**Proxy Design pattern** is one of the **Structural design pattern** and in my opinion one of the simplest pattern to understand. Proxy pattern intent according to GoF is:

*Provide a surrogate or placeholder for another object to control access to it.*

The definition itself is very clear and proxy pattern is used when we want to provide controlled access of a functionality. Let’s say we have a class that can run some command on the system. Now if we are using it, its fine but if we want to give this program to a client application, it can have severe issues because client program can issue command to delete some system files or change some settings that you don’t want. Here a proxy class can be created to provide controlled access of the program.

<http://www.journaldev.com/1572/proxy-design-pattern-in-java-example-tutorial>

Java RMI whole package uses proxy pattern.

**PROXY PATTTERN:**

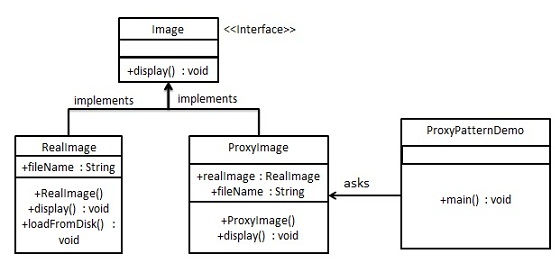
In Proxy pattern, a class represents functionality of another class. This type of design pattern comes under structural pattern.

In Proxy pattern, we create object having original object to interface its functionality to outer world.

Implementation

We're going to create a *Image* interface and concrete classes implementing the *Image* interface. *ProxyImage* is a proxy class to reduce memory footprint of *RealImage* object loading.

*ProxyPatternDemo*, our demo class will use *ProxyImage* to get a *Image* object to load and display as it needs.



Step 1

Create an interface.

*Image.java*

public interface Image {

void display();

}

Step 2

Create concrete classes implementing the same interface.

*RealImage.java*

public class RealImage implements Image {

private String fileName;

public RealImage(String fileName){

this.fileName = fileName;

loadFromDisk(fileName);

}

@Override

public void display() {

System.out.println("Displaying " + fileName);

}

private void loadFromDisk(String fileName){

System.out.println("Loading " + fileName);

}

}

*ProxyImage.java*

public class ProxyImage implements Image{

private RealImage realImage;

private String fileName;

public ProxyImage(String fileName){

this.fileName = fileName;

}

@Override

public void display() {

if(realImage == null){

realImage = new RealImage(fileName);

}

realImage.display();

}

}

Step 3

Use the *ProxyImage* to get object of *RealImage* class when required.

*ProxyPatternDemo.java*

public class ProxyPatternDemo {

public static void main(String[] args) {

Image image = new ProxyImage("test\_10mb.jpg");

//image will be loaded from disk

image.display();

System.out.println("");

//image will not be loaded from disk

image.display();

}

}

Step 4

Verify the output.

Loading test\_10mb.jpg

Displaying test\_10mb.jpg

Displaying test\_10mb.jpg

Difference between **Proxy** and **Decorator** pattern is that :

The difference is that, with the *Proxy pattern*, the relationship between a proxy and the real subject is typically set at compile time, whereas decorators can be recursively constructed at runtime. But I'm getting ahead of myself.

<http://www.javaworld.com/article/2074068/learn-java/take-control-with-the-proxy-design-pattern.html>

### [Proxy](http://en.wikipedia.org/wiki/Proxy_pattern) (recognizable by creational methods which returns an implementation of given abstract/interface type which in turn*delegates/uses* a *different* implementation of given abstract/interface type).

* [java.lang.reflect.Proxy](http://docs.oracle.com/javase/6/docs/api/java/lang/reflect/Proxy.html)
* [java.rmi.\*](http://docs.oracle.com/javase/6/docs/api/java/rmi/package-summary.html), the whole API actually.

<http://www.journaldev.com/1572/proxy-design-pattern-in-java-example-tutorial>