

# Penggunaan Proxy Object Dan Command Pattern Untuk Mengembangkan Service Layer Yang Handal

**Bernardus Irmanto**

birmanto@yahoo.com

<http://www.javaroller.com/page/crosshank>

## ***Lisensi Dokumen:***

*Copyright © 2003 IlmuKomputer.Com*

*Seluruh dokumen di IlmuKomputer.Com dapat digunakan, dimodifikasi dan disebarkan secara bebas untuk tujuan bukan komersial (nonprofit), dengan syarat tidak menghapus atau merubah atribut penulis dan pernyataan copyright yang disertakan dalam setiap dokumen. Tidak diperbolehkan melakukan penulisan ulang, kecuali mendapatkan ijin terlebih dahulu dari IlmuKomputer.Com.*

Dalam tahap analisis dan design sebuah aplikasi, kita pasti akan menemukan banyak business object yang membentuk sebuah kesatuan fungsional yang merupakan representasi dari requirement yang telah disetujui oleh calon pengguna aplikasi. Di antara business object-business object tersebut, ada beberapa yang mempunyai kelakuan dan struktur yang boleh dikatakan sama, sehingga kita memutuskan untuk membentuk sebuah interface yang mendefinisikan kelakuan dan struktur standard yang harus diikuti oleh business object yang bersangkutan. Seringkali juga kita menemui bahwa business object-business object tersebut mempunyai responsibility untuk melakukan fungsi-fungsi yang serupa. Sebagai contoh, misalkan kita sedang mengembangkan sebuah editor gambar dan text. Kita akan mengidentifikasi text dan gambar sebagai business object dalam aplikasi tersebut. Kedua object tersebut ternyata juga mempunyai responsibility serupa dalam konteks aplikasi seperti: draw dan delete. Problematika yang sering kita hadapi dalam mengembangkan aplikasi seperti ini adalah bagaimana kita bisa membuat sebuah mekanisme yang robust dan elegan untuk mengakses object-object dan fungsi-fungsi yang disediakan. Seringkali kita memutuskan untuk membuat class-class yang independent untuk setiap business object, dan menyediakan sebuah façade atau abstract factory sebagai pintu tunggal untuk mengelola life time dari object dan mengakses fungsi-fungsi yang disediakan object. Strategi itu tidaklah salah, dan sudah banyak digunakan dalam pengembangan aplikasi-aplikasi yang ada.

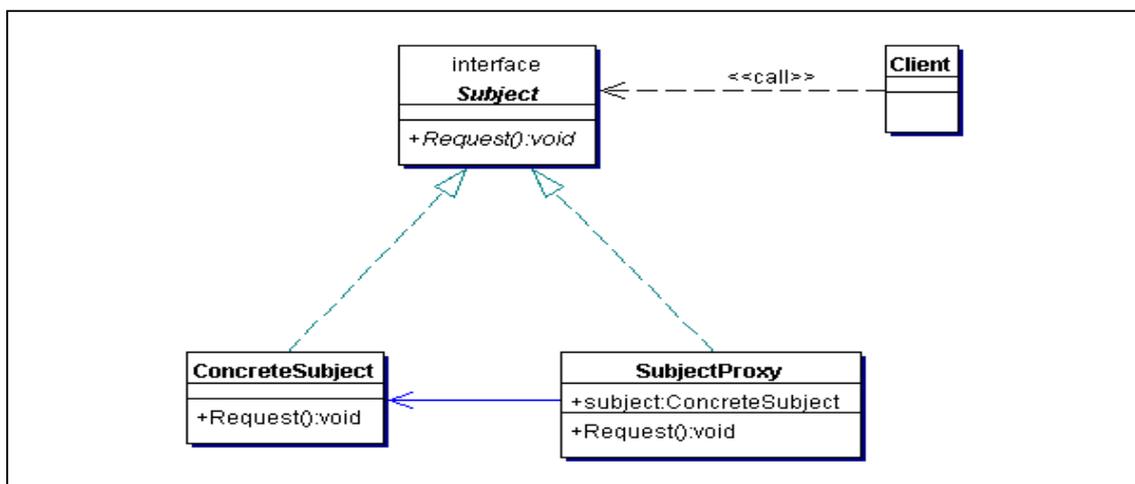
Dalam tulisan ini, akan diperkenalkan kemungkinan lain untuk membangun sebuah service layer yang handal dan robust sekaligus elegan, yaitu dengan menggunakan proxy object dan command pattern (GOF). Bahasa pemrograman java akan digunakan dalam tulisan ini untuk menerangkan konsep-konsep yang berkenaan dengan proxy object, command pattern dan service layer.

## PROXY OBJECT

Di dalam dunia komputasi, kita mengenal adanya proxy object. Proxy menyediakan sebuah mekanisme dimana pemanggilan fungsi yang disediakan oleh suatu object tidak dilakukan secara langsung, melainkan melalui proxy. Proxy berfungsi sebagai gerbang akses untuk object object yang berada di belakang proxy.

Penggunaan object proxy sering ditemui dalam berbagai aplikasi, baik sebagai façade, remote proxy, access proxy dan virtual proxy. Sebagai façade, proxy menyediakan interface tunggal untuk mengakses berbagai object. Dengan façade, akses ke fungsi yang disediakan object-object tersebut menjadi lebih mudah di pelihara, dan pengguna tidak perlu tahu detail dari fungsi-fungsi yang diakses. Remote proxy, digunakan untuk menyediakan sebuah mekanisme akses terhadap object-object yang baik secara konsep maupun fisik terletak 'jauh' dari object pemanggil. Dengan mekanisme ini, pengguna akan melihat object-object tersebut seakan-akan berada dalam proses yang sama. Aplikasi remote proxy banyak kita temui dalam aplikasi dengan arsitektur terdistribusi. Access proxy adalah proxy yang digunakan sebagai filter keamanan terhadap object-object tertentu. Proxy tersebut akan mengaplikasikan security policy terhadap semua akses terhadap object. Yang terakhir, virtual proxy, menyediakan sebuah mekanisme yang sering disebut sebagai 'place holder'. Proxy ini akan menyiapkan segala sesuatu yang dibutuhkan oleh object yang sesungguhnya, namun object tersebut sesungguhnya belum terbentuk. Baru pada saat di butuhkan, object tersebut akan di instantiate.

Java (mulai versi 1.3) menyediakan dynamic proxy API di dalam paket java.lang.reflect.Proxy. Disebut dynamic, karena java proxy object tersebut mempunyai kemampuan untuk mengelola akses ke berbagai interface yang berbeda. Dengan demikian java proxy object mendukung konsep 'loose couple', satu hal yang ditengarai oleh banyak pihak sebagai kelemahan dari proxy pattern (GOF). Dalam proxy pattern, implementasi dari proxy object sangat terkait dengan implementasi object yang akan diakses melalui proxy tersebut. Dalam UML diagram di gambar 1 dapat kita lihat, untuk setiap object, kita perlu mendefinisikan proxy object yang spesifik untuk object tersebut.



Gambar 1. Class diagram, proxy pattern(GOF) [<http://www.dofactory.com/Patterns>]

Dalam java proxy object, kelemahan tersebut diatasi dengan kemampuan untuk sifat dinamis yang dimilikinya. Untuk mendapatkan referensi kepada concrete object menggunakan java dynamic proxy, digunakan kode spt dibawah ini:

```
MyProxyInterface o = (MyProxyInterface)
    java.lang.reflect.Proxy.newProxyInstance(obj.getClass() .
    getClassLoader(), Class[] {MyProxyInterface.class}, new
```

```
MyDynamicProxyClass(obj));
```

Dalam fragmen kode diatas, untuk mendapatkan instance dari object proxy, kita harus memanggil static method `newProxyInstance` dari `java.lang.reflect.Proxy` class. Ada tiga argumen masukan dari method tersebut, yaitu :

- Class Loader dari concrete object yang akan di refer
- Interface yang diimplementasikan oleh concrete object
- Dynamic proxy object, yang mengimplementasikan `java.lang.reflect. InvocationHandler` interface.

Sifat dinamis dari proxy didapatkan dari kemampuan untuk menerima berbagai interface dan dari method `invoke` didefinisikan dalam interface `java.lang.reflect. InvocationHandler` dan harus diimplementasikan oleh dynamic proxy object. Method `invoke` tersebut mempunyai signature sebagai berikut:

```
public Object invoke(Object proxy, Method m, Object[] args)
    throws Throwable
```

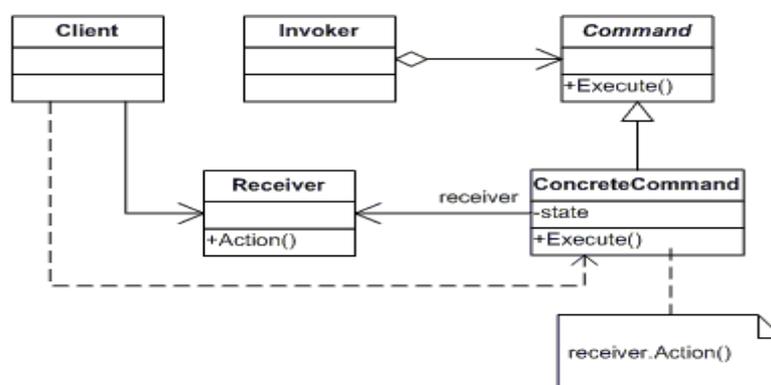
Yang harus di ingat adalah bahwa `newProxyInstance` method akan mengembalikan proxy object, bukan object yang sebenarnya. Proxy object ini bisa di cast menjadi tipe object yang kita inginkan. Pada saat kita mengeksekusi suatu fungsi yang disediakan oleh object tersebut, `invoke` method dari dynamic proxy object akan dipanggil untuk melakukan eksekusi. Didalam method `invoke`, pemanggilan fungsi tersebut akan diterjemahkan menjadi pemanggilan fungsi dari concrete object.

```
Object result = m.invoke(obj, args);
```

Dari gambaran diatas, kita bisa mengetahui bahwa dynamic proxy object menyediakan mekanisme yang elegan, scalable, dan maintainable untuk mengelola akses ke berbagai object dan fungsi yang disediakan oleh object tersebut. Untuk penjelasan yang lebih detail mengenai java proxy object bisa dibaca di <http://java.sun.com/j2se/1.4.1/docs/api/java/lang/reflect/Proxy.html>

## COMMAND PATTERN

Command pattern di dalam buku GOF dijelaskan sebagai sebuah mekanisme untuk mengenkapsulasi request sebagai sebuah object, sehingga penanganan akan request bisa dikelola secara terpusat, dengan menyediakan method khusus dengan request object sebagai parameter masukan. Selain itu command pattern juga memungkinkan mekanisme penantrean request dan request logging serta fasilitas undo.



Gambar 2. Class diagram, command pattern(GOF) [<http://www.dofactory.com/Patterns>]

Kalau kita tidak menggunakan command pattern, setiap request yang datang akan di cek dalam suatu block 'switch case' untuk menangani request tersebut. Setiap kali kita akan menambah jenis request yang bisa ditangani, kita harus menambah code pada blok 'switch-case' tersebut. Sebaliknya, dengan menggabungkan sifat polimorphism dari command pattern dan kemampuan dynamic loading dan mekanisme binding yang unik dari bahasa pemrograman java, kita bisa mengembangkan sistem yang lebih reliable dan extensible.

Di dalam diagram uml pada gambar 2, bisa kita lihat bahwa kunci dari command pattern adalah interface command. Interface command mendeklarasikan sebuah method : execute, yang harus di implementasikan oleh semua concrete class. Concrete command akan mendelegasikan penanganan request kepada receiver. Dalam hal ini, concrete command menjadi adapter antara invoker dan receiver.

Implementasi command pattern dengan menggunakan bahasa pemrograman java cukup sederhana. Berikut adalah sebuah contoh implementasi command pattern sederhana.

```
public interface Command {
    public abstract void execute();
}

public interface myObj{
    public abstract void draw();
    public abstract void delete()
}

public class Text implements myObj{
    public void draw(){
        System.out.println("Drawing a text")
    }

    public void delete(){
        System.out.println("Deleting a text")
    }
}

public class Image implements myObj{
    public void draw(){
        System.out.println("Drawing an image")
    }

    public void delete(){
        System.out.println("Deleting an image")
    }
}

public class DrawCommand implements Command{
    private myObj obj;

    public setObj(myObj o){
        obj=o;
    }

    public void execute(){
        obj.draw();
    }
}
```

```
}  
  
public class DeleteCommand implements Command{  
    private myObj obj;  
  
    public setObj(myObj o){  
        obj=o;  
    }  
  
    public void execute(){  
        obj.draw();  
    }  
  
}  
  
public class Painter {  
    Command drawCmd;  
    Command delCmd;  
  
    public Painter(Command cmd1, Command cmd2){  
        drawCmd=cmd1;  
        delCmd=cmd2;  
    }  
  
    public void draw(){  
        drawCmd.execute();  
    }  
  
    public void delete(){  
        delCmd.execute();  
    }  
  
}
```

Apabila kita ingin memanggil salah satu fungsi yang disediakan oleh salah satu receiver, kita harus meng-instantiate concrete command class yang bersangkutan. Sebagai contoh, apabila kita ingin mendelete sebuah image, kita bisa melakukannya seperti dibawah ini.

```
Image myImg = new Image();  
DeleteCommand delete = new DeleteCommand(myImg);  
DrawCommand draw = new DrawCommand(myImg);  
Painter paint = new Painter(draw,delete);  
Paint.delete();
```

## **APLIKASI PROXY DAN COMMAND PATTERN DALAM SERVICE LAYER**

Dalam mengembangkan aplikasi, pasti kita mempunyai object-object yang bertugas untuk menyediakan layanan bisnis untuk object-object yang lain. Layanan bisnis yang disediakan bisa berupa manipulasi informasi, kalkulasi bisnis dan lain sebagainya. Peng-enkapsulasi-an object object yang menyediakan layanan bisnis dalam layer terpisah (service layer) sudah menjadi best-practice dalam pengembangan aplikasi berskala menengah dan besar. Hal tersebut dimaksudkan supaya dependensi terhadap business object bisa di minimalkan, serta skalabilitas dan kemudahan dalam

pemeliharaan bisa ditingkatkan.

Mengambil studi kasus tentang aplikasi editor tetxt dan gambar yang telah disebutkan dalam bagian awal artikel ini, kita bisa mendefinisikan suatu service layer yang mengenkapsulasikan layanan 'draw' dan 'delete'. Kedua layanan tersebut akan dideklarasikan dalam sebuah interface myPaintObjMgr

```
public interface myPaintObjMgr {
    public void draw();
    public void delete();
    public void move();
    ...
}
```

Semua object yang akan dimanipulasi oleh aplikasi, akan mengimplementasikan interface diatas. Untuk merefer pada object yang sebenarnya, dibutuhkan sebuah service locator. Service locator akan mengembalikan concrete object dari myPaintObjMgr setiap kali dibutuhkan

```
Public interface ServiceLocator {
    Public Object getPaintObjMgr(Class myPaintObjMgr) throws
    ServiceLocatorException;
}
```

Karena untuk merefer seluruh concrete object didalam aplikasi kita hanya memerlukan satu locator, kita bisa mendefinisikan sebuah singleton class untuk mendapatkan instance dari ServiceLocator

```
public class ServiceLocatorMgr {

    private static ServiceLocatorImpl svc;

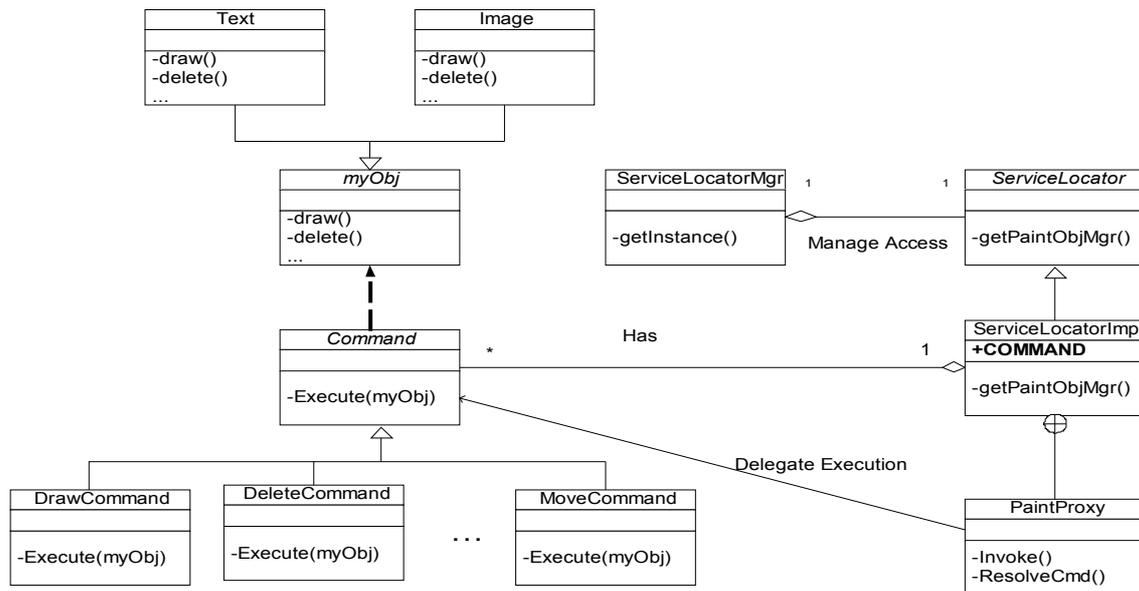
    public ServiceLocator getInstance(){ // synchronize the
call if needed

if(svc!=null)
    return svc;
}
else
{
    svc = new ServiceLocatorImpl();
    return svc;
}
}

}
```

ServiceLocatorImpl adalah class yang mengimplementasikan interface ServiceLocator. Pertanyaan selanjutnya yang muncul adalah bagaimana kita bisa mendapatkan referensi ke concrete object dan menggunakan fungsi-fungsi yang disediakan. Kita akan menggunakan kombinasi antara proxy object dan command pattern untuk membangun sebuah service layer yang handal Untuk setiap method yang dideklarasikan dalam myPaintObjMgr interface, kita bisa mendefinikan request object yang akan mengenkapsulasikan request terhadap fungsi tersebut. Penggunaan command pattern dalam hal ini, akan memberikan kemudahan apabila kita ingin menambah jenis dan jumlah request. Enkapsulasi request dalam object juga akan memberikan fleksibilitas dalam mendelegasikan eksekusi dari request.

Berikut adalah class diagram dari service layer yang akan kita bangun :



Gambar 3. Class Diagram, Implementasi Service Layer dengan proxy dan command pattern

1. Dalam class diagram diatas, bisa kita lihat bahwa ServiceLocatorImpl mempunyai satu Map collection COMMAND, yang berisi semua command yang kita definisikan. Collection tersebut, akan digunakan untuk me-resolve method yang dipanggil oleh pengguna. Proxy object akan digunakan untuk mengembalikan referensi ke concrete object (text atau image) didalam method getPaintObjMgr.

```

public Object getPaintObjMgr(Class myObjMgr)
    throws ServiceLocatorException {
    return Proxy.newProxyInstance(myObjMgr.getClassLoader(),
        new Class[]{myObjMgr},
        new PaintProxy());
}

```

Object proxy yang dikembalikan oleh fungsi ini bisa di 'cast' menjadi concrete object yang diinginkan. Ketika kita ingin memanggil fungsi yang disediakan oleh concrete object yang dimaksud, request terhadap fungsi tersebut akan di'intercept' oleh PaintProxy, yang mengimplementasikan interface InvocationHandler. Didalam class PaintProxy, kita mempunyai dua method penting yang perlu diperhatikan. Method pertama adalah resolveMethod. Method ini berfungsi untuk mencari method yang bersesuaian dengan request dari user di dalam COMMAND Map collection.

```

Private Command resolveCommand(Method method) {
    Command result = (Command) COMMAND.get(method.name)
    If(result != null)
        return result;
    else
        result = new DrawCommand();//default command
}

```

Method ini akan dipanggil didalam method invoke

```
Public Object invoke(Object proxy, Method method, object[]  
args) throws throwable{  
  
    Command command = resolveCommand(method); //resolving method  
    to be called  
  
    If(command == null){  
        throw new UnsupportedOperationException();  
    }  
    try{  
        command.setObj(proxy);  
        return command.execute();  
    }  
    catch(Exception e){  
        throw e;  
    }  
}
```

Dari potongan code diatas, kita bisa melihat bahwa dengan menggunakan proxy dan command pattern, kita bisa membangun service layer yang reliable, scalable, dan maintainable. Kita bisa dengan mudah menambahkan jenis layanan kedalam service layer yang ada. Demikian pula, apabila terjadi perubahan dalam requirement, kita hanya perlu mengubah bagian-bagian yang bersangkutan, tanpa harus mengubah concrete command class yang ada. Penggunaan Service Locator dan Proxy Object memberikan mekanisme pengelolaan yang terkendali terhadap business Object dalam aplikasi.

## Biografi Penulis



**Bernardus irmanto**, Lahir di Semarang, Agustus 1975. Menyelesaikan S1 di program studi ilmu komputer, Universitas Gadjah Mada pada tahun 1998. Bergabung dengan PT Freeport Indonesia beberapa saat sesudahnya, dan banyak berkecimpung dengan instrument interfacing, automatic data capturing, LIMS dan GIS selama bergabung dengan perusahaan tersebut. Mendapatkan beasiswa dari AusAid untuk melanjutkan studi ke jenjang S2 di School of Computer Science and Engineering, University of New South Wales(UNSW), Sydney, Australia dengan spesialisasi software engineering. Selama masa kuliah, bekerja sebagai freelance programmer untuk beberapa project komersial. Lulus dari UNSW bulan November 2002, kemudian sempat bekerja di Bali Camp dan Fujitsu System Indonesia, sebelum akhirnya bekerja sebagai software architect di PT. Newmont Nusa Tenggara. Mempunyai ketertarikan di bidang object oriented software development, design patterns dan rekayasa aplikasi berbasis web dengan J2EE framework maupun .NET framework.