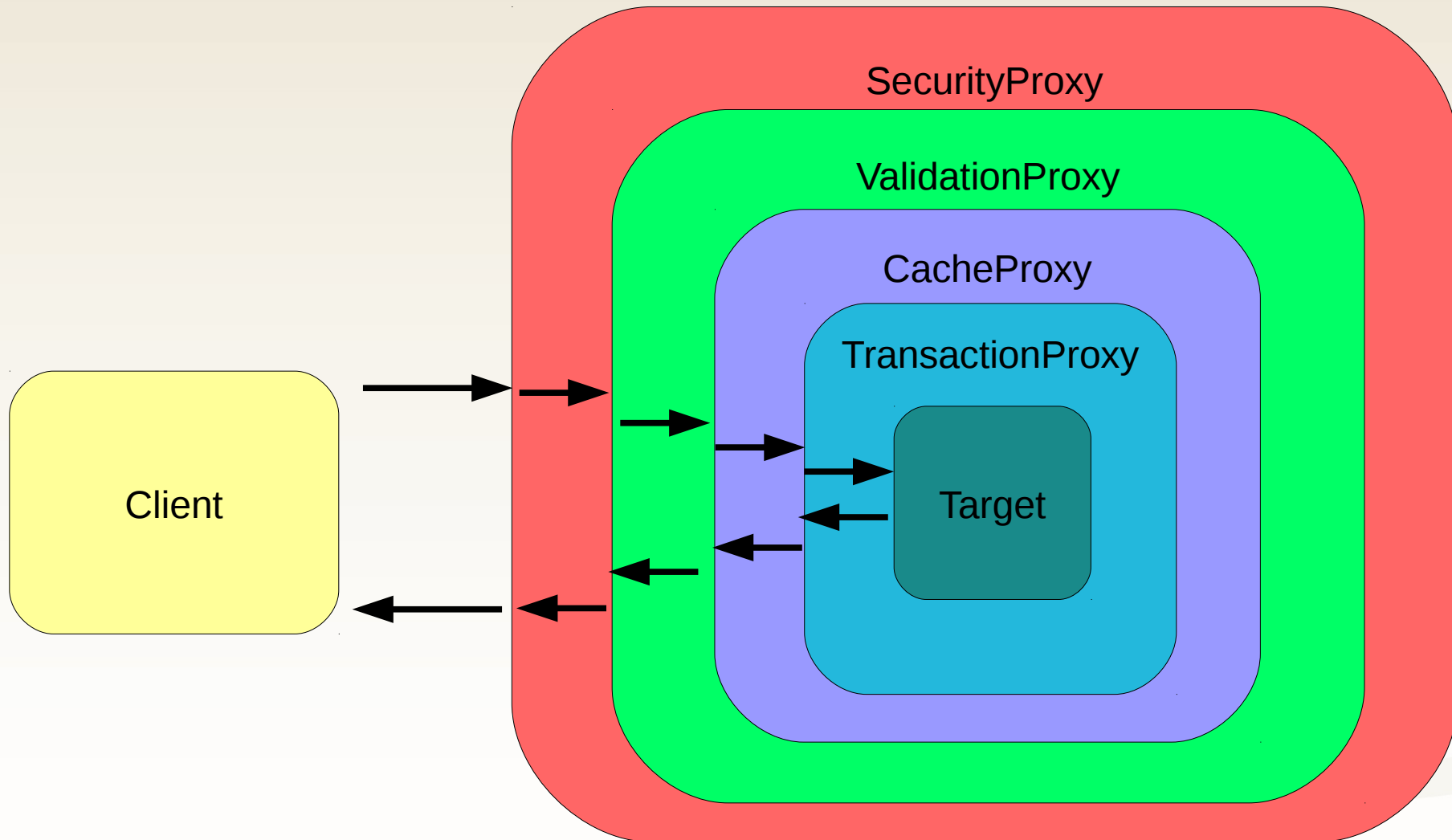
Spring İçerisinde Proxy Örüntüsünün Kullanımı

- Spring Application Framework'ün **pek çok kabiliyeti** proxy örüntüsü üzerine kuruludur
 - Transaction yönetimi
 - Bean scope kabiliyeti (request ve session scope bean'ler)
 - Aspect oriented programlama altyapısı (Spring AOP)
 - Metot düzeyinde validasyon ve caching
 - Remoting
 - Spring security'de metot düzeyinde yetkilendirme

Spring İçerisinde Proxy Örüntüsünün Kullanımı

- Spring, bu kabiliyetleri hayata geçirmek için genellikle uygulama geliştiricilerden habersiz **otomatik olarak proxy oluşturma** işini gerçekleştirir
- Diğer bean'lere de **bağımlılık olarak proxy nesne enjekte** edilir
- Diğer bean'ler proxy ile çalıştıklarının farkında **değillerdir**

Spring ve İç İçe Proxy Nesne Zinciri



Request ve Session Scope Bean'lerde Proxy Oluşturma

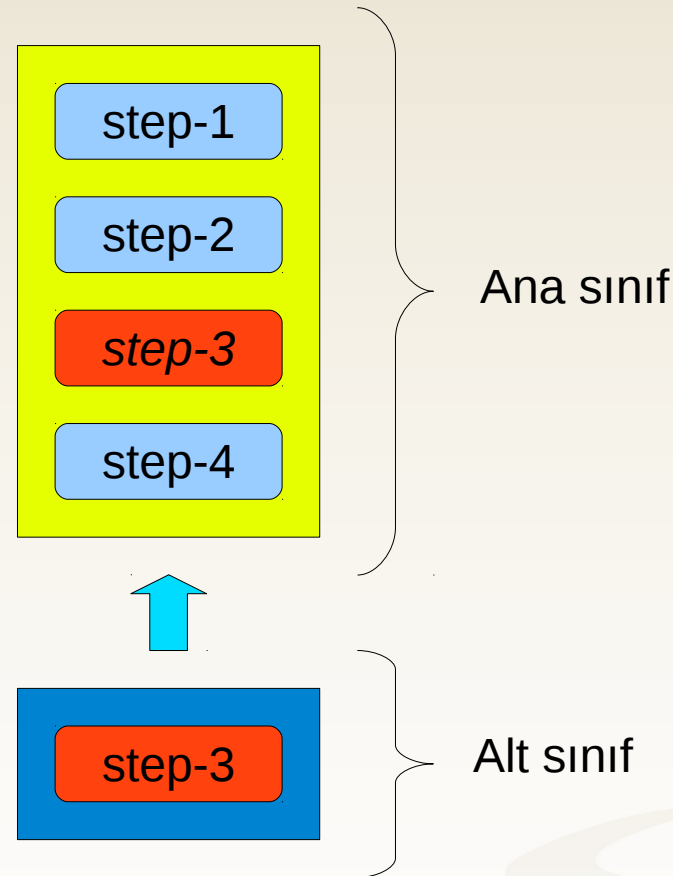
Request ve session scope bean'ları tanımlarken proxy modunun belirtilmesi gerekir. **Default proxy modu NO'dur.**

```
@Component
@Scope(value="session", proxyMode=ScopedProxyMode.INTERFACES)
public class UserPreferencesImpl implements UserPreferences {
    // ...
}
```

```
<context:component-scan
    base-package="com.javaegitimleri"
    scoped-proxy="targetClass"/>
```

Component scan sürecinde bu davranış bütün scoped bean'ler için geçerli olacak şekilde de değiştirilebilir

Template Method

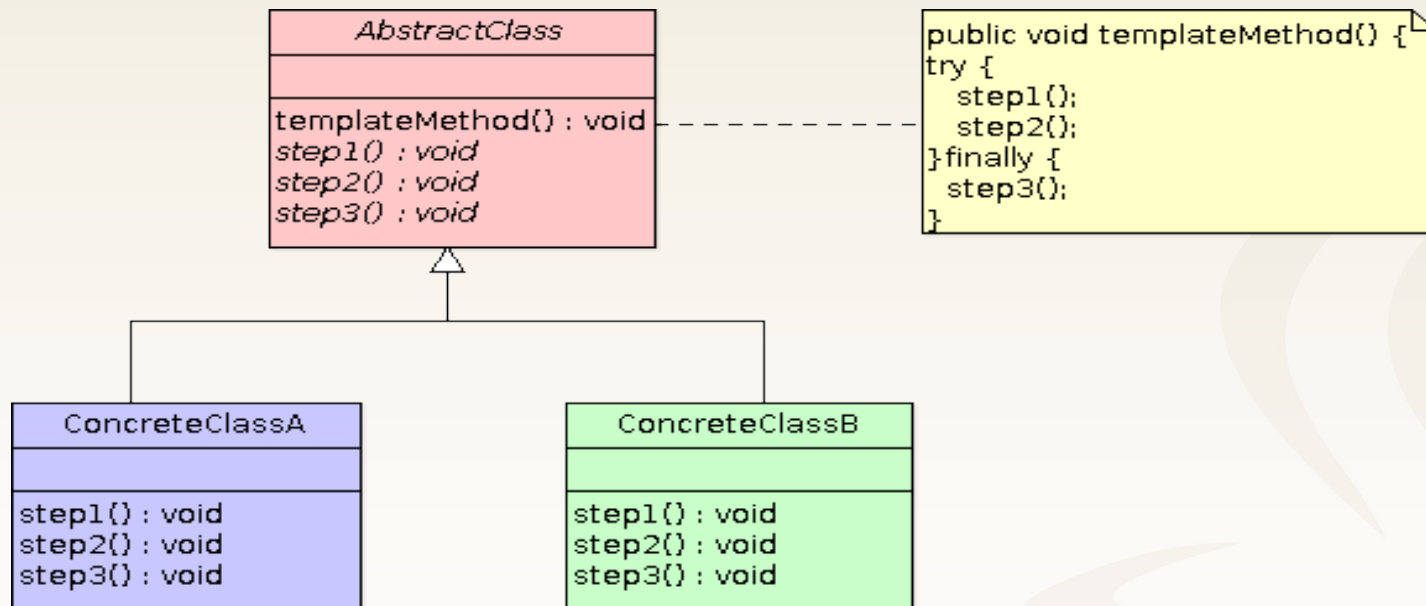


Algoritmanın **ana akışı** **encapsule** edilerek **spesifik adımların işleyişinin** alt sınıflar tarafından belirlenmesine imkan tanır

Template Method

- Bir algoritma **genel hatları** ile birden fazla sınıf için aynı olabilir
- Genelde her bir sınıf bu algorithmadaki **adımlardan bazılarının** farklı biçimde işletilmesini istemektedir
- Algoritmanın ana akışı değişmez biçimde tanımlanarak **sabitlenir** ve **adımları belli bir sıra ile işlettiğinden** emin olunur
- Değişen adımlar ise **alt sınıflar tarafından implement** edilir

Template Method Sınıf Diagramı



Örnek Problem: Template Method

- Java'da veritabanı işlemleri için JDBC API kullanılır. Veritabanında işletilecek SQL ifadeleri için her seferinde yapılması gereken rutin işlemler vardır.

Bunlar;

veritabanı bağlantısı kurulması

statement nesnesinin oluşturulması

SQL ifadesinin yazılması

SQL ifadesinin çalıştırılması

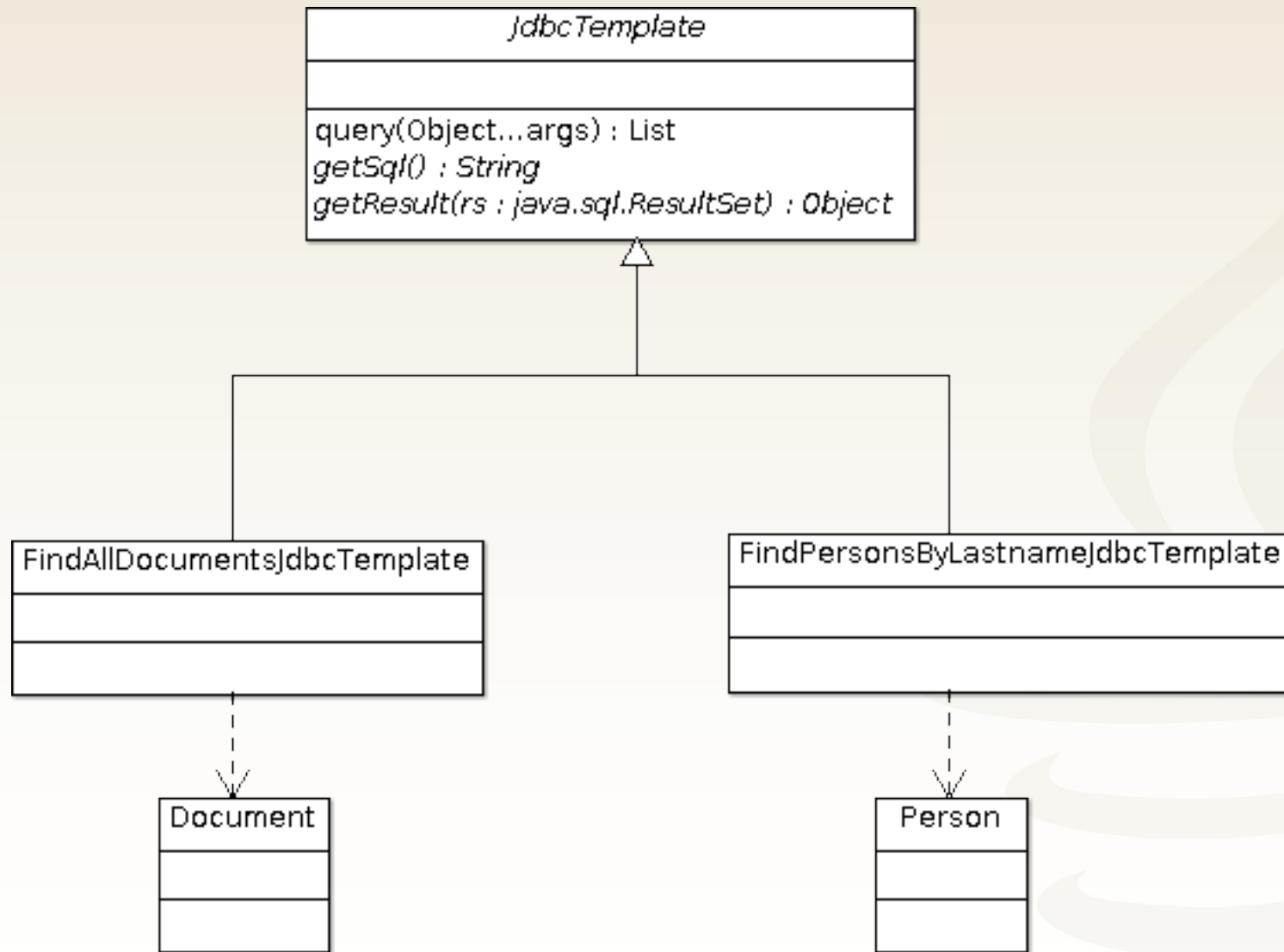
dönen ResultSet üzerinde iterate edilerek işlem yapılması

işi biten statement ve veritabanı bağlantı nesnelerinin kapatılması

Örnek Problem: Template Method

- Bu işlemlerden sadece **SQL ifadesinin yazılması** ve dönen **ResultSet üzerinde iterate edilerek işlem yapılması** probleme göre farklılık gösterir. Diğer adımlar her seferinde aynıdır
- Ana iskelet kodu sabitleyen ve her seferinde aynı sıra ile işleten, değişen iki adımı ise alt sınıfların tanımlamasına izin veren bir **JdbcTemplate** sınıfı yazılması istenmektedir

Örnek Problem: Template Method



Template Method Örüntüsünün Sonuçları

- Algoritmanın **ana işleyişi sabitlenmiş ve kontrol altına alınmış** olur
- Böylece alt sınıfların ata sınıfın davranışını **uygun olmayan biçimde override** etmelerinin önüne geçilmiş olunur

İletişim

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



harezmi
bilişim çözümleri

JAVA
Eğitimleri 