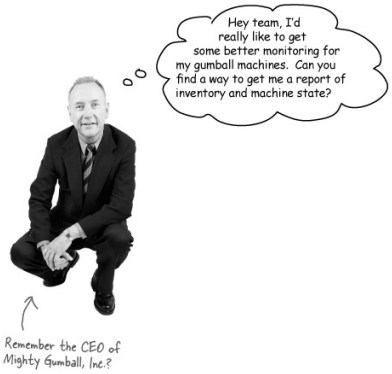
# Chapter 11. The Proxy Pattern: Controlling Object Access


**Ever play good cop, bad cop?** You're the good cop and you provide all your services in a nice and friendly manner, but you don't want *everyone* asking you for services, so you have the bad cop *control access* to you. That's what proxies do: control and manage access. As you're going to see, there are *lots* of ways in which proxies stand in for the objects they proxy. Proxies have been known to haul entire method calls over the Internet for their proxied objects; they've also been known to patiently stand in the place for some pretty lazy objects.



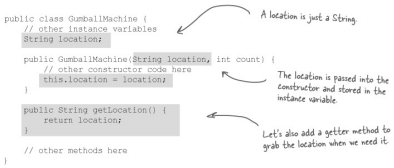

Sounds easy enough. If you remember, we've already got methods in the gumball machine code for getting the count of gumballs (getCount()), and getting the current state of the machine (getState()).

All we need to do is create a report that can be printed out and sent back to the CEO. Hmmm, we should probably add a location field to each gumball machine as well; that way the CEO can keep the machines straight.

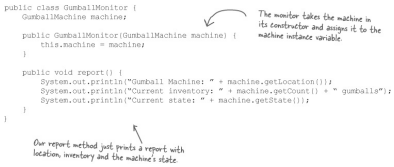
Let's just jump in and code this. We'll impress the CEO with a very fast turnaround.

## Coding the Monitor

Let's start by adding support to the GumballMachine class so that it can handle locations:



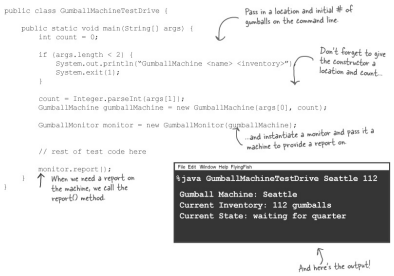

Now let's create another class, GumballMonitor, that retrieves the machine's location, inventory of gumballs and the current machine state and prints them in a nice little report:

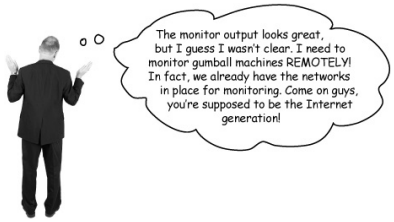



## Testing the Monitor

We implemented that in no time. The CEO is going to be thrilled and amazed by our development skills.

Now we just need to instantiate a GumballMonitor and give it a machine to monitor:





**Joe**: A remote what?

**Frank**: *Remote proxy*. Think about it: we've already got the monitor code written, right? We give the GumballMonitor a reference to a machine and it gives us a report. The problem is that monitor runs in the same JVM as the gumball machine and the CEO wants to sit at his desk and *remotely* monitor the machines! So what if we left our GumballMonitor class as is, but handed it a proxy to a *remote* object?

**Joe**: I'm not sure I get it.

**Jim**: Me neither.

**Frank**: Let's start at the beginning... a proxy is a stand in for a *real* object. In this case, the proxy acts just like it is a Gumball Machine object, but behind the scenes it is communicating over the network to talk to the real, remote GumballMachine.

**Jim:** So you're saying we keep our code as it is, and we give the monitor a reference to a proxy version of the GumballMachine...

**Joe**: And this proxy pretends it's the real object, but it's really just communicating over the net to the real object.

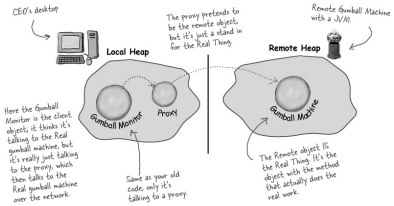
**Frank**: Yeah, that's pretty much the story.

**Joe**: It sounds like something that is easier said than done.

**Frank**: Perhaps, but I don't think it'll be that bad. We have to make sure that the gumball machine can act as a service and accept requests over the network; we also need to give our monitor a way to get a reference to a proxy object, but we've got some great tools already built into Java to help us. Let's talk a little more about remote proxies first...

## The role of the 'remote proxy'

A remote proxy acts as a *local representative to a remote object*. What's a "remote object?" It's an object that lives in the heap of a different Java Virtual Machine (or more generally, a remote object that is running in a different address space). What's a "local representative?" It's an object that you can call local methods on and have them forwarded on to the remote object.




**Your client object acts like it's making remote method calls. But what it's really doing is calling methods on a heap-local 'proxy' object that handles all the low-level details of network communication.**


**BRAIN POWER**

Before going further, think about how you'd design a system to enable remote method invocation. How would you make it easy on the developer so that she has to write as little code as possible? How would you make the remote invocation look seamless?

**BRAIN2 POWER**

Should making remote calls be totally transparent? Is that a good idea? What might be a problem with that approach?

## Adding a remote proxy to the Gumball Machine monitoring code

On paper this looks good, but how do we create a proxy that knows how to invoke a method on an object that lives in another JVM?

Hmmm. Well, you can't get a reference to something on another heap, right? In other words, you can't say:

```
Duck d = <object in another heap>
```

Whatever the variable **d** is referencing must be in the same heap space as the

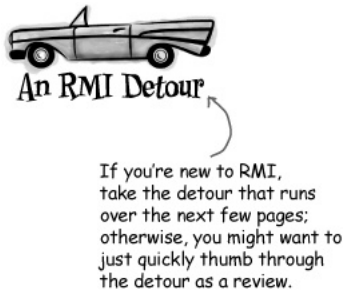
code running the statement. So how do we approach this? Well, that's where Java's Remote Method Invocation comes in... RMI gives us a way to find objects in a remote JVM and allows us to invoke their methods.

You may have encountered RMI in Head First Java; if not, we're going to take a slight detour and come up to speed on RMI before adding the proxy support to the Gumball Machine code.

So, here's what we're going to do:

1. **First, we're going to take the RMI Detour and check RMI out. Even if you are familiar with RMI, you might want to follow along and check out the scenery.**

2. **Then we're going to take our GumballMachine and make it a remote service that provides a set of methods calls that can be invoked remotely.**




3. **Then, we going to create a proxy that can talk to a remote GumballMachine, again using RMI, and put the monitoring system back together so that the CEO can monitor any number of remote machines.**

## Remote methods 101

**An RMI Detour**

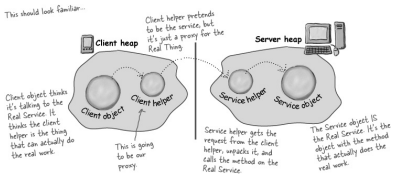
Let's say we want to design a system that allows us to call a local object that forwards each request to a remote object. How would we design it? We'd need a couple of helper objects that actually do the communicating for us. The helpers make it possible for the client to act as though it's calling a method on a local object (which in fact, it is). The client calls a method on the client helper, as if the client helper were the actual service. The client helper then takes care of forwarding that request for us.

In other words, the client object thinks it's calling a method on the remote service, because the client helper is pretending to be the service object. Pretending to be the thing with the method the client wants to call.

But the client helper isn't really the remote service. Although the client helper acts like it (because it has the same method that the service is advertising), the client helper doesn't have any of the actual method logic the client is expecting. Instead, the client helper contacts the server, transfers information about the method call (e.g., name of the method, arguments, etc.), and waits for a return from the server.

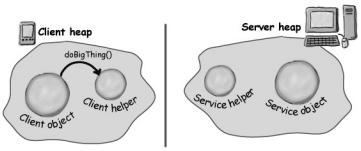
On the server side, the service helper receives the request from the client helper (through a Socket connection), unpacks the information about the call, and then invokes the real method on the real service object. So, to the service object, the call is local. It's coming from the service helper, not a remote client.

The service helper gets the return value from the service, packs it up, and ships it back (over a Socket's output stream) to the client helper. The client helper unpacks the information and returns the value to the client object.

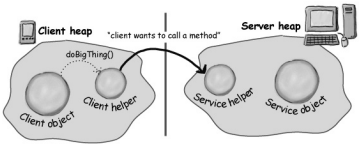



### How the method call happens

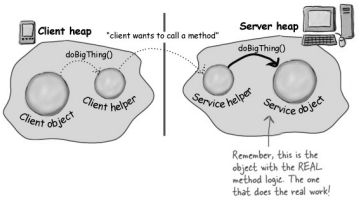
1. Client object calls doBigThing() on the client helper object.




2. Client helper packages up information about the call (arguments, method name, etc.) and ships it over the network to the service helper.

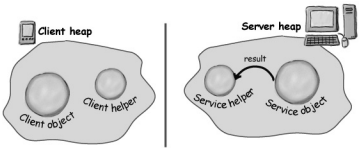



3. Service helper unpacks the information from the client helper, finds out which method to call (and on which object) and invokes the real method on the real service object.

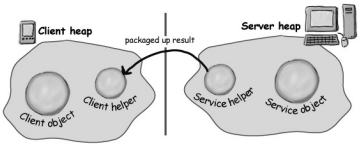



**An RMI Detour**

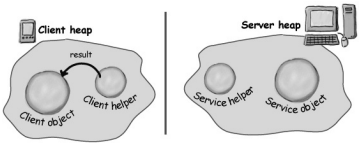
4. The method is invoked on the service object, which returns some result to the service helper.




5. Service helper packages up information returned from the call and ships it back over the network to the client helper.




6. Client helper unpackages the returned values and returns them to the client object. To the client object, this was all transparent.




### Java RMI, the Big Picture

Okay, you've got the gist of how remote methods work; now you just need to understand how to use RMI to enable remote method invocation.

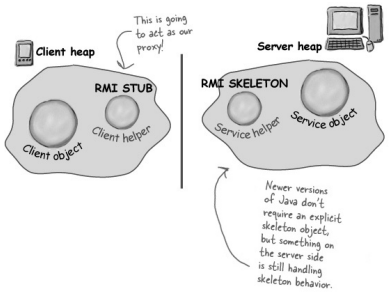
What RMI does for you is build the client and service helper objects, right down to creating a client helper object with the same methods as the remote service. The nice thing about RMI is that you don't have to write any of the networking or I/O code yourself. With your client, you call remote methods (i.e., the ones the Real Service has) just like normal method calls on objects running in the client's own local JVM.

RMI also provides all the runtime infrastructure to make it all work, including a lookup service that the client can use to find and access the remote objects.

There is one difference between RMI calls and local (normal) method calls. Remember that even though to the client it looks like the method call is local, the client helper sends the method call across the network. So there is networking and I/O. And what do we know about networking and I/O methods?

They're risky! They can fail! And so, they throw exceptions all over the place. As a result, the client does have to acknowledge the risk. We'll see how in a few pages.

**RMI Nomenclature: in RMI, the client helper is a 'stub' and the service helper is a 'skeleton'.**




**Now let's go through all the steps needed to make an object into a service that can accept remote calls and also the steps needed to allow a client to make remote calls.**

**You might want to make sure your seat belt is fastened; there are a lot of steps and a few bumps and curves – but nothing to be too worried about.**

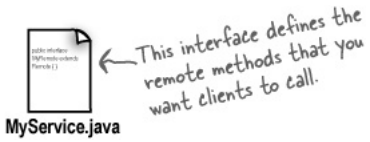
## Making the Remote service

**An RMI Detour**

This is an **overview** of the five steps for making the remote service. In other words, the steps needed to take an ordinary object and supercharge it so it can be called by a remote client. We'll be doing this later to our GumballMachine. For now, let's get the steps down and then we'll explain each one in detail.

**Step one:**

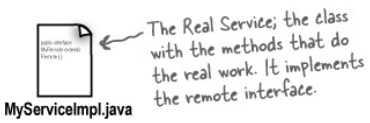
- Make a **Remote Interface**




The remote interface defines the methods that a client can call remotely. It's what the client will use as the class type for your service. Both the Stub and actual service will implement this!

**Step two:**

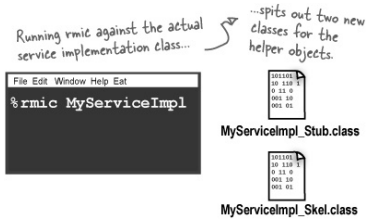
- Make a **Remote Implementation**




This is the class that does the Real Work. It has the real implementation of the remote methods defined in the remote interface. It's the object that the client wants to call methods on (e.g., our GumballMachine!).

**Step three:**

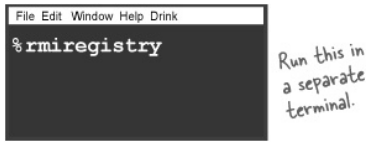
- Generate the **stubs** and **skeletons** using rmic




These are the client and server 'helpers'. You don't have to create these classes or ever look at the source code that generates them. It's all handled automatically when you run the rmic tool that ships with your Java development kit.

**Step four:**

- Start the **RMI registry** (rmiregistry)




The *rmiregistry* is like the white pages of a phone book. It's where the client goes to get the proxy (the client stub/helper object).

**Step five:**

- Start the **remote service**


You have to get the service object up and running. Your service implementation class instantiates an instance of the service and registers it with the RMI registry. Registering it makes the service available for clients.

### STEP ONE: MAKE A REMOTE INTERFACE

1. **Extend java.rmi.Remote**

   Remote is a 'marker' interface, which means it has no methods. It has special meaning for RMI, though, so you must follow this rule. Notice that we say 'extends' here. One interface is allowed to *extend* another interface.

```
public interface MyRemote extends Remote {
```
This tells us that the interface is going to be used to support remote calls.

2. Declare that all methods **throw a RemoteException**

   The remote interface is the one the client uses as the type for the service. In other words, the client invokes methods on something that implements the remote interface. That something is the stub, of course, and since the stub is doing networking and I/O, all kinds of Bad Things can happen. The client has to acknowledge the risks by handling or declaring the remote exceptions. If the methods in an interface declare exceptions, any code calling methods on a reference of that type (the interface type) must handle or declare the exceptions.

```
import java.rmi.*;
```
Remote interface is in java.rmi

```
public interface MyRemote extends Remote {
    public String sayHello() throws RemoteException;
}
```
Every remote method call is considered 'risky'. Declaring RemoteException on every method forces the client to pay attention and acknowledge that things might not work.

3. **Be sure arguments and return values are primitives or Serializable**

   Arguments and return values of a remote method must be either primitive or Serializable. Think about it. Any argument to a remote method has to be packaged up and shipped across the network, and that's done through Serialization. Same thing with return values. If you use primitives,

Strings, and the majority of types in the API (including arrays and collections), you'll be fine. If you are passing around your own types, just be sure that you make your classes implement Serializable.

> **NOTE**
>
> Check out Head First Java if you need to refresh your memory on Serializable.

```
public String sayHello() throws RemoteException;
```

This return value is gonna be shipped over the wire from the server back to the client, so it must be Serializable. That's how args and return values get packaged up and sent.

### STEP TWO: MAKE A REMOTE IMPLEMENTATION

**An RMI Detour**

1. **Implement the Remote interface**

   Your service has to implement the remote interface—the one with the methods your client is going to call.

```
public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
    public String sayHello() {
        return "Server says, 'Hey'";
    }
   // more code in class
}
```

The compiler will make sure that you've implemented all the methods from the interface you implement. In this case, there's only one.

2. **Extend UnicastRemoteObject**

   In order to work as a remote service object, your object needs some functionality related to 'being remote'. The simplest way is to extend UnicastRemoteObject (from the java.rmi.server package) and let that class (your superclass) do the work for you.

```
public class MyRemoteImpl extends UnicastRemoteObject implements
```

3. **Write a no-arg constructor that declares a RemoteException**

   Your new superclass, UnicastRemoteObject, has one little problem—its constructor throws a RemoteException. The only way to deal with this is to declare a constructor for your remote implementation, just so that you have a place to declare the RemoteException. Remember, when a class is instantiated, its superclass constructor is always called. If your superclass constructor throws an exception, you have no choice but to declare that your constructor also throws an exception.

```
public MyRemoteImpl() throws RemoteException { }
```

You don't have to put anything in the constructor. You just need a way to declare that your superclass constructor throws an exception.

4. **Register the service with the RMI registry**

   Now that you've got a remote service, you have to make it available to remote clients. You do this by instantiating it and putting it into the RMI registry (which must be running or this line of code fails). When you register the implementation object, the RMI system actually puts the *stub* in the registry, since that's what the client really needs. Register your service using the static rebind() method of the java.rmi.Naming class.

```
try {
    MyRemote service = new MyRemoteImpl();
    Naming.rebind("RemoteHello", service);
} catch(Exception ex) {...}
```

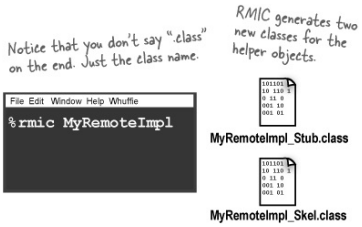
Give your service a name (that clients can use to look it up in the registry) and register it with the RMI registry. When you bind the service objects, RMI swaps the service for the stub and puts the stub in the registry.

### STEP THREE: GENERATE STUBS AND SKELETONS

1. **Run rmic on the remote implementation class (not the remote interface)**

   The rmic tool, which comes with the Java software development kit, takes a service implementation and creates two new classes, the stub and the skeleton. It uses a naming convention that is the name of your remote implementation, with either _Stub or _Skel added to the end. There are other options with rmic, including not generating skeletons, seeing what the source code for these classes looked like, and even using IIOP as the protocol. The way we're doing it here is the way you'll usually do it. The classes will land in the current directory (i.e. whatever you did a cd to). Remember, rmic must be able to see your implementation class, so you'll probably run rmic from the directory where your remote implementation is located. (We're deliberately not using packages here, to make it simpler. In the Real World, you'll need to account for package

directory structures and fully-qualified names).




#### STEP FOUR: RUN RMIREGISTRY

1. **Bring up a terminal and start the rmiregistry.**

   Be sure you start it from a directory that has access to your classes. The simplest way is to start it from your 'classes' directory.


#### STEP FIVE: START THE SERVICE

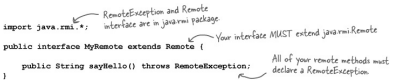
1. **Bring up another terminal and start your service**

   This might be from a main() method in your remote implementation class, or from a separate launcher class. In this simple example, we put the starter code in the implementation class, in a main method that instantiates the object and registers it with RMI registry.

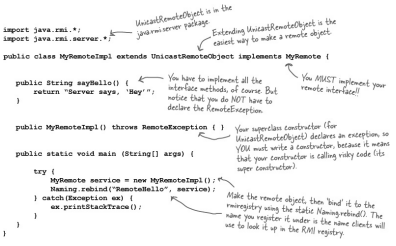

#### COMPLETE CODE FOR THE SERVER SIDE

**An RMI Detour**

**The Remote interface:**




**The Remote service (the implementation):**




## How does the client get the stub object?

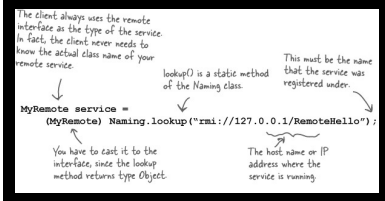
The client has to get the stub object (our proxy), since that's the thing the client will call methods on. And that's where the RMI registry comes in. The client does a 'lookup', like going to the white pages of a phone book, and essentially says, "Here's a name, and I'd like the stub that goes with that name."

Let's take a look at the code we need to lookup and retrieve a stub object.

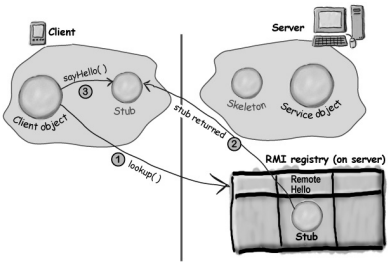
NOTE

Here's how it works.

CODE UP CLOSE




**An RMI Detour**




**How it works...**

1. **Client does a lookup on the RMI registry**

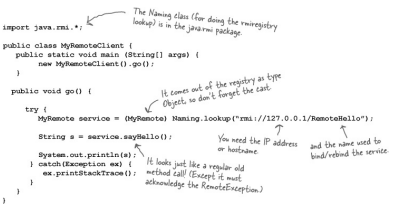
```
Naming.lookup("rmi://127.0.0.1/RemoteHello");
```

2. **RMI registry returns the stub object**

   (as the return value of the lookup method) and RMI deserializes the stub automatically. You MUST have the stub class (that rmic generated for you) on the client or the stub won't be deserialized.

3. **Client invokes a method on the stub, as if the stub IS the real service**

**Complete client code**



GEEK BITS

**How does the client get the stub class?**

Now we get to the interesting question. Somehow, some way, the client must have the stub class (that you generated earlier using rmic) at the time the client does the lookup, or else the stub won't be deserialized on the client and the whole thing blows up. The client also needs classes for any serialized objects returned by method calls to the remote object. In a simple system, you can simply hand-deliver these classes to the client.

There's a much cooler way, although it's beyond the scope of this book. But just in case you're interested, the cooler way is called "dynamic class downloading". With dynamic class downloading, Serialized objects (like the stub) are "stamped"with a URL that tells the RMI system on the client where to find the class file for that object. Then, in the process of deserializing an object, if RMI can't find the class locally, it uses that URL to do an HTTP Get to retrieve the class file. So you'd need a simple web server to serve up class files, and you'd also need to change some security parameters on the client. There are a few other tricky issues with dynamic class downloading, but that's the overview.

For the stub object specifically, there's *another* way the client can get the class. This is only available in Java 5, though. We'll briefly talk about this near the end of the chapter.

**An RMI Detour**

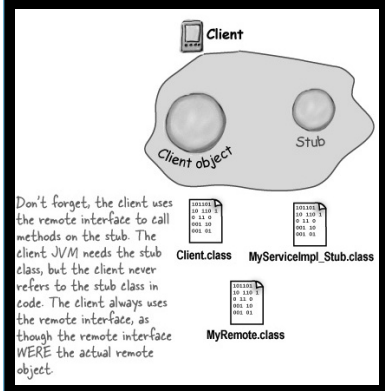
WATCH IT!

The top three things programmers do wrong with RMI are:

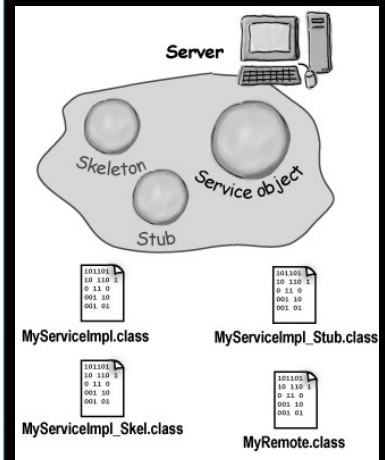
1) Forget to start rmiregistry before starting remote service (when the service is registered using Naming.rebind(), the rmiregistry must be running!)

2) Forget to make arguments and return types serializable (you won't know until runtime; this is not something the compiler will detect.)

3) Forget to give the stub class to the client.



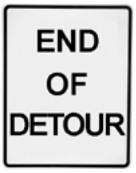

Don't forget, the client uses the remote interface to call methods on the stub. The client JVM needs the stub class, but the client never refers to the stub class in code. The client always uses the remote interface, as though the remote interface WERE the actual remote object.

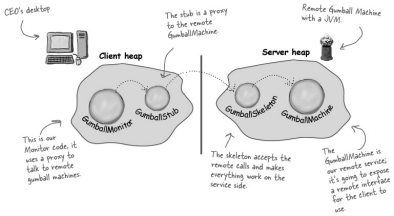



Server needs both the Stub and Skeleton classes, as well as the service and the remote interface. It needs the stub class because remember, the stub is substituted for the real service when the real service is bound to the RMI registry.

### Back to our GumballMachine remote proxy

**Okay, now that you have the RMI basics down, you've got the tools you need to implement the gumball machine remote proxy. Let's take a look at how the GumballMachine fits into this framework:**






## Getting the GumballMachine ready to be a remote service

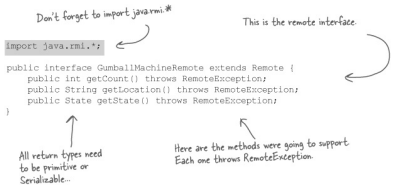
The first step in converting our code to use the remote proxy is to enable the GumballMachine to service remote requests from clients. In other words, we're going to make it into a service. To do that, we need to:

1) Create a remote interface for the GumballMachine. This will provide a set of methods that can be called remotely.

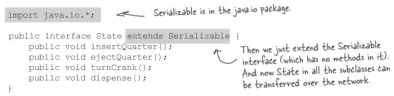
2) Make sure all the return types in the interface are serializable.

3) Implement the interface in a concrete class.

We'll start with the remote interface:



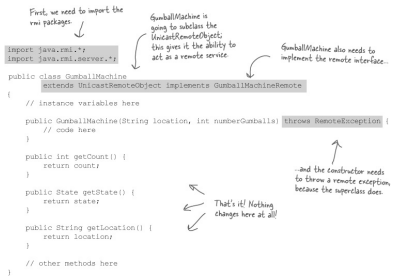

We have one return type that isn't Serializable: the State class. Let's fix it up...




Actually, we're not done with Serializable yet; we have one problem with State. As you may remember, each State object maintains a reference to a gumball machine so that it can call the gumball machine's methods and change its state. We don't want the entire gumball machine serialized and transferred with the State object. There is an easy way to fix this:


We've already implemented our GumballMachine, but we need to make sure it can act as a service and handle requests coming from over the network. To do that, we have to make sure the GumballMachine is doing everything it needs to implement the GumballMachineRemote interface.

As you've already seen in the RMI detour, this is quite simple, all we need to do is add a couple of things...




## Registering with the RMI registry...

That completes the gumball machine service. Now we just need to fire it up so it can receive requests. First, we need to make sure we register it with the RMI registry so that clients can locate it.

We're going to add a little code to the test drive that will take care of this for

us:

```
public class GumballMachineTestDrive {

    public static void main(String[] args) {
        GumballMachineRemote gumballMachine = null;
        int count;

        if (args.length < 2) {
            System.out.println("GumballMachine <name> <inventory>");
            System.exit(1);
        }
        try {
            count = Integer.parseInt(args[1]);

            gumballMachine =
                new GumballMachine(args[0], count);
            Naming.rebind("//" + args[0] + "/gumballmachine", gumballMachine);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

First we need to add a try/catch block around the gumball initialization because our constructor can now throw exceptions.

We also add the call to Naming.rebind, which publishes the GumballMachine stub under the name gumballmachine.

Let's go ahead and get this running..

Run this first.

This gets the RMI registry service up and running.

We're using the "official" Mighty Gumball machines; you should substitute your own machine name here.

```
File Edit Window Help Huh?
% rmiregistry
```

```
File Edit Window Help Huh?
% java GumballMachineTestDrive seattle.mightygumball.com 100
```

Run this second.

This gets the GumballMachine up and running and registers it with the RMI registry.

## Now for the GumballMonitor client...

Remember the GumballMonitor? We wanted to reuse it without having to rewrite it to work over a network. Well, we're pretty much going to do that, but we do need to make a few changes.

We need to import the RMI package because we are using the RemoteException class below.

```
import java.rmi.*;

public class GumballMonitor {
    GumballMachineRemote machine;

    public GumballMonitor(GumballMachineRemote machine) {
        this.machine = machine;
    }

    public void report() {
        try {
            System.out.println("Gumball Machine: " + machine.getLocation());
            System.out.println("Current inventory: " + machine.getCount() + " gumballs");
            System.out.println("Current state: " + machine.getState());
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

Now we've got to use our interface rather than the concrete GumballMachine class.

We also need to catch any remote exceptions that might happen as we try to invoke methods that are ultimately happening over the network.


## Writing the Monitor test drive

Now we've got all the pieces we need. We just need to write some code so the CEO can monitor a bunch of gumball machines:

Here's the monitor test drive. The CEO is going to run this!

Here's all the locations we're going to monitor.

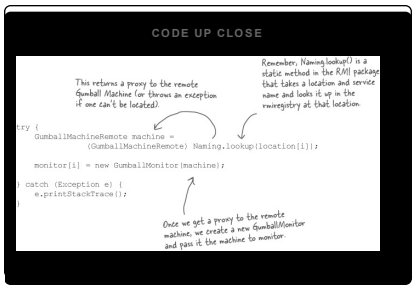
We create an array of locations, one for each machine.

```
import java.rmi.*;

public class GumballMonitorTestDrive {

    public static void main(String[] args) {
        String[] location = {"rmi://santafe.mightygumball.com/gumballmachine",
                             "rmi://boulder.mightygumball.com/gumballmachine",
                             "rmi://seattle.mightygumball.com/gumballmachine"};

        GumballMonitor[] monitor = new GumballMonitor[location.length];

        for (int i=0;i < location.length; i++) {
            try {
                GumballMachineRemote machine =
                        (GumballMachineRemote) Naming.lookup(location[i]);
                monitor[i] = new GumballMonitor(machine);
                System.out.println(monitor[i]);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        for(int i=0; i < monitor.length; i++) {
            monitor[i].report();
        }
    }
}
```

We also create an array of monitors.

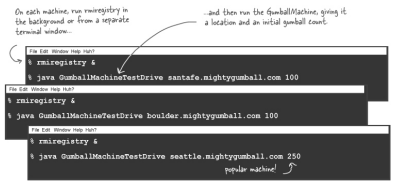
Now we need to get a proxy to each remote machine.

Then we iterate through each machine and print out its report.




## Another demo for the CEO of Mighty Gumball...

Okay, it's time to put all this work together and give another demo. First let's make sure a few gumball machines are running the new code:

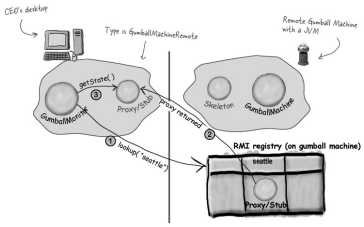



**And now let's put the monitor in the hands of the CEO. Hopefully this time he'll love it:**

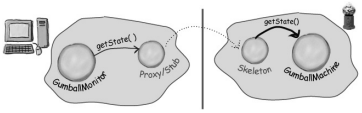

**By invoking methods on the proxy, a remote call is made across the wire and a String, an integer and a State object are returned. Because we are using a proxy, the GumballMonitor doesn't know, or care, that calls are remote (other than having to worry about remote exceptions).**

**Behind the Scenes**

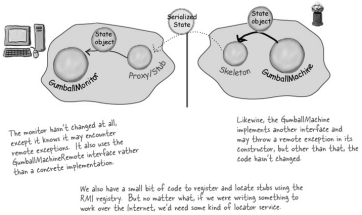

1. The CEO runs the monitor, which first grabs the proxies to the remote gumball machines and then calls getState() on each one (along with getCount() and getLocation()).




2. getState() is called on the proxy, which forwards the call to the remote service. The skeleton receives the request and then forwards it to the gumball machine.




3. GumballMachine returns the state to the skeleton, which serializes it and transfers it back over the wire to the proxy. The proxy deserializes it and returns it as an object to the monitor.




### The Proxy Pattern defined

We've already put a lot of pages behind us in this chapter; as you can see, explaining the Remote Proxy is quite involved. Despite that, you'll see that the definition and class diagram for the Proxy Pattern is actually fairly straightforward. Note that Remote Proxy is one implementation of the general Proxy Pattern; there are actually quite a few variations of the pattern, and we'll talk about them later. For now, let's get the details of the general pattern down.

Here's the Proxy Pattern definition:

NOTE

**The Proxy Pattern** provides a surrogate or placeholder for another object to control access to it.

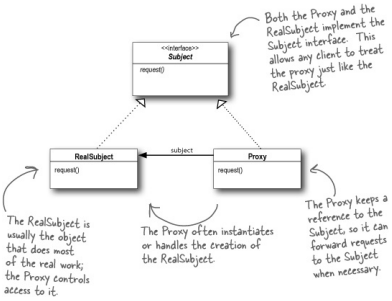
> **Use the Proxy Pattern to create a representative object that controls access to another object, which may be remote, expensive to create or in need of securing.**

Well, we've seen how the Proxy Pattern provides a surrogate or placeholder for another object. We've also described the proxy as a "representative" for another object.

But what about a proxy controlling access? That sounds a little strange. No worries. In the case of the gumball machine, just think of the proxy controlling access to the remote object. The proxy needed to control access because our client, the monitor, didn't know how to talk to a remote object. So in some sense the remote proxy controlled access so that it could handle the network details for us. As we just discussed, there are many variations of the Proxy Pattern, and the variations typically revolve around the way the proxy "controls access." We're going to talk more about this later, but for now here are a few ways proxies control access:

- As we know, a remote proxy controls access to a remote object.
- A virtual proxy controls access to a resource that is expensive to create.
- A protection proxy controls access to a resource based on access rights.

Now that you've got the gist of the general pattern, check out the class diagram...




Let's step through the diagram...

First we have a Subject, which provides an interface for the RealSubject and the Proxy. By implementing the same interface, the Proxy can be substituted for the RealSubject anywhere it occurs.

The RealSubject is the object that does the real work. It's the object that the Proxy represents and controls access to.

The Proxy holds a reference to the RealSubject. In some cases, the Proxy may be responsible for creating and destroying the RealSubject. Clients interact with the RealSubject through the Proxy. Because the Proxy and RealSubject implement the same interface (Subject), the Proxy can be substituted anywhere the subject can be used. The Proxy also controls access to the RealSubject; this control may be needed if the Subject is running on a remote machine, if the Subject is expensive to create in some way or if access to the subject needs to be protected in some way.

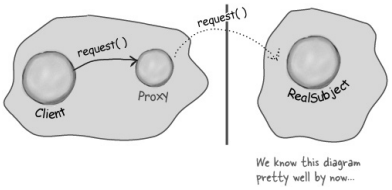
Now that you understand the general pattern, let's look at some other ways of using proxy beyond the Remote Proxy...

## Get ready for Virtual Proxy

Okay, so far you've seen the definition of the Proxy Pattern and you've taken a look at one specific example: the *Remote Proxy*. Now we're going to take a look at a different type of proxy, the *Virtual Proxy*. As you'll discover, the Proxy Pattern can manifest itself in many forms, yet all the forms follow roughly the general proxy design. Why so many forms? Because the proxy pattern can be applied to a lot of different use cases. Let's check out the Virtual Proxy and compare it to Remote Proxy:

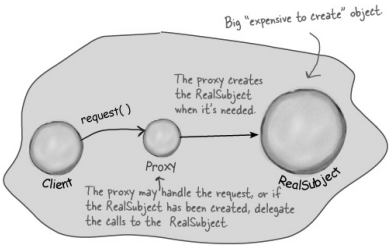
**Remote Proxy**

With Remote Proxy, the proxy acts as a local representative for an object that lives in a different JVM. A method call on the proxy results in the call being transferred over the wire, invoked remotely, and the result being returned back to the proxy and then to the Client.




**Virtual Proxy**

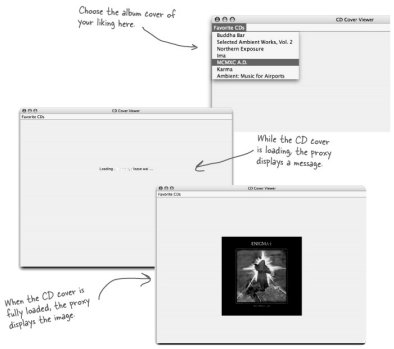
Virtual Proxy acts as a representative for an object that may be expensive to create. The Virtual Proxy often defers the creation of the object until it is needed; the Virtual Proxy also acts as a surrogate for the object before and while it is being created. After that, the proxy delegates requests directly to the RealSubject.




## Displaying CD covers

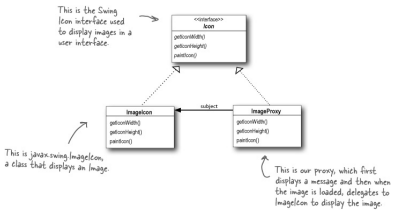
Let's say you want to write an application that displays your favorite compact disc covers. You might create a menu of the CD titles and then retrieve the images from an online service like Amazon.com. If you're using Swing, you might create an Icon and ask it to load the image from the network. The only problem is, depending on the network load and the bandwidth of your connection, retrieving a CD cover might take a little time, so your application should display something while you are waiting for the image to load. We also don't want to hang up the entire application while it's waiting on the image. Once the image is loaded, the message should go away and you should see the image.

An easy way to achieve this is through a virtual proxy. The virtual proxy can stand in place of the icon, manage the background loading, and before the image is fully retrieved from the network, display "Loading CD cover, please wait...". Once the image is loaded, the proxy delegates the display to the Icon.




## Designing the CD cover Virtual Proxy

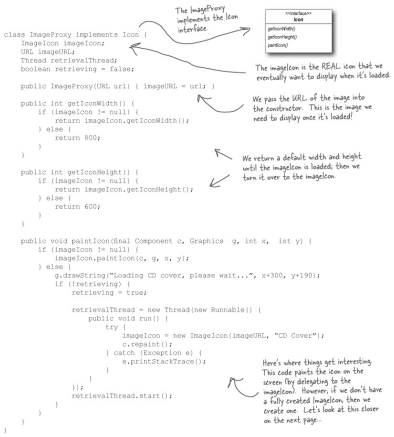
Before writing the code for the CD Cover Viewer, let's look at the class diagram. You'll see this looks just like our Remote Proxy class diagram, but here the proxy is used to hide an object that is expensive to create (because we need to retrieve the data for the Icon over the network) as opposed to an object that actually lives somewhere else on the network.




### How ImageProxy is going to work

1. **ImageProxy first creates an ImageIcon and starts loading it from a network URL.**
2. **While the bytes of the image are being retrieved, ImageProxy displays "Loading CD cover, please wait...".**
3. **When the image is fully loaded, ImageProxy delegates all method calls to the image icon, including paintIcon(), getWidth() and getHeight().**
4. **If the user requests a new image, we'll create a new proxy and start the process over.**

## Writing the Image Proxy




```
class ImageProxy implements Icon {
    ImageIcon imageIcon;
    URL imageURL;
    Thread retrievalThread;
    boolean retrieving = false;

    public ImageProxy(URL url) { imageURL = url; }

    public int getIconWidth() {
        if (imageIcon != null) {
            return imageIcon.getIconWidth();
        } else {
            return 800;
        }
    }

    public int getIconHeight() {
        if (imageIcon != null) {
            return imageIcon.getIconHeight();
        } else {
            return 600;
        }
    }

    public void paintIcon(final Component c, Graphics  g, int x,  int y) {
        if (imageIcon != null) {
            imageIcon.paintIcon(c, g, x, y);
        } else {
            g.drawString("Loading CD cover, please wait...", x+300, y+190);
            if (!retrieving) {
                retrieving = true;

                retrievalThread = new Thread(new Runnable() {
                    public void run() {
                        try {
                            imageIcon = new ImageIcon(imageURL, "CD Cover");
                            c.repaint();
                        } catch (Exception e) {
                            e.printStackTrace();
                        }
                    }
                });
                retrievalThread.start();
            }
        }
    }
}
```

## CODE UP CLOSE

This method is called when it's time to paint the icon on the screen.

```
public void paintIcon(final Component c, Graphics  g, int x,  int y) {
    if (imageIcon != null) {
        imageIcon.paintIcon(c, g, x, y);
    } else {
        g.drawString("Loading CD cover, please wait...", x+300, y+190);
        if (!retrieving) {
            retrieving = true;
            retrievalThread = new Thread(new Runnable() {
                public void run() {
                    try {
                        imageIcon = new ImageIcon(imageURL, "CD Cover");
                        c.repaint();
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            });
            retrievalThread.start();
        }
    }
}
```

If we've got an icon already, we go ahead and tell it to paint itself.

Otherwise we display the "loading" message.

Here's where we load the REAL icon image. Note that the image loading with IconImage is synchronous: the ImageIcon constructor doesn't return until the image is loaded. That doesn't give us much of a chance to do screen updates and have our message displayed, so we're going to do this asynchronously. See the "Code Way Up Close" on the next page for more...

## CODE WAY UP CLOSE

```
if (!retrieving) {
    retrieving = true;

    retrievalThread = new Thread(new Runnable() {
        public void run() {
            try {
                imageIcon = new ImageIcon(imageURL, "CD Cover");
                c.repaint();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    });
    retrievalThread.start();
}
```

If we aren't already trying to retrieve the image...

...then it's time to start retrieving it (in case you were wondering, only one thread calls paint, so we should be okay here in terms of thread safety!).

We don't want to hang up the entire user interface, so we're going to use another thread to retrieve the image.

In our thread we instantiate the Icon object. Its constructor will not return until the image is loaded.

When we have the image, we tell Swing that we need to be repainted.

### NOTE

So, the next time the display is painted after the ImageIcon is instantiated, the paintIcon method will paint the image, not the loading message.

## DESIGN PUZZLE

The ImageProxy class appears to have two states that are controlled by conditional statements. Can you think of another pattern that might clean up this code? How would you redesign ImageProxy?

```
class ImageProxy implements Icon {
    // instance variables & constructor here

    public int getIconWidth() {
        if (imageIcon != null) {
            return imageIcon.getIconWidth();
        } else {
            return 800;
        }
    }

    public int getIconHeight() {
        if (imageIcon != null) {
            return imageIcon.getIconHeight();
        } else {
            return 600;
        }
    }

    public void paintIcon(final Component c, Graphics  g, int x,  int y) {
        if (imageIcon != null) {
            imageIcon.paintIcon(c, g, x, y);
        } else {
            g.drawString("Loading CD cover, please wait...", x+300, y+190);
            // more code here
        }
    }
```

Two states

Two states

Two states

**READY-BAKE CODE: TESTING THE CD COVER VIEWER**

Okay, it's time to test out this fancy new virtual proxy. Behind the scenes we've been baking up a new ImageProxyTestDrive that sets up the window, creates a frame, installs the menus and creates our proxy. We don't go through all that code in gory detail here, but you can always grab the source code and have a look, or check it out at the end of the chapter where we list all the source code for the Virtual Proxy.

Here's a partial view of the test drive code:

```
public class ImageProxyTestDrive {
    ImageComponent imageComponent;
    public static void main (String[] args) throws Exception {
        ImageProxyTestDrive testDrive = new ImageProxyTestDrive();
    }

    public ImageProxyTestDrive() throws Exception{

        // set up frame and menus

        Icon icon = new ImageProxy(initialURL);
        imageComponent = new ImageComponent(icon);
        frame.getContentPane().add(imageComponent);
    }
```

Here we create an image proxy and set it to an initial URL. Whenever you choose a selection from the CD menu, you'll get a new image proxy.

Finally we add the proxy to the frame so it can be displayed.

Now let's run the test drive:

```
File Edit Window Help JustSomeOfTheCdsTheGeekAsThroughThisBook
% java ImageProxyTestDrive
```

Running ImageProxyTestDrive should give you a window like this

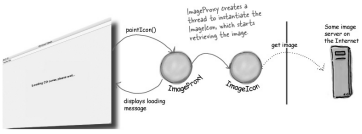
**Things to try...**

1. **Use the menu to load different CD covers; watch the proxy display "loading" until the image has arrived.**
2. **Resize the window as the "loading" message is displayed. Notice that the proxy is handling the loading without hanging up the Swing window.**
3. **Add your own favorite CDs to the ImageProxyTestDrive.**

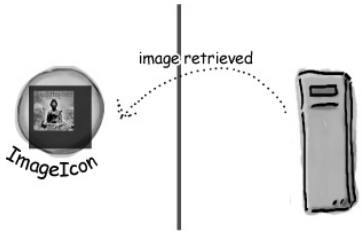
## What did we do?

**Behind the Scenes**

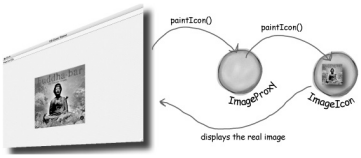
1. We created an ImageProxy for the display. The paintIcon() method is called and ImageProxy fires off a thread to retrieve the image and create the ImageIcon.




2. At some point the image is returned and the ImageIcon fully instantiated.




3. After the ImageIcon is created, the next time paintIcon() is called, the proxy delegates to the ImageIcon.



## THERE ARE NO DUMB QUESTIONS

**Q:** **Q: The Remote Proxy and Virtual Proxy seem so different to me; are they really ONE pattern?**

**A:** **A:** *You'll find a lot of variants of the Proxy Pattern in the real world; what they all have in common is that they intercept a method invocation that the client is making on the subject. This level of indirection allows us to do many things, including dispatching requests to a remote subject, providing a representative for an expensive object as it is created, or, as you'll see, providing some level of protection that can determine which clients should be calling which methods. That's just the beginning; the general Proxy Pattern can be applied in many different ways, and we'll cover some of the other ways at the end of the chapter.*

**Q:** **Q: ImageProxy seems just like a Decorator to me. I mean, we are basically wrapping one object with another and then delegating the calls to the ImageIcon. What am I missing?**

**A:** **A:** *Sometimes Proxy and Decorator look very similar, but their purposes are different: a decorator adds behavior to a class, while a proxy controls access to it. You might say, "isn't the loading message adding behavior?" In some ways it is; however, more importantly, the ImageProxy is controlling access to an ImageIcon. How does it control access? Well, think about it this way: the proxy is decoupling the client from the ImageIcon. If they were coupled the client would have to wait until each image is retrieved before it could paint its entire interface. The proxy controls access to the ImageIcon so that before it is fully created, the proxy provides another on screen representation. Once the ImageIcon is created the proxy allows access.*

**Q:** **Q: How do I make clients use the Proxy rather than the Real Subject?**

**A:** **A:** *Good question. One common technique is to provide a factory that instantiates and returns the subject. Because this happens in a factory method we can then wrap the subject with a proxy before returning it. The client never knows or cares that it's using a proxy instead of the real thing.*

**Q:** **Q: I noticed in the ImageProxy example, you always create a new ImageIcon to get the image, even if the image has already been retrieved. Could you implement something similar to the ImageProxy that caches past retrievals?**

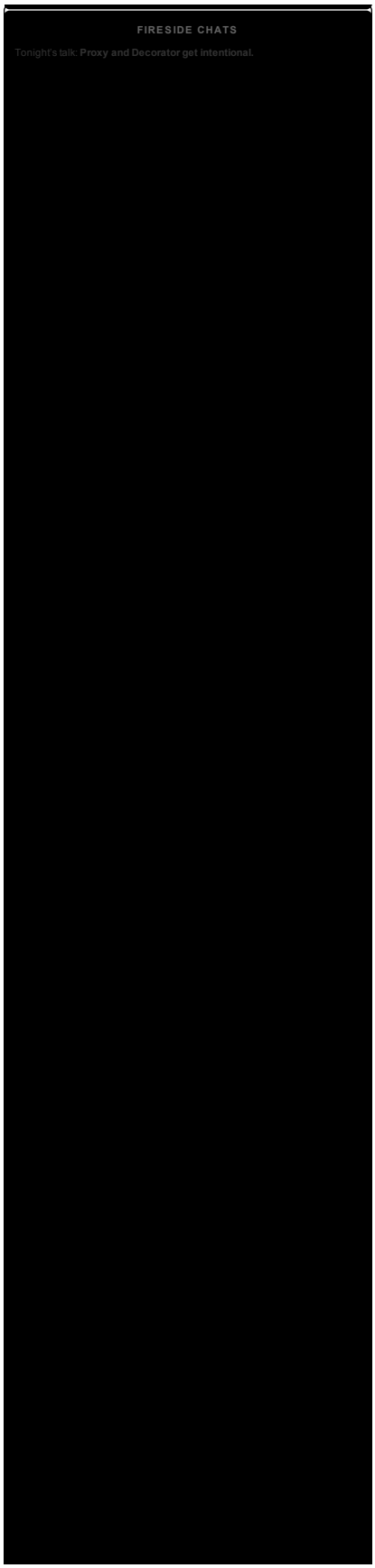
**A:** **A:** *You are talking about a specialized form of a Virtual Proxy called a Caching Proxy. A caching proxy maintains a cache of previous created objects and when a request is made it returns cached object, if possible.*

*We're going to look this and at several other variants of the Proxy Pattern at the end of the chapter.*

**Q:** **Q: I see how Decorator and Proxy relate, but what about Adapter? An adapter seems very similar as well.**

**A:** **A:** *Both Proxy and Adapter sit in front of other objects and forward requests to them. Remember that Adapter changes the interface of the objects it adapts, while the Proxy implements the same interface.*

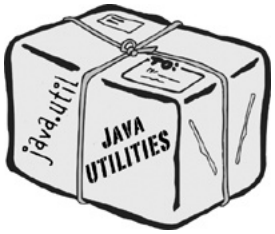
*There is one additional similarity that relates to the Protection Proxy. A Protection Proxy may allow or disallow a client access to particular methods in an object based on the role of the client. In this way a Protection Proxy may only provide a partial interface to a client, which is quite similar to some Adapters. We are going to take a look at Protection Proxy in a few pages.*


FIRESIDE CHATS
Tonight's talk: Proxy and Decorator get intentional.

| Proxy | Decorator |
| --- | --- |
| Hello, Decorator. I presume you're here because people sometimes get us confused? | |
| | Well, I think the reason people get us confused is that you go around pretending to be an entirely different pattern, when in fact, you're just a Decorator in disguise. I really don't think you should be copying all my ideas. |
| *Me* copying *your* ideas? Please. I control access to objects. You just decorate them. My job is so much more important than yours it's just not even funny. | |
| | "Just" decorate? You think decorating is some frivolous unimportant pattern? Let me tell you buddy, I add *behavior*. That's the most important thing about objects - what they *do*! |
| Fine, so maybe you're not entirely frivolous... but I still don't get why you think I'm copying all your ideas. I'm all about representing my subjects, not decorating them. | |
| | You can call it "representation" but if it looks like a duck and walks like a duck... I mean, just look at your Virtual Proxy; it's just another way of adding behavior to do something while some big expensive object is loading, and your Remote Proxy is a way of talking to remote objects so your clients don't have to bother with that themselves. It's all about behavior, just like I said. |
| I don't think you get it, Decorator. I stand in for my Subjects; I don't just add behavior. Clients use me as a surrogate of a Real Subject, because I can protect them from unwanted access, or keep their GUIs from hanging up while they're waiting for big objects to load, or hide the fact that their Subjects are running on remote machines. I'd say that's a very different intent from yours! | |
| | Call it what you want. I implement the same interface as the objects I wrap; so do you. |
| Okay, let's review that statement. You wrap an object. While sometimes we informally say a proxy wraps its Subject, that's not really an accurate term. | |
| | Oh yeah? Why not? |
| Think about a remote proxy... what object am I wrapping? The object I'm representing and | |

| | |
|---|---|
| controlling access to lives on another machine! Let's see you do that. | |
| | Okay, but we all know remote proxies are kinda weird. Got a second example? I doubt it. |
| Sure, okay, take a virtual proxy... think about the CD viewer example. When the client first uses me as a proxy the subject doesn't even exist! So what am I wrapping there? | |
| | Uh huh, and the next thing you'll be saying is that you actually get to create objects. |
| I never knew decorators were so dumb! Of course I sometimes create objects, how do you think a virtual proxy gets its subject! Okay, you just pointed out a big difference between us: we both know decorators only add window dressing; they never get to instantiate anything. | |
| | Oh yeah? Instantiate this! |
| Hey, after this conversation I'm convinced you're just a dumb proxy! | |
| | Dumb proxy? I'd like to see you recursively wrap an object with 10 decorators and keep your head straight at the same time. |
| Very seldom will you ever see a proxy get into wrapping a subject multiple times; in fact, if you're wrapping something 10 times, you better go back reexamine your design. | |
| | Just like a proxy, acting all real when in fact you just stand in for the objects doing the real work. You know, I actually feel sorry for you. |

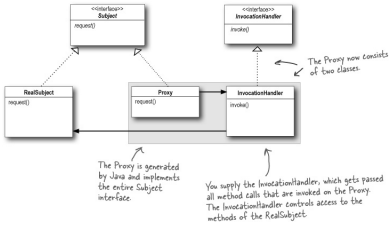
### Using the Java API's Proxy to create a protection proxy




Java's got its own proxy support right in the java.lang.reflect package. With this package, Java lets you create a proxy class *on the fly* that implements one or more interfaces and forwards method invocations to a class that you specify. Because the actual proxy class is created at runtime, we refer to this Java technology as a *dynamic proxy*.

We're going to use Java's dynamic proxy to create our next proxy

implementation (a protection proxy), but before we do that, let's quickly look at a class diagram that shows how dynamic proxies are put together. Like most things in the real world, it differs slightly from the classic definition of the pattern:



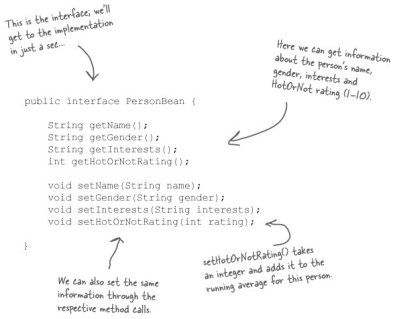

Because Java creates the Proxy class *for you*, you need a way to tell the Proxy class what to do. You can't put that code into the Proxy class like we did before, because you're not implementing one directly. So, if you can't put this code in the Proxy class, where do you put it? In an InvocationHandler. The job of the InvocationHandler is to respond to any method calls on the proxy. Think of the InvocationHandler as the object the Proxy asks to do all the real work after it's received the method calls.

Okay, let's step through how to use the dynamic proxy...

### Matchmaking in Objectville

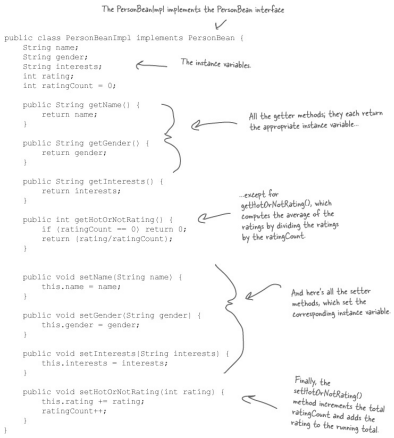

Every town needs a matchmaking service, right? You've risen to the task and implemented a dating service for Objectville. You've also tried to be innovative by including a "Hot or Not" feature in the service where participants can rate each other – you figure this keeps your customers engaged and looking through possible matches; it also makes things a lot more fun.

Your service revolves around a Person bean that allows you to set and get information about a person:




Now let's check out the implementation...

### The PersonBean implementation





While we suspect other factors may be keeping Elroy from getting dates, he is right: you shouldn't be able to vote for yourself or to change another customer's data. The way our PersonBean is defined, any client can call any of the methods.

This is a perfect example of where we might be able to use a Protection Proxy. What's a Protection Proxy? It's a proxy that controls access to an object based on access rights. For instance, if we had an employee object, a protection proxy might allow the employee to call certain methods on the object, a manager to call additional methods (like setSalary()), and a human resources employee to call any method on the object.

In our dating service we want to make sure that a customer can set his own information while preventing others from altering it. We also want to allow just the opposite with the HotOrNot ratings: we want the other customers to be able to set the rating, but not that particular customer. We also have a number of getter methods in the PersonBean, and because none of these return private information, any customer should be able to call them.

### Five minute drama: protecting subjects

The Internet bubble seems a distant memory; those were the days when all you needed to do to find a better, higher-paying job was to walk across the street. Even agents for software developers were in vogue...


## Big Picture: creating a Dynamic Proxy for the PersonBean

We have a couple of problems to fix: customers shouldn't be changing their own HotOrNot rating and customers shouldn't be able to change other customers' personal information. To fix these problems we're going to create two proxies: one for accessing your own PersonBean object and one for accessing another customer's PersonBean object. That way, the proxies can control what requests can be made in each circumstance.

To create these proxies we're going to use the Java API's dynamic proxy that you saw a few pages back. Java will create two proxies for us; all we need to do is supply the handlers that know what to do when a method is invoked on the proxy.

**Step one:**

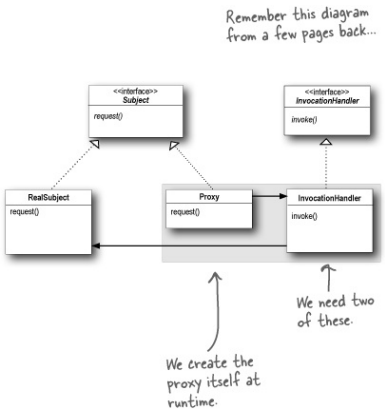
- Create two **InvocationHandlers.**

  InvocationHandlers implement the behavior of the proxy. As you'll see Java will take care of creating the actual proxy class and object, we just need to supply a handler that knows what to do when a method is called on it.

**Step two:**

- Write the code that creates the dynamic proxies.

  We need to write a little bit of code to generate the proxy class and instantiate it. We'll step through this code in just a bit.

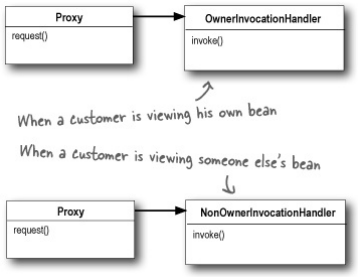



**Step three:**

- Wrap any PersonBean object with the appropriate proxy.

  When we need to use a PersonBean object, either it's the object of the customer himself (in that case, will call him the "owner"), or it's another user of the service that the customer is checking out (in that case we'll call him "non-owner").

  In either case, we create the appropriate proxy for the PersonBean.

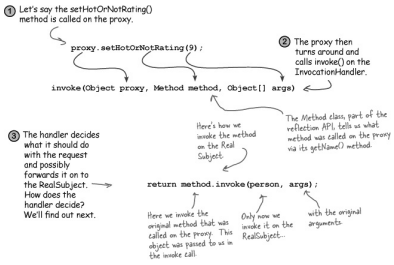



### Step one: creating Invocation Handlers

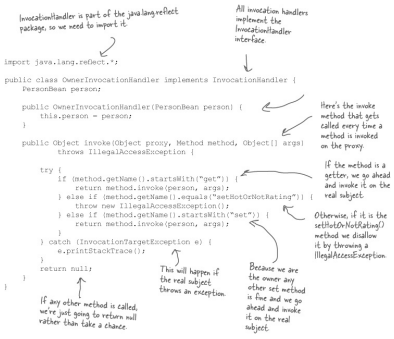
We know we need to write two invocation handlers, one for the owner and one for the non-owner. But what are invocation handlers? Here's the way to think about them: when a method call is made on the proxy, the proxy forwards that call to your invocation handler, but *not* by calling the invocation handler's corresponding method. So, what does it call? Have a look at the InvocationHandler interface:

| <<interface>><br>***OwnerInvocationHandler*** |
| --- |
| invoke() |

There's only one method, invoke(), and no matter what methods get called on the proxy, the invoke() method is what gets called on the handler. Let's see how this works:




When invoke() is called by the proxy, how do you know what to do with the call? Typically, you'll examine the method that was called on the proxy and make decisions based on the method's name and possibly its arguments. Let's implement the OwnerInvocationHandler to see how this works:

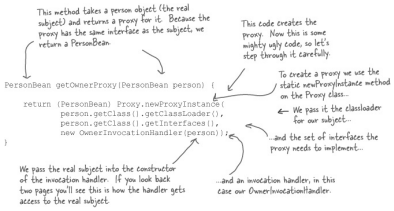



**EXERCISE**

The NonOwnerInvocationHandler works just like the OwnerInvocationHandler except that it *allows* calls to setHotOrNotRating() and it *disallows* calls to any other set method. Go ahead and write this handler yourself.

### Step two: creating the Proxy class and instantiating the Proxy object

Now, all we have left is to dynamically create the proxy class and instantiate the proxy object. Let's start by writing a method that takes a PersonBean and knows how to create an owner proxy for it. That is, we're going to create the kind of proxy that forwards its method calls to the OwnerInvocationHandler. Here's the code:



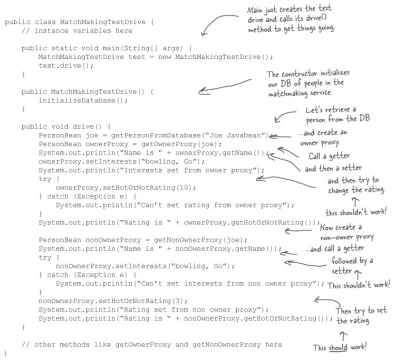

**SHARPEN YOUR PENCIL**

While it is a little complicated, there isn't much to creating a dynamic proxy. Why don't you write getNonOwnerProxy(), which returns a proxy for the NonOwnerInvocationHandler:

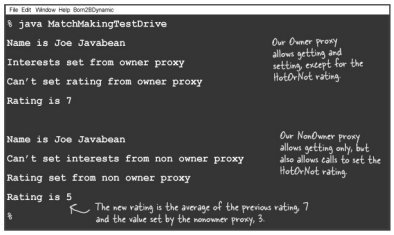
Take it further: can you write one method getProxy() that takes a handler and a person and returns a proxy that uses that handler?

## Testing the matchmaking service

Let's give the matchmaking service a test run and see how it controls access to the setter methods based on the proxy that is used.




## Running the code...



## THERE ARE NO DUMB QUESTIONS

**Q:** **Q: So what exactly is the "dynamic" aspect of dynamic proxies? Is it that I'm instantiating the proxy and setting it to a handler at runtime?**

**A:** **A:** *No, the proxy is dynamic because its class is created at runtime. Think about it: before your code runs there is no proxy class; it is created on demand from the set of interfaces you pass it.*

**Q:** **Q: My InvocationHandler seems like a very strange proxy, it doesn't implement any of the methods of the class it's proxying.**

**A:** **A:** *That is because the InvocationHandler isn't a proxy – it is a class that the proxy dispatches to for handling method calls. The proxy itself is created dynamically at runtime by the static Proxy.newProxyInstance() method.*

**Q:** **Q: Is there any way to tell if a class is a Proxy class?**

**A:** **A:** *Yes. The Proxy class has a static method called isProxyClass(). Calling this method with a class will return true if the class is a dynamic proxy class. Other than that, the proxy class will act like any other class that implements a particular set of interfaces.*

**Q:** **Q: Are there any restrictions on the types of interfaces I can pass into newProxyInstance()?**

**A:** **A:** *Yes, there are a few. First, it is worth pointing out that we always pass newProxyInstance() an array of interfaces – only interfaces are allowed, no classes. The major restrictions are that all non-public interfaces need to be from the same package. You also can't have interfaces with clashing method names (that is, two interfaces with a method with the same signature). There are a few other minor nuances as well, so at some point you should take a look at the fine print on dynamic proxies in the javadoc.*

**Q:** **Q: Why are you using skeletons? I thought we got rid of those back in Java 1.2.**

**A:** **A:** *You're right; we don't need to actually generate skeletons. As of Java 1.2, the RMI runtime can dispatch the client calls directly to the remote service using reflection. But we like to show the skeleton, because conceptually it helps you to understand that there is something under the covers that's making that communication between the client stub and the remote service happen.*

**Q:** **Q: I heard that in Java 5, I don't even need to generate stubs anymore either. Is that true?**

**A:** **A:** *It sure is. In Java 5, RMI and Dynamic Proxy got together and now stubs are generated dynamically using Dynamic Proxy. The remote object's stub is a java.lang.reflect.Proxy instance (with an invocation handler) that is automatically generated to handle all the details of getting the local method calls by the client to the remote object. So, now you don't have to use rmic at all; everything you need to get a client talking to a remote object is handled for you behind the scenes.*

**WHO DOES WHAT?**

Match each pattern with its description:

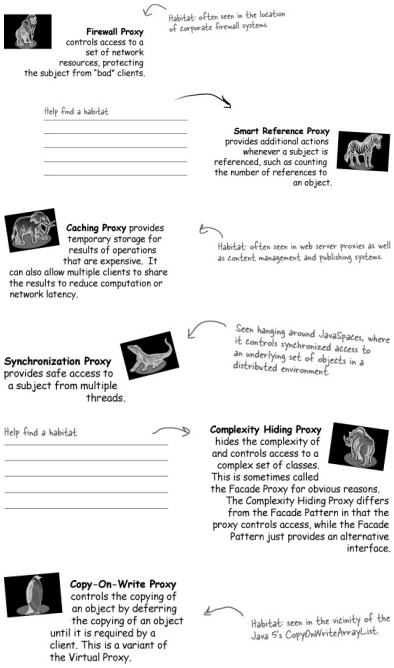
| | |
|---|---|
| Decorator | Wraps another object and provides a different interface to it |
| Facade | Wraps another object and provides additional behavior for it |
| Proxy | Wraps another object to control access to it |
| Adapter | Wraps a bunch of objects to simplify their interface |

## The Proxy Zoo

Welcome to the Objectville Zoo!

You now know about the remote, virtual and protection proxies, but out in the wild you're going to see lots of mutations of this pattern. Over here in the Proxy corner of the zoo we've got a nice collection of wild proxy patterns that we've captured for your study.

Our job isn't done; we are sure you're going to see more variations of this pattern in the real world, so give us a hand in cataloging more proxies. Let's take a look at the existing collection:




Field Notes: please add your observations of other proxies in the wild here:

_____

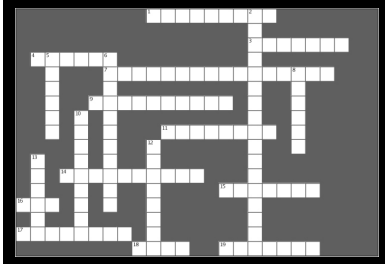
_____

_____

_____

_____

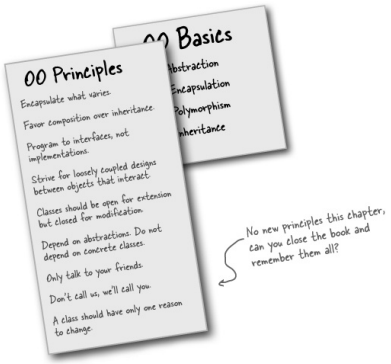
It's been a LONG chapter. Why not unwind by doing a crossword puzzle before it ends?



| Across | Down |
|---|---|
| 1. Group of first CD cover displayed (two words)<br>3. Proxy that stands in for expensive objects<br>4. We took one of these to learn RMI<br>7. Remote _____ was used to implement the gumball machine monitor (two words)<br>9. Software developer agent was being this kind of proxy<br>11. In RMI, the object that takes the network requests on the service side<br>14. Proxy that protects method calls from unauthorized callers<br>15. A _____ proxy class is created at runtime<br>16. Place to learn about the many proxy variants<br>17. Commonly used proxy for web services (two words)<br>18. In RMI, the proxy is called this<br>19. The CD viewer used this kind of proxy | 2. Java's dynamic proxy forwards all requests to this (two words)<br>5. Group that did the album MCMXC A.D.<br>6. This utility acts as a lookup service for RMI<br>8. Why Elroy couldn't get dates<br>10. Similar to proxy, but with a different purpose<br>12. Objectville Matchmaking gimmick (three words)<br>13. Our first mistake: the gumball machine reporting was not_____ |

## Tools for your Design Toolbox

**Your design toolbox is almost full; you're prepared for almost any design problem that comes your way.**




**BULLET POINTS**

- The Proxy Pattern provides a representative for another object in order to control the client's access to it. There are a number of ways it can manage that access.
- A Remote Proxy manages interaction between a client and a remote object.
- A Virtual Proxy controls access to an object that is expensive to instantiate.
- A Protection Proxy controls access to the methods of an object based on the caller.
- Many other variants of the Proxy Pattern exist including caching proxies, synchronization proxies, firewall proxies, copy-on-write proxies, and so on.
- Proxy is structurally similar to Decorator, but the two differ in their purpose.
- The Decorator Pattern adds behavior to an object, while a Proxy controls access.
- Java's built-in support for Proxy can build a dynamic proxy class on demand and dispatch all calls on it to a handler of your choosing.
- Like any wrapper, proxies will increase the number of classes and objects in your designs.

## Exercise Solutions

**EXERCISE**

The NonOwnerInvocationHandler works just like the OwnerInvocationHandler, except that it allows calls to setHotOrNotRating() and it disallows calls to any other set method. Go ahead and write this handler yourself:

```
import java.lang.reflect.*;

public class NonOwnerInvocationHandler implements InvocationHandler
    PersonBean person;

    public NonOwnerInvocationHandler(PersonBean person) {
        this.person = person;
    }

    public Object invoke(Object proxy, Method method, Object[] args
            throws IllegalAccessException {

        try {
            if (method.getName().startsWith("get")) {
                return method.invoke(person, args);
            } else if (method.getName().equals("setHotOrNotRating")
                return method.invoke(person, args);
            } else if (method.getName().startsWith("set")) {
                throw new IllegalAccessException();
            }
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

### DESIGN CLASS

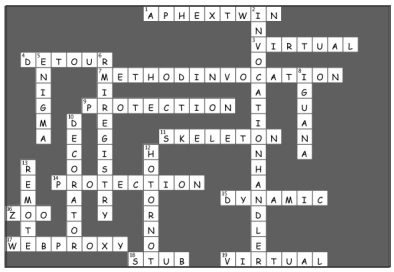
Our ImageProxy class appears to have two states that are controlled by conditional statements. Can you think of another pattern that might clean up this code? How would you redesign ImageProxy?

Use State Pattern: implement two states, ImageLoaded and ImageNotLoaded. Then put the code from the if statements into their respective states. Start in the ImageNotLoaded state and then transition to the ImageLoaded state once the ImageIcon had been retrieved.

### SHARPEN YOUR PENCIL

While it is a little complicated, there isn't much to creating a dynamic proxy. Why don't you write getNonOwnerProxy(), which returns a proxy for the NonOwnerInvocationHandler:

```
PersonBean getNonOwnerProxy(PersonBean person) {

    return (PersonBean) Proxy.newProxyInstance(
            person.getClass().getClassLoader(),
            person.getClass().getInterfaces(),
            new NonOwnerInvocationHandler(person));
}
```



**READY-BAKE CODE: THE CODE FOR THE CD COVER VIEWER**

```
package headfirst.proxy.virtualproxy;

import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;
public class ImageProxyTestDrive {
    ImageComponent imageComponent;
    JFrame frame = new JFrame("CD Cover Viewer");
    JMenuBar menuBar;
    JMenu menu;
    Hashtable cds = new Hashtable();

    public static void main (String[] args) throws Exception {
        ImageProxyTestDrive testDrive = new ImageProxyTestDrive();
    }

    public ImageProxyTestDrive() throws Exception{
        cds.put("Ambient: Music for Airports","http://images.amazon
B000003S2K.01.LZZZZZZZ.jpg");
        cds.put("Buddha Bar","http://images.amazon.com/images/P/B00
jpg");
        cds.put("Ima","http://images.amazon.com/images/P/B000005IRM
        cds.put("Karma","http://images.amazon.com/images/P/B000005D
        cds.put("MCMXC A.D.","http://images.amazon.com/images/P/B00
jpg");
        cds.put("Northern Exposure","http://images.amazon.com/image
LZZZZZZZ.jpg");
        cds.put("Selected Ambient Works, Vol. 2","http://images.ama
B000002MNZ.01.LZZZZZZZ.jpg");
        cds.put("oliver","http://www.cs.yale.edu/homes/freeman-elis
sm.jpg");

        URL initialURL = new URL((String)cds.get("Selected Ambient
        menuBar = new JMenuBar();
        menu = new JMenu("Favorite CDs");
        menuBar.add(menu);
        frame.setJMenuBar(menuBar);

        for(Enumeration e = cds.keys(); e.hasMoreElements();) {
            String name = (String)e.nextElement();
            JMenuItem menuItem = new JMenuItem(name);
            menu.add(menuItem);
            menuItem.addActionListener(new ActionListener() {
                  public void actionPerformed(ActionEvent event) {
                     imageComponent.setIcon(new ImageProxy(getCDUrl
mand()));
                    frame.repaint();
                  }
            });
        }

        // set up frame and menus

        Icon icon = new ImageProxy(initialURL);
        imageComponent = new ImageComponent(icon);
        frame.getContentPane().add(imageComponent);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(800,600);
        frame.setVisible(true);

    }
    URL getCDUrl(String name) {
        try {
            return new URL((String)cds.get(name));
        } catch (MalformedURLException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

```
package headfirst.proxy.virtualproxy;

import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class ImageProxy implements Icon {
    ImageIcon imageIcon;
    URL imageURL;
    Thread retrievalThread;
    boolean retrieving = false;

    public ImageProxy(URL url) { imageURL = url; }

    public int getIconWidth() {
        if (imageIcon != null) {
            return imageIcon.getIconWidth();
        } else {
            return 800;
        }
    }

    public int getIconHeight() {
        if (imageIcon != null) {
            return imageIcon.getIconHeight();
        } else {
            return 600;
        }
    }

    public void paintIcon(final Component c, Graphics g, int x, int
        if (imageIcon != null) {
            imageIcon.paintIcon(c, g, x, y);
        } else {
            g.drawString("Loading CD cover, please wait...", x+300,
```

```
            if (!retrieving) {
                retrieving = true;

                retrievalThread = new Thread(new Runnable() {
                    public void run() {
                        try {
                            imageIcon = new ImageIcon(imageURL, "C
                            c.repaint();
                        } catch (Exception e) {
                            e.printStackTrace();
                        }
                    }
                });
                retrievalThread.start();
            }
        }
    }
}
```

```
package headfirst.proxy.virtualproxy;

import java.awt.*;
import javax.swing.*;

class ImageComponent extends JComponent {
    private Icon icon;

    public ImageComponent(Icon icon) {
        this.icon = icon;
    }

    public void setIcon(Icon icon) {
        this.icon = icon;
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int w = icon.getIconWidth();
        int h = icon.getIconHeight();
        int x = (800 - w)/2;
        int y = (600 - h)/2;
        icon.paintIcon(this, g, x, y);
    }
}
```

PREV
10. The State Pattern: The State of Things

NEXT
12. Compound Patterns: Patterns of Patterns