

# Proxy (Vekil) Tasarım Şablonu

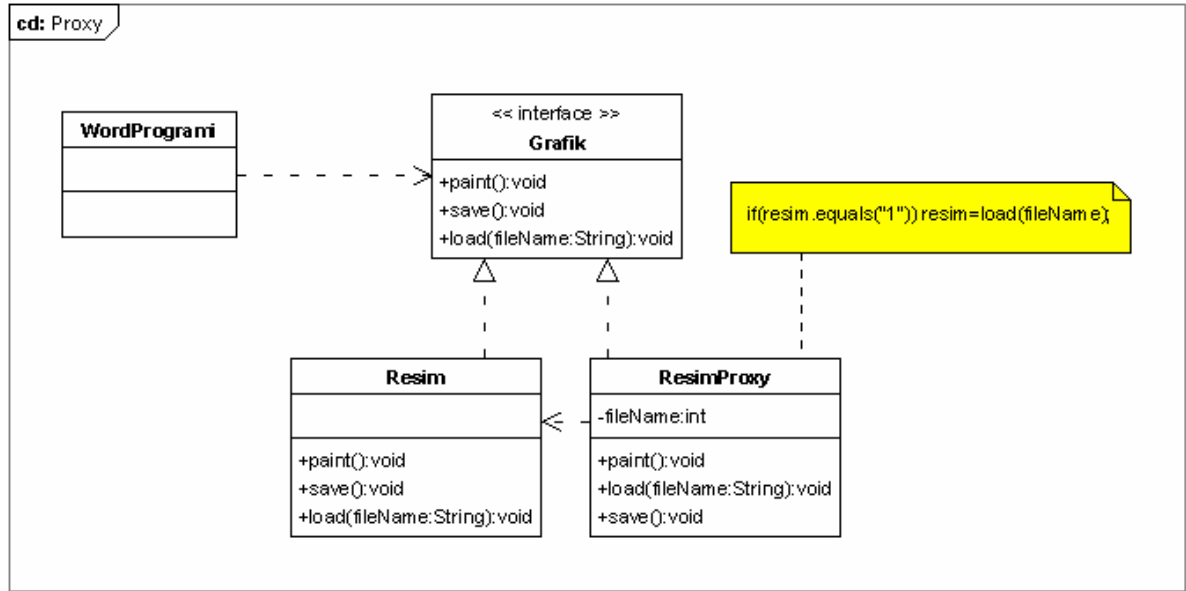
**KurumsalJava.com**

Özcan Acar  
Bilgisayar Mühendisi  
<http://www.ozcanacar.com>

Oluşturulmaları zaman alıcı ve sistem kaynaklarını zorlayan nesnelere vekalet eden nesnelere proxy nesneleri adı verilir. Bu nesneler vekil oldukları nesnelerin tüm metodlarına sahiptirler ve kullanıcı sınıf ile vekil olunan nesne arasında aracılık yaparlar. Vekil olan nesne, kullanıcı sınıfa, vekil olunan nesne gibi davranır ve kullanıcı sınıftan gelen tüm istekleri vekil olunan nesneye iletir. Böyle bir yapının kullanılmasının sebebi, gerek olmadığı sürece vekil olunan nesnenin oluşturulmasını engellemektir ya da vekil olunan nesneyi gizlemektir. Böylece vekil olunan nesneye dışardan erişimlerde kontrol altına alınmış olur. Yazılan programın yapısına göre, değişik tipte proxy nesneler kullanılabilir. Bunlar virtual (sanal), remote (uzak bir nesne) ve dynamic (dinamik nesne) proxy nesneler olabilir. Değişik proxy tiplerini yakından inceliyelim.

## Virtual Proxy (Sanal Vekil)

İçinde yüzlerce sayfanın ve resmin bulunduğu bir Word dokümanı, virtual proxy nesnelerin kullanımını açıklamak için iyi bir örnektir. Doküman açıldığında, doküman içinde bulunan tüm resimler bir seferde Word programı tarafından yüklenmez. Eğer böyle olsa idi, dokümanın ilk sayfasını görmek çok uzun zaman alırdı, çünkü Word programı uzun bir süre resimleri yüklemek ile meşgul olurdu. Word programının nasıl çalıştığını tam olarak bilmemekle beraber, proxy nesnelerin kullanıldığını düşünebiliriz. Doküman içinde yer alan tüm grafikler için bir proxy nesnesi oluşturulur. Bir grafik için bir proxy nesnesinin oluşturulması, temsil ettiği grafiğin yüklenmesi ve gösterilmesi anlamına gelmez. Daha ziyade proxy nesne, doküman içinde grafik yerine oluşturulan bir yer tutucu öge (placeholder) olarak düşünülebilir.



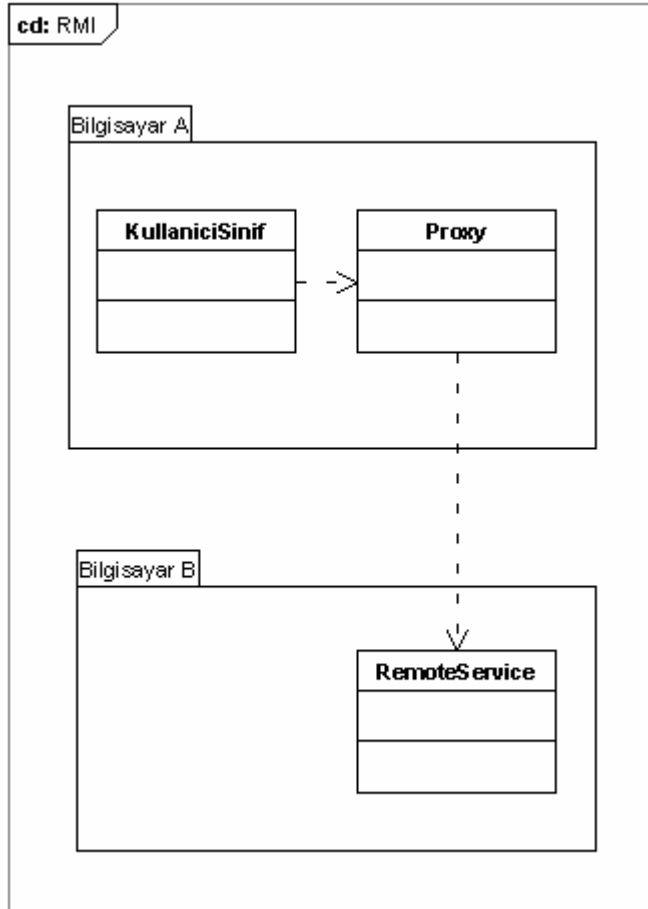
Proxy, yerini aldığı grafiğin boyutlarını tanıyor olabilir ve Word programına, yüklenmesi gereken grafiğin eni ve boyu hakkında bilgi verebilir. Grafiğin yüklenme işlemi ise, Word programı içinde grafiğin yer aldığı sayfanın gösterimi ile gerçekleşir. Grafiğin bulunduğu sayfa gösterilmediği sürece, proxy nesne grafiğin yerini alır ve grafiği temsil eder. Word programı, grafiğin bulunduğu sayfaya gelindiğinde proxy nesnesinin `ciz()` ya da `goster()` metodunu kullanarak, grafiğin sayfa içinde gösterilmesini sağlar. Aslında grafiğin gösterilme işlemi proxy nesne tarafından değil, proxy nesnesinin `ciz()` ya da `goster()` metoduna gelen çağırının asıl grafik nesnesinin bu isimdeki metoduna deleğe edilmesiyle gerçekleşir. Bu sayede gerek duyulmadığı sürece, yani grafiğin yer aldığı sayfa gösterimde olmadığı sürece

asıl grafik nesnesi oluşturulmaz. Bu da sistem kaynaklarının daha idareli kullanılmasını sağlar ve programın reaksiyon süresini azaltır.

## Remote Proxy (Uzaktaki Vekil)

New operatörü ile oluşturduğumuz tüm nesneler kullandığımız bilgisayarın hafızasında yer alırlar. Yeni internet teknolojileri ile, kullandığımız servisler birden fazla bilgisayar ve serverin üzerinde çalışmaktadır. Bu servisleri kullanabilmek için, üzerinde bulundukları bilgisayarlara bağlantı kurulması gerekmektedir. Java dilinde, başka bir bilgisayarın hafızasında bulunan bir nesnenin sunduğu servise ulaşabilmek için RMI<sup>1</sup> teknolojisi kullanılır. RMI, bir bilgisayardan diğer bir bilgisayara TCP/IP<sup>2</sup> Protokolü ile bağlantı kurup, bir nesnenin sahip olduğu metodları, o nesnenin, aynı adres alanı içinde bulunuyormuşçasına kullanımını sağlayan bir protokoldür. Jdk (java programlama araçları) içinde RMI ile programlama yapabilmek için tüm araçlar mevcuttur. Kısaca RMI ile başka bir bilgisayarın hafızasında bulunan bir nesneyi, new operatörü kullanırmışçasına kendi adres alanımızda kullanabiliriz. Nesneler ve bilgisayarlar arası bağlantıyı RMI gerçekleştirir.

RMI programlarında remote proxy nesneler kullanılır.



Proxy, Bilgisayar B üzerinde bulunan nesneye vekillik eder. Bilgayar A üzerinde bulunan **KullaniciSinif**, Bilgisayar B üzerinde bulunan nesnenin metodlarını kullandığını düşünür ama

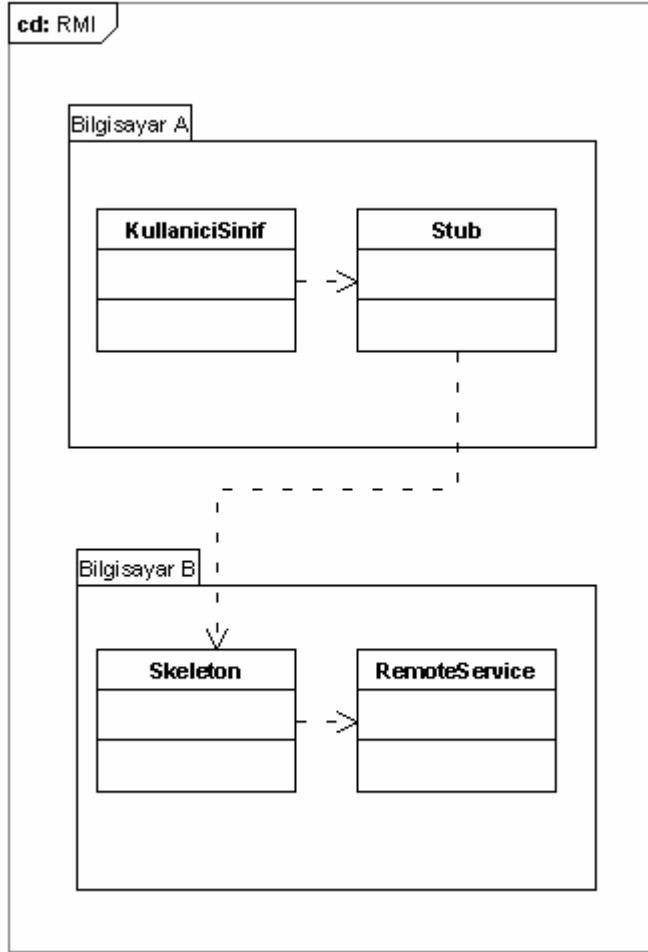
<sup>1</sup> Remote Method Invocation (RMI). Bakınız:

<http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp>

<sup>2</sup> Bakınız: <http://www.tcpsecurity.com/doc/genel/tcp.html>

aslında Bilgisayar B üzerinde bulunan nesne ile bağlantı kurmak için kendi adres alanında bulunan proxy nesneyi kullanır. Proxy nesne, Bilgisayar B üzerinde bulunan nesneye nasıl bağlantı yapması gerektiğini bilir ve **KullaniciSinif**'in isteklerini direk bu nesneye yönlendirir.

RMI kullanarak, Bilgisayar A üzerinde bulunan proxy nesneyi, Bilgisayar B üzerinde bulunan servisi ve nesneler arası bağlantıyı oluşturabiliriz. İki bilgisayar arasındaki bağlantıyı sağlamak için RMI Stub ve RMI Skeleton sınıflarının oluşturulması gerekmektedir.



Stub, Bilgisayar A üzerinde bulunan proxy nesnedir. **KullaniciSinif**'tan gelen istekleri bu proxy nesne karşılar ve metod ismini ve kullanılan parametreleri bilgisayar ağı üzerinden transfer edilebilir hale getirdikten sonra, Bilgisayar B üzerinde bulunan Skeleton nesnesine iletir. Skeleton, metod ismini ve parametreleri **RemoteService** tarafından anlaşılır bir hale getirdikten sonra verileri **RemoteService** nesnesinin gerekli metoduna iletir. Skeleton nesnesi metodun çalışmasıyla oluşan verileri ağ üzerinden Stub nesnesine, Stub nesneside bu verileri **KullaniciSinif**'a iletir. Bu şekilde iki değişik bilgisayarın hafızaları içinde bulunan nesneler arası kullanım sağlanmış olur.

Bilgisayar B üzerinde bulunan **RemoteService** isimli servis sınıfını oluşturmadan önce, bu servisin dış dünyaya sunduğu metodların bulunduğu bir remote interface sınıfının oluşturulması gerekmektedir. Bu interface sınıfı içinde, servis bünyesinde bulunan metodlar tanımlanır. Bu metodlar, Bilgisayar A üzerinde bulunan **KullaniciSinif** tarafından kullanılacak metodlardır.

Klasik merhaba dünya örneği ile RMI programın hazırlanışını yakından inceliyelim. İlk önce remote interface sınıfını oluşturuyoruz:

```
package org.javatasarim.pattern.proxy.rmi;

import java.rmi.Remote;
import java.rmi.RemoteException;

/**
 * Remote servisin metodlarının
 * tanımlandığı remote interface.
 *
 * @author Oezcan Acar
 */
public interface MerhabaDunya extends Remote
{
    /**
     * Remote metod.
     * @return String
     */
    public String merhaba() throws RemoteException;
}
```

**java.rmi.Remote**, içinde metod bulunmayan ama RMI açısından önemli bir interface sınıftır. Bizim oluşturduğumuz remote interface sınıfı **java.rmi.Remote** interface'ini extend etmek zorundadır. Sadece bu durumda bir remote interface olabilir.

Remote interface (**MerhabaDunya**) içinde tanımlanan tüm metodların **RemoteException** sınıfını kullanmaları gerekmektedir. Bilgisayar ağları üzerinden yapılan nesneler arası iletişimde birçok şey düşünüldüğü şekilde gelişebilir. İki bilgisayar arasındaki ağ çalışmaz hale gelebilir, ya da Stub başka bir nedenden dolayı Skeleton nesnesine bağlanamayabilir. Bu gibi durumların **RemoteException** sınıfı ile kontrol edilmesi gerekmektedir. Remote interface içinde tanımlanan tüm metodlarda **RemoteException** kullanılmak zorundadır.

Metodlarda kullanılan parametrelerin primitif java tipinde (int, String, long, double...) ya da zerialize (serializable) edilebilir nesnelerden oluşması gerekmektedir. Aksi takdirde parametreler bilgisayar ağı üzerinden transfer edilemez.

*merhaba()* metodunu remote servis bünyesinde implemente edilmek üzere **MerhabaDunya** interface sınıfında tanımlıyoruz. Bu metod daha sonra Bilgisayar A üzerinde bulunan **KullaniciSinif** tarafından kullanılacaktır.

Remote servisi oluşturabilmek için **MerhabaDunya** remote interface sınıfını implemente eden bir sınıfın oluşturulması gerekmektedir. Implementasyonu **MerhabaDunyaImpl** sınıfı bünyesinde gerçekleştiriyoruz.

```
package org.javatasarim.pattern.proxy.rmi;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

/**
```

```

* Remote servis implementasyonu.
*
* @author Oezcan Acar
*
*/
public class MerhabaDunyaImpl extends UnicastRemoteObject
    implements MerhabaDunya
{
    /**
     * Default konstruktör
     * @throws RemoteException
     */
    protected MerhabaDunyaImpl() throws RemoteException
    {
        super();
    }

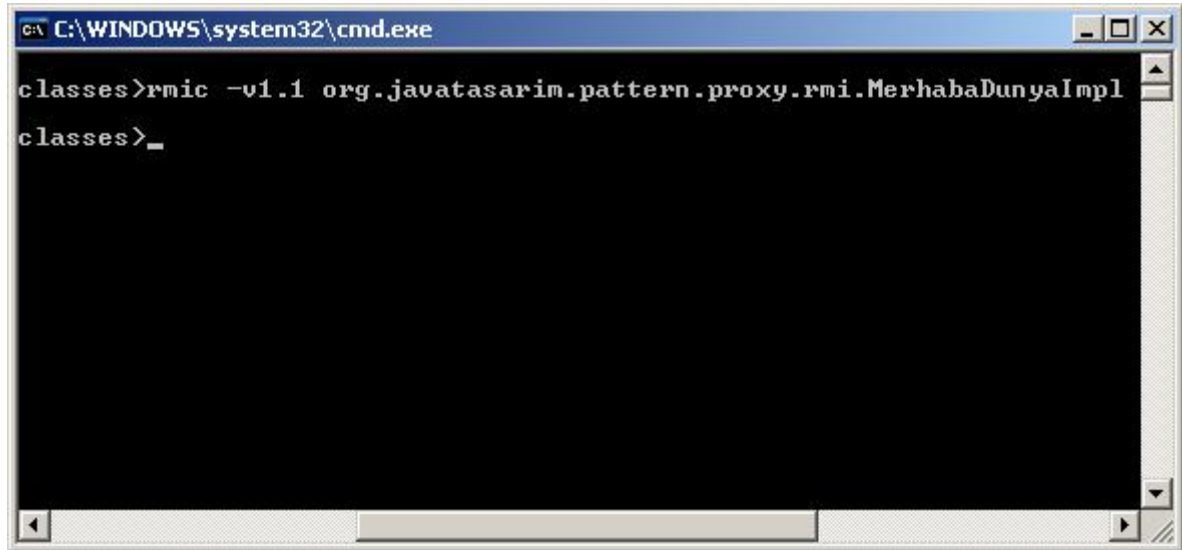
    /**
     * Implemente edilen merhaba
     * metodu.
     */
    public String merhaba() throws RemoteException
    {
        return "Merhaba Dünya";
    }
}

```

**MerhabaDunyaImpl** sınıfı **MerhabaDunya** remote interface sınıfını implemente ediyor. Bir nesnenin remote servis nesnesi olabilmesi için bazı özelliklere sahip olması gerekmektedir. Bu özellikleri edinmek için **java.rmi.server.UnicastRemoteObject** nesnesini extend etmemiz gerekiyor.

**MerhabaDunyaImpl** sınıfı bünyesinde *merhaba()* metodunu implemente ediyoruz. Bilgisayar A üzerinde bulunan **KullaniciSinif** remote proxy (RMI Stub) üzerinden *merhaba()* metodunu kullandığında, proxy tarafından bu istek **MerhabaDunyaImpl** ile aynı adres alanında bulunan RMI Skeleton nesnesine iletilecektir. Skeleton aranan remote servisi (**MerhabaDunyaImpl**) lokalize ettikten sonra, bu servisin *merhaba()* metodunu kullanacaktır.

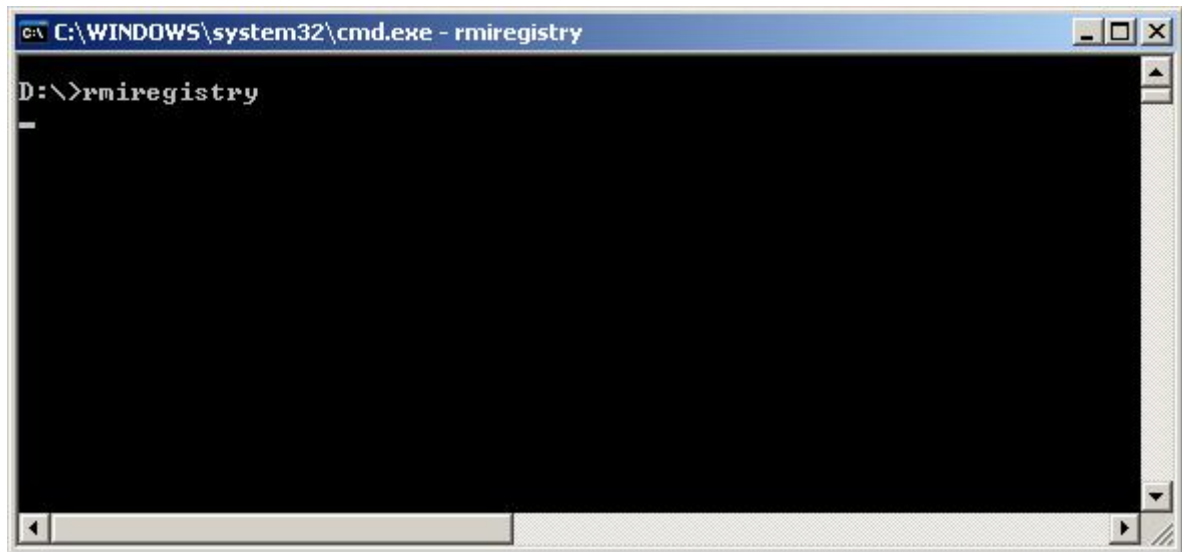
Stub ve Skeleton sınıflarını programlamamız gerekmiyor. **rmic** (jdk içinde bulunan bir program) komutu ile bu sınıfları otomatik olarak oluşturabiliriz. **MerhabaDunya.class** ve **MerhabaDunyaImpl.class** dosyalarının bulunduğu dizine giderek, *rmic MerhabaDunyaImpl* komutunu girmemiz, bu servis için Stub ve Skeleton sınıflarının oluşturulmasını sağlayacaktır.



```
C:\WINDOWS\system32\cmd.exe
classes>rmic -v1.1 org.javatasarim.pattern.proxy.rmi.MerhabaDunyaImpl
classes>_
```

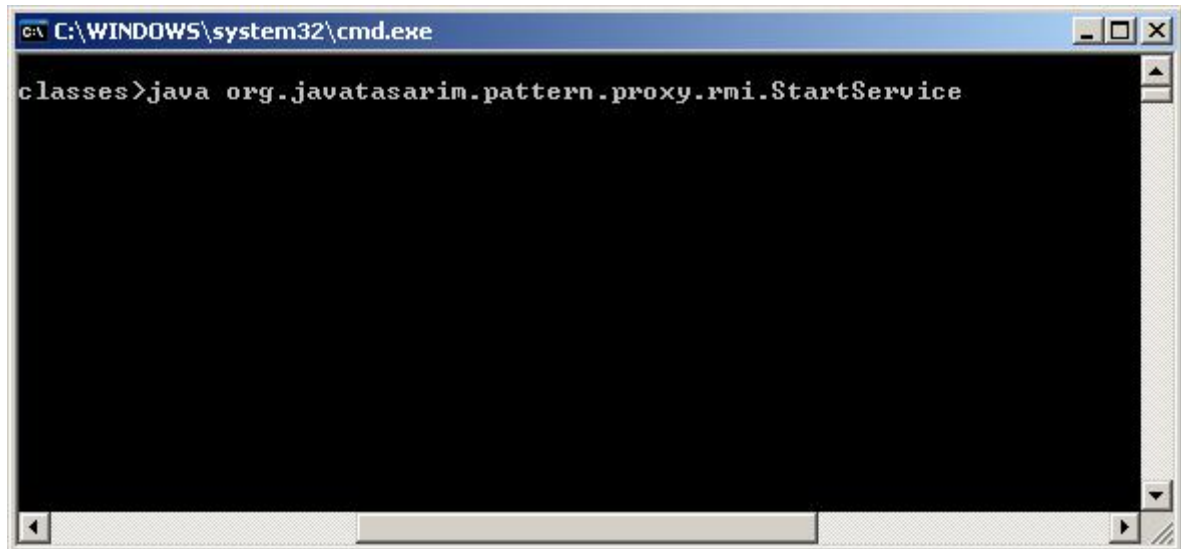
Bu işlemin ardından **MerhabaDunyaImpl\_Skel.class** ve **MerhabaDunyaImpl\_Stub.class** dosyaları oluşturulur. Yeni Java versiyonlarında Skeleton sınıfları obsolet ( gerek kalmamıştır) olmuştur ve oluşturulmazlar. rmic -v1.1 parametresi ile Skelaton sınıfın oluşturulmasını sağlayabilirsiniz.

**MerhabaDunyaImpl** sınıfından oluşan servisin bulunup, kullanılabilmesi için bir sicile (RMI Registry) kayıtlanması gerekmektedir. Jdk içinde bulunan program, **MerhabaDunyaImpl** servisinin kayıtlandığı sicili oluşturur ve Bilgisayar B üzerinde çalıştırılır.



```
C:\WINDOWS\system32\cmd.exe - rmiregistry
D:\>rmiregistry
D:\>_
```

Sicil oluşturulduktan sonra, **StartService.main()** metodu ile MerhabaDunyaImpl sınıfından bir nesne, **Naming.rebind()** komutu ile sicile eklenir. Bu andan itibaren bu nesne bulunduğu bilgisayarın IP adresi üzerinden erişilebilir hale gelir.



```
C:\WINDOWS\system32\cmd.exe

classes>java org.javatasarim.pattern.proxy.rmi.StartService
```

```
package org.javatasarim.pattern.proxy.rmi;

import java.rmi.Naming;

/**
 * MerhabaDunyaImpl nesnesini
 * olusturan ve sicile baglayan
 * program.
 *
 * @author Oezcan Acar
 */
public class StartService
{
    public static void main(String[] args)
    {
        try
        {
            MerhabaDunya md = new MerhabaDunyaImpl();
            Naming.rebind("MerhabaDunya", md);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Sicil oluřturulduktan ve **MerhabaDunyaImpl** sınıfından bir nesne oluřturulup, sicile eklendikten sonra, bu nesne **rmi://127.0.0.1/MerhabaDunya** adresi üzerinden eriřilebilir hale gelir. 127.0.0.1 yerine, bilgisayarın sahip olduėu aė adresi de kullanılabilir, orneėin 192.168.1.1.

**MerhabDunya** servisine baėlanmak iin ařaėıda yer alan Test sınıfını kullanabiliriz:



```

package org.javatasarim.pattern.proxy.rmi;

import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;

public class Test
{
    public static void main(String[] args) throws
        MalformedURLException, RemoteException,
        NotBoundException
    {
        MerhabaDunya service =
            (MerhabaDunya)Naming.lookup("rmi://127.0.0.1/" +
                "MerhabaDunya");
        System.out.println(service.merhaba());
    }
}

```

**KullaniciSinif** (örneğimizde Test sınıfı) remote servise bağlanmak için her zaman remote interface sınıfını kullanır (MerhabaDunya service=...). Bu sebepten dolayı **KullaniciSinif**, remote servisin hangi isim altında implemente edildiğini bilmek zorunda değildir (**MerhabaDunyaImpl**). Diğer tasarım şablonlarında gördüğümüz gibi, interface sınıfları kullanarak program yazılması, sınıflar arası bağı daha esnek kılar ve komponentlerin birbirlerini etkilemeden değiştirilmesine izin verir. Servis implementasyon sınıfını her zaman değiştirebiliriz. Bu durumdan **KullaniciSinif** hiçbir zaman etkilenmez ve kodunun tekrar derlenmesine gerek kalmaz.

Bilgisayar B üzerinde bulunan **MerhabaDunya** servisine bağlanmak için kullandığımız adresi yakından inceleyelim:

```

MerhabaDunya service =
    (MerhabaDunya)Naming.lookup("rmi://127.0.0.1/" +
        "MerhabaDunya");

```

- **rmi://127.0.0.1/MerhabaDunya:** RMI protokolü kullanıldığı için adresin rmi:// ile başlaması gerekmektedir. Protokol isminin ardından, servisin bulunduğu bilgisayarın IP adresi yer alır (127.0.0.1). Adresin son bölümünde **StartService.java** sınıfında tanımladığımız servisin ismi yer alır (**MerhabaDunya**). **MerhabaDunyaImpl** sınıfından bir nesneyi **StartService.main()** metodunda oluşturduktan sonra, bu nesneyi **MerhabaDunya** ismi altında sicile (rmiregistry) kayıtlamıştık. Servisin ismini istediğimiz şekilde seçebilirsiniz. **MerhabaDunya** yerine ABC isminide seçebilirdik. Bu durumda nesneye rmi://127.0.0.1/ABC adresinden ulaşabilirdik.
- **Naming.lookup(...):** Naming sınıfının *lookup()* metodu ile, sicile kayıtlı bir nesneye bağlantı gerçekleştirilir. Lookup metodu parametre olarak nesnenin RMI adresini alır. Bu adres aynı zamanda Bilgisayar B üzerinde çalışan sicilin (rmiregistry) adresidir. Naming sınıfı sicile bağlanarak, **MerhabaDunya** için oluşturulmuş olan Stub nesnesini alır. Stub nesnesi zerialize edilerek ağ üzerinden Bilgisayar A'ya gönderilir. Bu andan itibaren **KullaniciSinif** bir proxy nesneye sahiptir. Yukarda yer alan örneğimizde MerhabaDunya service =... şeklinde tanımlama ile proxy nesne için bir

değişken oluşturuyoruz, yani service ismindeki değişken Bilgisayar B den gelen Stub nesnesidir.

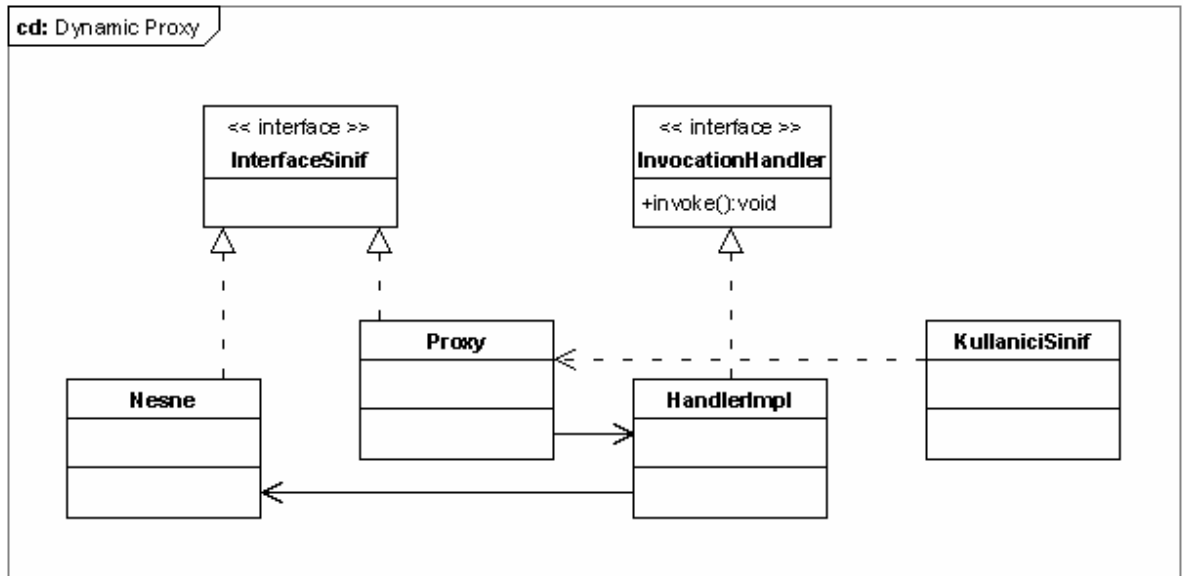
Proxy nesne oluşturulduktan sonra (**MerhabaDunya**) *merhaba()* metodunu kullanabiliriz. **KullaniciSinif** (Test) *service.merhaba()* ile şimdi kendi adres alanında bulunan servis nesnesinin (proxy) *merhaba()* metodunu kullanabilir. **KullaniciSinif** açısından servis nesnesi diğer nesnelerden farklı değildir. **KullaniciSinif** gerçek nesne ile değil de, bir proxy nesne ile çalıştığının farkına varmaz.

RMI programları yazarken dikkat edilmesi gereken hususlar vardır. Bunlar:

- Metod parametreleri ve metodların geri verdikleri değerlerin zerialize edilebilir olmaları gerekmektedir. Aksi takdirde veriler ağ üzerinden transfer edilemez. Primitif java tipleri olan String, int, long yapı itibari ile zerialize edilebilir özelliktedir. Transfer edilmek istenen diğer nesnelerin “implements Serializable” komutu kullanmaları gerekir.
- Servis sunan nesnelerin sicile (rmiregistry) kayıtlanmaları gerekmektedir. Servis nesnesini sicile kayıtlamadan önce rmiregistry ile sicil aktive edilir.

## Protection Proxy (Koruyan vekil)

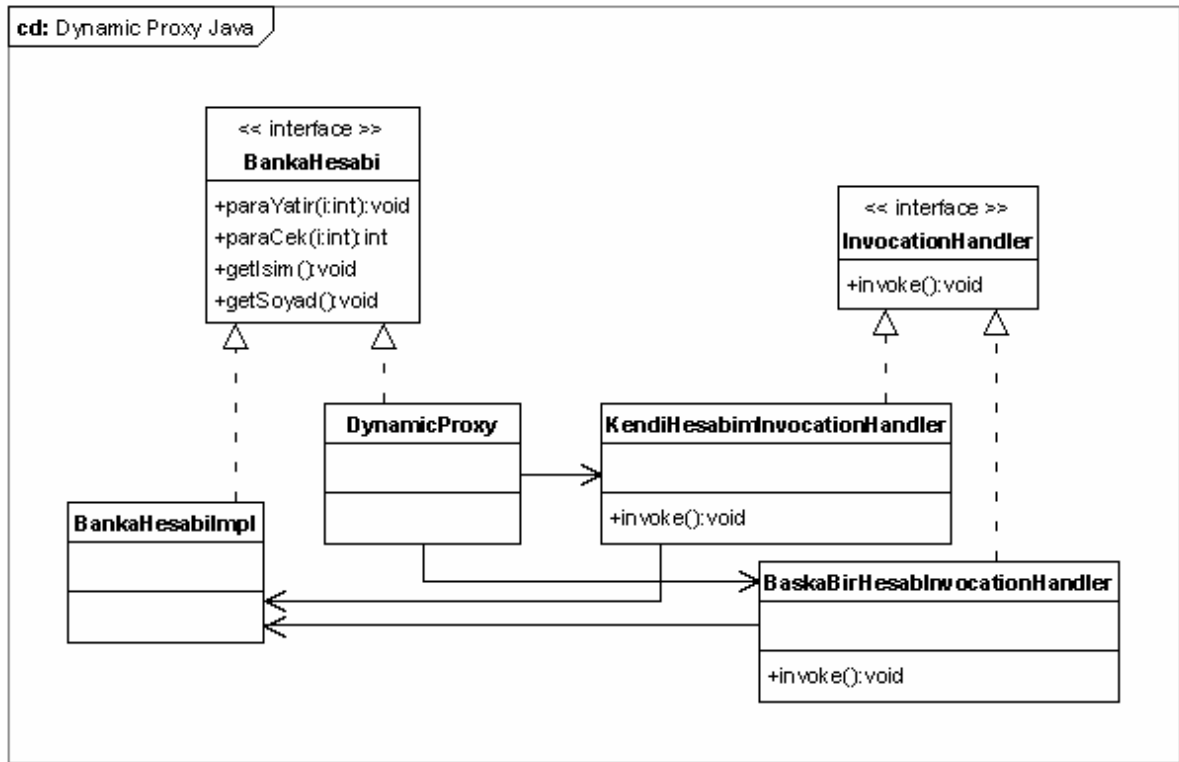
Bir nesnenin başka nesneler tarafından kontrolsüz bir şekilde kullanımını engellemek için protection proxy nesneler kullanılır. Java dilinde protection proxy nesneleri oluşturmak için **java.lang.reflect** paketinde tanımlanmış olan sınıflar kullanılır. Bu sınıflar yardımı ile programın çalışma esnasında (at runtime) proxy nesneleri oluşturulup, korunması gereken nesneler kontrol altına alınır. Kullanılan proxy nesnesi, programın çalışma esnasında (at runtime) oluşturulduğu için dinamik (dynamic) proxy ismini alır.



Uml diagramında yer alan **Nesne**, Proxy tarafından korunan nesnedir. Nesne sınıfı **InterfaceSinif**'i implemente eder. Bu durumda Nesne'yi korumak için oluşturacağımız Proxy nesnesinin de bu interface sınıfını implemente etmesi gerekmektedir. Sadece bu durumda Proxy nesnesi **Nesne** nesnesi gibi davranabilir.

Proxy nesnesi Java tarafından otomatik olarak oluşturulur. Bu sınıfı programlamamız gerekmiyor. Gerekli kodu Proxy sınıfına ekleyemeyeceğimiz için, ikinci bir sınıfın oluşturulması gerekmektedir. Java'da dynamic proxy'ler kullanabilmek için **InvocationHandler** sınıfını implemente eden bir sınıf oluşturmamız gerekmektedir. **InvocationHandler** sınıfının görevi, proxy nesnesinin isteklerini karşılamaktır. Proxy nesnesinin bir metodu kullanıldığında, bu proxy nesnesi tarafından **InvocationHandler** nesnesine yönlendirilir. Akabinde **InvocationHandler**, korunan nesnenin aynı isimdeki metodunu kullanarak, korunan nesne üzerinde gerekli işlemi gerçekleştirir.

Protection ve dynamic proxy'lerin Java'da nasıl kullanıldığını bir örnekle yakından inceleyelim:



Bir banka hesabını modellemek için **BankaHesabi** interface sınıfını tanımlıyoruz. Bu interface içinde `paraYatir()`, `paraCek()`, `getIsim()`, `getSoyad()` gibi metodlar mevcut. Bir müşteriye açılan banka hesabını temsil etmek için **BankaHesabiImpl** isminde, **BankaHesabi** interface sınıfını implemente eden bir sınıf oluşturuyoruz.

Kendi hesabıma para yatırabilir ve hesaptan para çekebilirim. Başka bir şahsın banka hesabına para yatırabilir ama hesaptan para çekemem, çünkü bu hesap bana ait değildir. Her banka müşterisi için bir **BankaHesabiImpl** nesnesi oluşturulacağından, nesnelerin kullanımının kontrol altına alınması gerekmektedir. Sistemin, benim başka bir müşterinin hesabından para çekmemi engellemesi gerekiyor. Başka bir hesaptan para çekme işlemi engellemek için bir **InvocationHandler** hazırlamamız gerekiyor. Bunun yanısıra kendi hesabım üzerinde işlemler yapabilmek için ikinci bir **InvocationHandler** sınıfına ihtiyacım var. Bunlar:

- **KendiHesabimInvocationHandler:** Bu InvocationHandler sınıfı, kendi hesabım üzerinde *paraCek()*, *paraYatir()* ve *get* ile başlayan tüm metodları kullanmama izin vermektedir.
- **BaskaBirHesabInvocationHandler:** Bu InvocationHandler sınıfı, başka bir şahsın hesabından para çekmeme izin vermez. Başka bir hesaba para yatırabilirim.

BankaHesabi interface sınıfı aşağıdaki yapıdadır:

```
package org.javatasarim.pattern.proxy.dynamic;

/**
 * Bir banka hesabini modelleyen
 * interface sınıf.
 *
 * @author Oezcan Acar
 */
public interface BankaHesabi
{
    public int paraCek(int miktar) throws Exception;
    public void paraYatir(int miktar);
    public String getIsim();
    public void setIsim(String isim);
    public String getSoyad();
    public void setSoyad(String soyad);
    public int getHesapDurumu();
    public void setHesapDurumu(int hesapDurumu);
    public boolean iptalEt();
}
```

**BankaHesabiImpl** sınıfı **BankaHesabi** interface sınıfını implemente ederek, müşteriler için banka hesabı oluşturulmasında kullanılır.

```
package org.javatasarim.pattern.proxy.dynamic;

/**
 * Bir sahsin banka hesabini temsil
 * eden sınıf.
 *
 * @author Oezcan Acar
 */
public class BankaHesabiImpl implements BankaHesabi
{
    private String isim;
    private String soyad;
    private int hesapDurumu = 0;
    private boolean iptalEdildi=false;

    public BankaHesabiImpl(String isim, String soyad,
                           int miktar)
    {
        setIsim(isim);
        setSoyad(soyad);
        setHesapDurumu(miktar);
    }

    public BankaHesabiImpl()
    {
    }
}
```

```

{

}

/**
 * Belirli bir miktar parayi
 * hesaptan ceker.
 * @param miktar int cekilen para
 * @return miktar int cekilen para
 * @throws Exception
 */
public int paraCek(int miktar) throws Exception
{
    if(getHesapDurumu()-miktar > 0)
    {
        setHesapDurumu(getHesapDurumu()-miktar);
    }
    else
    {
        throw new Exception("Hesabinizda yeterinde " +
            "para bulunmuyor.");
    }
    return miktar;
}

/**
 * Hesaba bir miktar para yatirmek için kullanılan
 * metod.
 * @param miktar int yatırılan para
 */
public void paraYatir(int miktar)
{
    setHesapDurumu(getHesapDurumu()+miktar);
}

/**
 * Bir banka hesabini iptal etmek
 * için kullanılan metod. Sadece
 * banka calisanlari tarafından
 * bu metod kullanılabilir.
 * Hesap sahibi kendi hesabini
 * iptal edemez.
 */
public boolean iptalEt()
{
    setIptalEdildi(true);
    return isIptalEdildi();
}

public String getIsim()
{
    return isim;
}

public void setIsim(String isim)
{
    this.isim = isim;
}

```

```

    public String getSoyad()
    {
        return soyad;
    }

    public void setSoyad(String soyad)
    {
        this.soyad = soyad;
    }

    public int getHesapDurumu()
    {
        return hesapDurumu;
    }

    public void setHesapDurumu(int hesapDurumu)
    {
        this.hesapDurumu = hesapDurumu;
    }

    public boolean isIptalEdildi()
    {
        return iptalEdildi;
    }

    public void setIptalEdildi(boolean iptalEdildi)
    {
        this.iptalEdildi = iptalEdildi;
    }
}

```

Kendi banka hesabım üzerinde işlem yapabilmek için **KendiHesabimInvocationHandler** isminde bir **InvocationHandler** oluşturuyorum. Bu **InvocationHandler** sınıfı, kendi banka hesabım üzerinde gerekli tüm işlemleri yapmama izin verecek şekilde programlanacak. Sistem tarafından otomatik olarak oluşturulacak Proxy nesnesi, bu sınıfın *invoke()* metodunu kullanarak, hesap nesnesi üzerinde işlem yapacaktır.

```

package org.javatasarim.pattern.proxy.dynamic;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

/**
 * Kendi banka hesabım üzerinde
 * gerekli işlemlerin yapılmasına
 * izin veren InvocationHandler sınıfı
 *
 * @author Oezcan Acar
 */
public class KendiHesabimInvocationHandler
    implements InvocationHandler
{
    private BankaHesabi hesap;

    public KendiHesabimInvocationHandler(BankaHesabi bh)
    {

```

```

        setHesap(bh);
    }

    /**
     * Dynamic proxy, invoke metodunu kullanarak,
     * hesap nesnesi üzerinde işlem yapılabilir.
     */
    public Object invoke(Object proxy, Method method,
        Object[] args)
        throws Throwable
    {
        try
        {
            if(method.getName().startsWith("paraYatir"))
            {
                return method.invoke(getHesap(), args);
            }
            else if(method.getName().startsWith("paraCek"))
            {
                return method.invoke(getHesap(), args);
            }
            else if(method.getName().startsWith("get"))
            {
                return method.invoke(getHesap(), args);
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
        return null;
    }

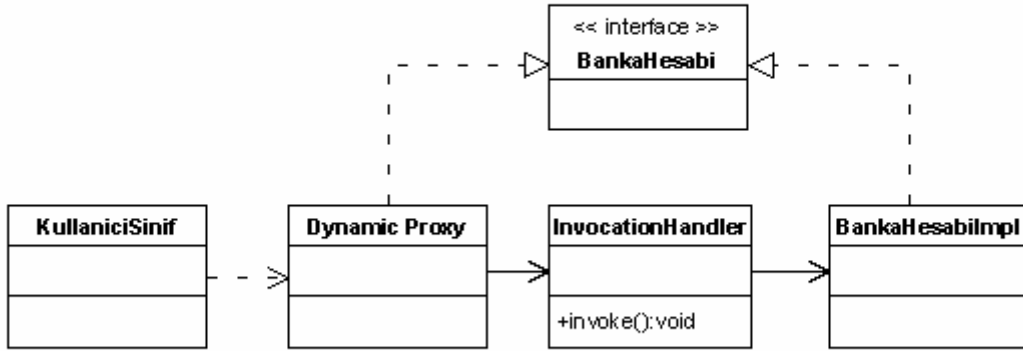
    public BankaHesabi getHesap()
    {
        return hesap;
    }

    public void setHesap(BankaHesabi hesap)
    {
        this.hesap = hesap;
    }
}

```

Nesnelerin ve sahip oldukları metodların kullanım sırasını tekrar bir Uml diagramında görelim:

cd: Dynamic Proxy Java 2



Herhangi bir kullanıcı sınıf (**KullaniciSinif**) sistem tarafından oluşturulmuş dynamic proxy üzerinde herhangi bir **BankaHesabi** metodunu kullanmak istiyor. Dynamic proxy, **BankaHesabi** interface sınıfını implemente ettiği için, **KullaniciSinif** bu interface sınıfında tanımlanan tüm metodları kullanabilir, örneğin *paraYatir()*. Proxy nesnesinin herhangi bir metodu kullanılmak istendiğinde, proxy bu isteği otomatik olarak sahip olduğu **InvocationHandler** sınıfına delege eder, yani **InvocationHandler** sınıfının *invoke()* metodu kullanır.

Daha sonra detaylı olarak göreceğimiz Test sınıfı bünyesinde, aşağıdaki şekilde bir dynamic proxy nesnesi oluşturabiliriz.

```
protected BankaHesabi getKendiHesabimProxy(
    BankaHesabi hesap)
{
    return (BankaHesabi)Proxy.newProxyInstance(
        hesap.getClass().getClassLoader(),
        hesap.getClass().getInterfaces(),
        new KendiHesabimInvocationHandler(hesap));
}
```

**InvocationHandler** ile korumak istediğimiz nesne (**BankaHesabiImpl**) arasındaki bağı oluşturmak için, **KendiHesabimInvocationHandler** konstruktörüne korunmasını istediğimiz **BankaHesabi** nesnesini parametre olarak veriyoruz. **KendiHesabimInvocationhandler** sınıfı, proxy den gelen tüm istekleri, bünyesinde barındırdığı **BankaHesabi** nesnesine delege eder.

Şimdi **InvocationHandler** içinde bir banka hesap nesnesinin nasıl korunduğunu inceliyelim. Hazırladığımız **InvocationHandler** tipine göre, bazı metodların kullanılmasını engelliyebilir, bazılarının kullanılmasına izin verebiliriz. Bu işlemi **InvocationHandler.invoke()** metodu içinde gerçekleştiriyoruz:

```
/**
 * Dynamic proxy invoke metodunu kullanarak,
 * hesap nesnesi üzerinde işlem yapabilir.
 */
public Object invoke(Object proxy, Method method,
```



```

        Object[] args)
        throws Throwable
    {
        try
        {
            if(method.getName().startsWith("paraYatir"))
            {
                return method.invoke(getHesap(), args);
            }
            else if(method.getName().startsWith("paraCek"))
            {
                return method.invoke(getHesap(), args);
            }
            else if(method.getName().startsWith("get"))
            {
                return method.invoke(getHesap(), args);
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
        return null;
    }

```

Yukarda yer alan satırlar **KendiHesabimInvocationHandler** sınıfından alıntıdır. Proxy tarafından **InvocationHandler.invoke()** metodu kullanıldığında proxy'nin kendisi, proxy üzerinde kullanılan metod ve parametre listesi *invoke()* metoduna iletilir. Bu veriler sayesinde **InvocationHandler**, korunan nesnenin hangi metodunu hangi parametrelerle kullanılması gerektiğini tespit edebilir.

**method.getName()** ile, proxy üzerinde kullanılan metodun ismini tespit edebiliriz. Simdi **InvocationHandler** sınıfının istenilen metodu korunan nesneye nasıl ilettiğini görelim:

```

if(method.getName().startsWith("paraYatir"))
{
    return method.invoke(getHesap(), args);
}

```

Eger proxy üzerinde **proxy.paraYatir()** metodu kullanıldı ise, bunu if ile kontrol edip, tespit ediyoruz. Hesap nesnesi **InvocationHandler** sınıfında sınıf değişkeni olarak mevcut olduğundan (nesneyi InvocationHandler konstruktörüne parametre olarak vermiştik) *getHesap()* ile bu nesneye ulaşabiliriz. Örneğin **proxy.paraYatir(1000)** şeklinde bir metod kullanımı olmuşsa, *method* değişkeninde “paraYatir” String’i, *args* içinde 1000 int degeri olacaktır.

```

return method.invoke(getHesap(), args);
= (esittir)
return getHesap().paraYatir(1000);

```

**InvocationHandler** hesap nesnesinin metodunu kullandıktan sonra, neticeyi proxy nesnesine geri verir ve görevini tamamlamış olur.

**KendiHesabimInvocationHandler**.*invoke()* metodu içinde *paraYatir()*, *paraCek* ve *get()* ile başlayan tüm metodların kullanımına izin verilmektedir. Başka bir şahsın hesabından para çekilmesini engellemek için **BaskaBirHesapInvocationHandler** sınıfını oluşturuyoruz.

```
package org.javatasarim.pattern.proxy.dynamic;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

/**
 * Baska bir hesap üzerinde
 * para cekme islemini
 * engelleyen InvocationHandler sinifi
 *
 * @author Oezcan Acar
 */
public class BaskaBirHesapInvocationHandler
    implements InvocationHandler
{
    private BankaHesabi hesap;

    public BaskaBirHesapInvocationHandler(BankaHesabi bh)
    {
        setHesap(bh);
    }

    /**
     * Dynamic proxy invoke metodunu kullanarak,
     * hesap nesnesi üzerinde islem yapabilir.
     */
    public Object invoke(Object proxy, Method method,
        Object[] args)
        throws Throwable
    {
        try
        {
            if(method.getName().startsWith("paraYatir"))
            {
                return method.invoke(getHesap(), args);
            }
            else if(method.getName().startsWith("paraCek"))
            {
                throw new Exception("Baska bir hesaptan " +
                    "para cekemezsiniz!");
            }
            else if(method.getName().startsWith("get"))
            {
                return method.invoke(getHesap(), args);
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

```

        return null;
    }

    public BankaHesabi getHesap()
    {
        return hesap;
    }

    public void setHesap(BankaHesabi hesap)
    {
        this.hesap = hesap;
    }
}

```

*invoke()* metodunda yer aldığı gibi, *paraYatir()* metodunun kullanımına izin verilmiştir, ama para çekme işlemi yasaklanmıştır. Eğer başka bir hesaptan para çekmeye çalışırsak, bu durumda bir Exception ile uyarı alırız.

Her örneğimizde olduğu gibi bir Test sınıfı ile, hazırladığımız programı test edelim:

```

package org.javatasarim.pattern.proxy.dynamic;

import java.lang.reflect.*;

/**
 * Test programi.
 *
 * @author Oezcan Acar
 */
public class Test
{
    /**
     * Main
     * @param args
     * @throws Exception
     */
    public static void main(String[] args) throws Exception
    {
        Test test = new Test();
        test.start();
    }

    /**
     * Start
     * @throws Exception
     */
    public void start() throws Exception
    {
        BankaHesabi hesap = new BankaHesabiImpl("Ahmet",
            "Yildirim", 1000);
        BankaHesabi kendiHesabimProxy =
            getKendiHesabimProxy(hesap);
        kendiHesabimProxy.paraYatir(2000);
        kendiHesabimProxy.paraCek(100);

        System.out.println("Hesabtaki para miktarı: " +
            kendiHesabimProxy.getHesapDurumu());
    }
}

```

```

        BankaHesabi baskaHesapProxy =
            getBaskaHesapProxy(hesap);

        baskaHesapProxy.paraYatir(500);
        baskaHesapProxy.paraCek(200);
    }

    /**
     * Kendi banka hesabim üzerinde
     * islem yapabilmek için oluşturulan
     * dynamic proxy.
     *
     * @param hesap BankaHesabi hesap
     * @return BankaHeabi dynamic proxy
     */
    protected BankaHesabi getKendiHesabimProxy(
        BankaHesabi hesap)
    {
        return (BankaHesabi)Proxy.newProxyInstance(
            hesap.getClass().getClassLoader(),
            hesap.getClass().getInterfaces(),
            new KendiHesabimInvocationHandler(hesap));
    }

    /**
     * Baska bir sahsin banka hesabim
     * üzerinde islem yapabilmek için
     * oluşturulan dynamic proxy.
     *
     * @param hesap BankaHesabi hesap
     * @return BankaHeabi dynamic proxy
     */
    protected BankaHesabi getBaskaHesapProxy(
        BankaHesabi hesap)
    {
        return (BankaHesabi)Proxy.newProxyInstance(
            hesap.getClass().getClassLoader(),
            hesap.getClass().getInterfaces(),
            new BaskaBirHesapInvocationHandler(hesap));
    }
}

```

Test sınıfı çalıştırıldığında, aşağıdaki ekran görüntüsü oluşacaktır:

```

Hesabtaki para miktarı: 2900
java.lang.Exception: Baska bir hesaptan para cekemezsiniz!
    at
org.javatasarim.pattern.proxy.dynamic.BaskaBirHesapInvocationHandler.
invoke(BaskaBirHesapInvocationHandler.java:41)
    at $Proxy0.paraCek(Unknown Source)
    at org.javatasarim.pattern.proxy.dynamic.Test.start(Test.java:46)
    at org.javatasarim.pattern.proxy.dynamic.Test.main(Test.java:22)
Exception in thread "main" java.lang.NullPointerException
    at $Proxy0.paraCek(Unknown Source)
    at org.javatasarim.pattern.proxy.dynamic.Test.start(Test.java:46)
    at org.javatasarim.pattern.proxy.dynamic.Test.main(Test.java:22)

```

*start()* metodu bünyesinde Ahmet Yıldırım isimli bir şahsın banka hesabını ediniyoruz. **KendiHesabimInvocationHandler** yardımı ile bir dynamic proxy oluşturduğumuz için, bu hesap üzerinde birçok işlemi gerçekleştirebiliriz. Örnekte yer aldığı gibi önce hesaba 2000 TL yatırıyor ve akabinde 100 TL hesaptan çekiyoruz. **KendiHesabimInvocationHandler** kullanıldığı için Ahmet Yıldırım kendi hesabına para yatırabiliyor ve çekebiliyor. Daha sonra **BaskaBirHesapInvocationHandler** yardımı ile başka bir şahsın banka hesabı üzerinde işlem yapıyoruz. *paraYatir()* metodunu kullanarak bu hesaba 500 TL yatırıyoruz. *paraCek()* metodunu kullanmak istediğimizde yukarda yer alan Exception oluşuyor, çünkü **BaskaBirHesapInvocationHandler** yabancı bir hesaptan para çekmemizi *invoke()* metodunda engelliyor.

Test sınıfının ekran çıktısında dikkat çeken bir nokta daha bulunmaktadır:

```
at $Proxy0.paraCek(Unknown Source)
```

\$Proxy0, Java Reflection API tarafından oluşturulan dynamic proxy sınıfıdır. Stacktrace içinde görüldüğü gibi, Test sınıfının 46. satırında sonra işlem \$Proxy0 sınıfına devredilmektedir. Test sınıfının 46. satırında aşağıdaki komut yer almaktadır:

```
baskaHesapProxy.paraCek(200);
```

Stacktrace içinde görüldüğü gibi *baskaHesapProxy.paraCek(200)*, *\$Proxy0.paraCek()* halini almıştır. *\$Proxy0.paraCek()* satırının bir üstünde **BaskaBirHesapInvocationHandler** sınıfının 41. satırında bir Exception oluşmuştur. Bu *invoke()* metodunda *paraCek()* metodunun kontrol edildiği satırdır. **BaskaBirHesapInvocationHandler** yabancı bir hesaptan para çekilmesine izin vermediği için, bu sınıfın 41. satırında bir Exception oluşmaktadır. Stacktrace bünyesinde yer aldığı gibi, kullandığımız program \$Proxy0 nesnesinin metodunu kullanmakta, \$Proxy0 bu işlemi kullanılan **InvocationHandler** sınıfına devretmekte ve **InvocationHandler** sınıfıda korunan nesnenin gerekli metodunu işletmektedir.

Proxy tasarım şablonu ne zaman kullanılır?

- Başka bir bilgisayar (server) üzerinde bulunan bir nesneye ulaşmak için remote proxy tasarım şablonu kullanılır.
- Oluşturulması zaman alıcı ve sistem kaynaklarını zorlayan nesnelere vekil olmak için virtual proxy tasarım şablonu kullanılır.
- Bir nesneyi ve metodlarını dışardan erişimlere karşı korumak için protection proxy tasarım şablonu kullanılır.

İlişkili tasarım şablonları:

- Adapter: Adapter tasarım şablonu adapte ettiği nesnenin metod isimlerini değiştirir. Proxy tasarım şablonu ise, kullandığı ve koruduğu nesne ile aynı metodlara sahiptir çünkü nesne ile aynı interface sınıfını implemente eder. Bu protection proxy nesneleri için geçerli değildir. Protection proxy, bazı metodların kullanılmasını engelleyebilir. Bu gibi durumlarda protection proxy, koruduğu nesnenin sahip olduğu metodların bir alt kümesine sahip olacaktır.

- Decorator: Decorator ve proxy nesneleri birbirlerine benzer implemente ediliyor olmalarına rağmen, iki tasarım şablonu değişik amaçlar için kullanılır. Decorator tasarım şablonu bir nesneye yeni metodlar eklerken, proxy tasarım şablonu bir nesneyi korumak için kullanılır.

*EOF (End Of Fun)*

*Özcan Acar*