

Tecniche di Progettazione: Design Patterns

GoF: Proxy

Revisit the Gumball machine example

- ▶ The same example covered in the State pattern
- ▶ Now we want to add some monitor to a collection of Gumball machines



Gumball Class

```
public class GumballMachine {  
    // other instance variables  
    String location;
```

A location is just a String.

```
    public GumballMachine(String location, int count) {  
        // other constructor code here  
        this.location = location;  
    }
```

The location is passed into the constructor and stored in the instance variable.

```
    public String getLocation() {  
        return location;  
    }
```

Let's also add a getter method to grab the location when we need it.

```
    // other methods here  
}
```



Gumball Monitor

```
public class GumballMonitor {
    GumballMachine machine;

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

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



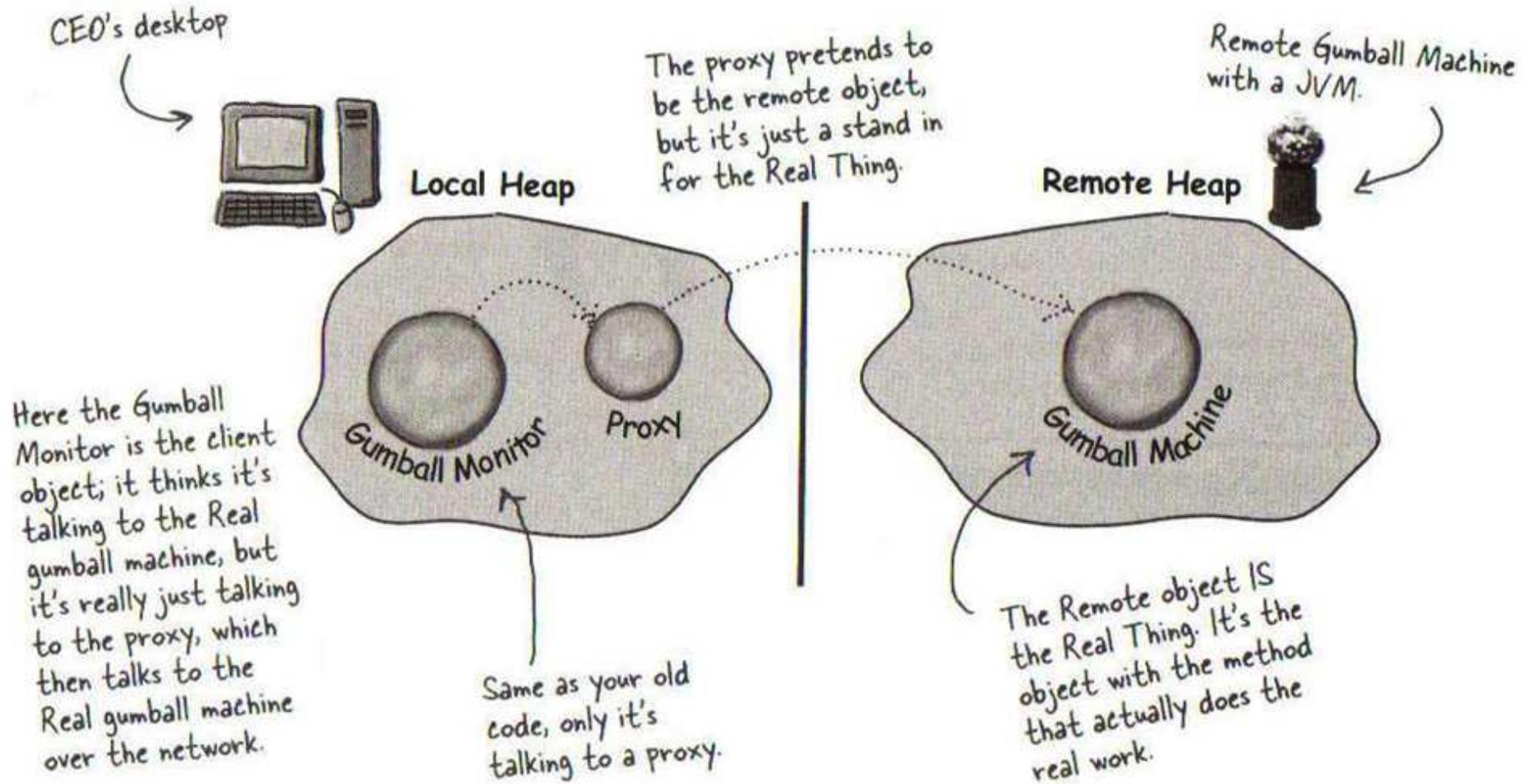
The monitor takes the machine in its constructor and assigns it to the machine instance variable.



Our report method just prints a report with location, inventory and the machine's state.



Role of the remote Proxy

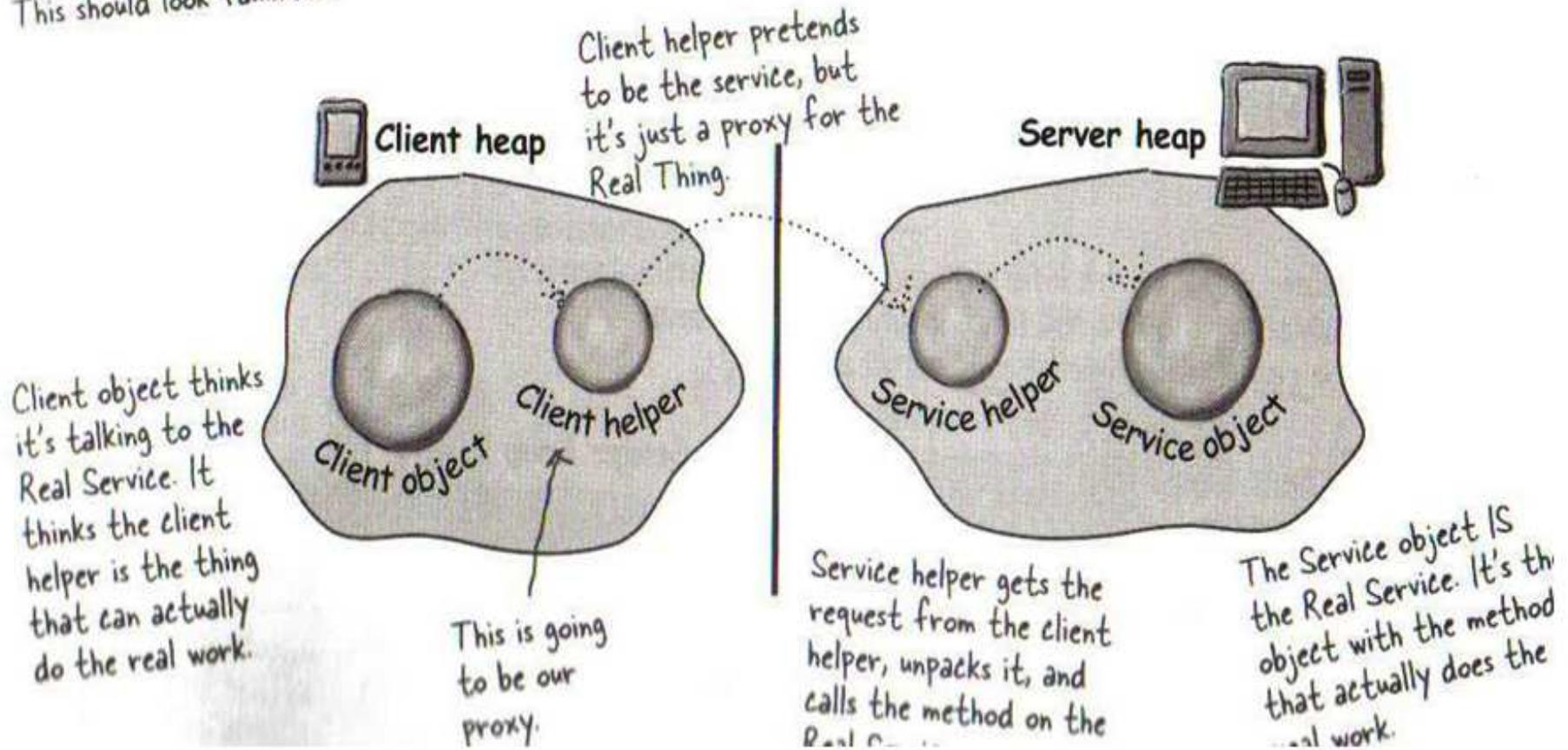


RMI Detour in looking at Proxy Pattern

- 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

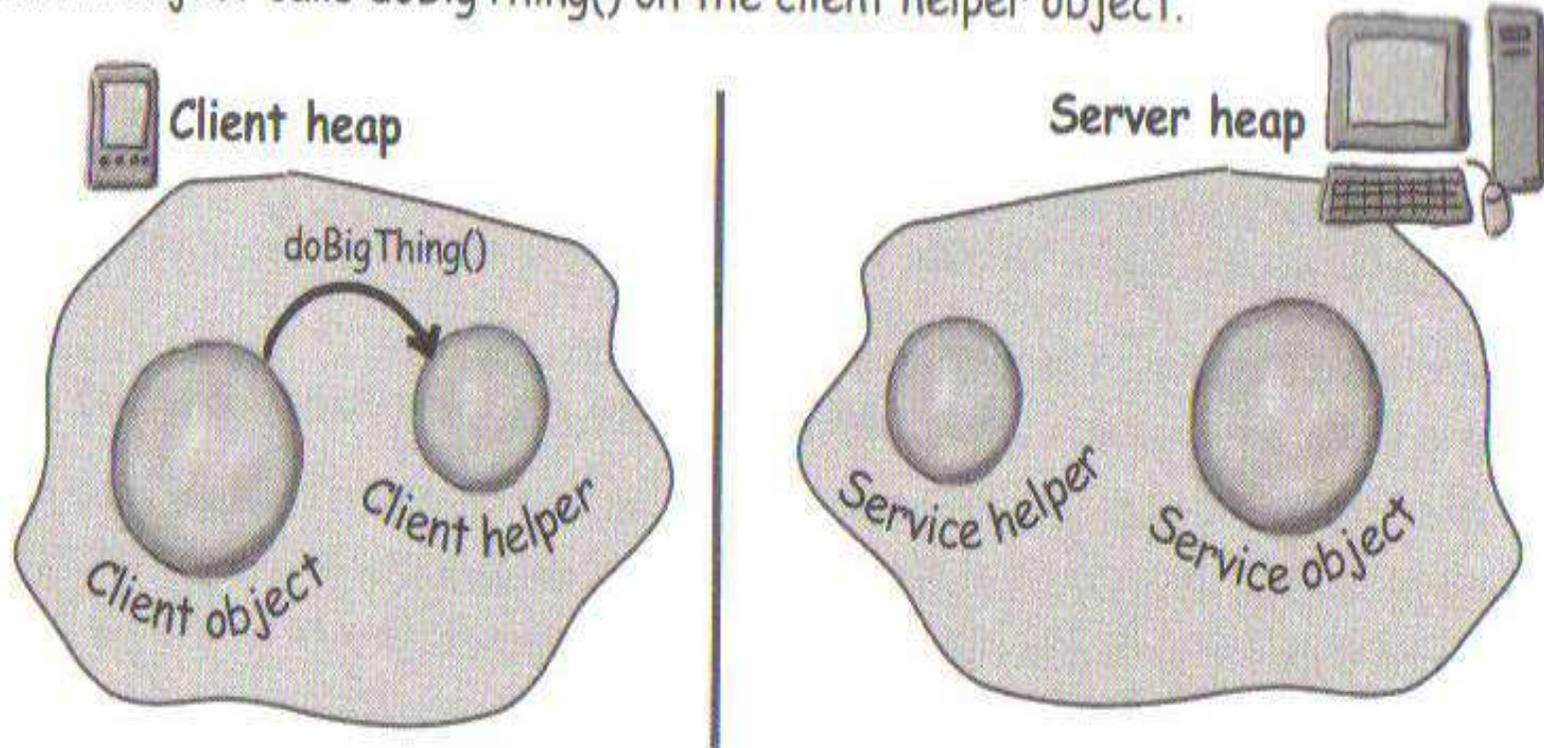
This should look familiar...



How the method call happens

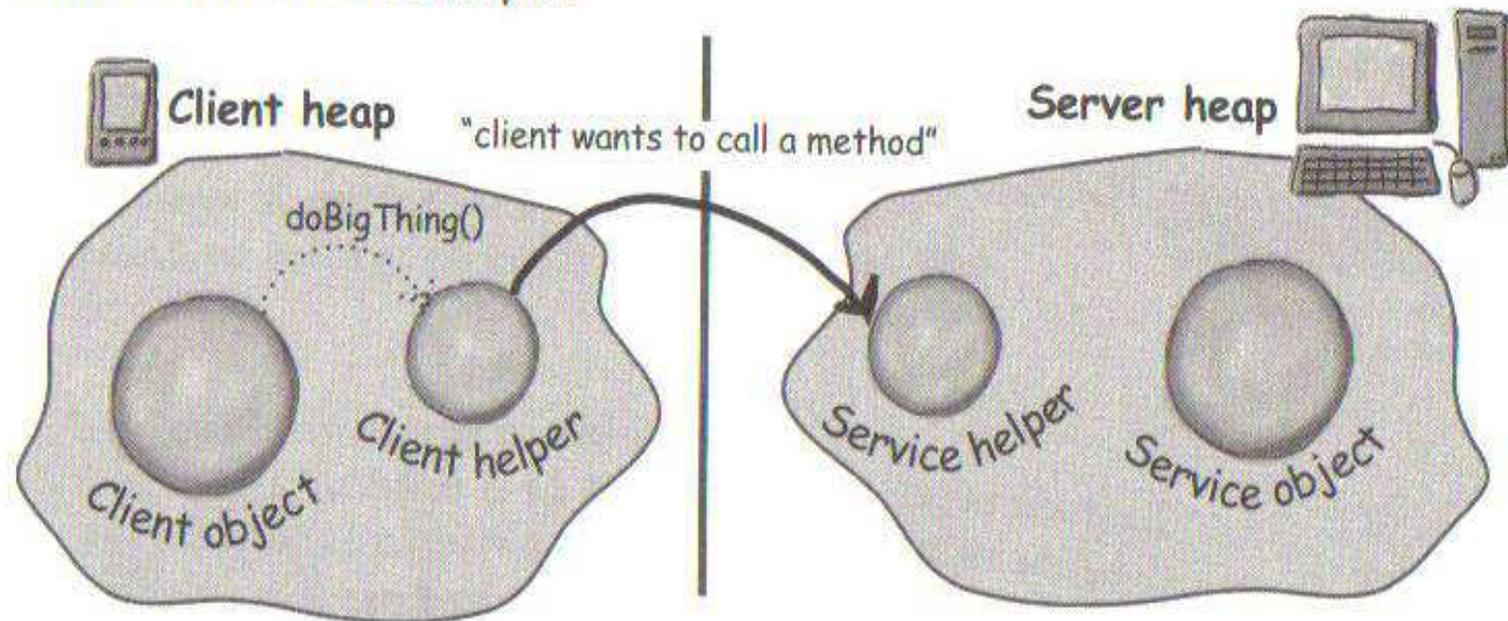
Client calls method

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



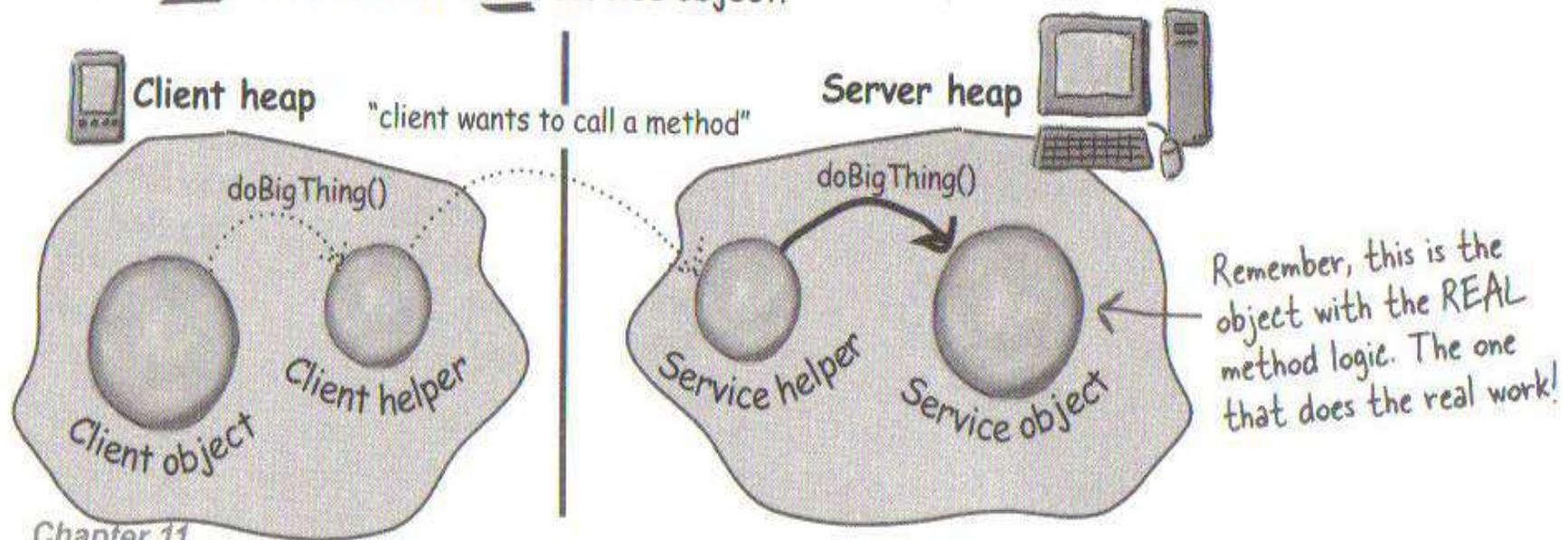
Client Helper forwards to service helper

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



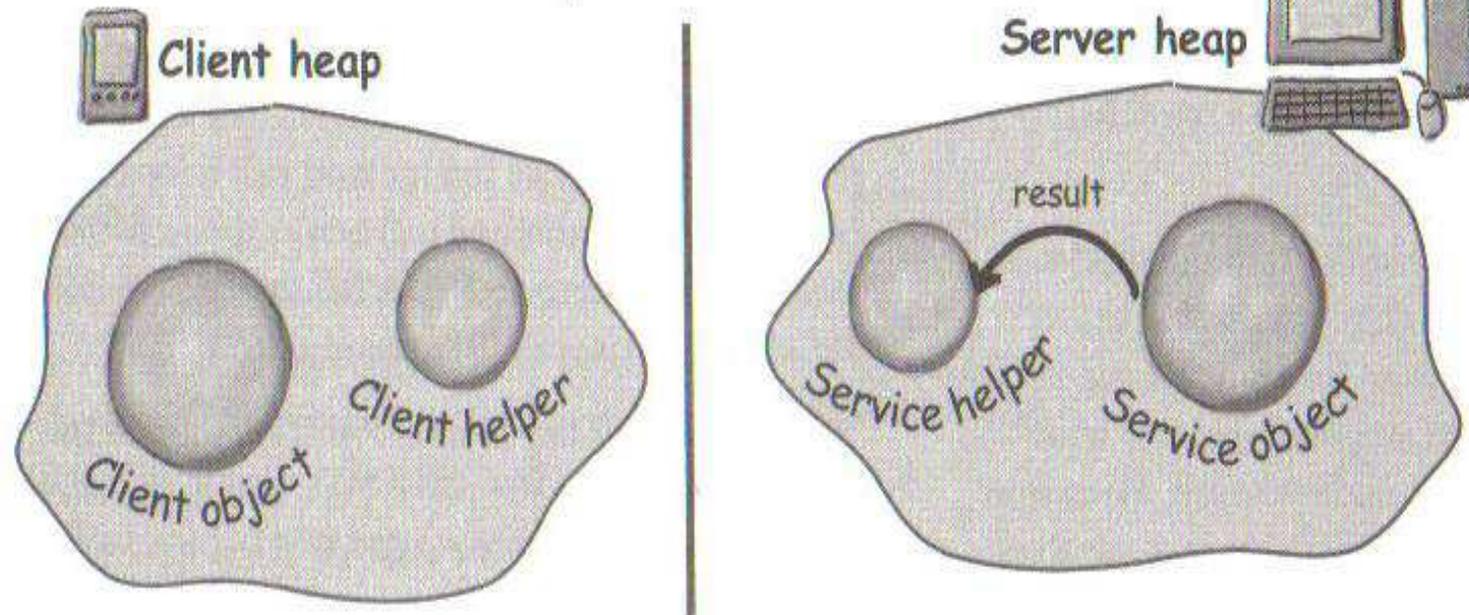
Service helper calls the real object

- 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.



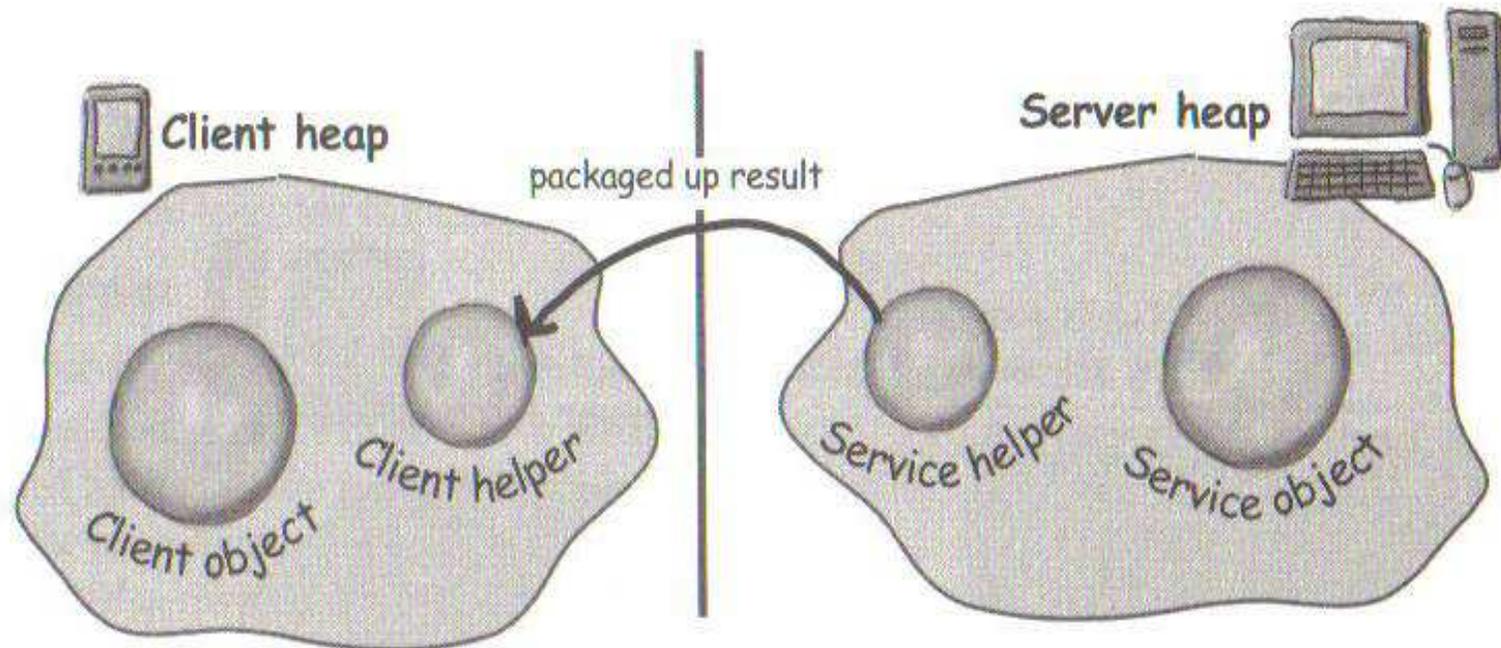
Real object returns result

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



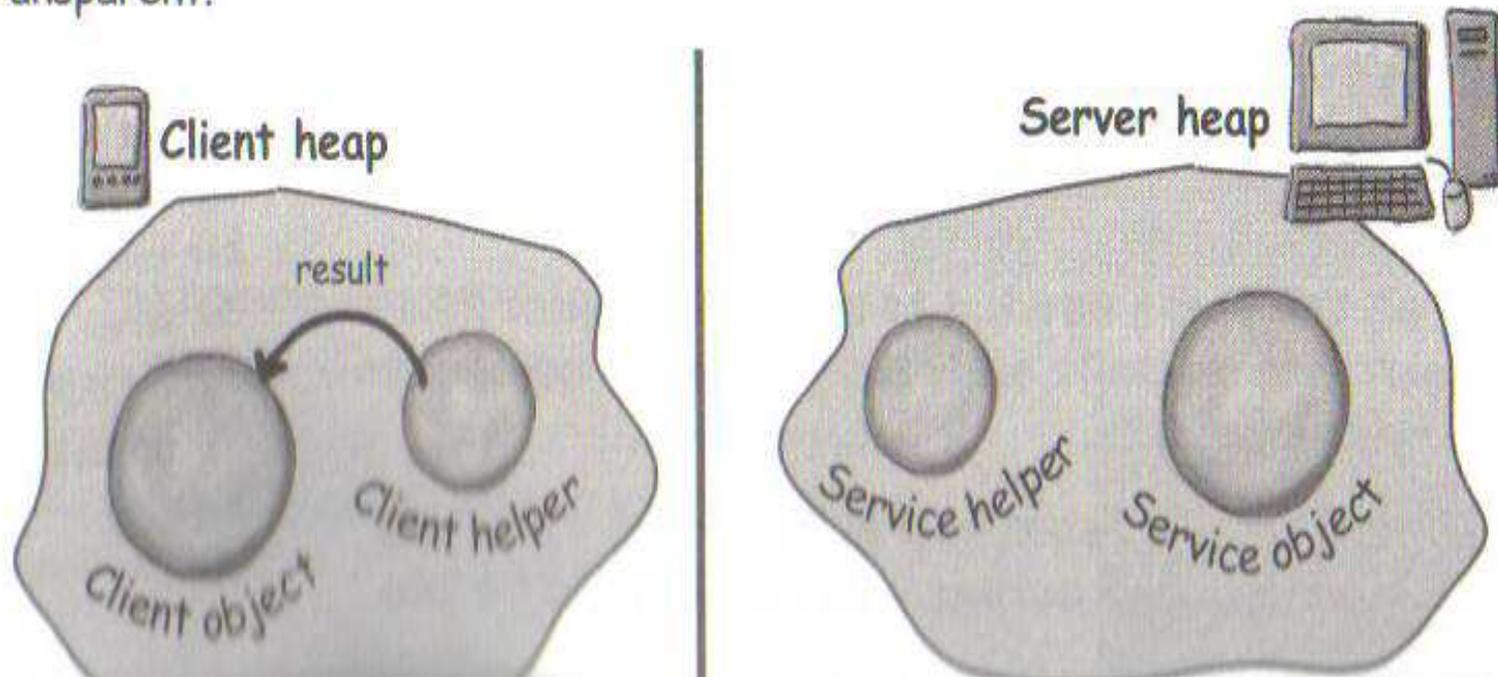
Service helper forwards result to client helper

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

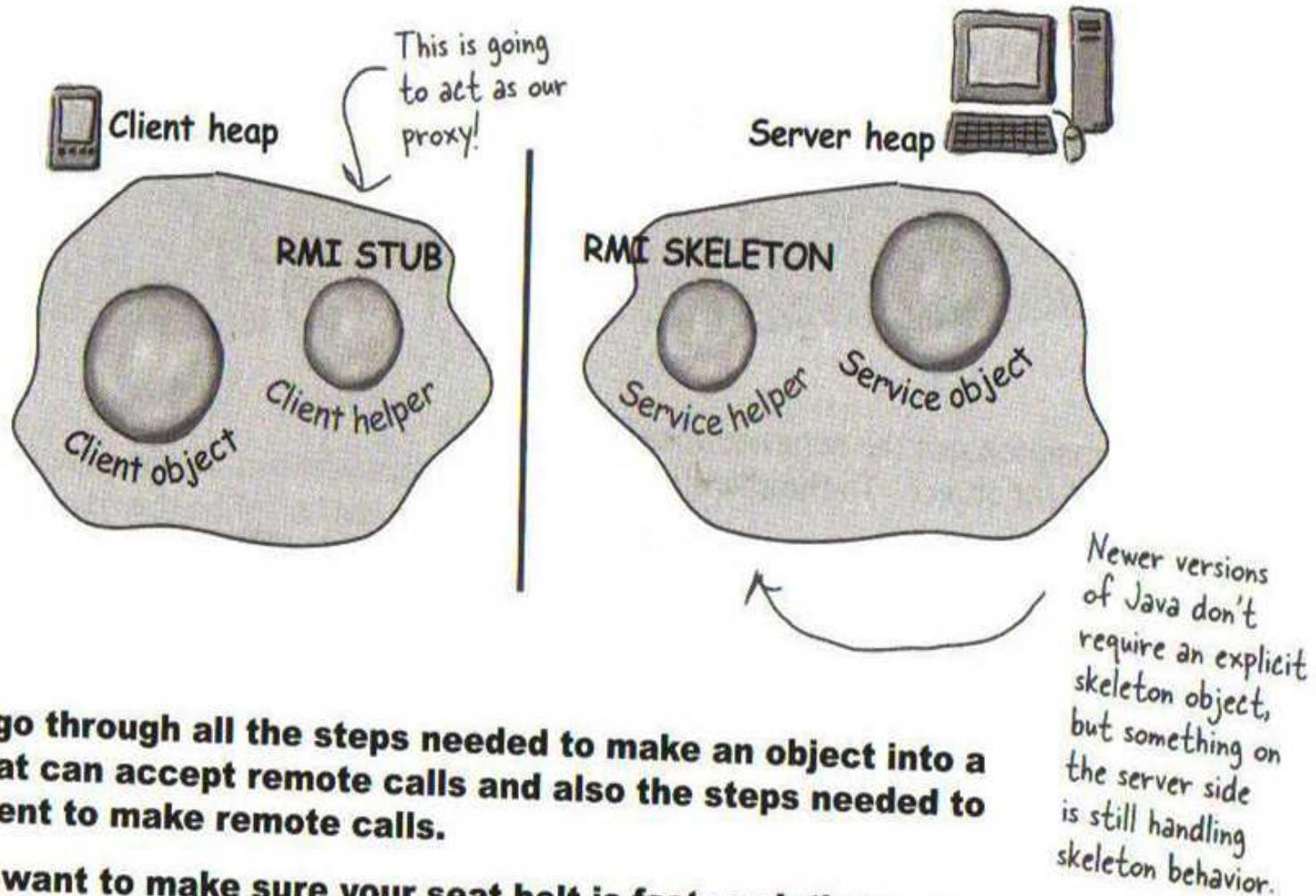


Client helper returns result to client

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



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.

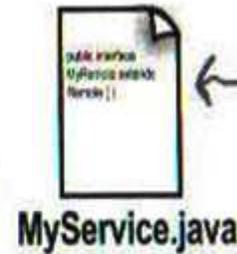
You might want to make sure your seat belt is fastened; there are a lot of steps and a few bumps and curves - but nothing to be too worried about.

Steps in using Java RMI

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!

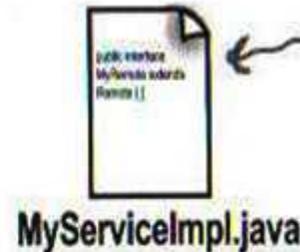


← This interface defines the remote methods that you want clients to call.

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!).



← The Real Service; the class with the methods that do the real work. It implements the remote interface.

Additional steps

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:

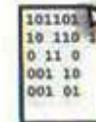
Start the **remote service**

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.

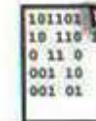
Running `rmic` against the actual service implementation class...

```
File Edit Window Help Eat
%rmic MyServiceImpl
```

...spits out two new classes for the helper objects:



MyServiceImpl_Stub.class



MyServiceImpl_Skel.class

```
File Edit Window Help Drink
%rmiregistry
```

Run this in a separate terminal.

```
File Edit Window Help BeMerry
%java MyServiceImpl
```

STEP 1 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.

STEP 1 Remote Interface

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.

```
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 2 Remote Implementation

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 MyRemote {
```



STEP 2 Remote Implementation

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 object, RMI swaps the service for the stub and puts the stub in the registry.

STEP 3 Create Stubs & Skeletons

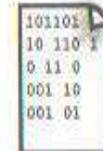
① Run `rmic` on the remote implementation class (not the remote interface)

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).

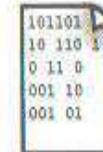
Notice that you don't say ".class" on the end. Just the class name.

```
File Edit Window Help Whuffie
%rmic MyRemoteImpl
```

RMIC generates two new classes for the helper objects.



MyRemoteImpl_Stub.class

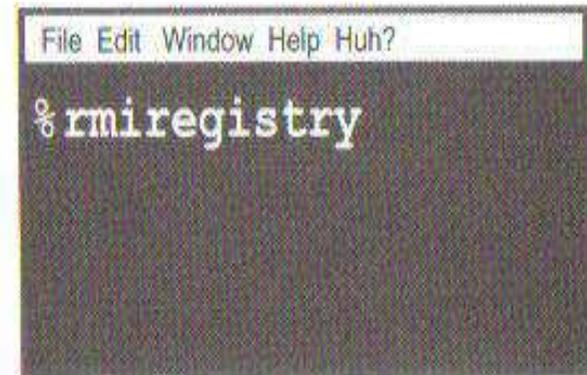


MyRemoteImpl_Skel.class

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.

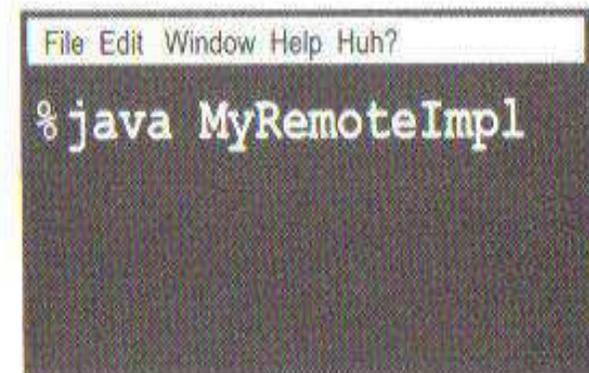


```
File Edit Window Help Huh?  
% rmiregistry
```

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.



```
File Edit Window Help Huh?  
% java MyRemoteImpl
```

Complete code for the server side



The Remote interface:

```
import java.rmi.*;  
  
public interface MyRemote extends Remote {  
    public String sayHello() throws RemoteException;  
}
```

RemoteException and Remote interface are in java.rmi package.

Your interface MUST extend java.rmi.Remote

All of your remote methods must declare a RemoteException.

The Remote service (the implementation):

```
import java.rmi.*;  
import java.rmi.server.*;
```

← UnicastRemoteObject is in the java.rmi.server package.

↘ Extending UnicastRemoteObject is the easiest way to make a remote object.

```
public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
```

```
    public String sayHello() {  
        return "Server says, 'Hey'";  
    }  
}
```

← You have to implement all the interface methods, of course. But notice that you do NOT have to declare the RemoteException.

↖ You MUST implement your remote interface!!

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

↖ Your superclass constructor (for UnicastRemoteObject) declares an exception, so YOU must write a constructor, because it means that your constructor is calling risky code (its super constructor).

```
    public static void main (String[] args) {
```

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

↖ Make the remote object, then 'bind' it to the rmi registry using the static Naming.rebind(). The name you register it under is the name clients will use to look it up in the RMI registry.

Client talks to the stub



Code Up Close

The client always uses the remote implementation as the type of the service. In fact, the client never needs to know the actual class name of your remote service.



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

You have to cast it to the interface, since the lookup method returns type Object.

lookup() is a static method of the Naming class.



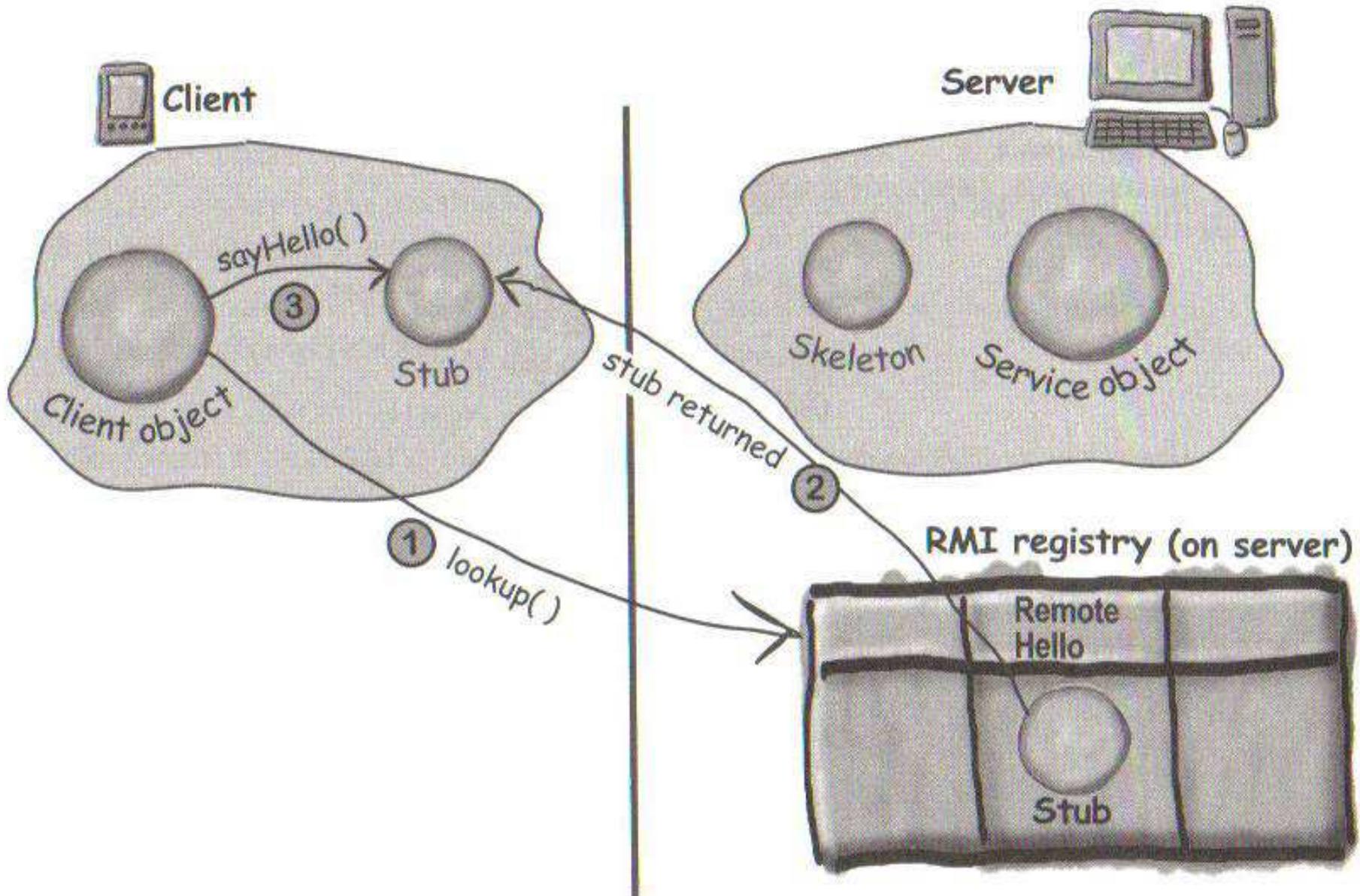
This must be the name that the service was registered under.



The host name or IP address where the service is running.



Hooking up client and server objects



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

```
import java.rmi.*;

public class MyRemoteClient {
    public static void main (String[] args) {
        new MyRemoteClient().go();
    }

    public void go() {
        try {
            MyRemote service = (MyRemote) Naming.lookup("rmi://127.0.0.1/RemoteHello");

            String s = service.sayHello();

            System.out.println(s);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

The Naming class (for doing the rmi registry lookup) is in the java.rmi package.

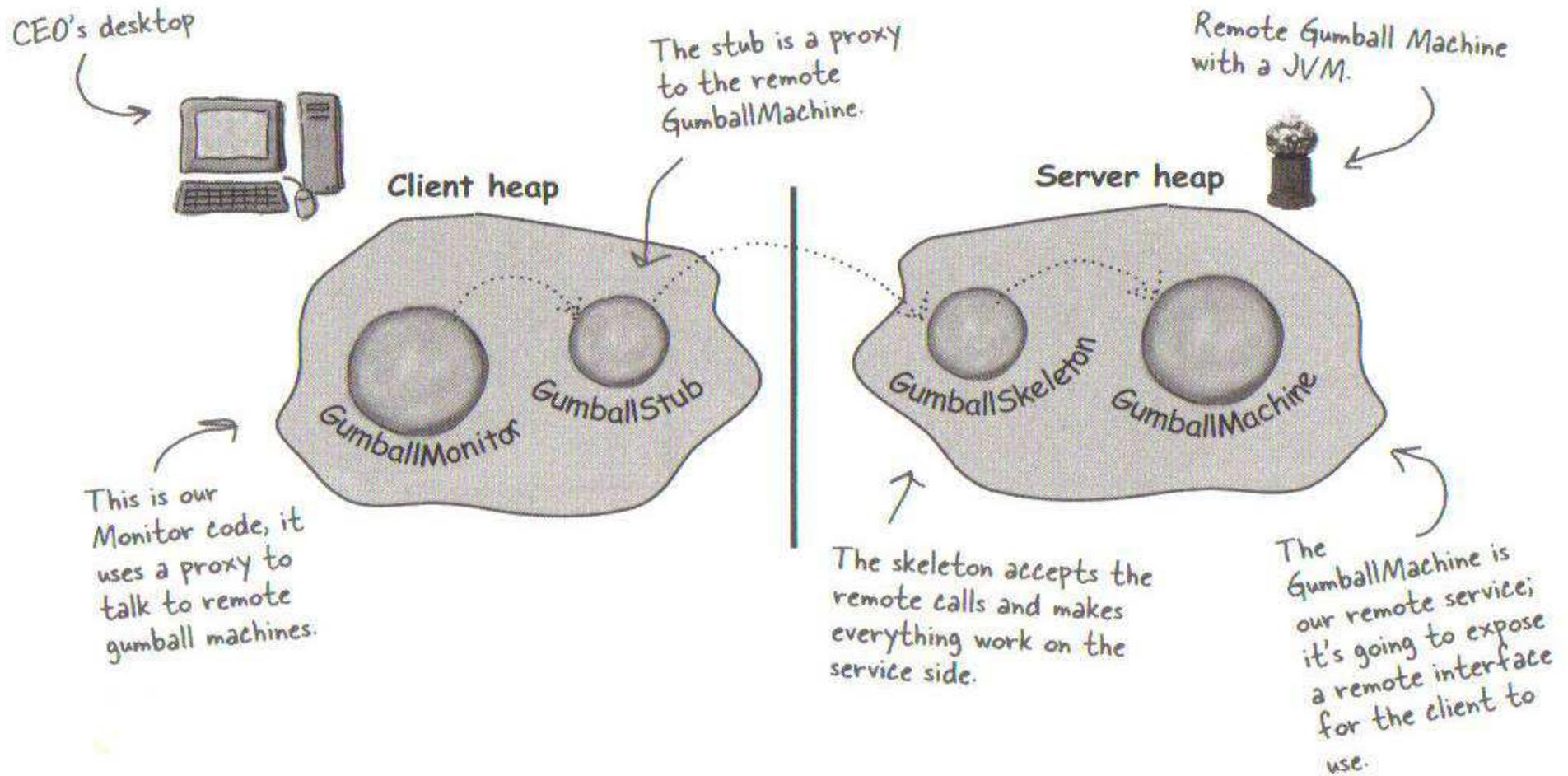
It comes out of the registry as type Object, so don't forget the cast.

You need the IP address or hostname.

and the name used to bind/rebind the service.

It looks just like a regular old method call! (Except it must acknowledge the RemoteException.)

Back to Gumball machine problem



Gumball Machine remote interface

- ▶ `import java.rmi.*;`
- ▶
- ▶ `public interface GumballMachineRemote extends Remote`
 - `{`
 - ▶ `public int getCount() throws RemoteException;`
 - ▶ `public String getLocation() throws RemoteException;`
 - ▶ `public State getState() throws RemoteException;`
 - ▶ `}`



State interface extends Serializable

- ▶ `import java.io.*;`
- ▶
- ▶ `public interface State extends Serializable {`
- ▶ `public void insertQuarter();`
- ▶ `public void ejectQuarter();`
- ▶ `public void turnCrank();`
- ▶ `public void dispense();`
- ▶ `}`



Use of keyword “transient”

```
public class NoQuarterState implements State {
    transient GumballMachine gumballMachine;

    public NoQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You inserted a quarter");
        gumballMachine.setState(gumballMachine.getHasQuarterState());
    }

    // other methods
}
```

The use of transient to ensure that the serialization does not involve this object as well.



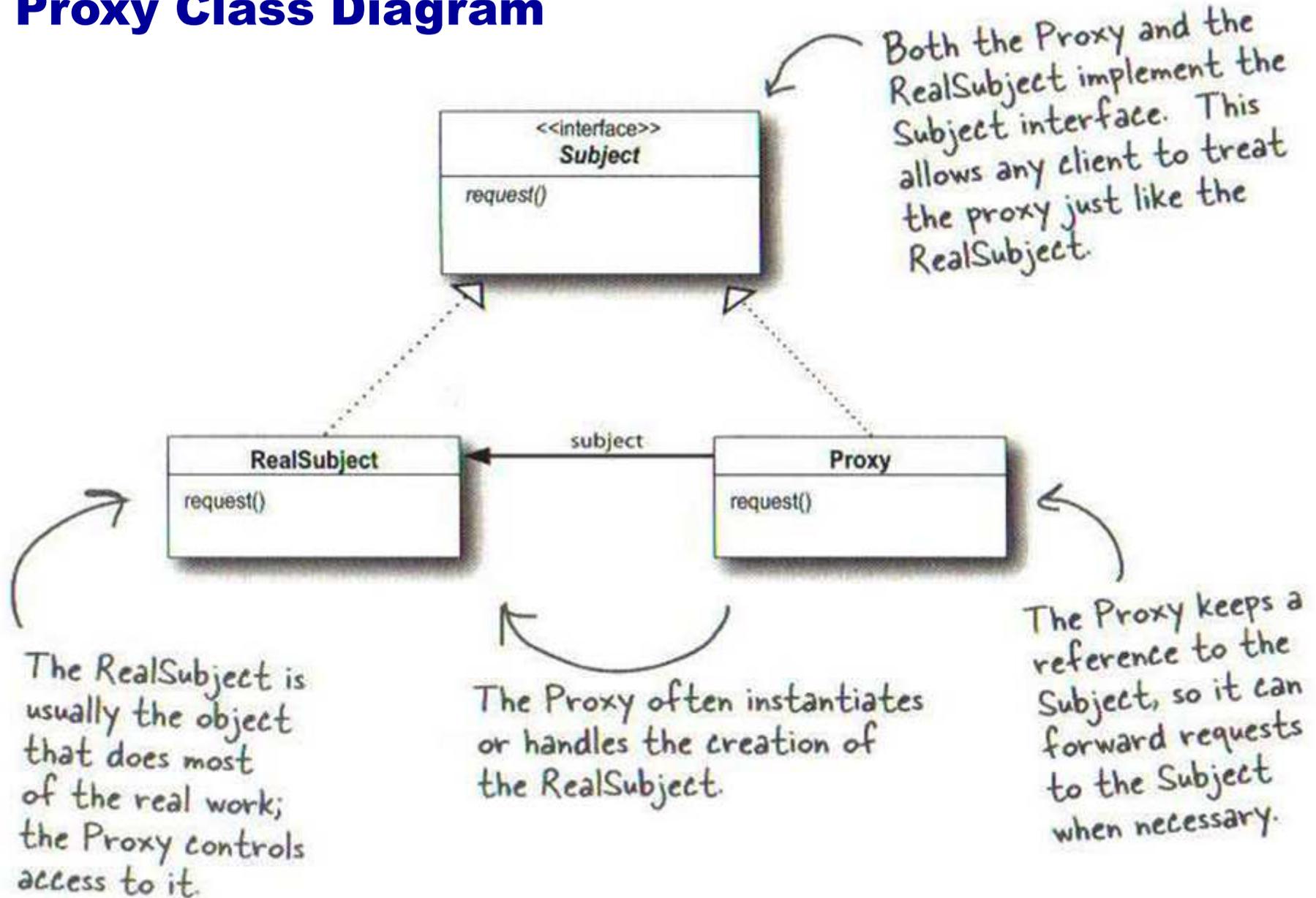
Proxy Pattern defined

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

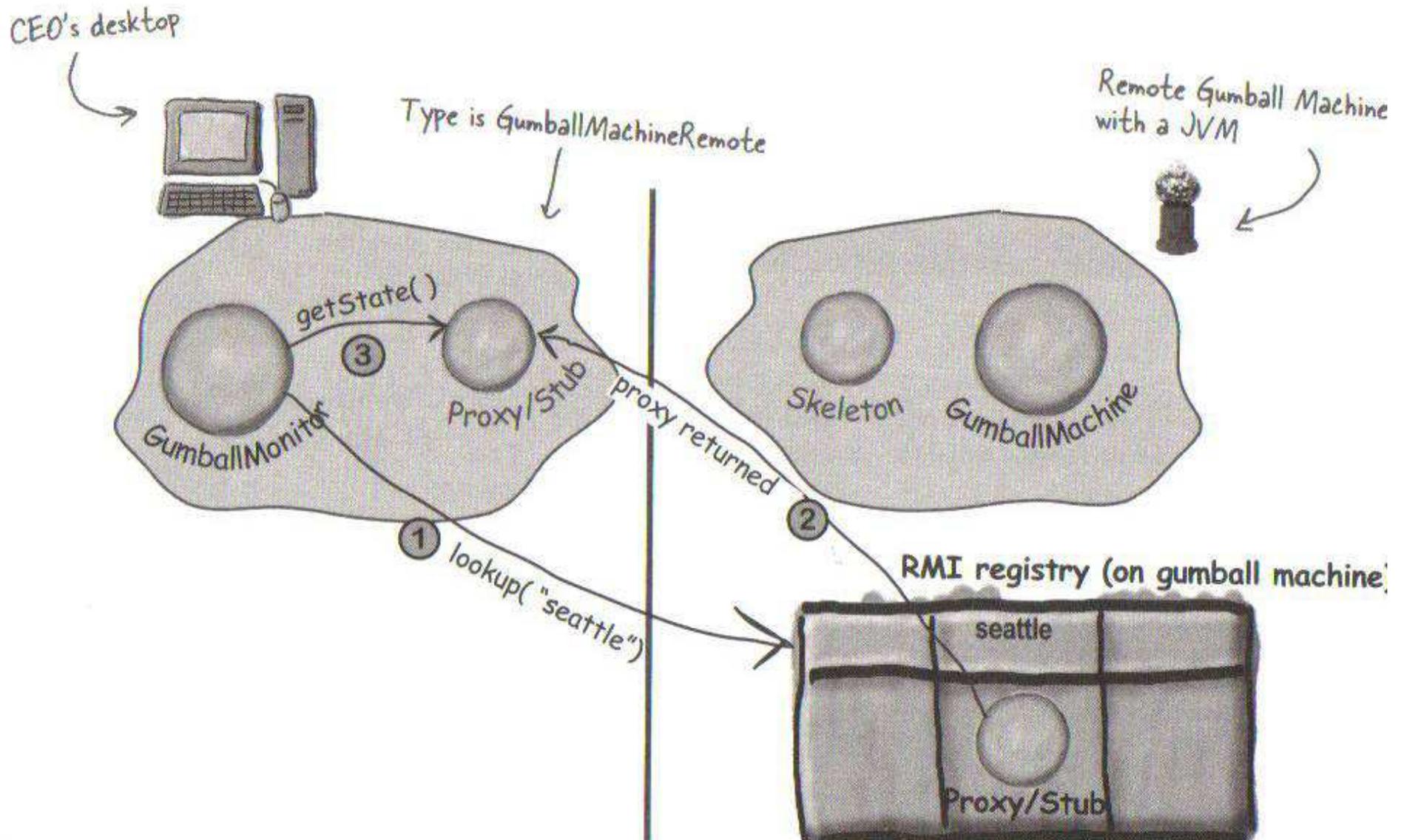
The proxy pattern is used to create a representative object that controls access to another object, which may be remote, expensive to create or in need of securing.



Proxy Class Diagram

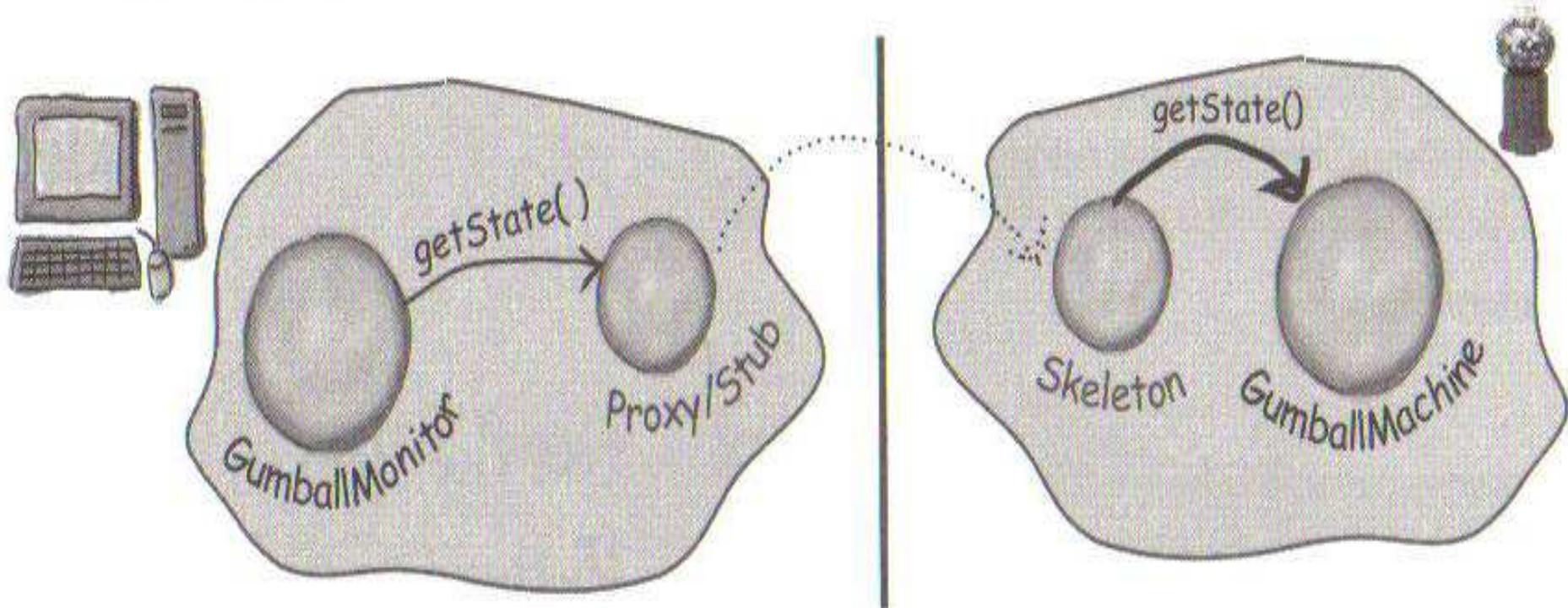


- 1 The CEO runs the monitor, which first grabs the proxies to the remote gumball machines and then calls `getState()` on each one (along with `getCount()` and `getLocation()`).

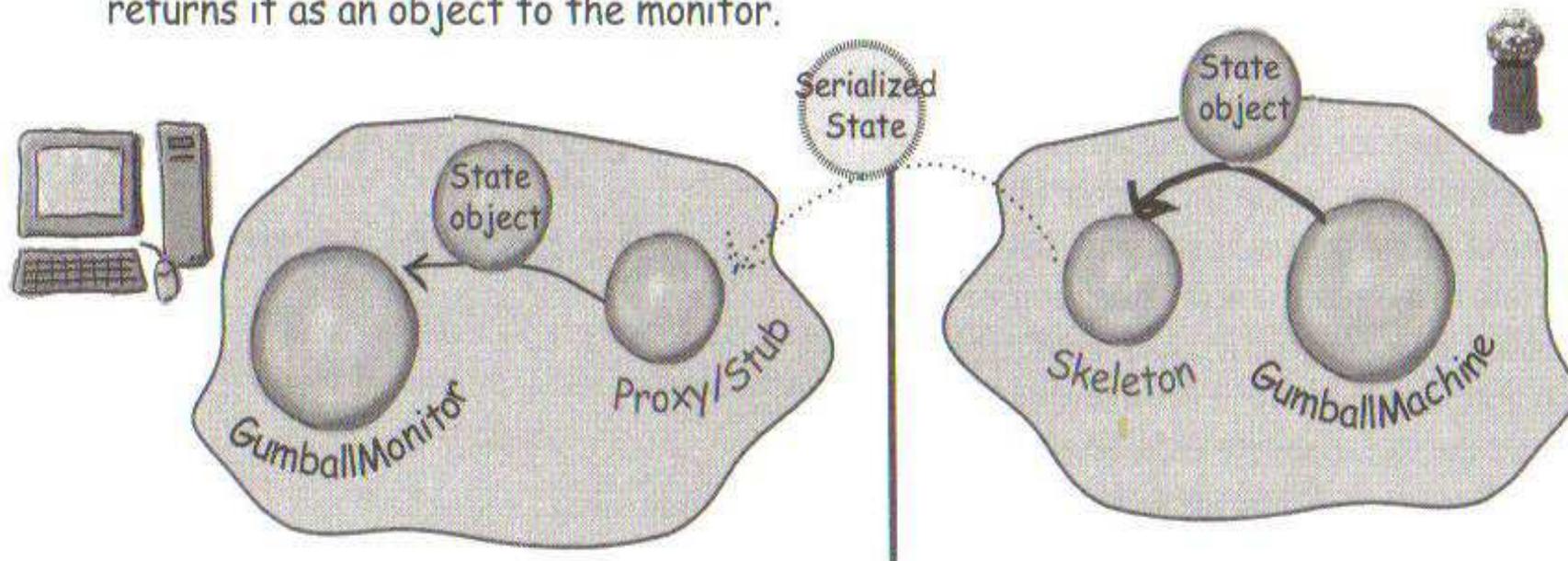


Making the call

- 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 gunball 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 monitor hasn't changed at all, except it knows it may encounter remote exceptions. It also uses the `GumballMachineRemote` interface rather than a concrete implementation.

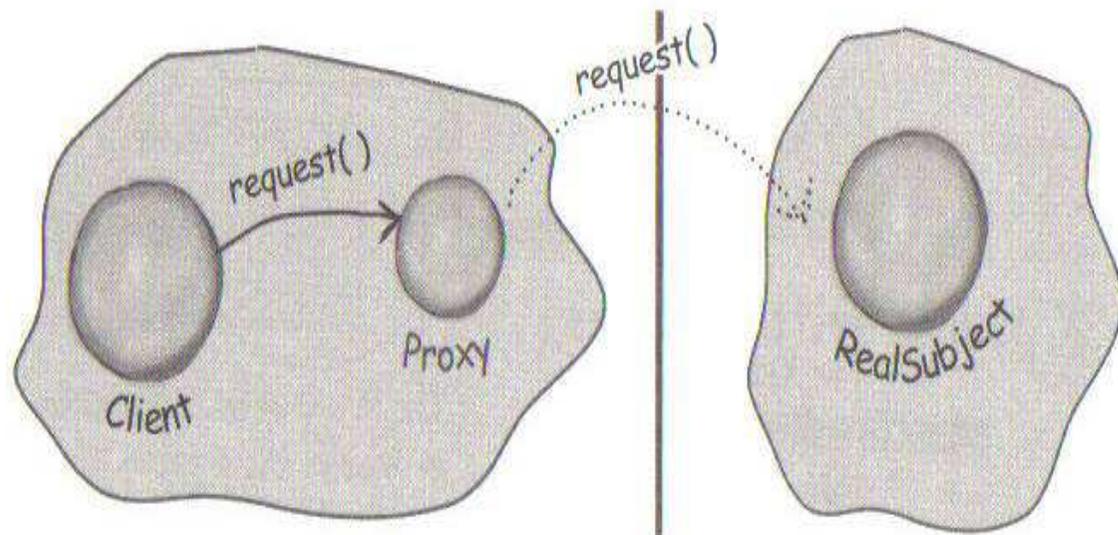
Likewise, the `GumballMachine` implements another interface and may throw a remote exception in its constructor, but other than that, the code hasn't changed.

We also have a small bit of code to register and locate stubs using the RMI registry. But no matter what, if we were writing something to work over the Internet, we'd need some kind of locator service.

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.



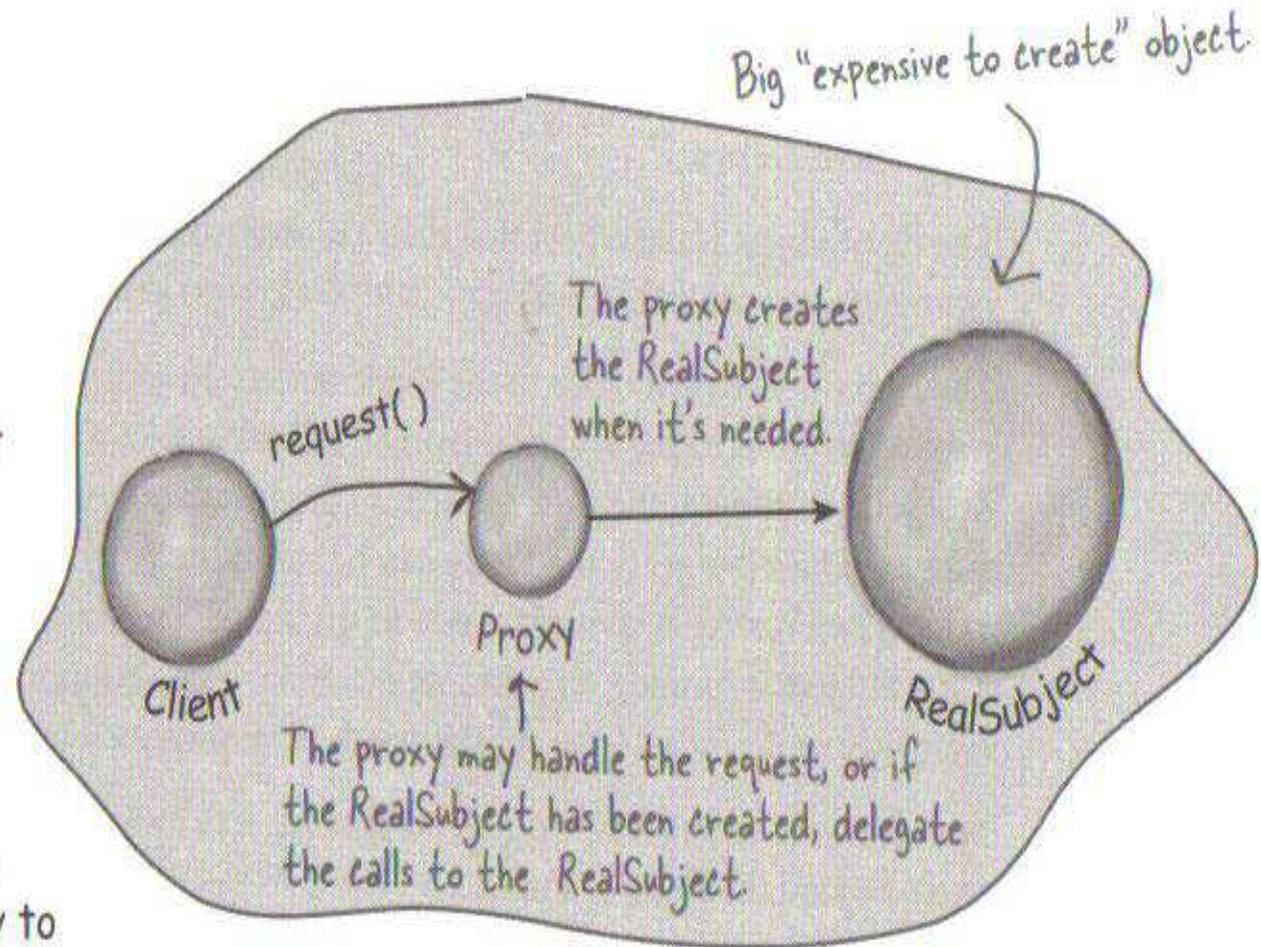
We know this diagram pretty well by now...



Virtual Proxy

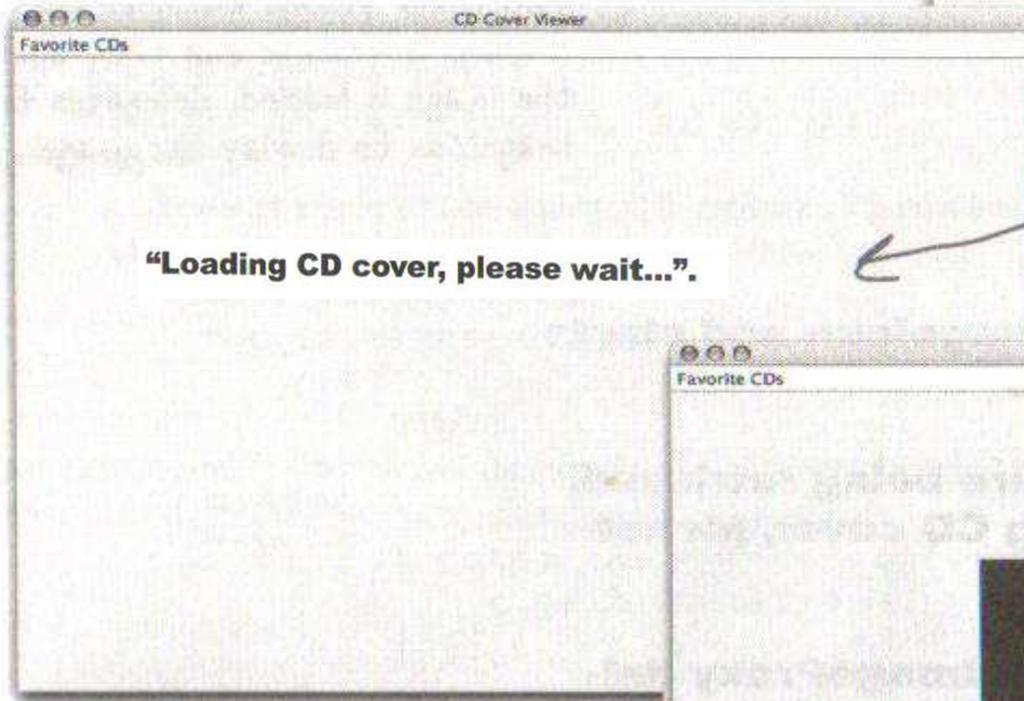
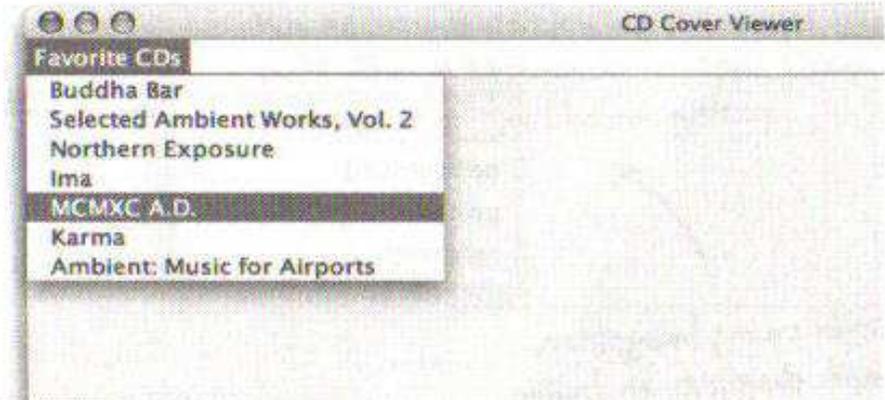
Virtual Proxy

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.



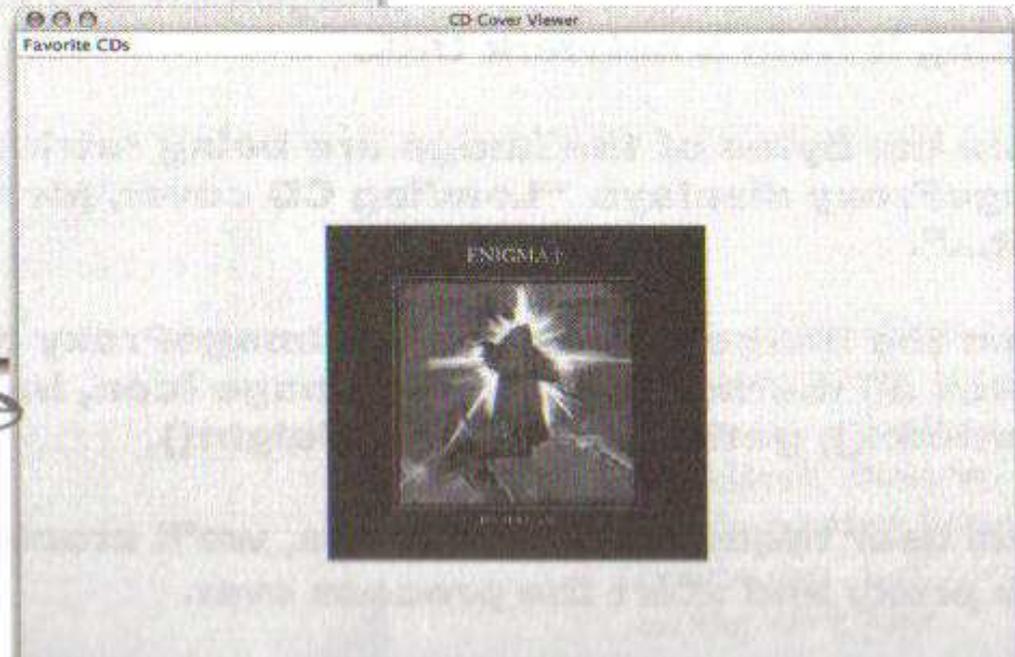
Choose the album cover of your liking here.

Playing CD Covers



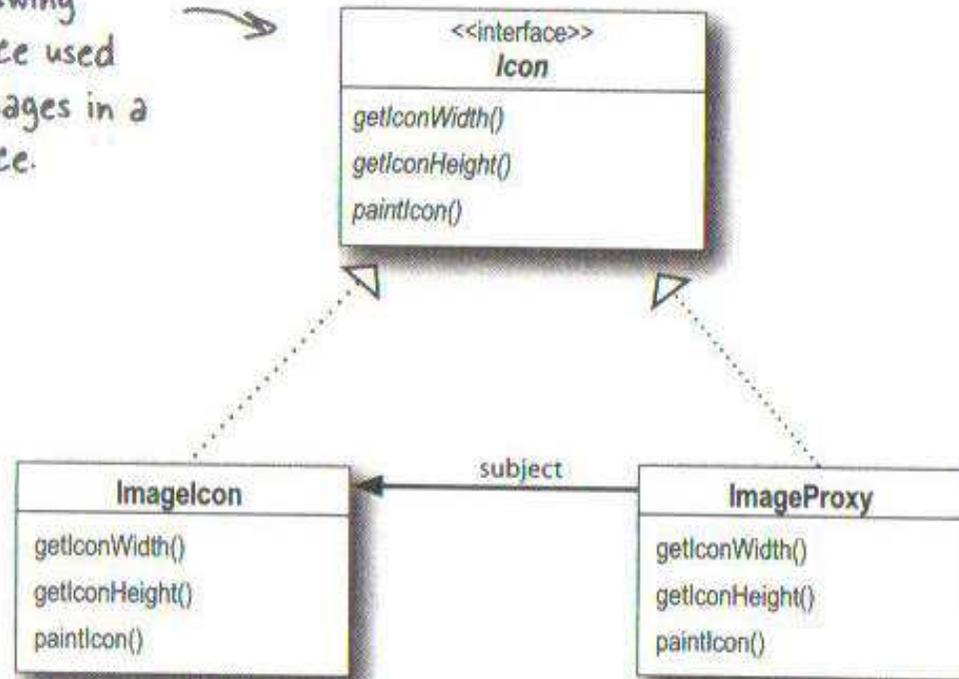
While the CD cover is loading, the proxy displays a message.

When the CD cover is fully loaded, the proxy displays the image.



Playing CD Cover Proxy

This is the Swing Icon interface used to display images in a user interface.



This is `javax.swing.ImageIcon`, a class that displays an image.

This is our proxy, which first displays a message and then when the image is loaded, delegates to `ImageIcon` to display the image.

ImageProxy process

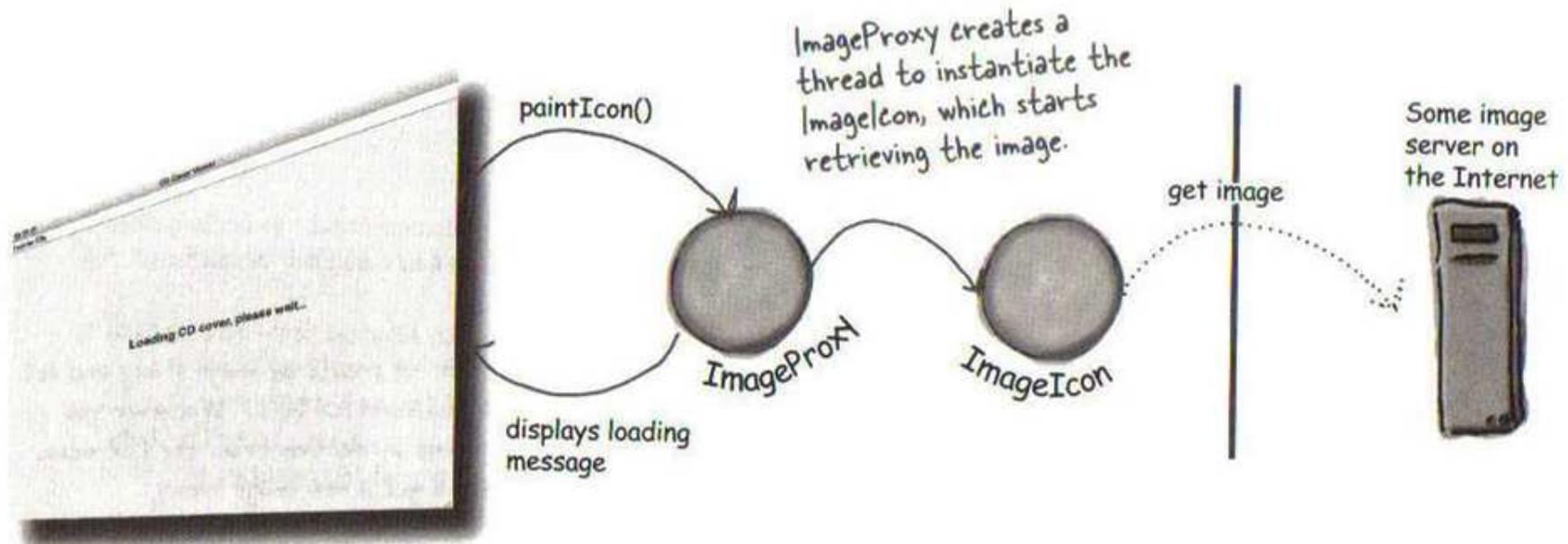
- 1 ImageProxy first creates an ImageIcon and starts loading it from a network URL.**
- 2 While the bytes of the image are being retrieved, ImageProxy displays “Loading CD cover, please wait...”.**
- 3 When the image is fully loaded, ImageProxy delegates all method calls to the image icon, including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

ImageProxy process

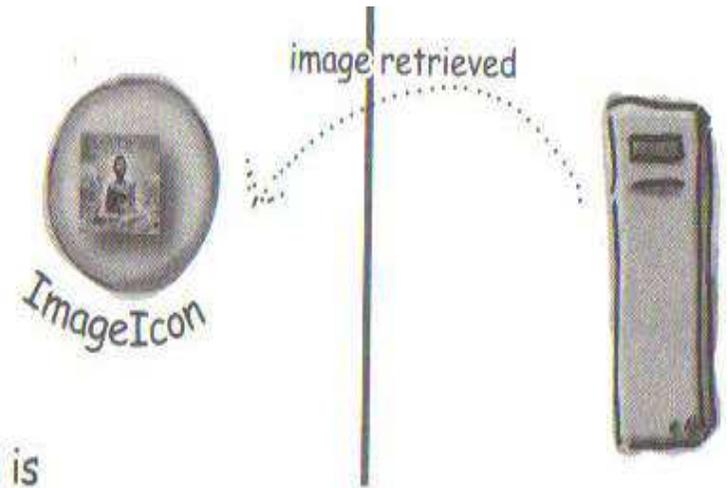
What did we do?

- 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.

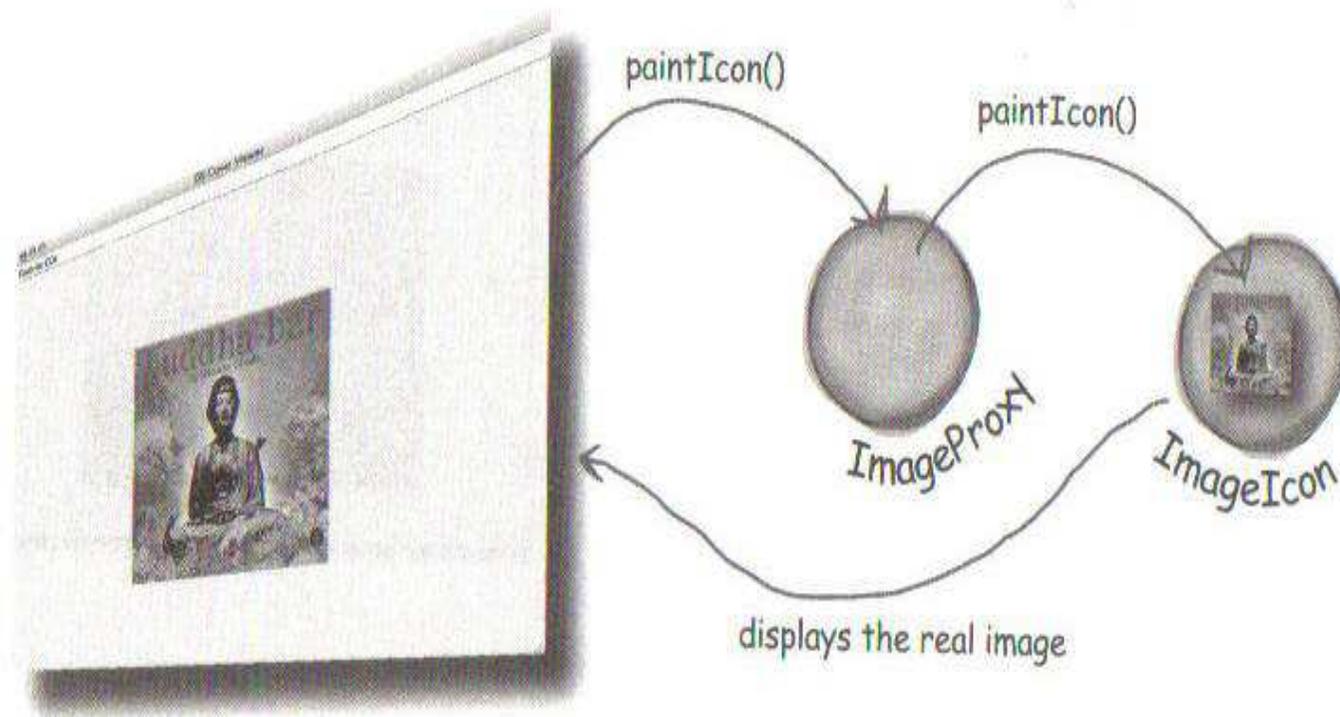
Behind the Scenes



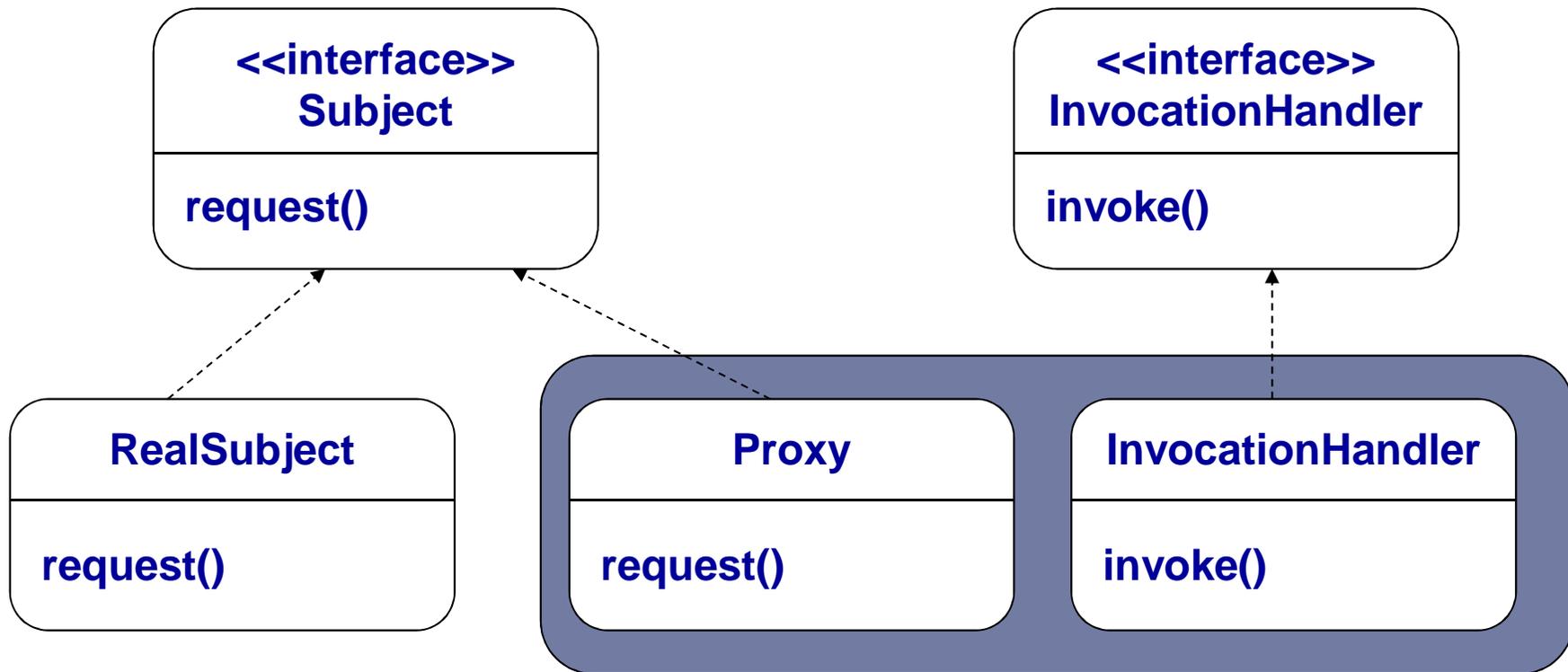
2 At some point the image is returned and the ImageIcon fully instantiated.



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



Using Java API's Proxy to create a protection proxy



The proxy zoo (1 / 2)

- ▶ **Firewall proxy**
 - ▶ Protects targets from bad clients (or viceversa)
- ▶ **Smart Reference proxy**
 - ▶ E.g. counts the number of references to the target object
- ▶ **Caching proxy**
 - ▶ Provides temporary storage of the result of expensive target operations so that multiple clients can share the results.
- ▶ **Synchronization Proxy**
 - ▶ Provides multiple accesses to a target object.

The proxy zoo (2/2)

- ▶ **Complexity hiding Proxy**
 - ▶ Similar to façade pattern, it also controls accesses
- ▶ **Protection (Access) Proxy**
 - ▶ Provides different clients with different levels of access to a target object
- ▶ **Copy-on-write Proxy**
 - ▶ Form of virtual proxy
 - ▶ Defers coping (cloning) a target object until required by a client action.



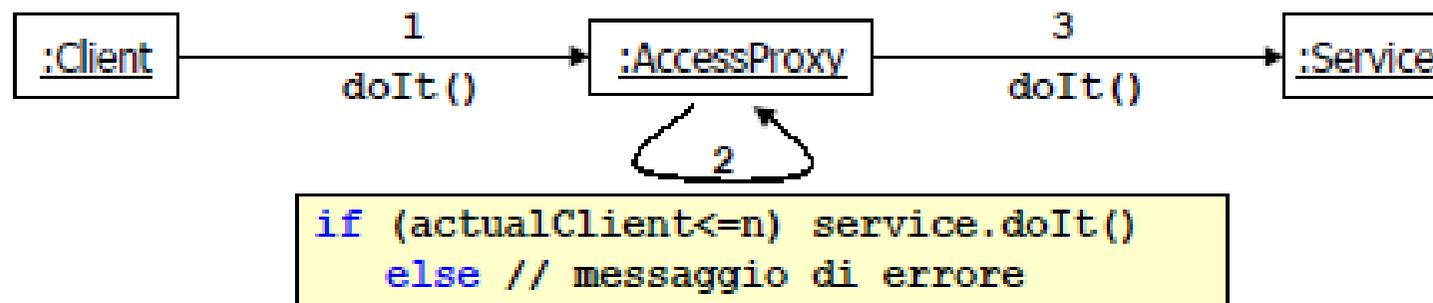
Il pattern Access Proxy

100%

1. Access Proxy (non documentato su Patterns in Java)

Il pattern Access Proxy viene utilizzato per far rispettare una politica di sicurezza nell'accesso ad oggetti che erogano determinati servizi.

Esempio: Il servizio può essere erogato ad un massimo di n client contemporaneamente





Il pattern Broker/Proxy

2. **Broker** [non documentato su Grand98]

Il pattern Proxy è a volte usato con il pattern Broker per fornire un sistema trasparente finalizzato al ridirezionamento di una richiesta di servizio verso un service object selezionato dall'oggetto Broker/Proxy.

Esempio: Il Broker/Proxy ridireziona i client con una politica Round Robin sulle istanze Service multithread disponibili

