JAVA Eğitimleri

Tasarım Örüntüleri ile Spring Eğitimi 1

harezmi bilişim çözümleri

Tasarım Örüntüleri Nedir?

JAVA Eğitimleri

- Farklı ortamlarda/bağlamlarda karşımıza çıkan **benzer problemleri çözmek için uygulanan çözüm şablonları**dır
- Bu şablonlardaki **çözümün özü hep aynı**dır, ancak **şablonun uygulanışı** yapısal ve davranışsal olarak **her problem için farklılıklar** arz edebilir
- Bu şablonlar, zaman içerisinde benzer problemlere tekrar tekrar uygulanarak **evrilmiş ve olgunlaşmış, çalışırlığı sınanmış çözümler**dir

Tasarım Örüntülerinin Çıkışı

- Tasarım örüntülerinin çıkışı **mimari** ve **antropolojiye** dayanır
- Christopher Alexander isimli mimar "**kalite nesnel veya subjektif bir olgu mudur?**" sorusuna cevap aramıştır
- Alexander'ın kanaati kaliteli/yaşanabilir mimarisel sistemleri teşhis etmek için **belirli nesnel kriterler söz konusu**dur
- Benzer problemleri çözmek için inşa edilmiş yapılarda **kaliteyi belirleyen ortak nesnel özellikler** vardır

Tasarım Örüntülerinin Çıkışı

JAVA Eğitimleri

- Alexander, bu **ortak nesnel özelliklere** de **örüntü** (pattern) adını vermiştir
- Her bir örüntü öncelikle etrafımızda **sürekli olarak ortaya çıkan bir problem**i tanımlar
- Daha sonra da bu problem için **uygulanabilecek çözümün özü**nü ortaya koyar
- Öyle ki bu **çözüm her seferinde farklı bir hal alabilir**

GOF ve Tasarım Örüntüleri

- Yazılım sistemlerini de **örüntüler etrafında tasarlamak** mümkün müdür?
- Bu konu üzerinde pek çok değişik çalışma yapılmıştır
- Ancak en çok ses getiren “**Design Patterns: Elements of Reusable Object-Oriented Software**” isimli kitap olmuştur

GOF ve Tasarım Örüntüleri

JAVA Eğitimleri

- Bu kitap örüntüler için **katalog** oluşturup, bunları **tasvir** etmeye çalışmıştır
- Kitapta **23 adet örüntü** katalog içerisine alınmıştır
- Yazarların bu örüntüleri **sıfırdan üretmedikleri**ni bilmek önemlidir
- Yaptıkları iş **değişik yazılım sistemlerini inceleyerek** bu örüntüleri tespit etmektir

Örüntülerin Tasviri

- Örüntü tanımı **4 ana başlığa** sahiptir
  - Örüntünün ismi
  - Örüntünün amacı ve çözdüğü problem
  - Bu çözümün nasıl uygulanabileceği
  - Çözüm ile ortaya çıkan sonuçlar
- Hemen her **tasarım problemi** için bir takım **örüntüler mevcut**tur
- Örüntüler bir araya getirilerek **daha karmaşık problemleri çözmek** için de kullanılabilirler

Tasarım Örüntülerinin Faydaları

- Ekip içinde ve yazılım geliştiriciler arasında **ortak bir terminoloji** oluşmasını sağlar, ortak **bir bakış açısı** getirir
- Hazır çözümler probleme sıfırdan başlamayı, ve **olası hatalara düşmeyi önler**
- Çözümlerin **yeniden kullanılması**nı sağlar
- Diğer yazılım geliştiricilerin **deneyimlerinden faydalanma**yı sağlar

Tasarım Örüntülerinin Faydaları

JAVA Eğitimleri

- Bu örüntüler zaman içerisinde **evrilmiş ve olgunlaşmış** çözümlerdir
- Bu nedenle üzerlerinde **değişiklik yapmak daha kolay ve hızlı**dır
- Tasarım ve nesne yönelimli modelleme işlemlerine **bir üst perspektiften bakma**yı sağlar
- Bu sayede daha ilk aşamada **gereksiz detay ve ayrıntılar** içinde boğulmanın önüne geçilmesi mümkün olur

Tasarım Örüntülerindeki Temel Tasarım Prensipleri

- Değişen ne ise bul ve **encapsule** et
- Mümkün olduğunca **inheritance yerine composition**'ı tercih et
- Her zaman **soyut tipler**e (interface veya abstract class) depend et

Değişen Ne ise Encapsule Edilmeli

JAVA Eğitimleri

- Encapsulation sıklıkla **verinin gizlenmesi** olarak tanımlanır
- Ancak encapsulation **sadece data-hiding değil**dir
- Encapsulation, veri gizlemeden **daha fazlası**dır
- Örneğin, aynı zamanda **concrete sınıfların soyut sınıflar ile de gizlenmesi**dir
- Encapsulation genel olarak **değişen herhangi tür birşeyin sabit/değişmeyen bir şey arkasında gizlenmesi**dir

Değişen Ne ise Encapsule Edilmeli

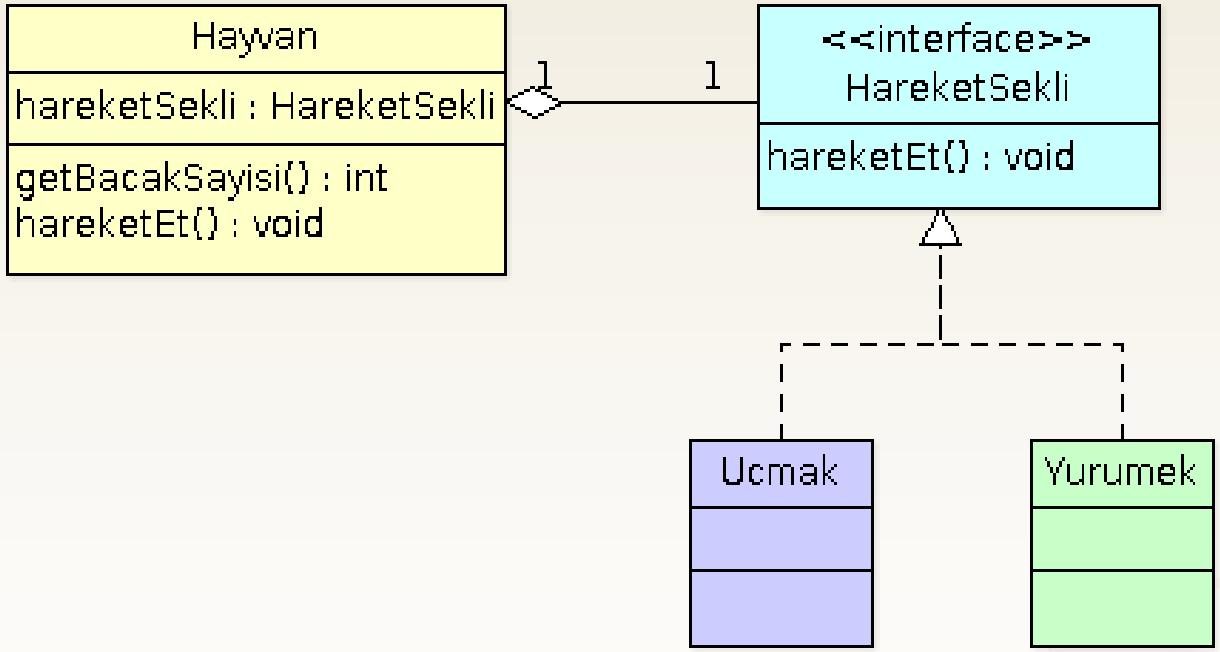
JAVA Eğitimleri

- Tasarım içerisinde hangi noktaların sonradan **değişikliğe gitmeden değişebilir/farklılaşabilir olması** gerektiği tespit edilmelidir
- Burada **değişen olgu veya konsept** üzerine odaklanılmalıdır
- Bu olgu veya konsept **değişmeyen/sabit başka bir yapı** arkasında **encapsule** edilmelidir
- **Değişmeyen bu yapı** bir metot, arayüz veya sınıf olabilir

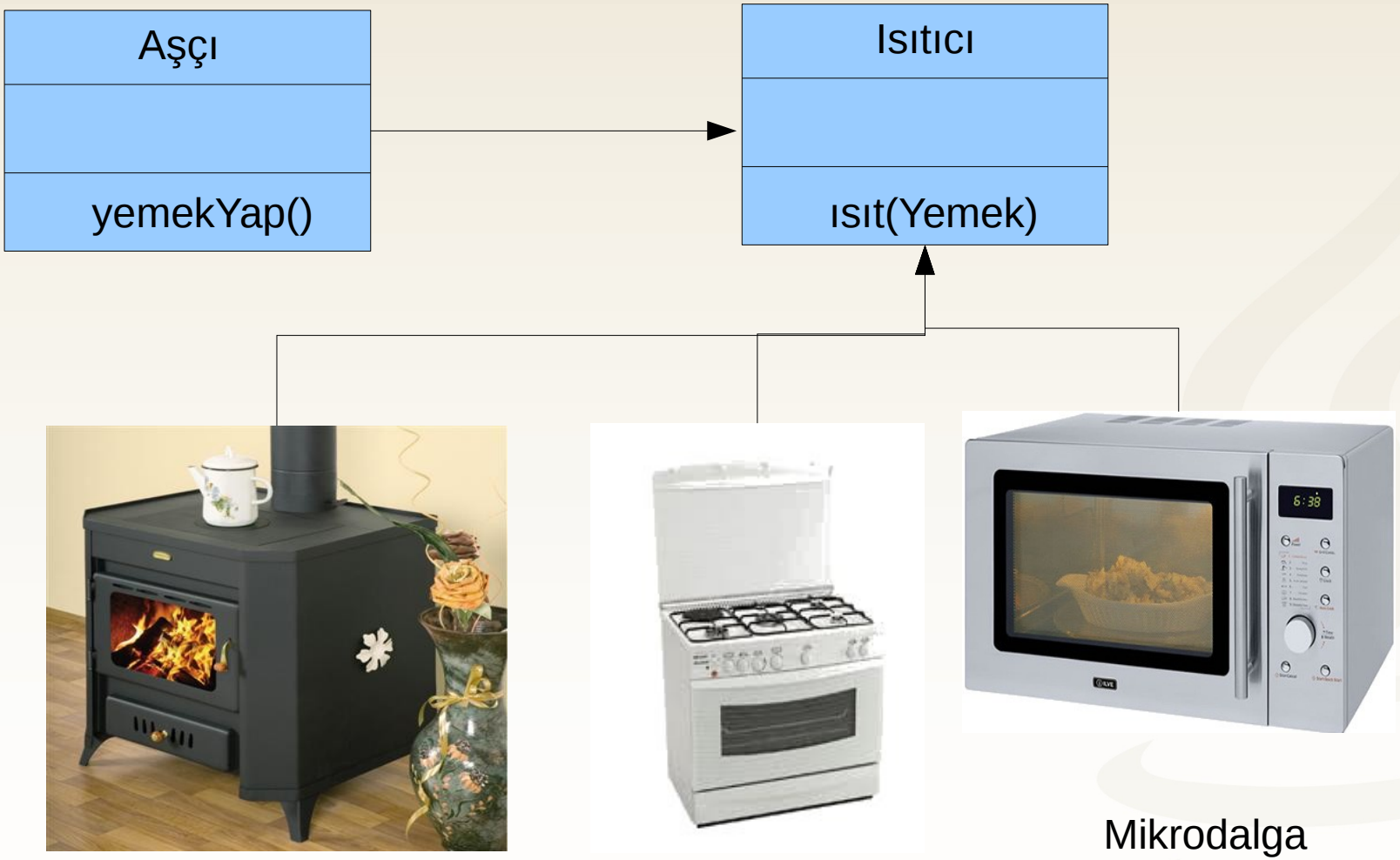
Değişen Ne ise Encapsule Edilmeli

- "Hayvanlar aleminde farklı hayvanların farklı sayıda bacakları vardır. Her bir hayvan türünün kendine özgü bir hareket şekli olabilir."
- Herhangi bir hayvan türü değişik sayıda bacağa sahip olabilir --> **verinin değişmesi**
- Herhangi bir hayvan türü değişik bir hareket tarzına sahip olabilir --> **davranışın değişmesi**

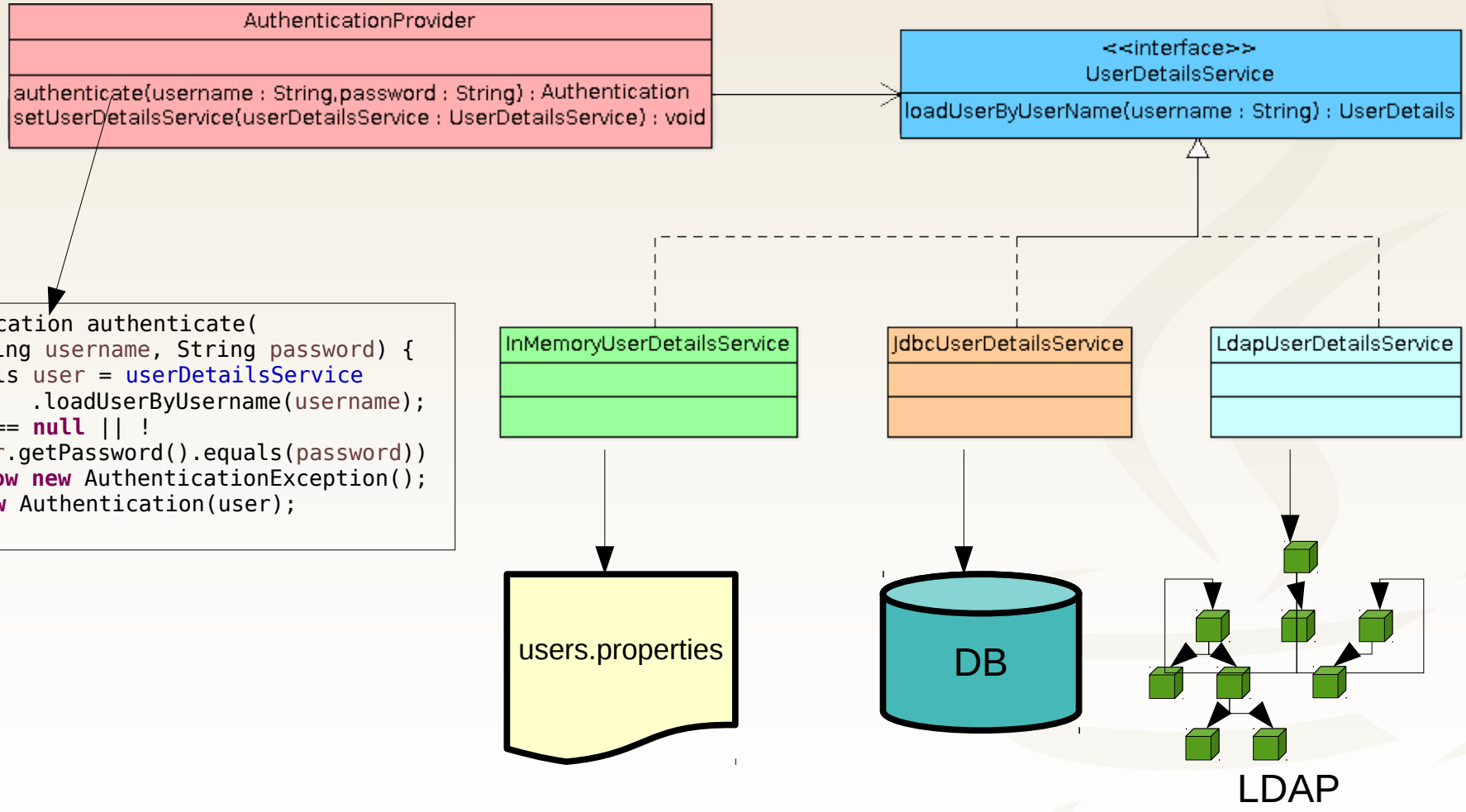
Değişen Ne ise Encapsule Edilmeli



Strategy

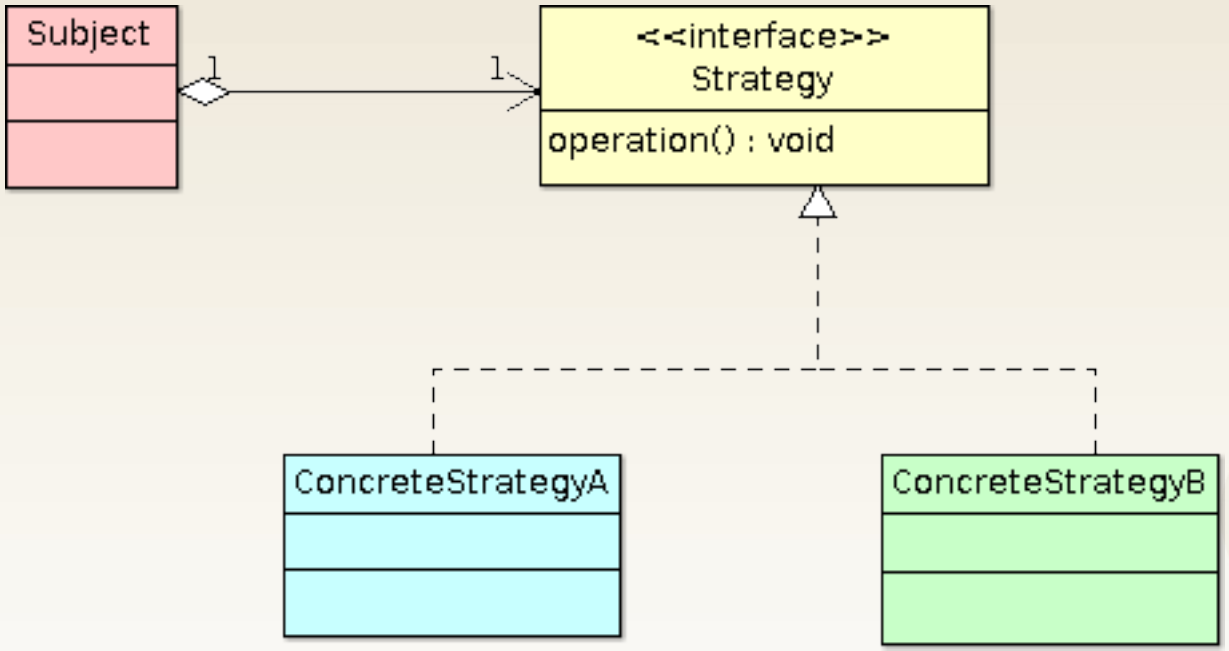


Strategy




```
public Authentication authenticate(
            String username, String password) {
      UserDetails user = userDetailsService
                  .loadUserByUsername(username);
      if (user == null || !
            user.getPassword().equals(password))
            throw new AuthenticationException();
      return new Authentication(user);
}
```

Strategy Sınıf Diagramı



- Algoritmanın **seçimi ile implemantasyonu birbirinden ayrı** tutulur
- Kullanılacak algoritma **istemciye veya eldeki veriye göre** değişiklik gösterebilir

Strategy Örüntüsünün Sonuçları

- Şartlı ifadeleri azaltması veya ortadan kaldırması algoritmayı veya **iş mantığı**nı daha **kolay anlaşılabilir** yapabilir
- Çalışma zamanında **algoritmanın dinamik olarak değiştirilmesi** sağlanabilir
- **Bağlam bilgisi**nin nasıl elde edileceği işleri biraz daha **zor veya karmaşık** bir hale sokabilir

Spring Application Framework'e Giriş

Spring Framework Nedir?

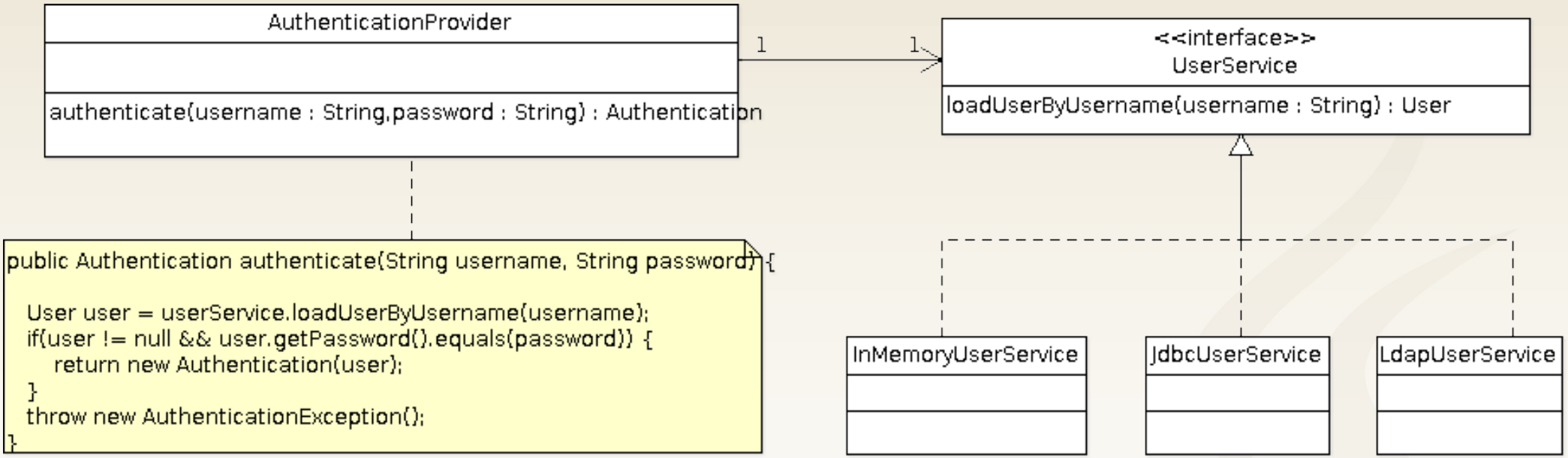
- **Kurumsal Java** uygulamalarını
  - **kolay**,
  - **hızlı** ,
  - **test edilebilir**

  biçimde geliştirmek ve Tomcat, Jetty, Resin gibi her tür ortamda çalıştırabilmek için ortaya çıkmış bir “**framework**”tür

Spring'i Öne Çıkaran Özellikler

- **POJO** tabanlı bir programlama modeli sunar
- "**Program to interface**" yaklaşımını temel ilke kabul etmiştir
- **Test edilebilirlik** her noktada **ön planda**dır
- **Modüler** **bir framework**tür, sadece ihtiyaç duyulan modüller istenilen kapsamda kullanılabilir

Program to Interface Yaklaşımı Nedir?



- Sınıflar arasındaki bağımlılıklar **arayüz ve soyut sınıflara** doğrudur
- Farklı ortamlardaki **farklı davranışlar** daha kolay biçimde ele alınabilir
- Bu sayede sınıfların **test edilebilirliği** de artmaktadır
- Testlerde **mock ve stub nesneler** daha rahat kullanılabilir

Spring IoC Container'a Doğru Yolculuk

Nesne bağımlılıklarını kendi içerisinde yaratır. Bu durumda farklı platformlar için farklı gerçekleştirimlerin yaratılması ve kullanılması problem olacaktır

```
public class AuthenticationProvider {
    private UserService userService =
                new JdbcUserService(new JdbcDataSource());
    private RoleService roleService = new RoleServiceImpl();

    public Authentication authenticate(
                String username, String password)
                                    throws AuthenticationException {
        User user = userService.loadUserByUsername(username);
        if(user != null && user.getPassword().equals(password)) {
            List<Role> roles =
                        roleService.findRolesByUsername(username);
            return new Authentication(user, roles);
        }
        throw new AuthenticationException("Authentication failed");
    }
}
```

Spring IoC Container'a Doğru Yolculuk

Nesne içerisinde basit bir factory metot içerisinde farklı platformlar için farklı gerçekleştirimleri oluşturma işlemi ele alınmaya çalışılır. Tabi bu bağımlılıkların da kendilerine ait bağımlılıkları vardır. Bunlarında ele alınması gerekecektir.

```
public class AuthenticationProvider {
    private UserService userService = createUserService();
    private RoleService roleService = new RoleServiceImpl();

    private UserService createUserService() {
        String targetPlatform =
                    System.getProperty("targetPlatform");
        if("dev".equals(targetPlatform)) {
            return new InMemoryUserService();
        } else if("test".equals(targetPlatform) ||
                    "prod".equals(targetPlatform)) {
            return new JdbcUserService(new JdbcDataSource());
        } else {
            return new LdapUserService(new LdapTemplate(
                                        new LdapContextSource()));
        }
    }
...
}
```

Spring IoC Container'a Doğru Yolculuk

Bağımlılıkları oluşturma ve platforma göre yönetme işi ayrı bir sınıfa çekilir. ServiceLocator isimli bu sınıf uygulama genelinde ihtiyaç duyulan bağımlılıklara erişim sağlar

```
public class ServiceLocator {

    private static final ServiceLocator
                            INSTANCE = new ServiceLocator();


    public static final ServiceLocator getInstance() {
        return INSTANCE;
    }

    private UserService userService;
    private RoleService roleService;
    private DataSource dataSource;

    ...
}
```

ServiceLocator genellikle singleton olur ve bootstrap esnasında target platforma göre (dev,test veya prod) servis nesnelerini genellikle reflection api ile yaratarak kullanıma hazır hale getirir

Spring IoC Container'a Doğru Yolculuk

JAVA Eğitimleri

targetPlatform'a göre property dosyalarından yüklenen sınıf bilgileri Reflection API kullanılarak ilgili servis nesnelerine dönüştürülür

```
public class ServiceLocator {
    private ServiceLocator() {
        try {
            String targetPlatform = System.getProperty("targetPlatform");

            InputStream is = this.getClass().getClassLoader()
                .getResourceAsStream("service." + targetPlatform + ".properties");

            Properties properties = new Properties();
            properties.load(is);

            String usClassName = properties.getProperty("userService");
            String rsClassName = properties.getProperty("roleService");
            String dsClassName = properties.getProperty("dataSource");

            userService = (UserService) Class.forName(usClassName).newInstance();
            roleService = (RoleService) Class.forName(rsClassName).newInstance();
            dataSource = (DataSource) Class.forName(dsClassName).newInstance();
        } catch (Exception ex) {
            throw new ServiceLocatorException();
        }
    }...
}
```

Spring IoC Container'a Doğru Yolculuk

Servis nesnelerini oluşturacak sınıfların FQN service.dev.properties gibi classpath'deki dosyalarda yönetilir

```
userService=com.javaegitimleri.example.JdbcUserService
roleService=com.javaegitimleri.example.RoleServiceImpl
dataSource=org.h2.jdbcx.JdbcDataSource
```

```
public class ServiceLocator {
    ...

    public final UserService getUserService() {
        return userService;
    }

    public RoleService getRoleService() {
        return roleService;
    }

    public DataSource getDataSource() {
        return dataSource;
    }
}
```

Servisler getter metotlar üzerinden erişilebilirler. bazı ServiceLocator varyasyonlarında **getService(String serviceName)** gibi genel bir metot üzerinden de servislere erişim sağlanabilmektedir

Spring IoC Container'a Doğru Yolculuk

```
public class AuthenticationProvider {

    private UserService userService =
                ServiceLocator.getInstance().getUserService();

    private RoleService roleService =
                ServiceLocator.getInstance().getRoleService();

    ...
}
```

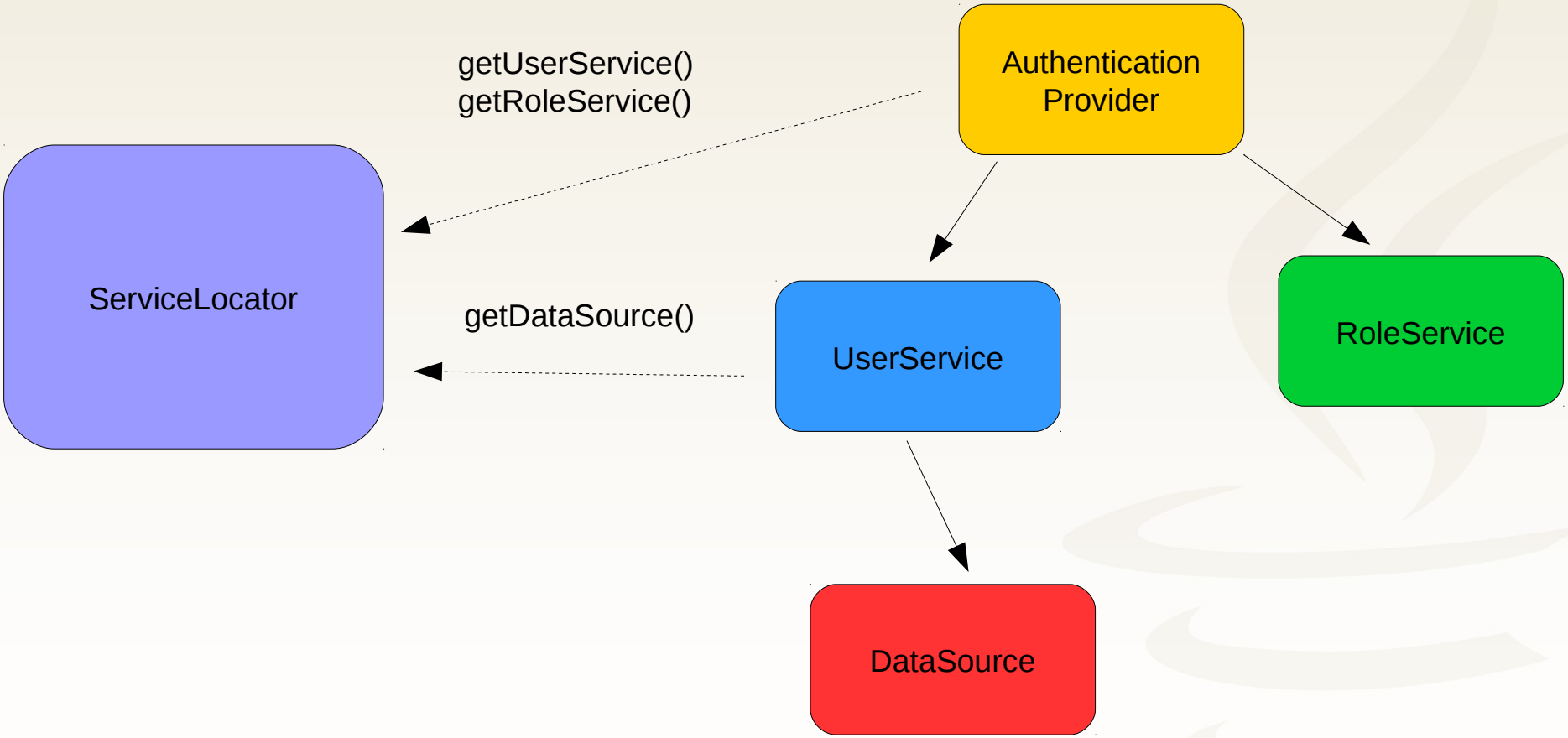
Nesneler ihtiyaç duydukları bağımlılıkları **ServiceLocator** üzerinden elde ederler

Spring IoC Container'a Doğru Yolculuk

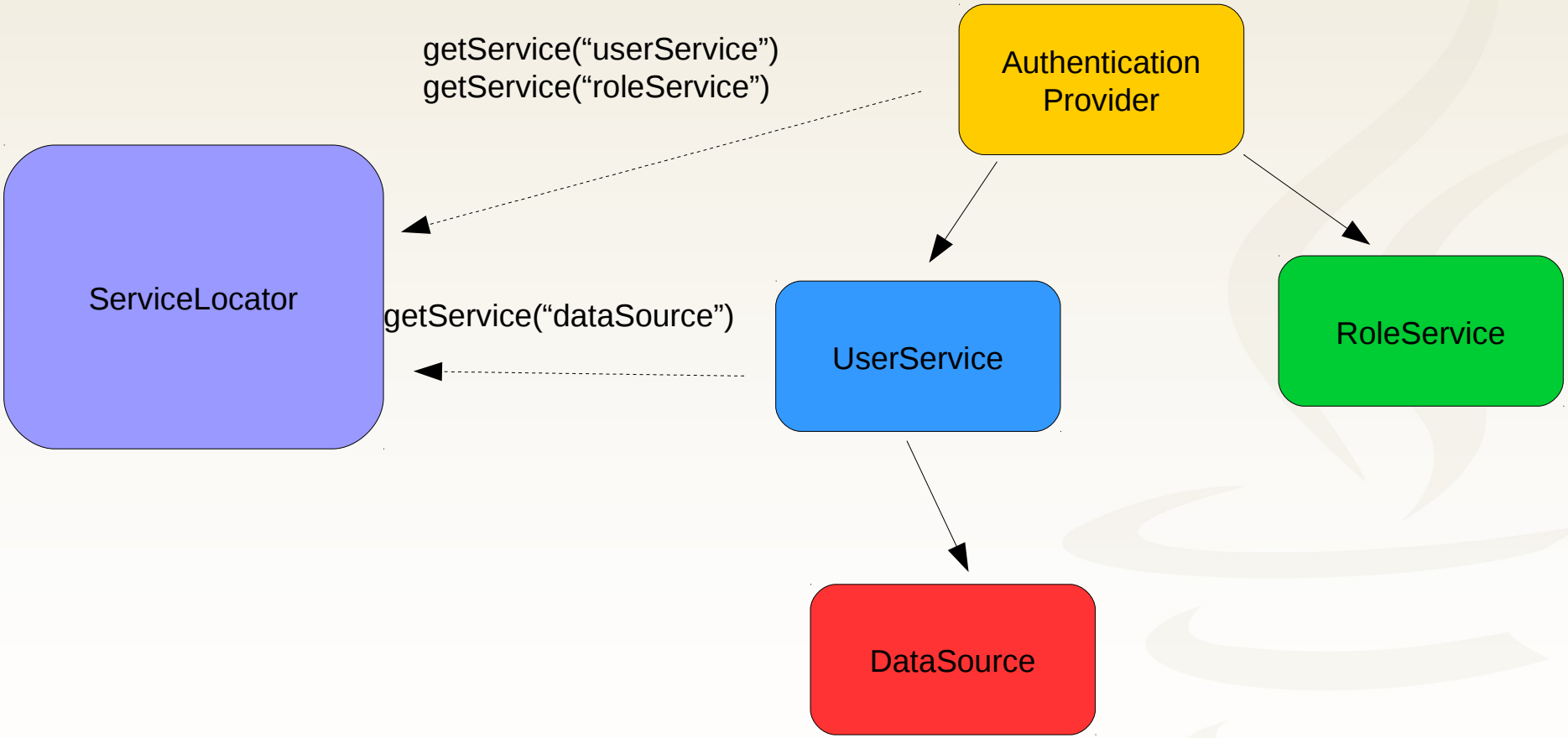
```
public class JdbcUserService implements UserService {

    private DataSource dataSource =
                    ServiceLocator.getInstance().getDataSource();

...
}
```

Bağımlılık hiyerarşisindeki bütün diğer nesneler de kendi bağımlılıklarını aynı şekilde ServiceLocator üzerinden çözümlerler

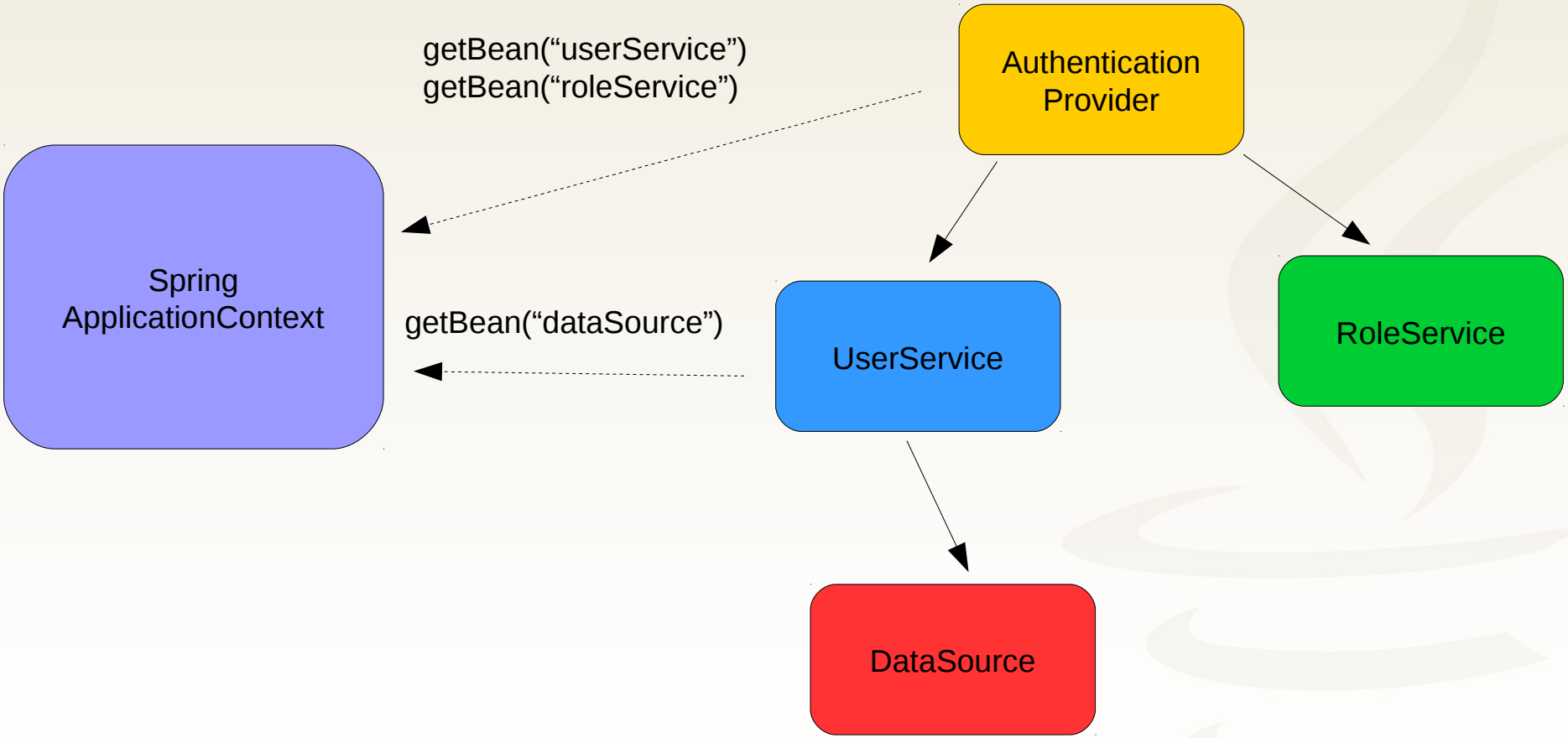
Spring IoC Container'a Doğru Yolculuk



Spring IoC Container'a Doğru Yolculuk



Spring IoC Container'a Doğru Yolculuk



Spring IoC Container ve Dependency Injection

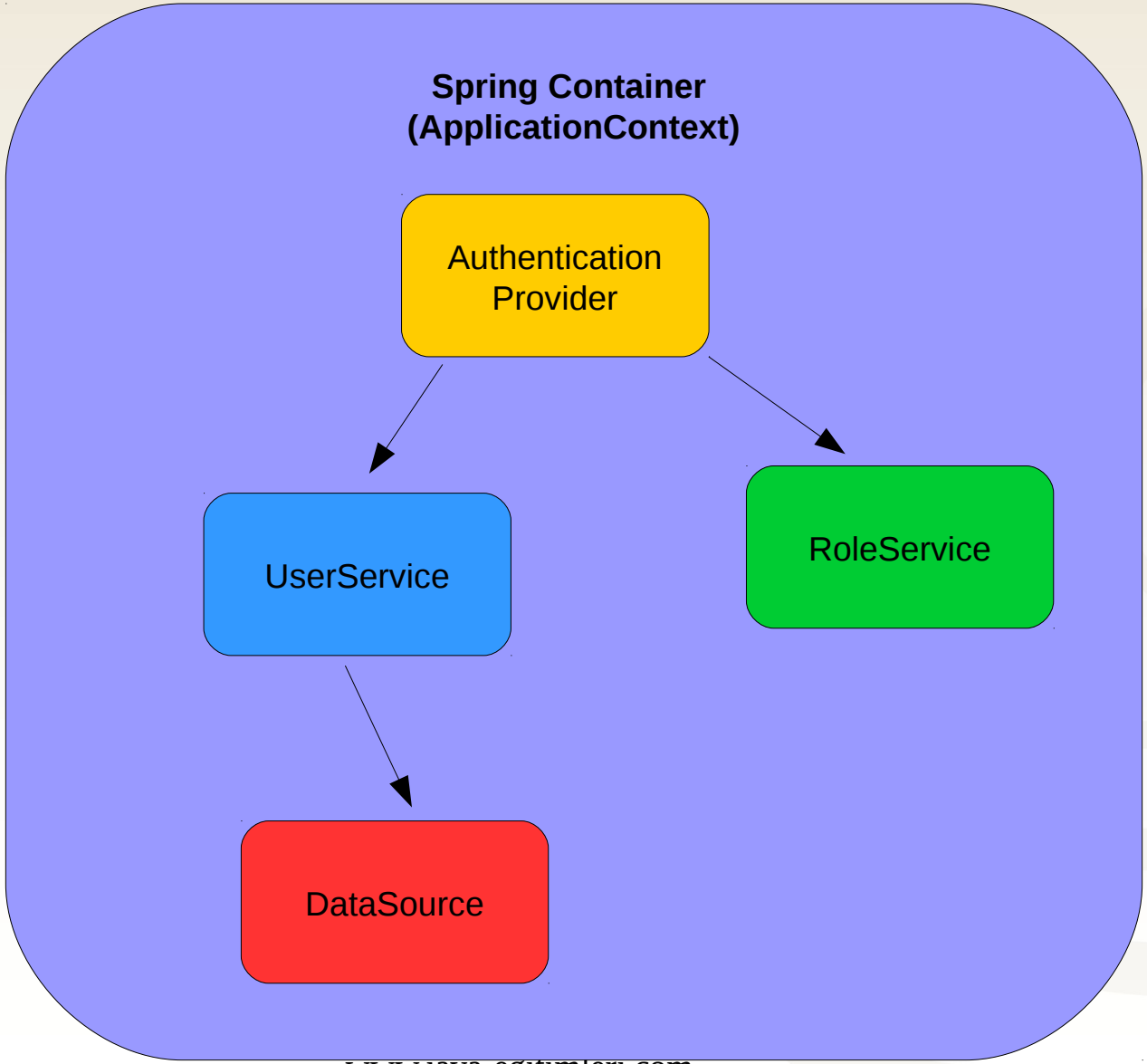
JAVA Eğitimleri

- Spring **nesneleri oluşturma, bir araya getirme ve yönetme işi**ne ServiceLocator'dan çok daha sistematik ve kapsamlı bir yol sunmaktadır
- Bağımlılıkları **oluşturma ve yönetme işi** nesnenin kendi içinden çıkıp, **Spring Container'a geçmektedir**
- Nesneler bağımlılıkların hangi **concrete sınıflar**la sağlandıklarını bilmezler

Spring IoC Container ve Dependency Injection

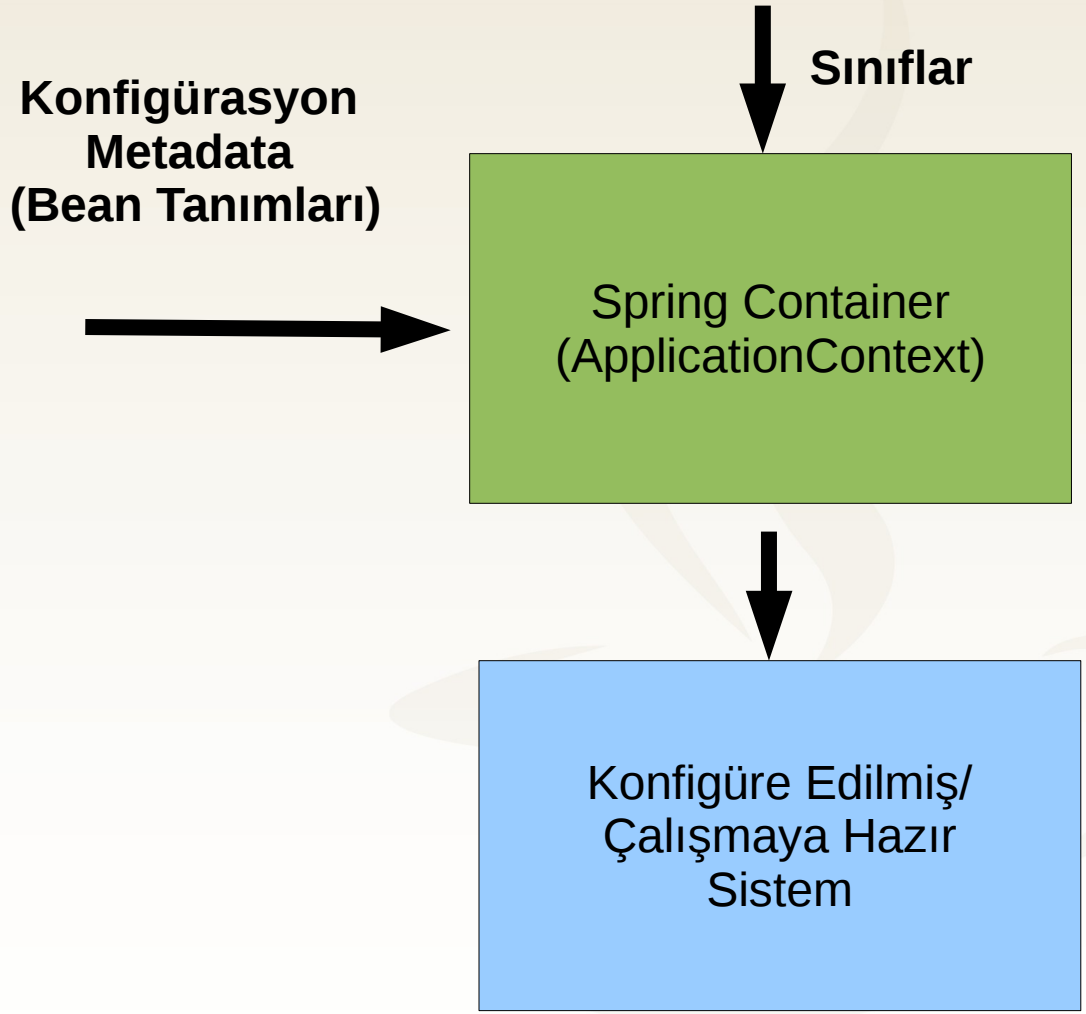
- Bağımlı olunan nesnelerin kim tarafından oluşturulduğu, nereden geldiği de bilinmez
- **Bağımlılıkların yönetimi nesnelerden container'a geçmiş**tir
- Spring IoC container tarafından yönetilen nesnelere “**bean**” adı verilir
- Spring bean'lerinin **sıradan Java nesneleri**nden hiç bir farkı yoktur

Spring IoC Container ve Dependency Injection



Spring IoC Container ve Konfigürasyon Metadata

- Bean'lerin yaratılması, bağımlılıkların enjekte edilmesi için Spring Container **bir bilgi**ye ihtiyaç duyar
- Bean ve bağımlılık tanımları **konfigürasyon metadata**'sını oluşturur



Spring IoC Container'ın Oluşturulması

JAVA Eğitimleri

- Spring Container'ın diğer adı **ApplicationContext**'dir
- ApplicationContext'i oluşturmak için **programatik** veya **dekleratif** yöntemler mevcuttur
- **Standalone** uygulamalarda **programatik** yöntem kullanılır
- **Web** uygulamalarında ise **dekleratif** yöntem kullanılır

Spring IoC Container'ın Oluşturulması

- XML, ApplicationContext oluşturmanın **geleneksel** yoludur
- Ancak **tek değil**dir
  - Java annotasyonları
  - Java kodu
- Spring IoC Container **konfigürasyon metadata formatından bağımsız**dır
- **Farklı metadata formatları**nı birlikte kullanarak da ApplicationContext oluşturulabilir

ApplicationContext Türleri

- Spring Framework farklı ApplicationContext **implemantasyonlarına** sahiptir
- Standalone uygulamalarda genellikle **ClassPathXmlApplicationContext** kullanılır
- Web uygulamalarında ise **WebApplicationContext** kullanılır

ApplicationContext'in Yaratılması ve Kullanımına Örnek

JAVA Eğitimleri

1

```
ApplicationContext context =
    new ClassPathXmlApplicationContext("beans-dao.xml",
"beans-service.xml");
```

ApplicationContext yaratılır , Container bu aşamadan itibaren kullanıma hazırdır

2

```
PetClinicService service = context.getBean(
                "petClinicService", PetClinicService.class);
```

Bean lookup ile ilgili bean'a erişilir

3

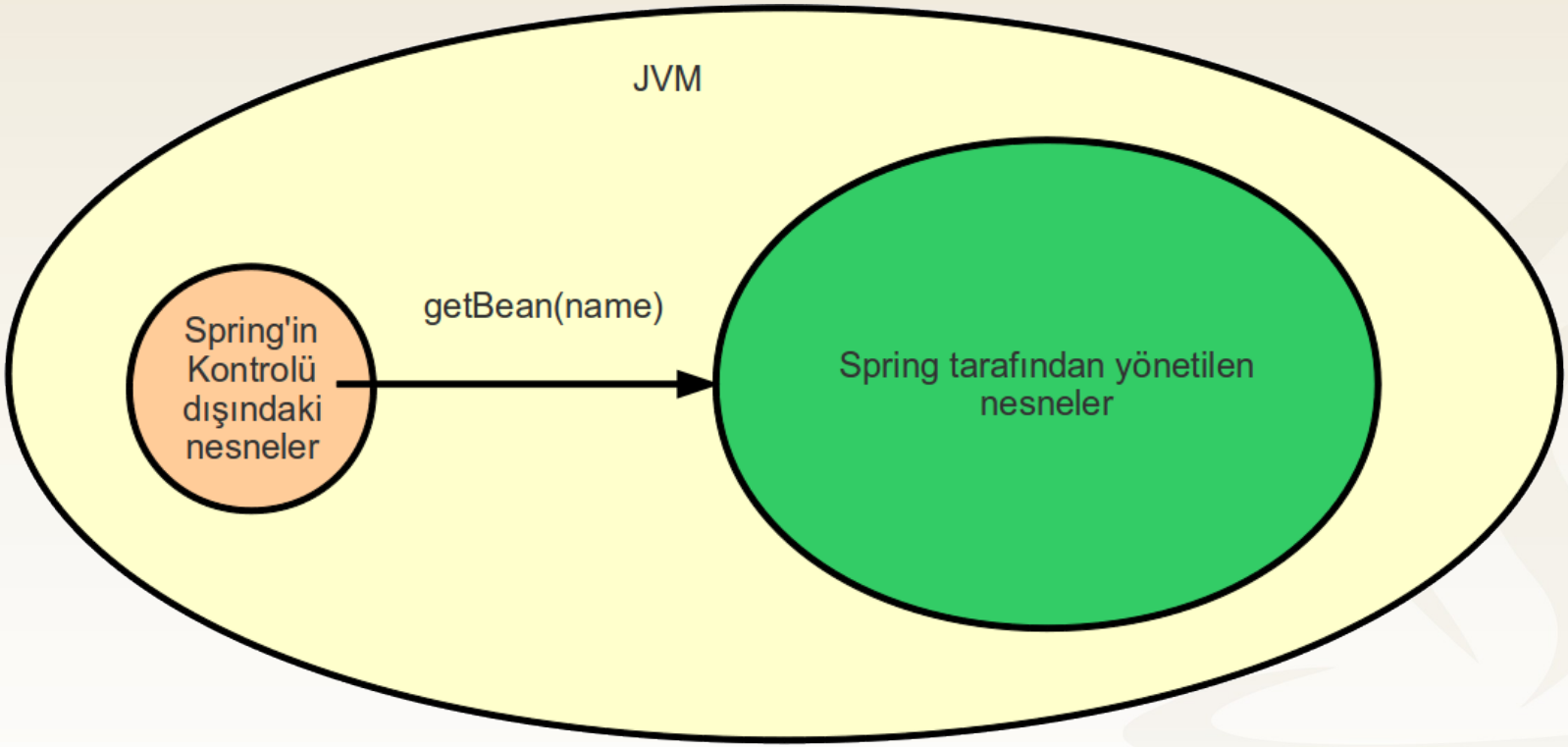
```
List vets = service.getVets();
```

Artık bean'ler uygulama içerisinde istenildiği gibi kullanılabilir

JVM ve Spring Container İlişkisi

- Aslında Spring Container veya diğer adı ile ApplicationContext'de **JVM içerisinde bir Java nesnesi**dir
- Standalone veya web tabanlı Java uygulaması içerisinde **bir yerde ApplicationContext oluşturulur**
- ApplicationContext içerisinde de **uygulamanın diğer Java nesneleri** yaratılır ve yönetilir

JVM ve Spring Container İlişkisi



JVM içerisinde hedefimiz mümkün olduğunca fazla sayıda nesnenin Spring Container tarafından yönetilmesidir, böylece bu nesneler Spring'in sunduğu kabiliyetlerden yararlanabilirler. Ancak JVM içerisinde Spring Container tarafından yönetilemeyecek nesneler de olacaktır. Spring tarafından yönetilmeyen nesnelerde ApplicationContext lookup yaparak Spring Container'daki nesnelere erişilebilir

Tasarım Örüntüleri'ne Devam...

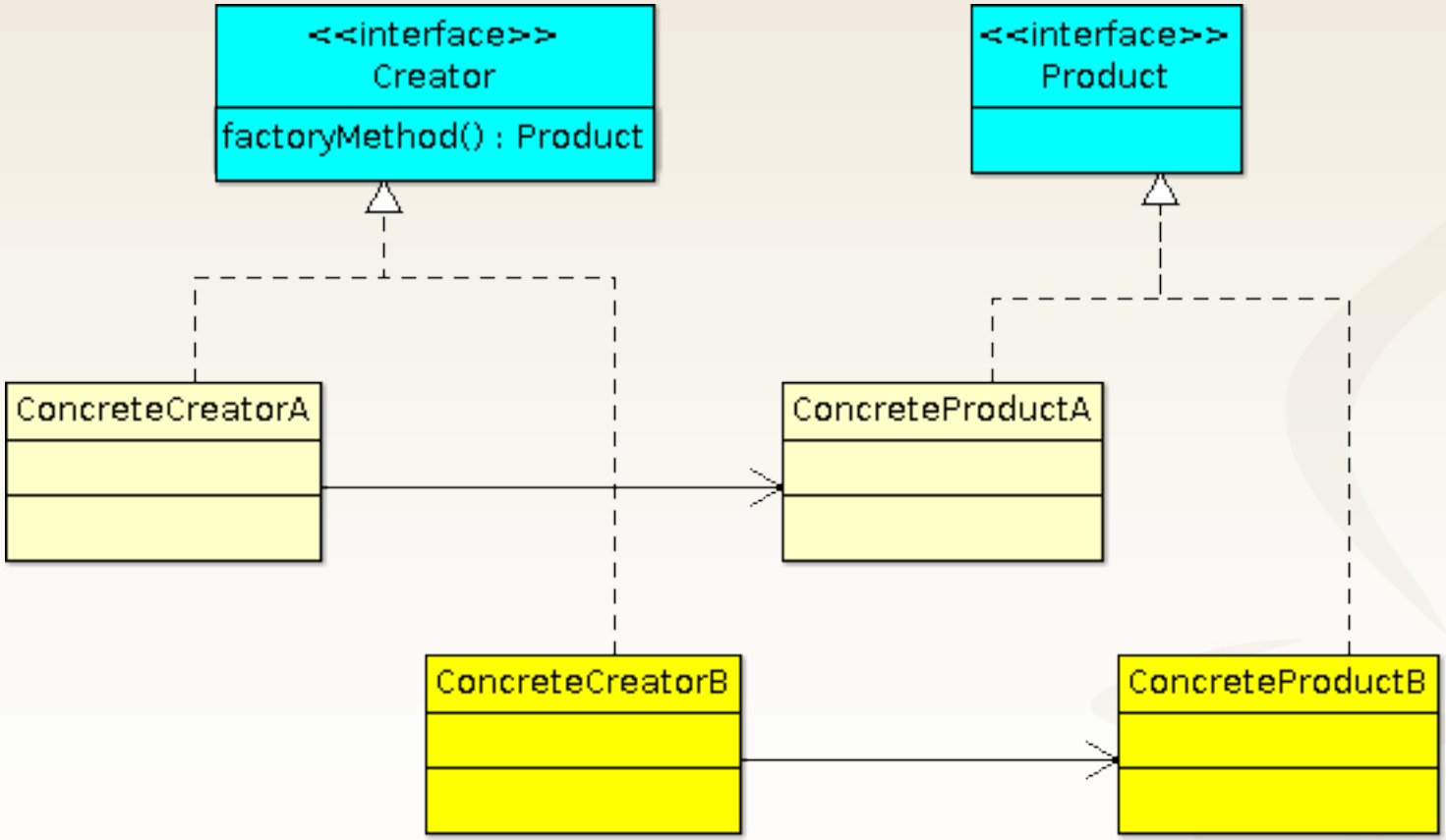
Nesne Yaratım İşinin Encapsule Edilmesi

Nesne **yaratım süreci** de encapsule edilebilir...

Factory Method

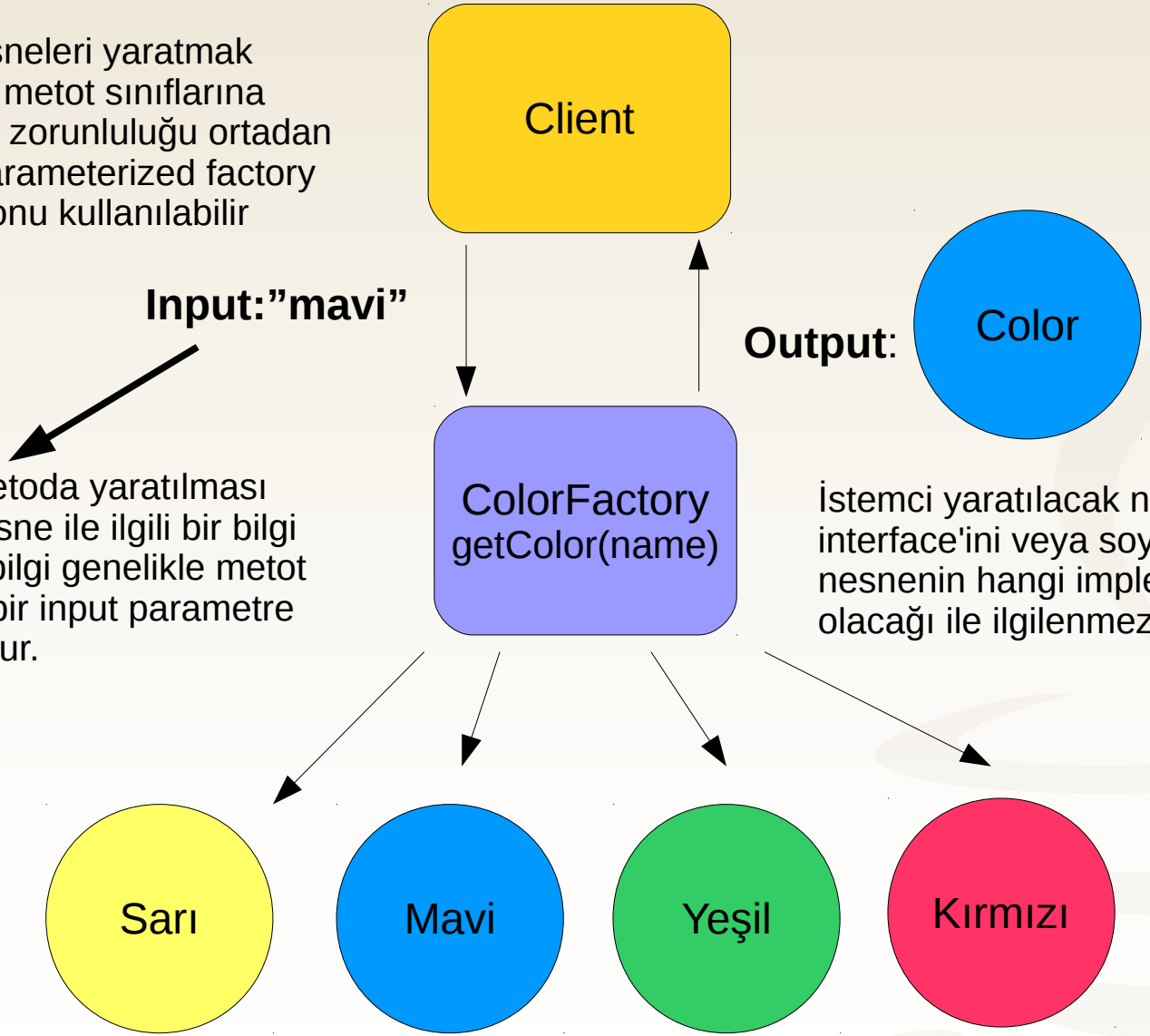
- İstemcinin ihtiyaç duyduğu **nesneyi yaratan bir yaratıcı** sınıf vardır
- Yaratıcı sınıf içerisinde nesne **yaratma işinden sorumlu bir metot** vardır
- İstemci yeni bir nesneye ihtiyaç duyduğunda **bu metodu çağır**ır
- İstemci ihtiyaç duyduğu **nesnenin concrete tipini bilmez,** yaratıcı metot soyut tipte bir değer döner

Factory Method Sınıf Diagramı



Parametrik Factory Method

Farklı türden nesneleri yaratmak için farklı factory metot sınıflarına ihtiyaç vardır. Bu zorunluluğu ortadan kaldırmak için parameterized factory method varyasyonu kullanılabilir



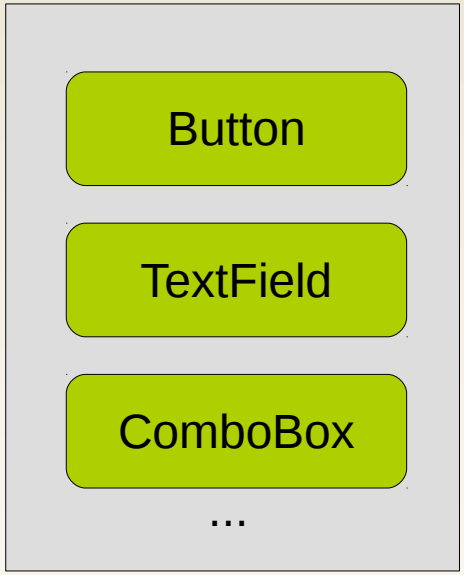
Factory metoda yaratılması istenen nesne ile ilgili bir bilgi verilir. Bu bilgi genelikle metot ismi veya bir input parametre şeklinde olur.

İstemci yaratılacak nesnenin sadece interface'ini veya soyut tipini bilir. Yaratılan nesnenin hangi implemantasyondan olacağı ile ilgilenmez.

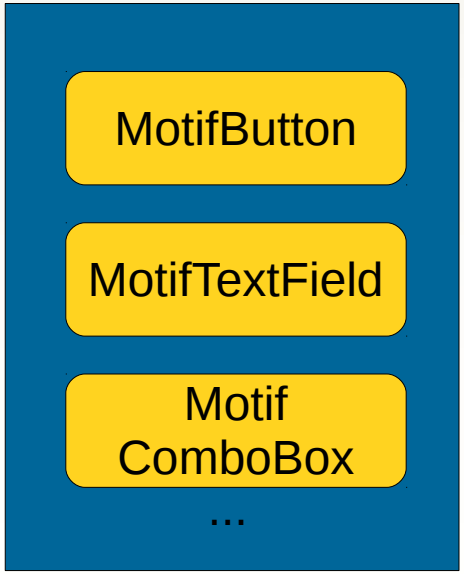
Abstract Factory

- Bazen **bir grup nesnenin koordineli biçimde yaratılması** söz konusu olabilir
- Örneğin Swing gibi bir **grafik toolkit**, UI bileşenleri için **birden fazla "look & feel"** desteği sunabilir
- Her farklı "look & feel" için **ayrı bir sınıf seti** vardır
- Çalışma zamanında da **her UI bileşeni kendi setinden** bir sınıftan **yaratılmalı**dır

Abstract Factory

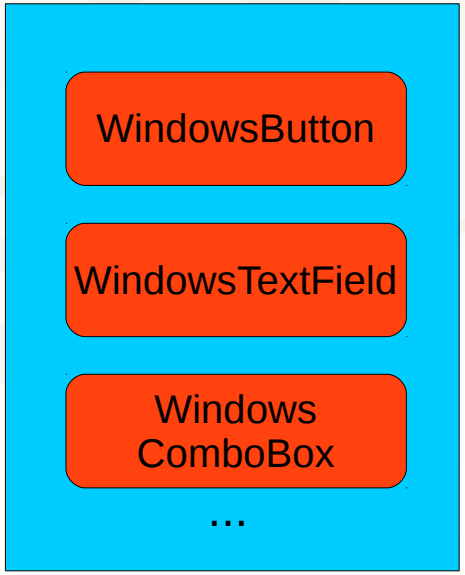


Grafik toolkit her bir farklı UI bileşeni için soyut sınıflar tanımlar ancak GUI'nin ilgili platformda çalışabilmesi için concrete UI sınıflarına ihtiyaç vardır



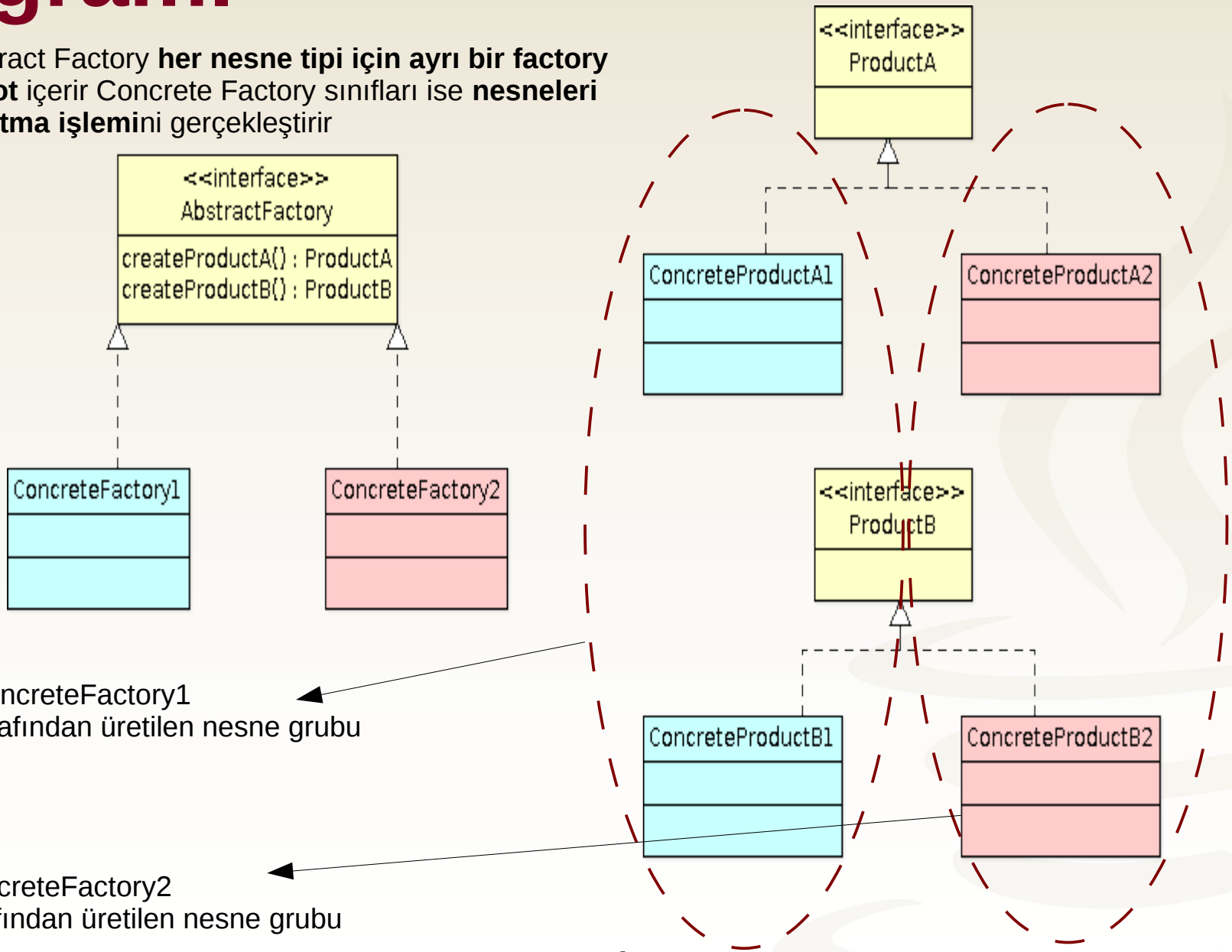
Motif ve Windows gibi iki farklı look & feel oluşturulmak istendiğinde soyut UI sınıflarının hepsi bu look & feel'e uygun biçimde implement edilmeliler

İstemciler de bu concrete implemantasyonları birbirleri ile karıştırmadan kullanmalılar



Abstract Factory Sınıf Diagramı

Abstract Factory **her nesne tipi için ayrı bir factory metot** içerir Concrete Factory sınıfları ise **nesneleri yaratma işlemi**ni gerçekleştirir



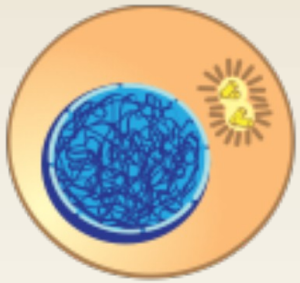
Factory Method Örüntüsünün Sonuçları

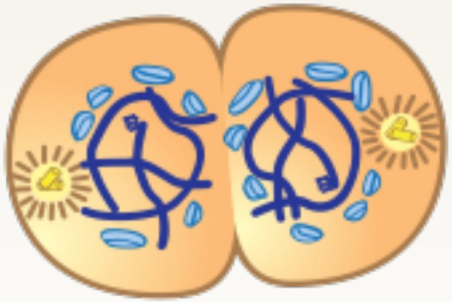
- Nesneyi **yaratan** kısım ile nesneyi **kullanan** kısım farklı yerlerdir
- Bu sayede nesnenin **yaratılma süreci** nesneyi **kullanan yerden bağımsız** hale gelmiş olur
- Nesneyi kullanan kısım concrete sınıflara değil, **soyut sınıflara bağımlı** kalır

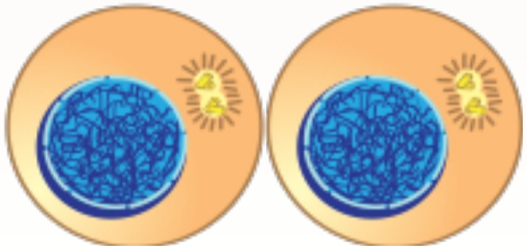
Abstract Factory Örüntüsünün Sonuçları

- Birbirleri ile **ilişkili nesne grupları**nı oluşturmayı kolaylaştırır
- İstemci tarafının nesneleri **yanlış sınıflar**dan oluşturmasının önüne geçer
- Farklı **nesne setleri arasında geçiş**i kolaylaştırır

Prototype





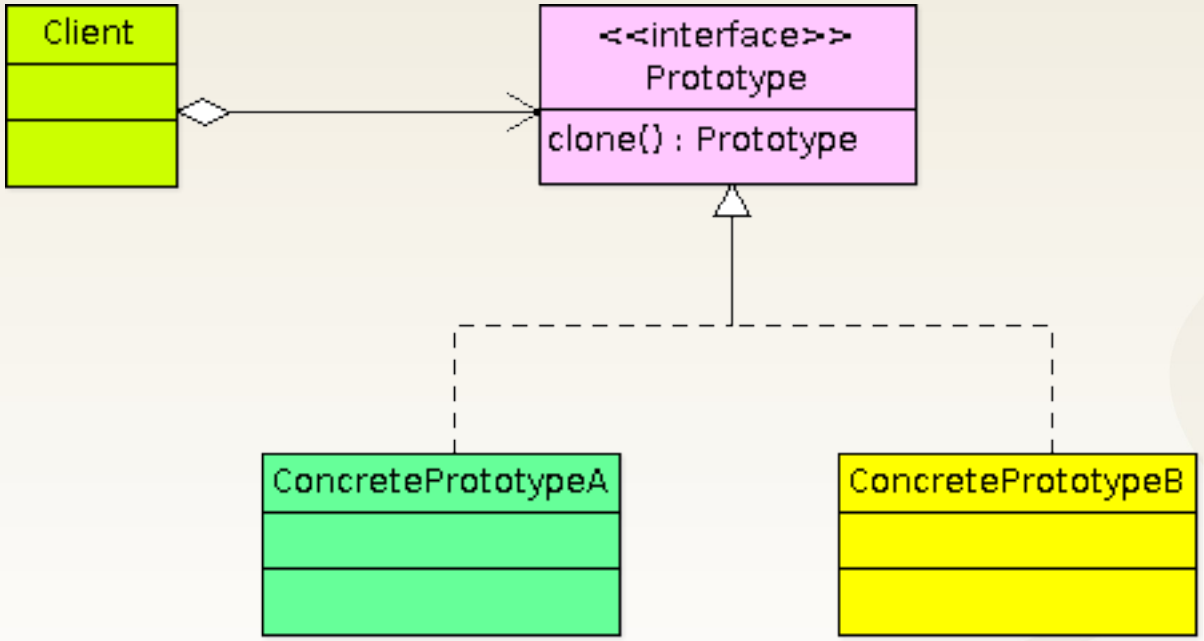


- Bazı durumlarda nesnelerin kullanılabilir duruma gelebilmeleri için **new operatörü** ile yaratılması ve **initialize edilmesi** birkaç adımdan oluşabilir ve uzun sürebilir
- Bu tür senaryolarda önceden bir nesne **initialize edilmiş biçimde** hazırlanarak tutulur
- Yeni nesneler bu prototip nesne **klonlanarak** elde edilir

Prototype

- Örneğin bir **grafik editör aracı**nda bir resmi edit ettiğimizi düşünelim. **Resim** üzerinde **border oluşturma**, **bevel ekleme**, **tonlama** gibi işlemler yaptığımızı farz edelim
- Bu aşamada resmin bu halini Ctrl+C, Ctrl+V ile **kopyala yapıştır** yapmaya çalışalım
- Uygulamanın kopyala yapıştır için izleyeceği yol, hali hazırda belirli bir aşamaya gelmiş bu nesnenin bire bir aynı kopyasını (**klonu**) oluşturması olacaktır

Prototype Sınıf Diagramı



Java ve Prototype

- Herhangi bir nesnenin prototip olarak oluşturulması ve bu nesneden klonlama yolu ile diğer yeni nesnelerin oluşturulması için Java, **clone** metodunu sunmaktadır
- Ancak **java.lang.Object** sınıfındaki bu **clone** metodu **tasarımsal olarak problemli**dir
- Bu problemler de geliştiricilerin **kolaylıkla bazı hatalara düşmeleri**ne neden olmaktadır

Clone Metodunun Problemleri

- Herhangi bir nesnenin klonlanabilir olması için öncelikle sınıfının **java.lang.Cloneable** arayüzünü implement etmesi gerekir
- Ancak clone metodu bu arayüzde tanımlı değildir, aksine doğrudan **java.lang.Object** sınıfında tanımlıdır
- Fakat burada da **protected** olarak tanımlıdır
- Dolayısı ile clone metodunu **public modifer ile override etmeden çağırmak** mümkün değildir

Clone Metodunun Problemleri

- Normal şartlarda bir arayüz, istemcilerine onu implement eden sınıfın **ne kabiliyete sahip olacağını,** içerdiği metot tanımları ile ilan eder
- **Cloneable** arayüzü ise Object sınıfındaki **clone metodunun davranışını** değiştirir
- Cloneable arayüzü implement edili değilken bir nesnenin clone metodu çağrılırsa **CloneNotSupportedException** fırlatılır
- Eğer arayüz implement edilirse metot bu durumda nesnenin **attribute'ları teker teker kopyalayarak** kopya nesneyi oluşturur

Clone Metodunun Problemleri

- Clone metodunun spesifikasyonuna göre **x.clone().getClass() == x.getClass()** olmalıdır
- Ayrıca nesnenin kopyası yaratılırken **herhangi bir constructor çağrılmayacağı** belirtilmiştir
- Ancak bunları clone metodunu **override eden alt sınıflarda** garanti edecek bir mekanizma mevcut değildir
- Aslında **eğer sınıf final ise alt sınıf olamayacağı için** constructor çağrısı ile nesne kopyası oluşturmak herhangi bir sorun da çıkarmayacaktır

Clone Metodunun Problemleri

- Ancak final olmayan sınıflarda clone metodu override edildiğinde nesne kopyası mutlaka **super.clone()** çağrısı ile oluşturulmalıdır
- Klonlanan nesnenin attribute'ları kopyalanırken **yüzeysel kopyalama** (swallow copy) yapılır
- Bu nedenle asıl nesne ve kopya nesne **aynı nesnelere refer** ediyor olabilir
- Attribute'lardaki **mutable nesneler** için bu ciddi sorun teşkil edebilir

Clone Metodunun Problemleri

- Böyle durumlarda **derin kopyalama** (deep copy) yapmak ve attribute'lara değerlerini bu derin kopyalama sırasında **ayrı ayrı oluşturmak ve atamak** gerekebilir
- Bunun için de attribute'ların **final tanımlanmamış olması** gerekir
- Constructor'larda olduğu gibi clone metodunda da **final olmayan metotların invoke edilmesi** inheritance nedeni ile risk teşkil edebilir
- Burada meydana gelebilecek bir hata **nesnenin yaratımı**nı başarısız kılacaktır

Clone Metoduna Alternatif: Copy Constructor

- Clone metodunu kullanmak yerine önerilen çözümlerden birisi “**copy constructor**” yaklaşımıdır
- Bir sınıf **constructor argümanı** olarak **kendi tipinden bir nesne** kabul eder
- Constructor içerisinde **input nesnenin state'i** yeni yaratılan nesneye kopyalanır

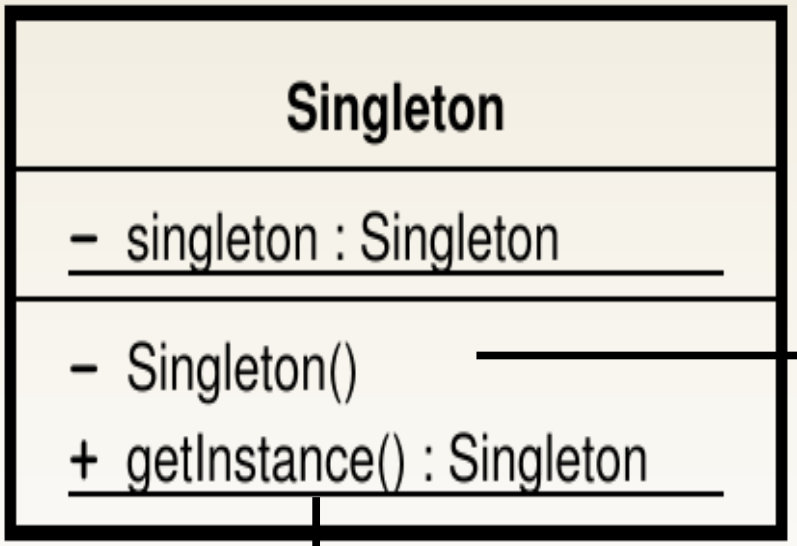
Prototype Örüntüsünün Sonuçları

- Halihazırda yaratılmış, initialize edilmiş ve belli bir state'e gelmiş **nesnenin o anki gösteriminin elde edilmesi** çok daha kolay ve hızlı olur
- Yaratılacak nesnenin **sınıfının çalışma zamanında belirlenmesi**ne de imkan tanır
- Clone metodunun implemantasyonu nesne hiyerarşisinin **derin kopya**sının oluşturulmasında veya **döngüsel bağımlılıklar**da zor olabilir

Singleton

- Bir sınıftan **uygulama genelinde tek bir nesne**nin olmasını sağlar
- Ayrıca bu nesneyi uygulama içerisinde **herhangi bir yerden de erişilebilir** kılar

Singleton Sınıf Diagramı



Nesne **private static** değişkende tutulur

Constructor private yapılarak sınıf dışından herhangi bir biçimde başka instance oluşturulmasının önüne geçilir

public static getInstance() metodu ile uygulama içerisinde Herhangi bir yerden bu instance'a erişmek mümkün olur

Java ve Singleton: Eager Initialization


```
public class Foo {
    public static final Foo INSTANCE = new Foo();

    private Foo() {
    }

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

```
public class Foo {
    private static final Foo INSTANCE = new Foo();

    private Foo() {
    }

    public static final Foo getInstance() {
        return INSTANCE;
    }

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

Klasik eager initialization yöntemi ile singleton oluşturma örnekleridir

İlk örnekte statik final değişken public yapılarak dış dünyanın doğrudan erişimine açılmıştır

İkinci örnekte ise singleton instance static final getInstance() metodu ile erişilebilir kılınmıştır

Java ve Singleton: Lazy Initialization

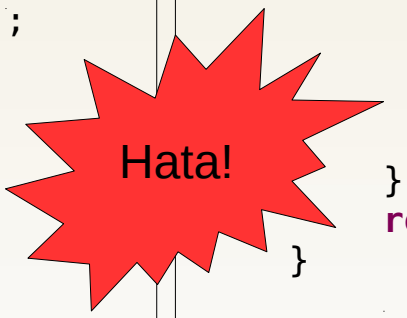
```
public class Foo {
    private static Foo INSTANCE;

    private Foo() {
    }

    public synchronized static final Foo getInstance() {
        if(INSTANCE == null) {
            INSTANCE = new Foo();
        }
        return INSTANCE;
    }

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

Burada ise getInstance() metodunda lazy initialization yapılamaktadır.
Metodun synchronized olması önemlidir!
Ancak bütün metodun synchronized yapılması maliyetlidir.

Java ve Singleton: Double Checked Locking Idiom

JAVA Eğitimleri

```
public class Foo {
     private static Foo INSTANCE;

     private Foo() {
     }

     public static final Foo getInstance() {
          if(INSTANCE == null) {
               synchronized (Foo.class) {
                    INSTANCE = new Foo();

               }
          }
          return INSTANCE;
     }

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




```
public class Foo {
     private static Foo INSTANCE;

     private Foo() {
     }

     public static final Foo getInstance() {
          if(INSTANCE == null) {
               synchronized (Foo.class) {
                    if(INSTANCE == null) {
                         INSTANCE = new Foo();
                    }
               }
          }
          return INSTANCE;
     }

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

Bütün metodu synchronized yapmak yerine sadece singleton instance'ın yaratıldığı bölümü synchronized yapmak daha efektif bir çözüm olabilir. Ancak yukarıdaki iki çözüm de problemlidir!

Java ve Singleton: Double Checked Locking Idiom


```
public class Foo {
    private static volatile Foo INSTANCE;

    private Foo() {
    }

    public static final Foo getInstance() {
        if(INSTANCE == null) {
            synchronized (Foo.class) {
                if(INSTANCE == null) {
                    INSTANCE = new Foo();
                }
            }
        }
        return INSTANCE;
    }

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

Foo içerisideki bütün diğer alanlarda volatile olmalıdır!

JVM'in shared/global hafıza alanının yanı sıra, her bir Thread'in kendine özel hafıza alanı(CPU/Thread cache) vardır. Thread'ler global hafıza alanındaki değerleri kendi cache'lerine kopyalarlar, cache üzerinde işlemleri yaparlar ve daha sonra cache'deki değişiklikleri global alana yazarlar. **JVM'de nesne oluşturma, nesne adresinin değişkene yazılması gibi işlemlerde atomik olmayan birden fazla adımda yürütülebilirler**. Örneğin, singleton instance'ın hafıza alanının ayrılması ve adresin değişkene atanması ve constructor'ın çalıştırılarak instance'ın yaratılması farklı adımlarla gerçekleşebilir. **Dolayısı ile global hafıza alanında değişkenin NULL olmadığı ama nesne'nin de tam olarak construct edilmediği bir durumda diğer Thread bu değişkene erişip işlem yapabilir**. Volatile anahtar kelimesi ile JVM'e değişkenin global hafıza alanında tutulacağı,okuma/yazmaların doğrudan buradan yapılacağı söylenebilir.

Java ve Singleton: Singleton Holder

```
public class Foo {

    private static final class FooHolder {
        private static final Foo INSTANCE = new Foo();
    }

    private Foo() {
    }

    public static final Foo getInstance() {
        return FooHolder.INSTANCE;
    }

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

Java'da inner sınıflar erişilmedikleri müddetçe yüklenmezler. Dolayısı ile FooHolder inner sınıfı da getInstance() metodu çağrılmadığı müddetçe yüklenmeyecek, böylece INSTANCE değişkeni de initialize olmayacaktır. Bu da Java'da az bilinen bir lazy initialization yöntemidir.

Java ve Singleton: Enum Singleton

```
public enum Foo {
    INSTANCE;

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

- Java 5 ile birlikte gelen Enum kabiliyeti ile tek instance'a sahip enum tipleri oluşturulabilir

Java ve Singleton: Serializable Singleton

- Singleton sınıfları **Serializable** veya **Externalizable** arayüzlerini implement edebilirler
- Serialize/deserialize işlemi ile Singleton instance'tan **yeni bir kopya** elde etmek mümkündür
- Bu durumda uygulama genelinde **tek bir instance kuralı ihlal edilmiş** olur
- Bunun önüne geçmek için **readResolve/writeReplace** metotları kullanılabilir

Java ve Singleton: Serializable Singleton

```
public class Foo implements Serializable {

    private static final Foo INSTANCE = new Foo();

    private Foo() {
    }

    public static final Foo getInstance() {
        return INSTANCE;
    }

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


```
Foo f1 = Foo.INSTANCE;
ByteArrayOutputStream bout = new ByteArrayOutputStream();
ObjectOutputStream out = new ObjectOutputStream(bout);
out.writeObject(f1);

ByteArrayInputStream bin = new
ByteArrayInputStream(bout.toByteArray());
ObjectInputStream in = new ObjectInputStream(bin);

Foo f2 = (Foo) in.readObject();

System.out.println(f1 == f2);
```

f1 ve f2 instance'larının birbirlerinden farklı oldukları görülecektir!

Java ve Singleton: Serializable Singleton

```
public class Foo implements Serializable {

    private static final Foo INSTANCE = new Foo();


    private Foo() {
    }

    public static final Foo getInstance() {
        return INSTANCE;
    }

    private Object readResolve()
                    throws ObjectStreamException {
        return INSTANCE;
    }

    private Object writeReplace()
                    throws ObjectStreamException {
        return INSTANCE;
    }

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

ObjectInputStream'den okunan nesnenin bu metot tarafından dönülen nesne ile değiştirilmesini sağlar. Böylece deserialization sırasında oluşan farklı nesne yerine asıl singleton instance dönülebilir.

ObjectOutputStream'e yazılan nesnenin bu metot tarafından dönülen nesne ile değiştirilmesini sağlar. Böylece İstenirse asıl nesneden farklı bir Nesnenin asıl nesne yerine serialize edilmesi mümkündür.

Java ve Singleton: ClassLoaders

- JVM'de sınıflar **ClassLoader** nesneleri tarafından yüklenmektedir
- Dolayısı ile bir singleton sınıf **iki farklı ClassLoader nesnesi** tarafından yüklenirse **iki singleton instance** yaratılması kaçınılmazdır
- **Standalone uygulamalar**da genellikle tek ClassLoader olduğu için bu sorun teşkil etmez

Java ve Singleton: ClassLoaders

- **Web uygulamaları**nda ise uygulama sunucusu birden fazla ClassLoader ile çalışmaktadır
- Bu yüzden **farklı ClassLoader instance'ları**nın aynı singleton sınıfı birden fazla yükleme ihtimali ortaya çıkabilir
- Web sunucusunda veya kütüphanelerde yapılacak düzenlemeler ile bu durumun **önüne geçilmesi** gerekir

Singleton Örüntüsünün Sonuçları

- Sistem genelinde **bir sınıftan tek bir nesne** olmasını garanti eder
- Bazı senaryolar için **bu çok önemli**dir. Örneğin, cache, file sistem yöneticisi vb.
- **Birim testler** ile çalışmayı **zor**laştırabilir
- Tek instance'ı garanti etmek için **class loading** ve **serialization** gibi konuları da dikkate almak gerekir

İletişim

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