Tecniche di Progettazione: Design Patterns

GoF: Proxy

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

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

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

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

    public String getLocation() {
        return location;
    }

    // other methods here
}
```

A location is just a String.

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

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

Gumball Monitor

```java
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());
    }
}
```
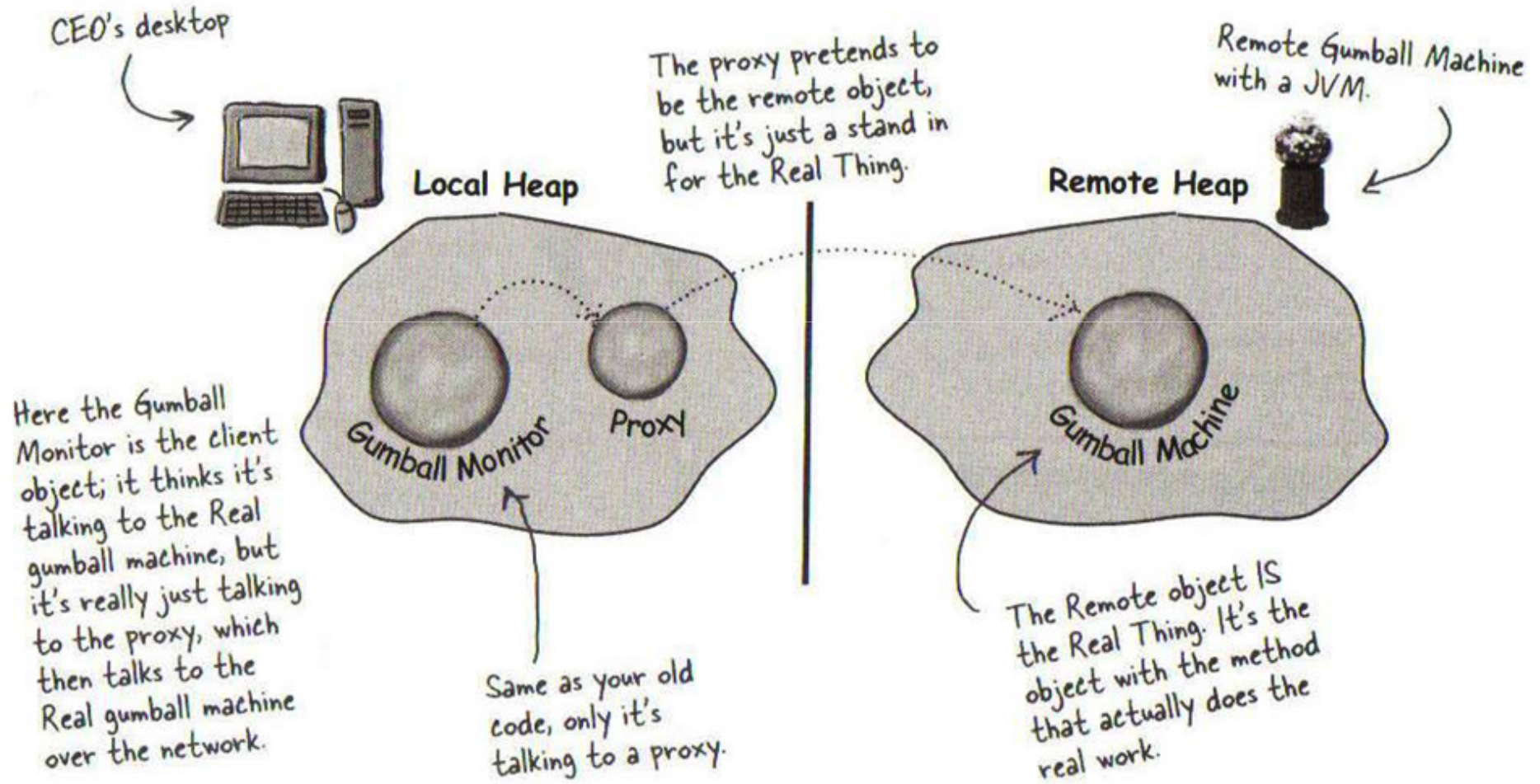
The monitor takes the machine in its constructor and assigns it to the machine instance variable.

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

Role of the remote Proxy

CEO's desktop

Local Heap

The proxy pretends to be the remote object, but it's just a stand in for the Real Thing.

Remote Gumball Machine with a JVM.

Remote Heap

Gumball Monitor

Proxy

Gumball Machine

Here the Gumball Monitor is the client object; it thinks it's talking to the Real gumball machine, but it's really just talking to the proxy, which then talks to the Real gumball machine over the network.

Same as your old code, only it's talking to a proxy.

The Remote object IS the Real Thing. It's the object with the method that actually does the real work.

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.
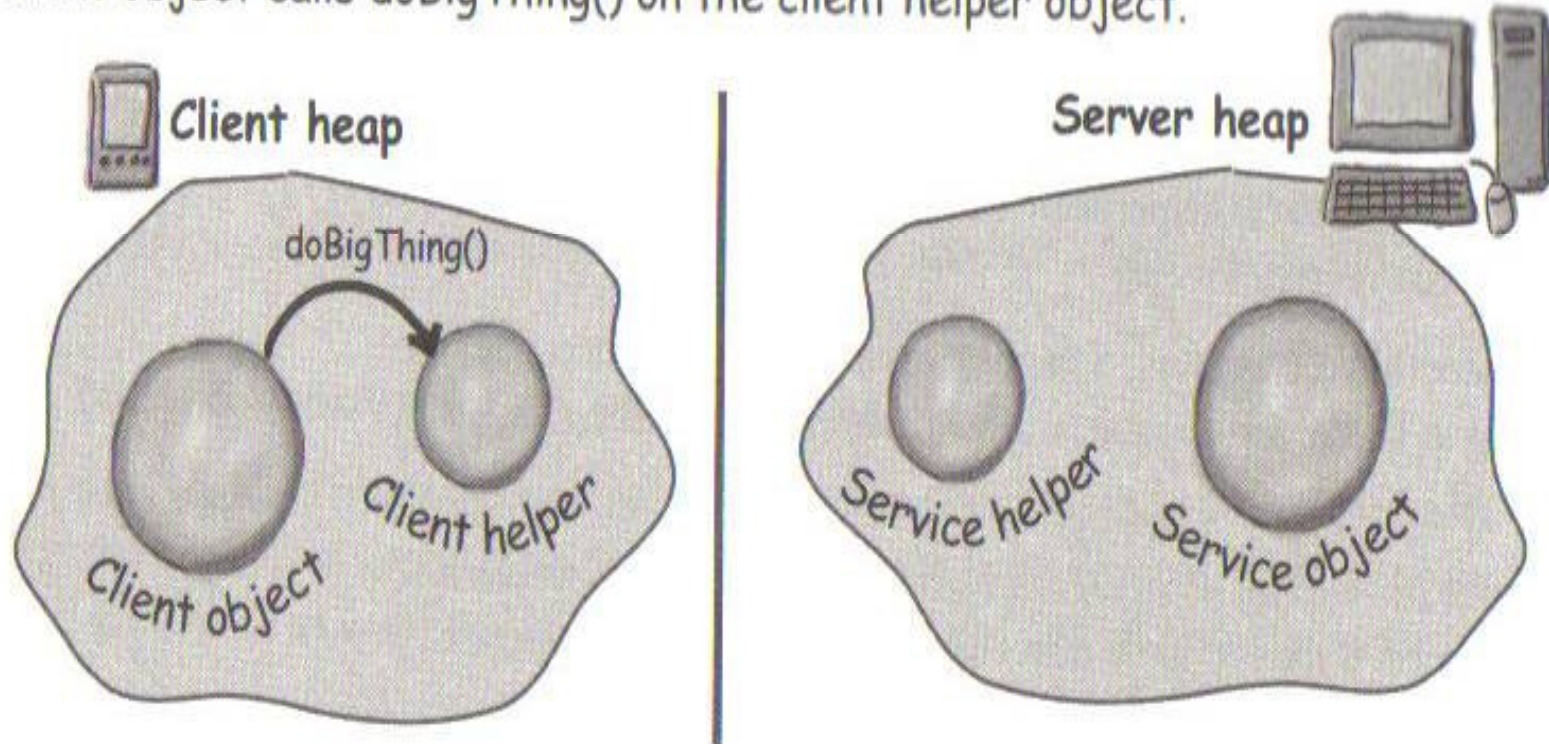
Remote Methods
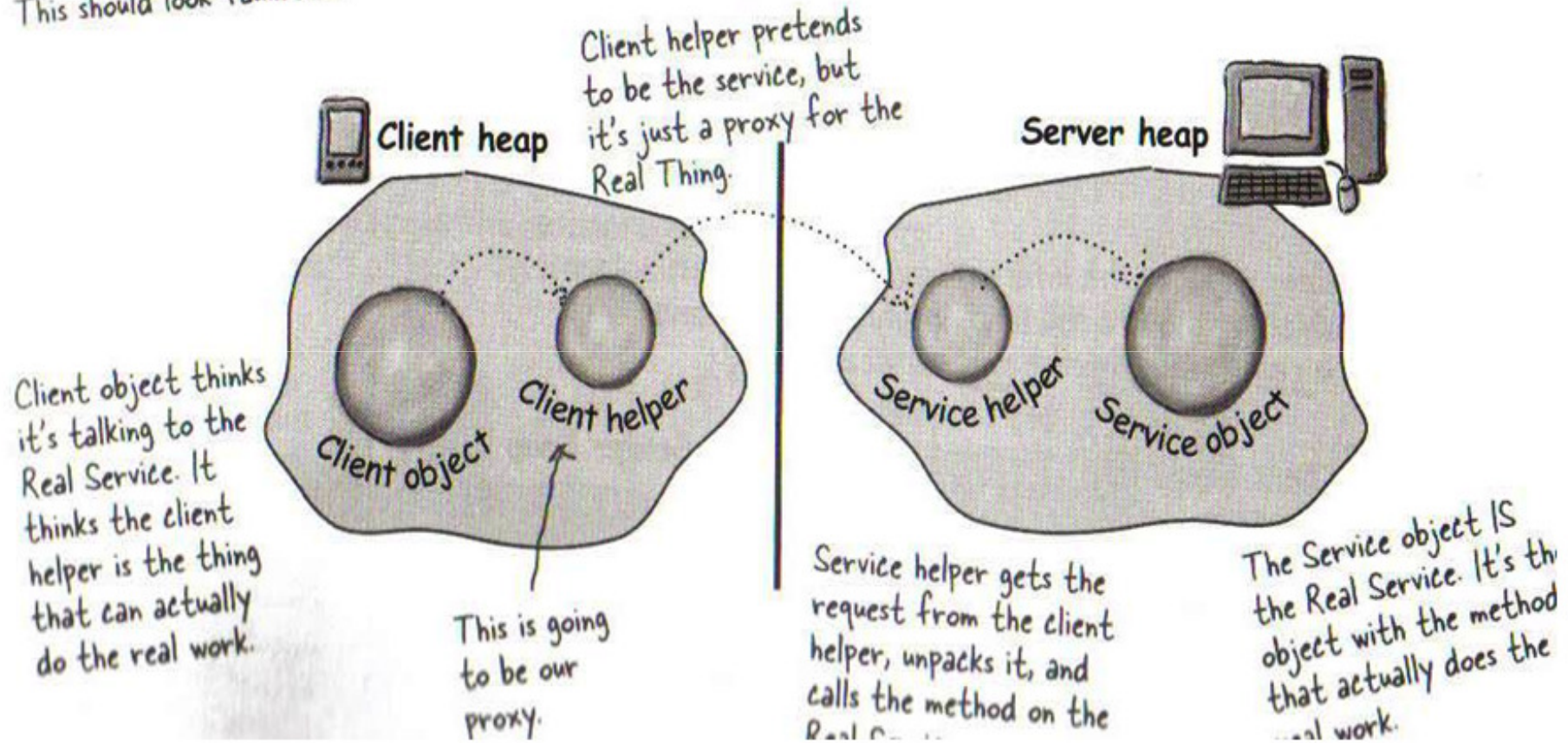
How the method call happens
Client calls method
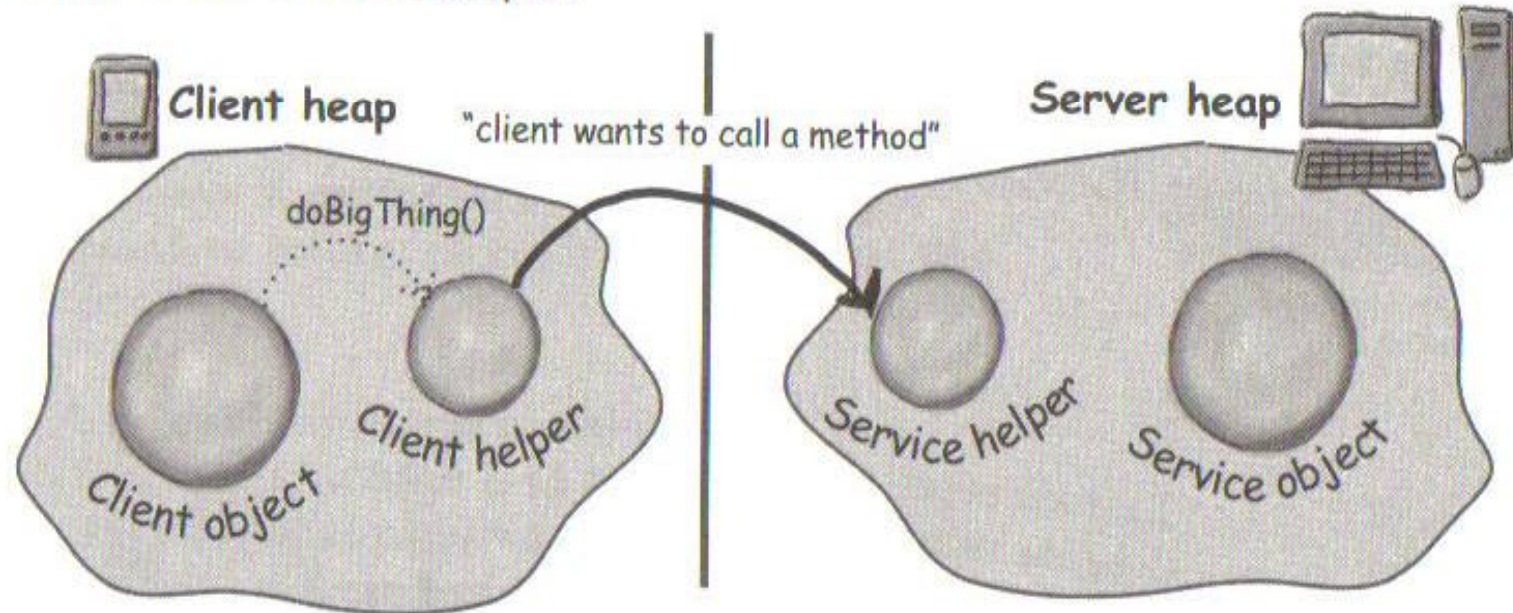

1. Client object calls doBigThing() on the client helper object.

Client heap

Server heap

doBigThing()

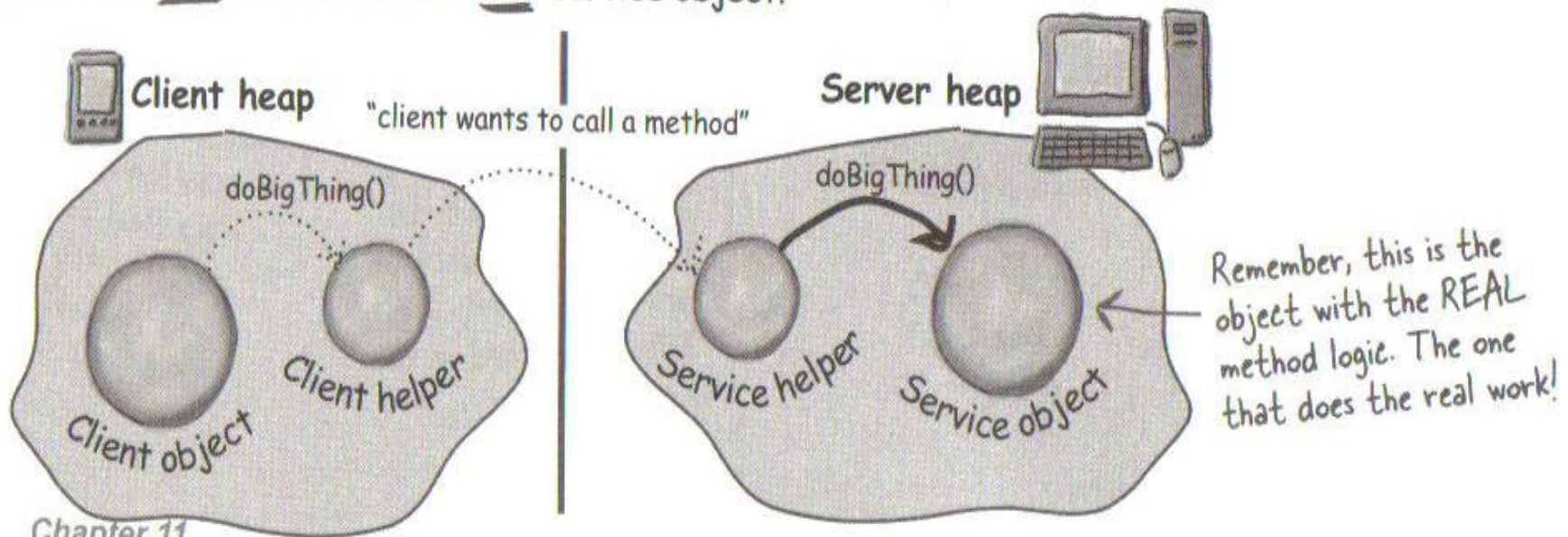Client object

Client helper

Service helper

Service 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.


Client heap   "client wants to call a method"   Server heap

doBigThing()

doBigThing()

Client object   Client helper

Service helper   Service object

Remember, this is the object with the REAL method logic. The one that does the real work!

Real object returns result

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

Client heap

Server heap

result

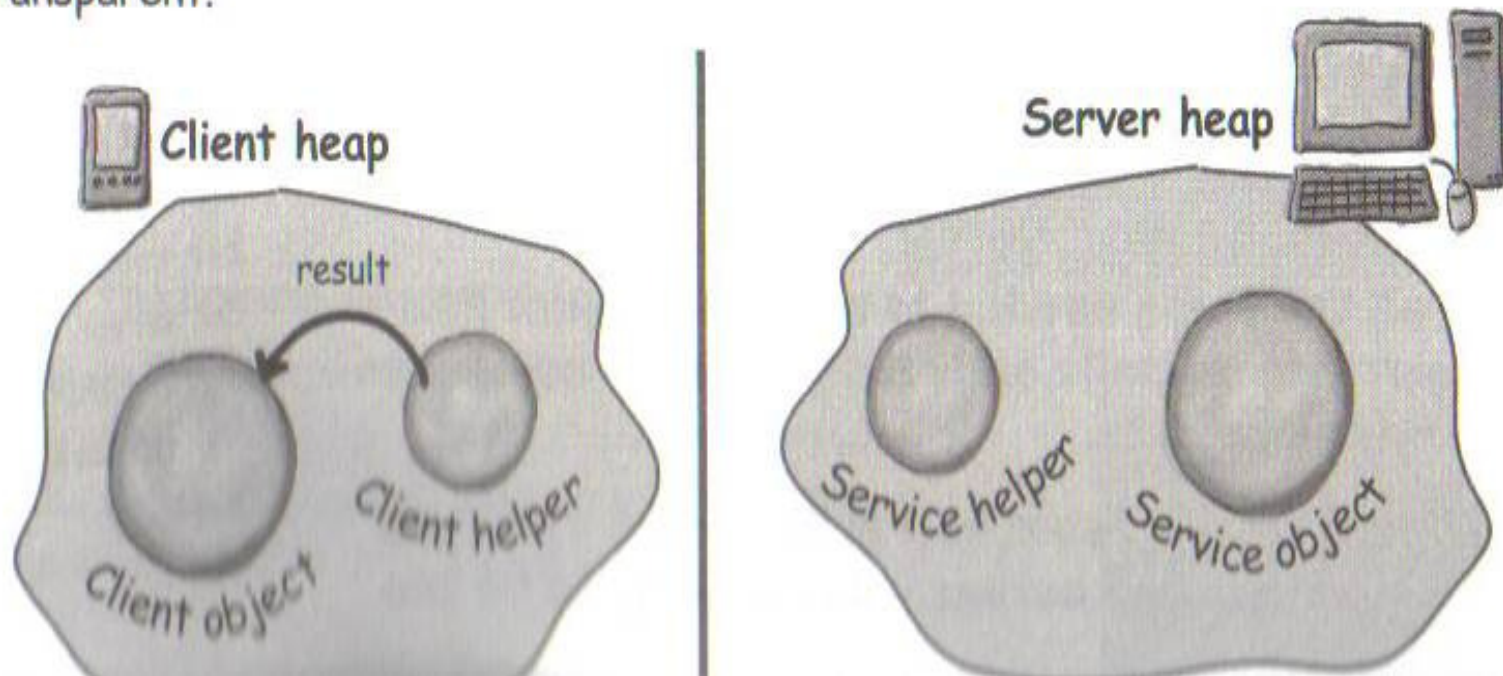Client object

Client helper

Service helper

Service object
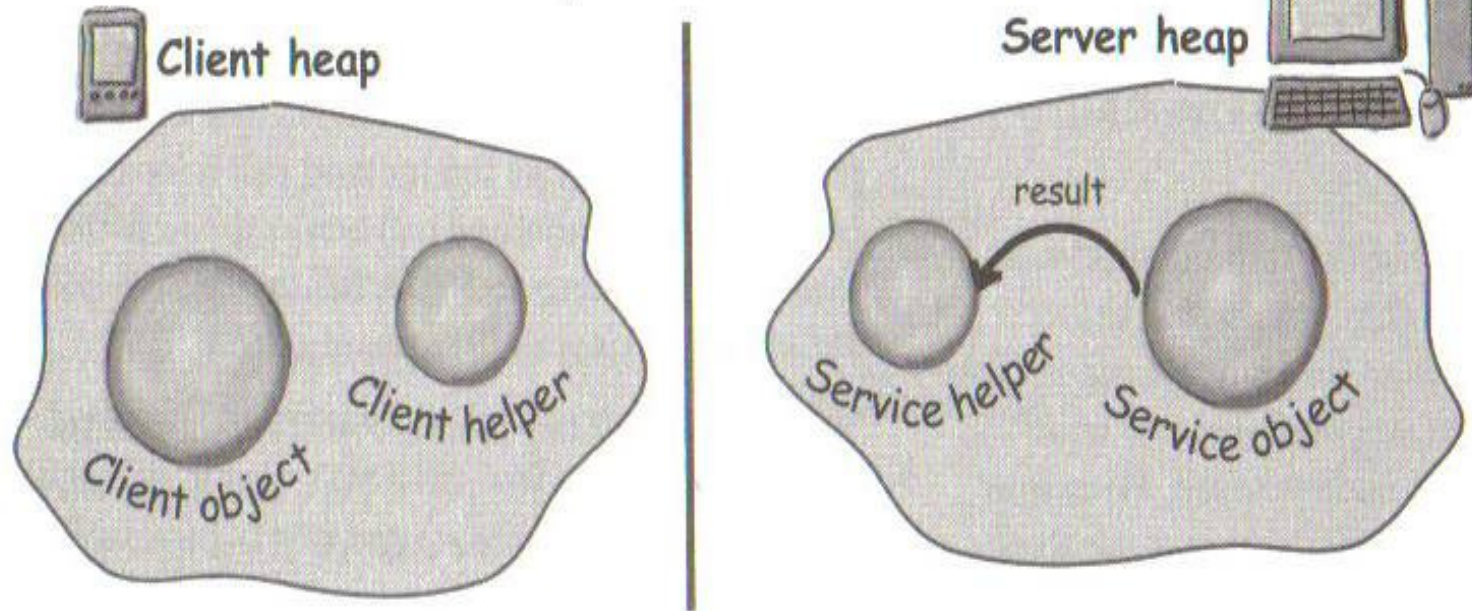
Service helper forwards result to client helper

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

Client heap

Server heap

packaged up result

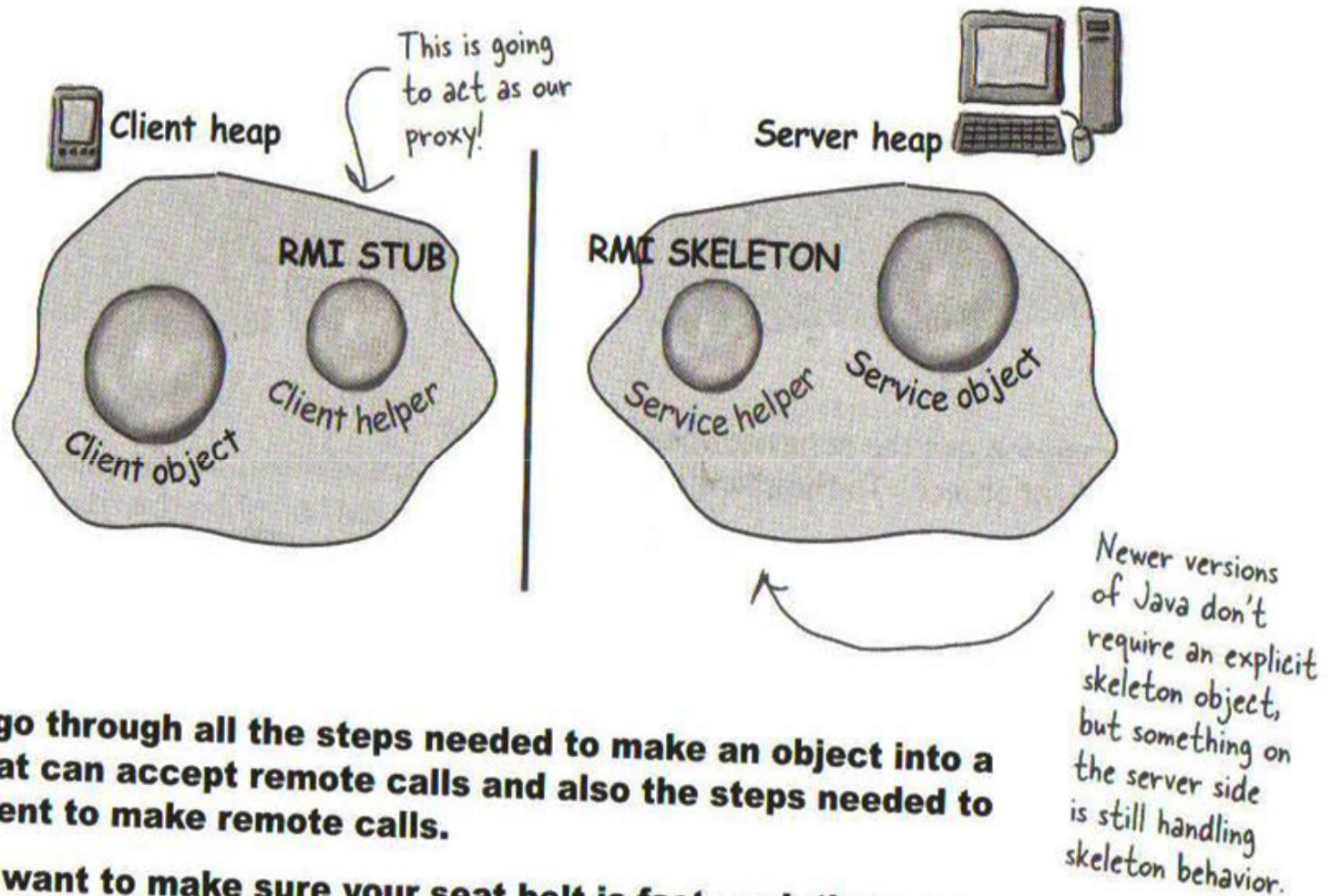Client helper

Client object

Service helper
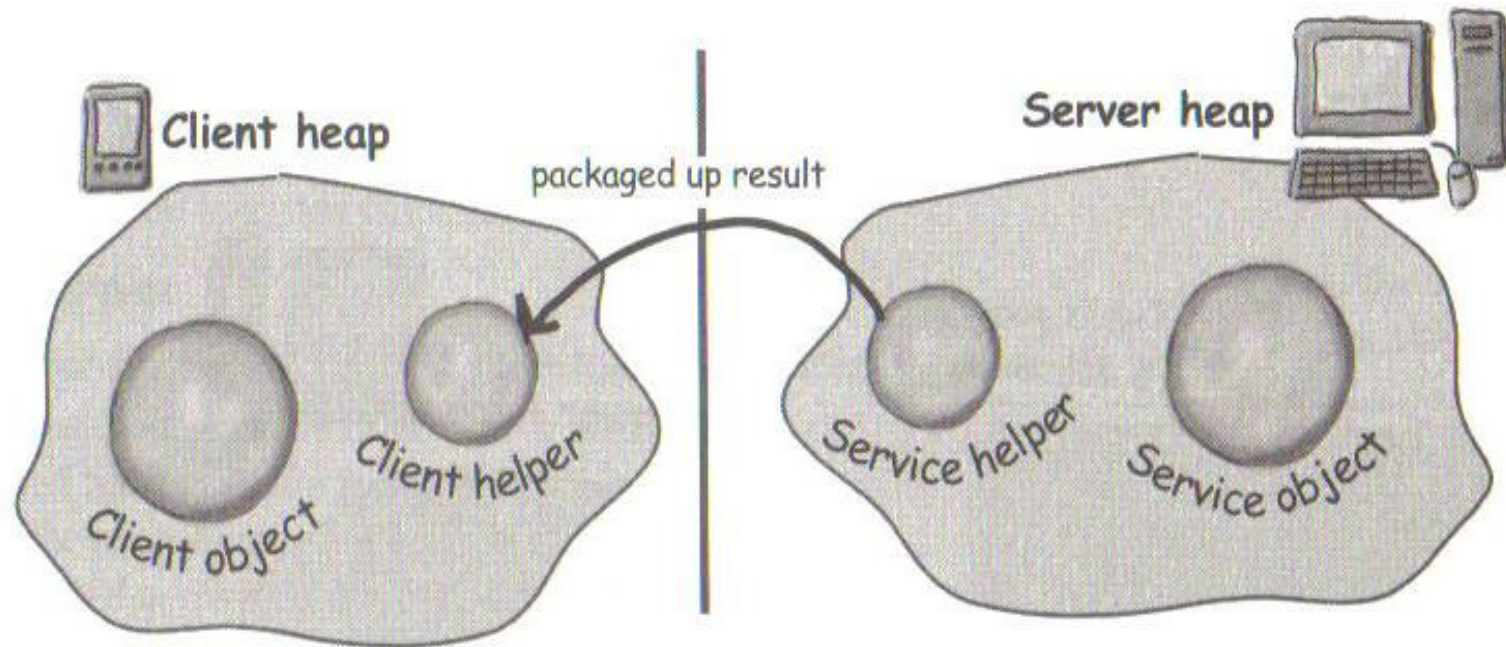
Service object

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'.

This is going to act as our proxy!

Client heap

Server heap

RMI STUB

RMI SKELETON

Client helper

Client object

Service helper

Service object

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.

Newer versions of Java don't require an explicit skeleton object, but something on the server side is still handling skeleton behavior.

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!
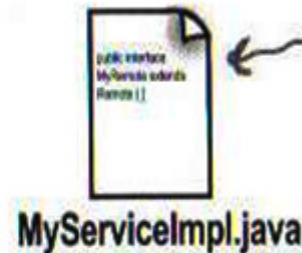
MyService.java

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!).

MyServiceImpl.java

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.

Running rmic against the actual service implementation class...

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

```
File Edit Window Help Eat

% rmic MyServiceImpl
```

```
101101
10 110 1
0 11 0
001 10
001 01
```

MyServiceImpl_Stub.class

```
101101
10 110 1
0 11 0
001 10
001 01
```

MyServiceImpl_Skel.class

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).

```
File Edit Window Help Drink

% rmiregistry
```

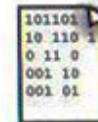Run this in a separate terminal.

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.
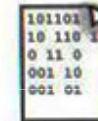
```
File Edit Window Help BeMerry

% java MyServiceImpl
```

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.

```
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.

① Implement the Remote interface

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

```java
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.

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

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.

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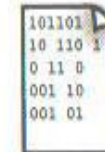
Notice that you don't say ".class" on the end. Just the class name.

RMIC generates two new classes for the helper objects.

File Edit Window Help Whuffie
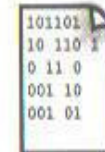
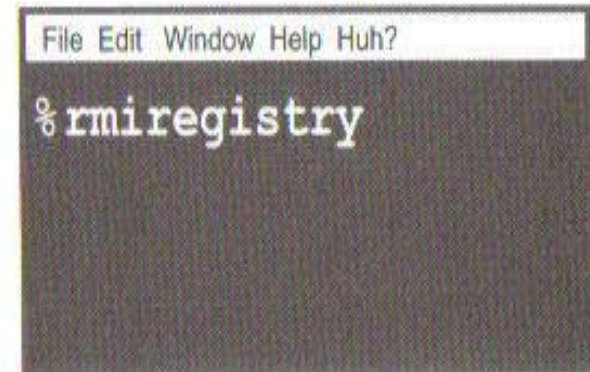% rmic MyRemoteImpl

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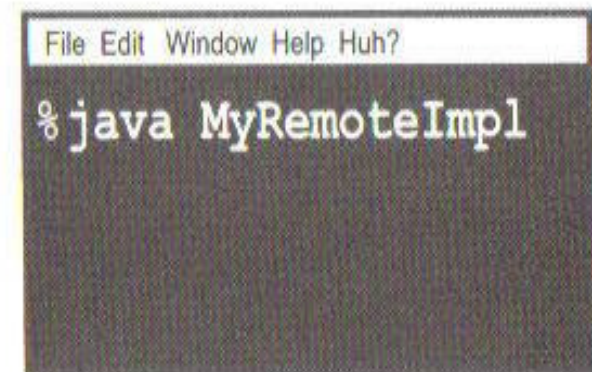
```
File  Edit  Window  Help  Huh?
%rmiregistry
```

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):

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

public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {

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

   public MyRemoteImpl() throws RemoteException { }

   public static void main (String[] args) {

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

UnicastRemoteObject is in the java.rmi.server package.

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

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!!

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

Make the remote object, then 'bind' it to the rmiregistery 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.

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

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

```
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.

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

Hooking up client and server objects
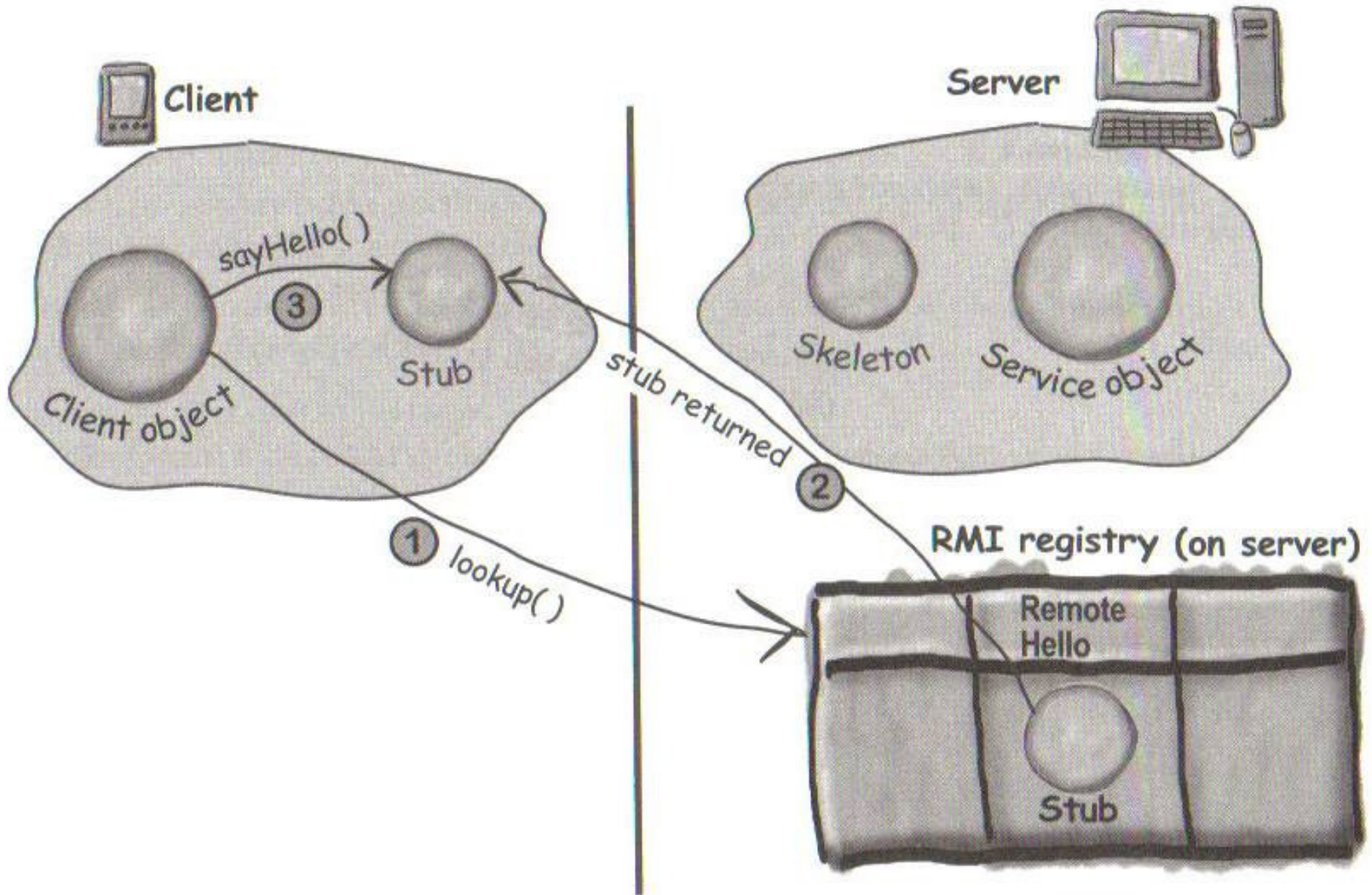
How it works...

(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();
        }
    }
}
```

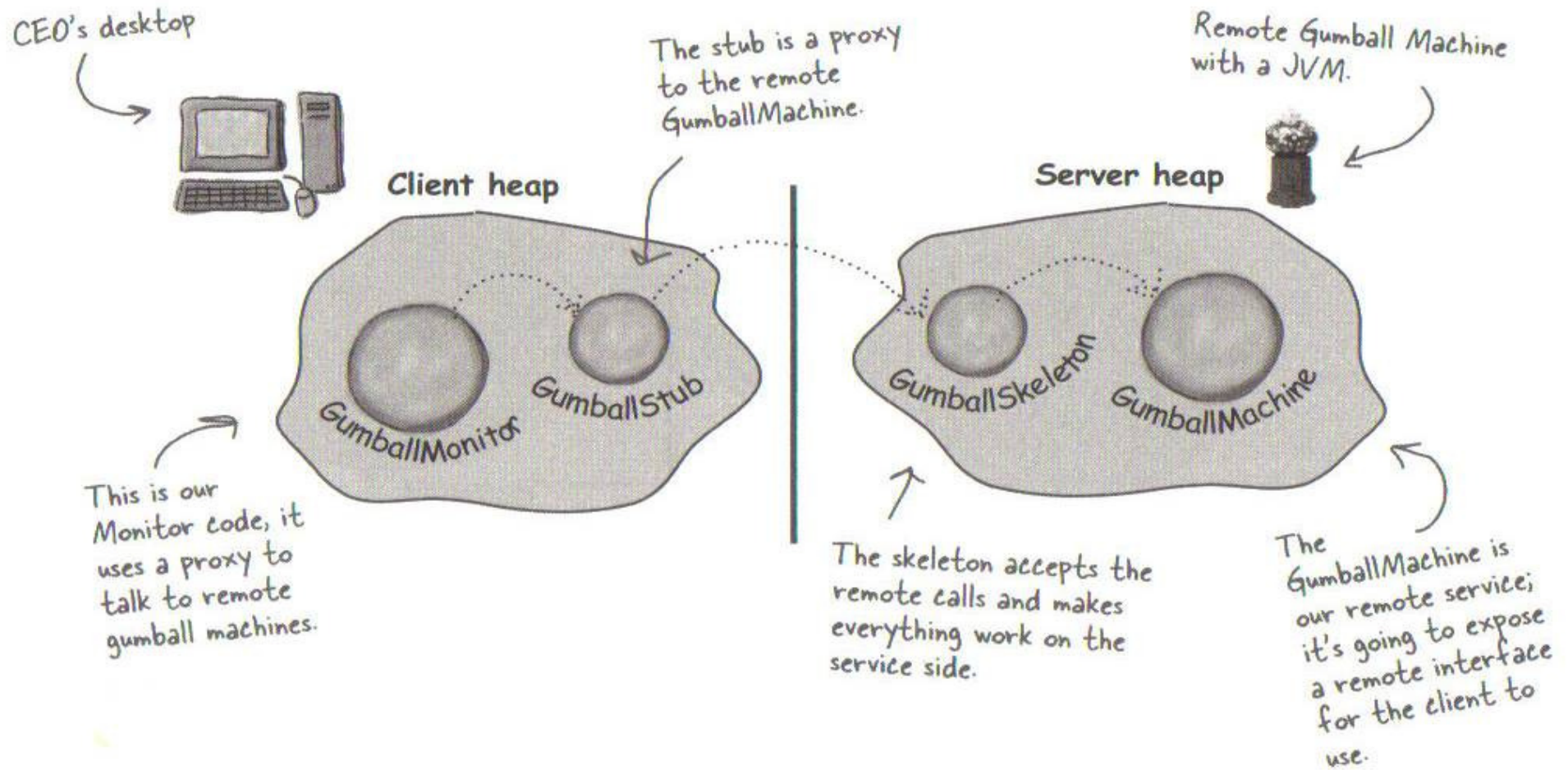*The Naming class (for doing the rmiregistry lookup) is in the java.rmi package.*

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

CEO's desktop

The stub is a proxy to the remote GumballMachine.

Remote Gumball Machine with a JVM.

Client heap

Server heap

GumballMonitor

GumballStub

GumballSkeleton

GumballMachine

This is our Monitor code, it uses a proxy to talk to remote gumball machines.

The skeleton accepts the remote calls and makes everything work on the service side.

The GumballMachine is our remote service; it's going to expose a remote interface for the client to use.

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"

```java
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.
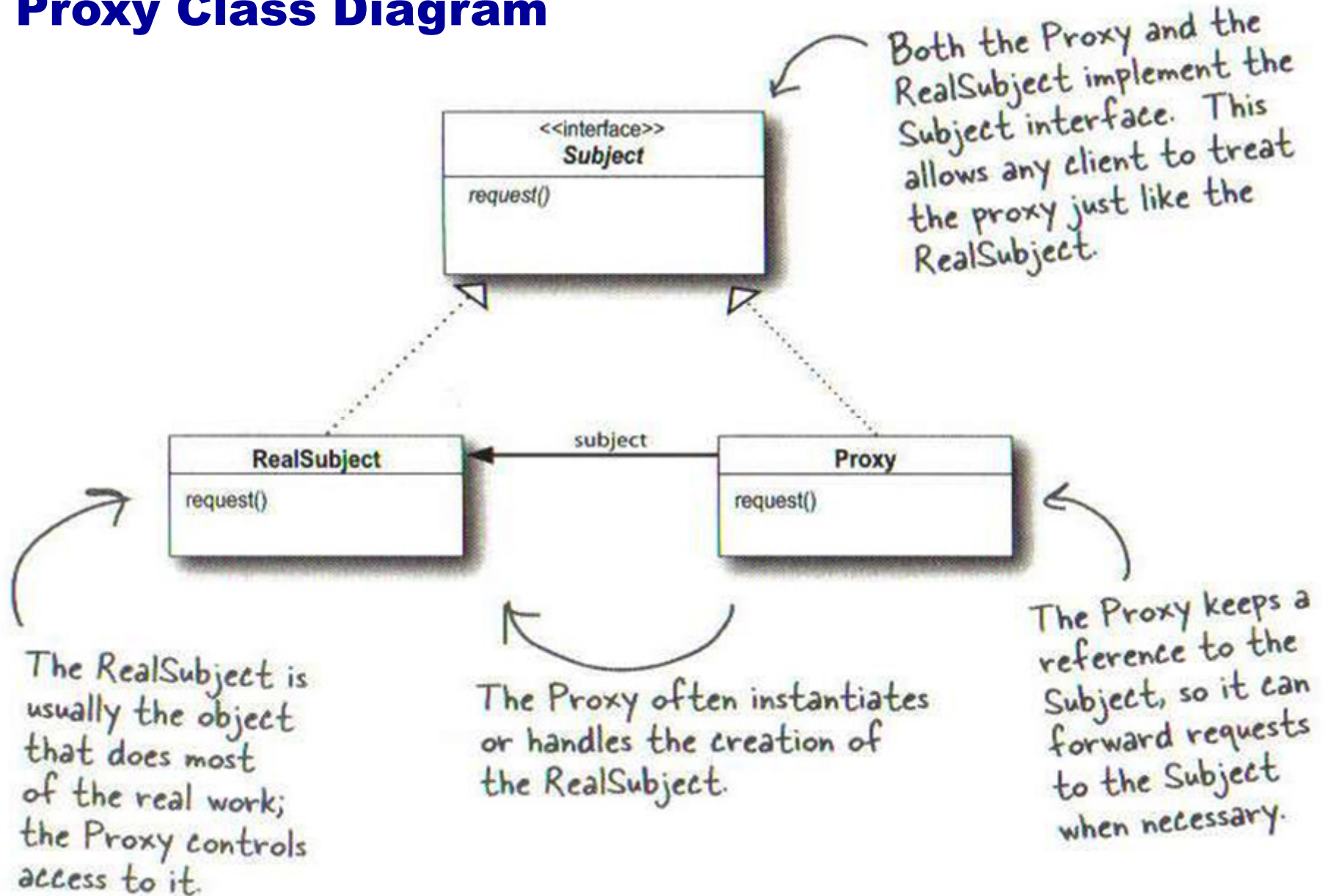
Proxy Pattern defined

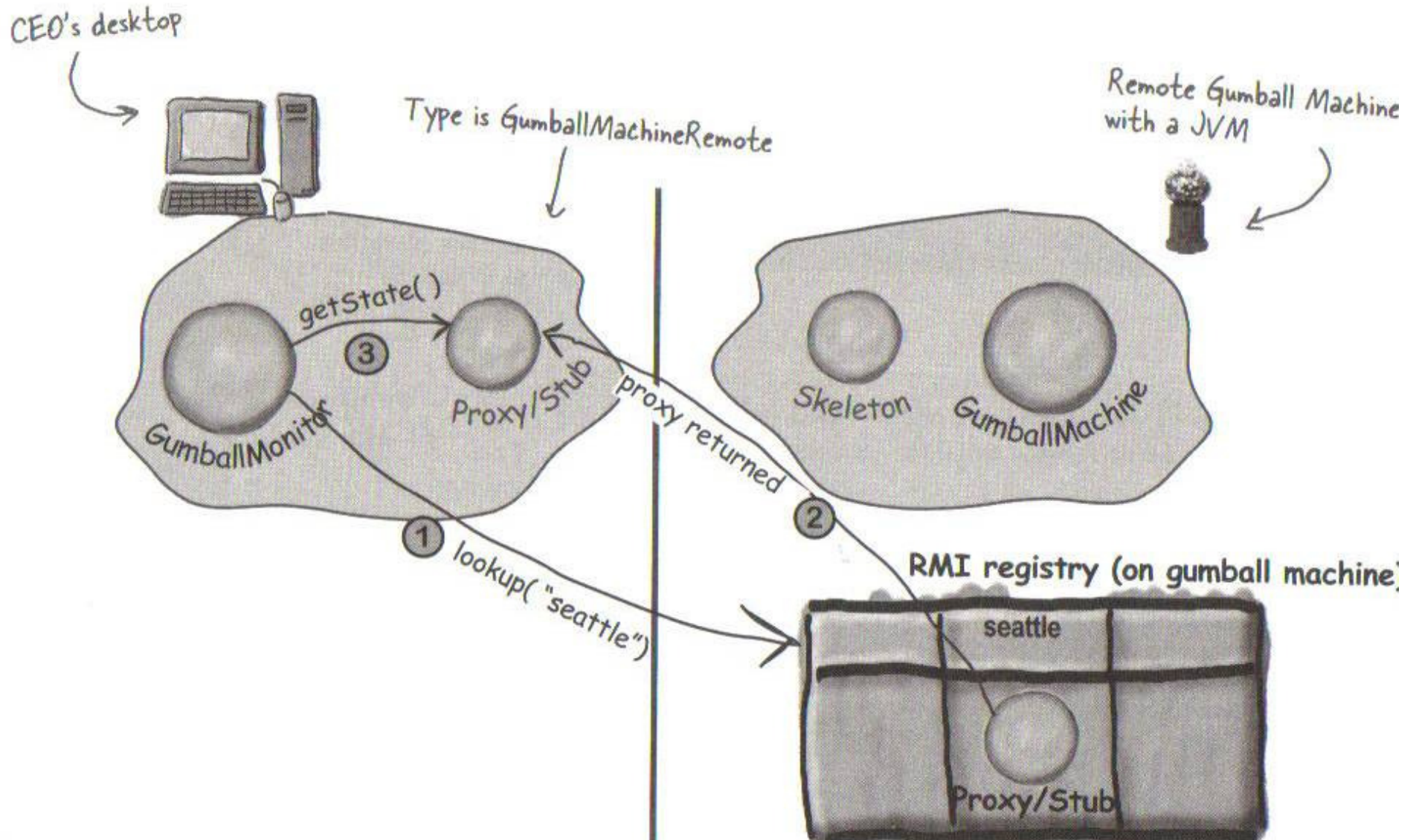**The Proxy Pattern provides a surrogate or placeholder for another object to control access to it.**

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

Both the Proxy and the RealSubject implement the Subject interface. This allows any client to treat the proxy just like the RealSubject.

```
          <<interface>>
            Subject
         ─────────────
          request()
```

```
     RealSubject              subject            Proxy
    ─────────────      ◄──────────────      ─────────────
     request()                                request()
```

The RealSubject is usually the object that does most of the real work; the Proxy controls access to it.

The Proxy often instantiates or handles the creation of the RealSubject.

The Proxy keeps a reference to the Subject, so it can forward requests to the Subject when necessary.

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()).


CEO's desktop

Type is GumballMachineRemote

Remote Gumball Machine with a JVM

getState()

③

Proxy/Stub

proxy returned

GumballMonitor

Skeleton

GumballMachine

① lookup("seattle")

②

RMI registry (on gumball machine)

seattle

Proxy/Stub

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 gumball machine.
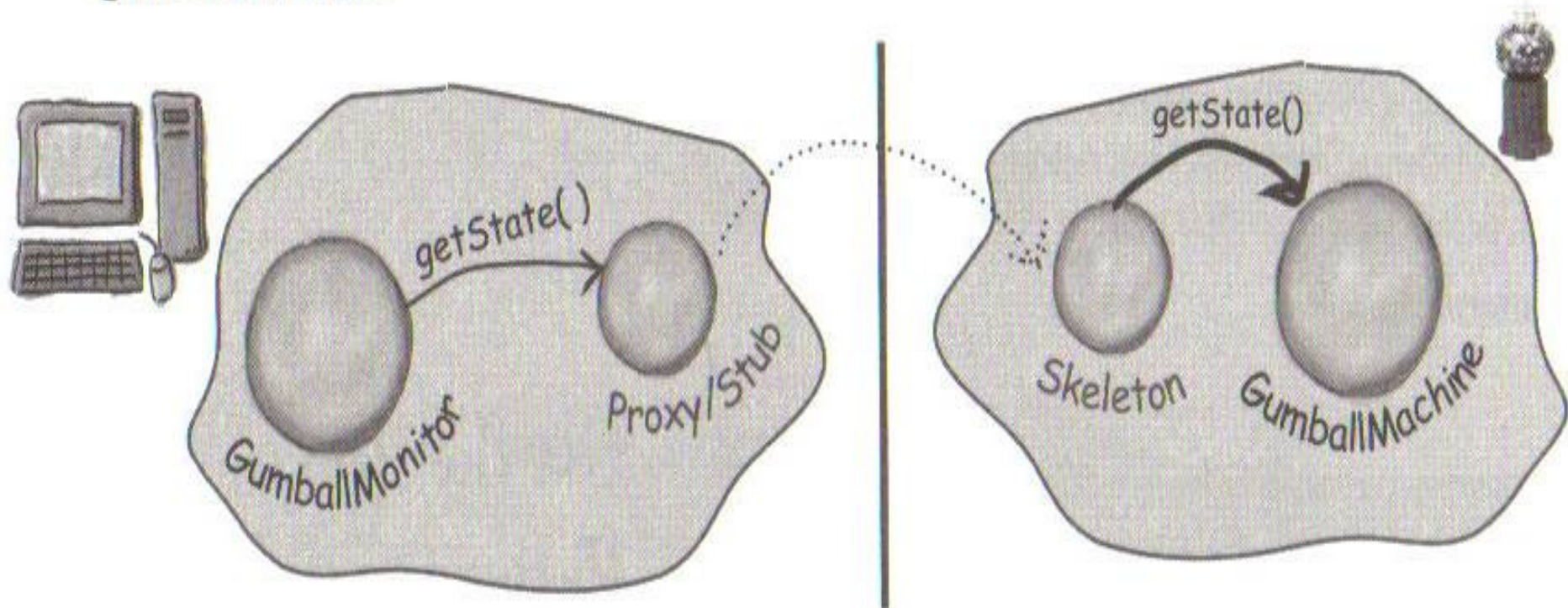
3 GumballMachine returns the state to the skeleton, which serializes it and transfers it back over the wire to the proxy. The proxy deserializes it and returns it as an object to the monitor.
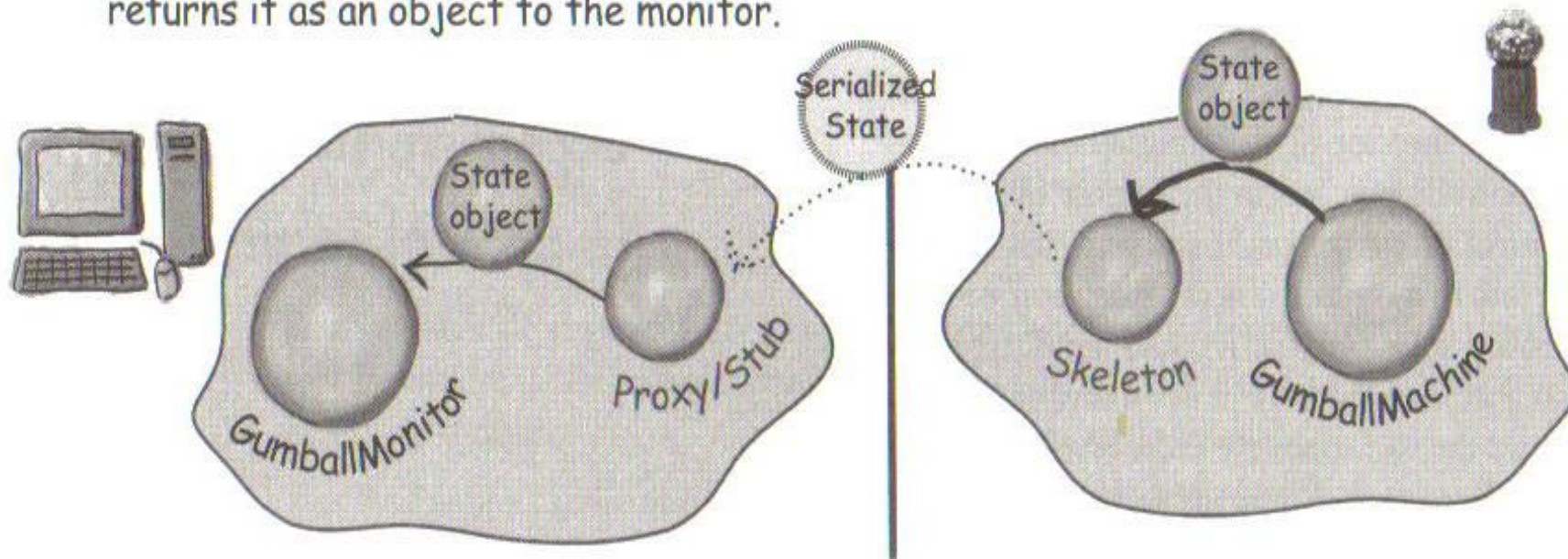
The monitor hasn't changed at all, except it knows it may encounter remote exceptions. It also uses the GumballMachineRemote interface rather than a concrete implementation.

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.
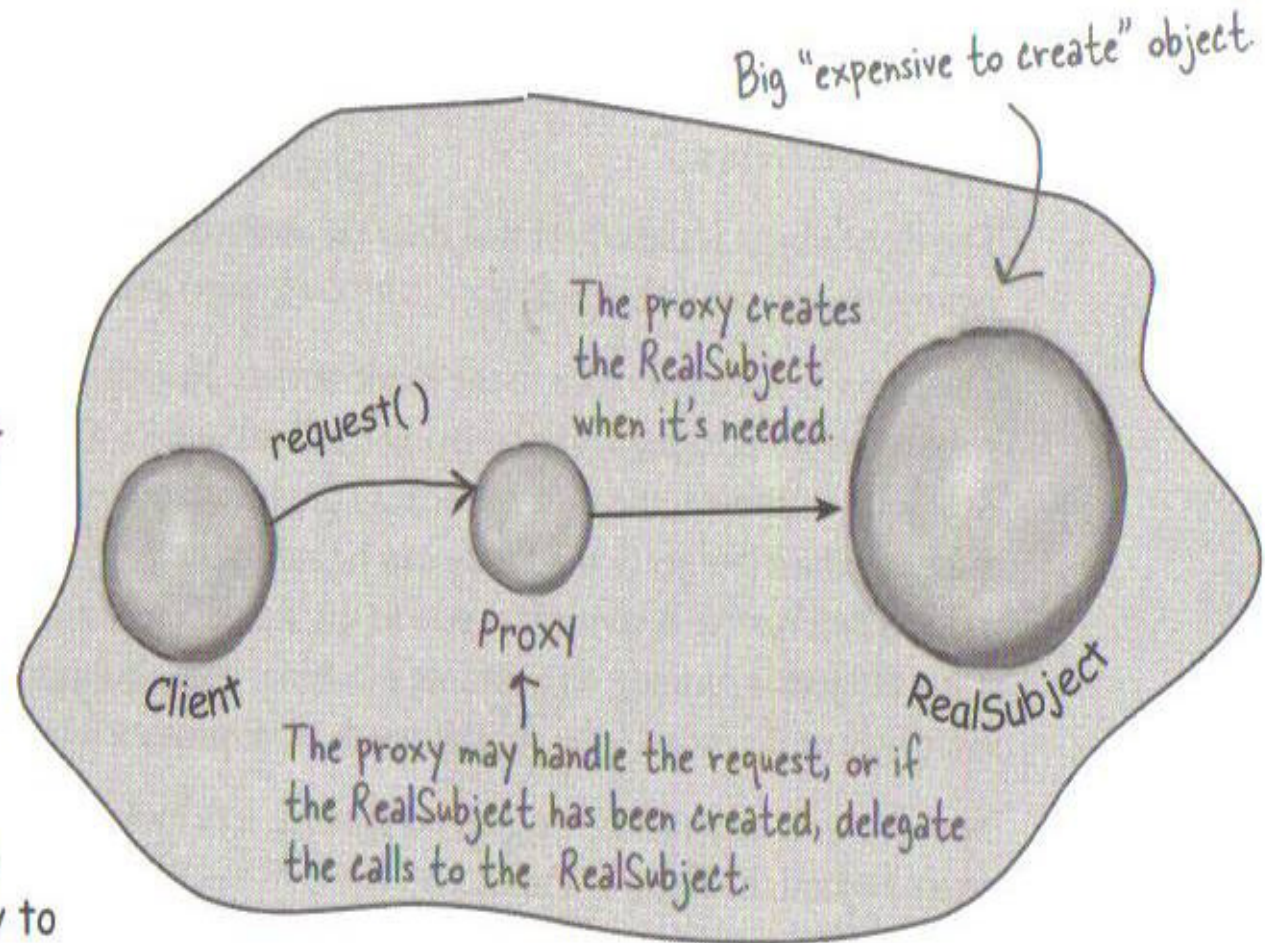

request()

request()

Client

Proxy

RealSubject
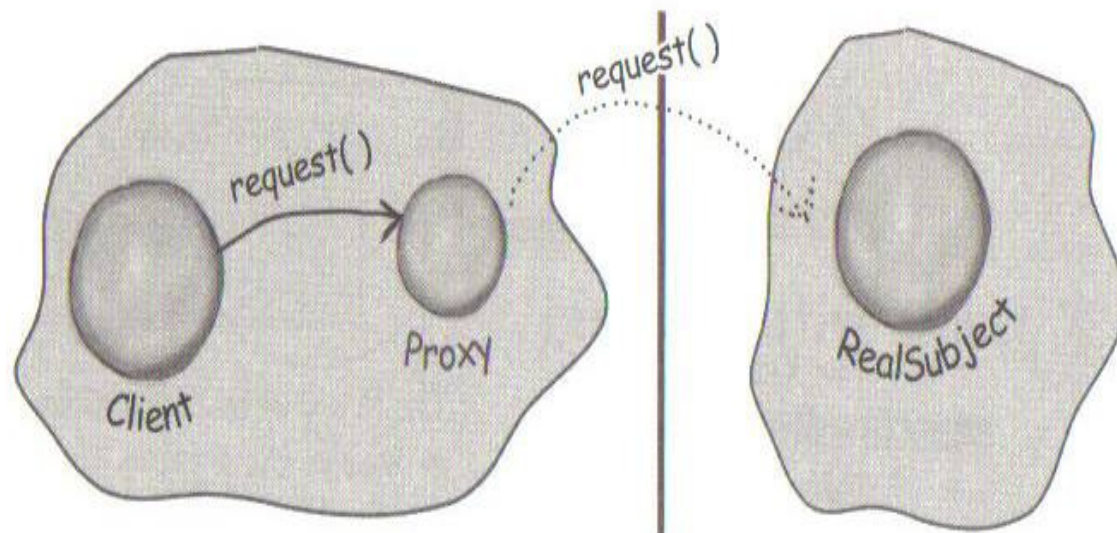
We know this diagram pretty well by now...
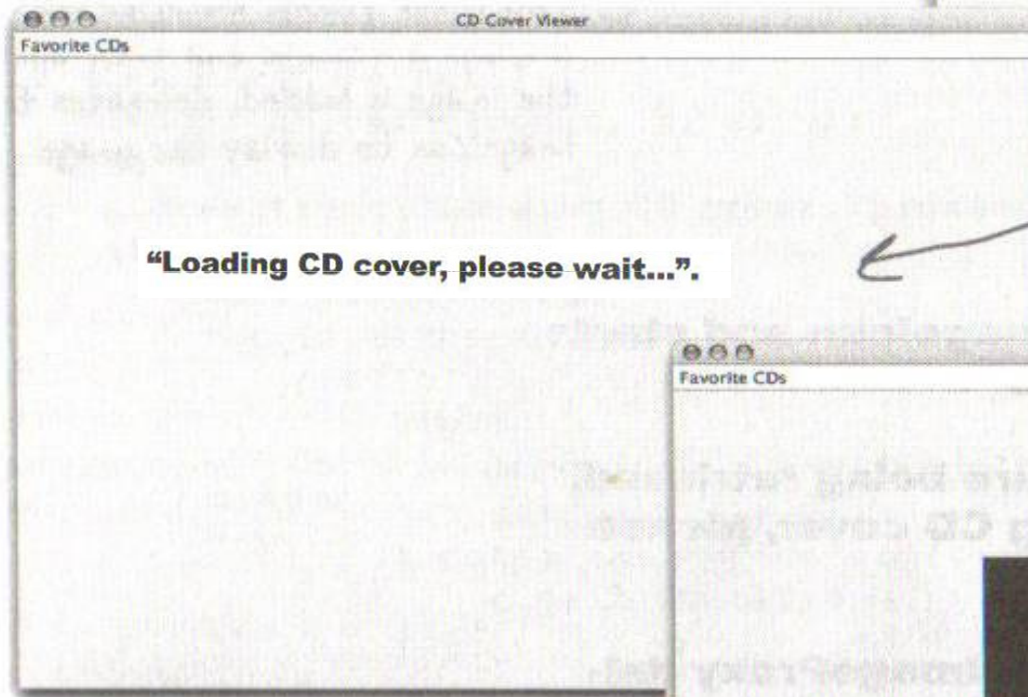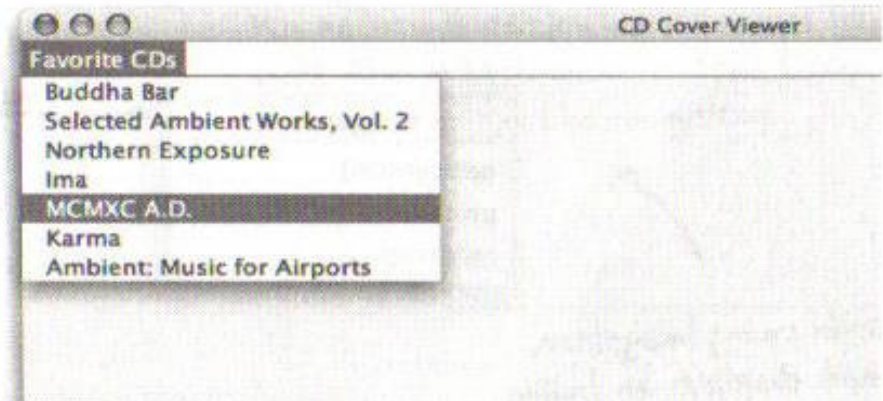
Virtual Proxy

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.

Big "expensive to create" object.

The proxy creates the RealSubject when it's needed.

request()

Client

Proxy

The proxy may handle the request, or if the RealSubject has been created, delegate the calls to the RealSubject.

RealSubject

Playing CD Covers

Choose the album cover of your liking here.

CD Cover Viewer

Favorite CDs
Buddha Bar
Selected Ambient Works, Vol. 2
Northern Exposure
Ima
MCMXC A.D.
Karma
Ambient: Music for Airports

CD Cover Viewer

Favorite CDs

"Loading CD cover, please wait...".

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

CD Cover Viewer

Favorite CDs

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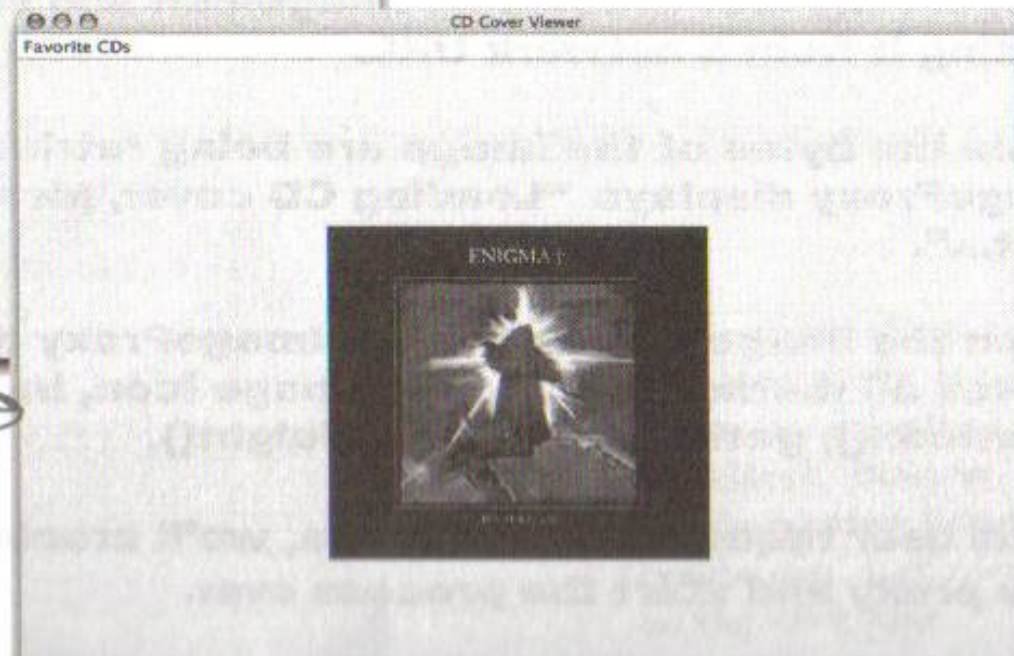
Playing CD Cover Proxy

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

<<interface>>
Icon

getIconWidth()
getIconHeight()
paintIcon()

ImageIcon

getIconWidth()
getIconHeight()
paintIcon()

subject

ImageProxy

getIconWidth()
getIconHeight()
paintIcon()
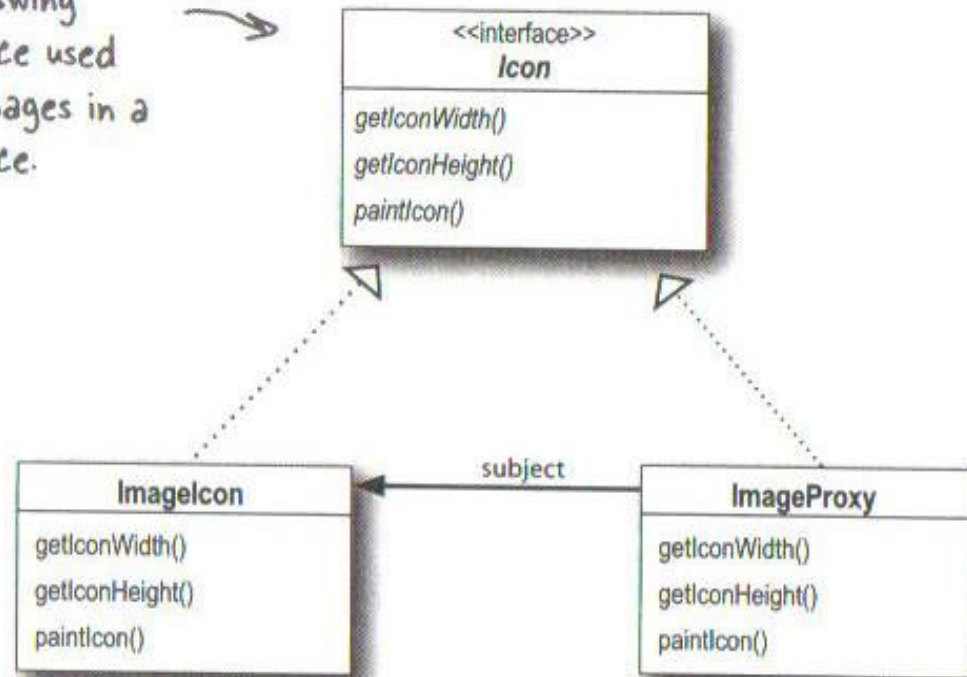
This is javax.swing.ImageIcon, a class that displays an Image.

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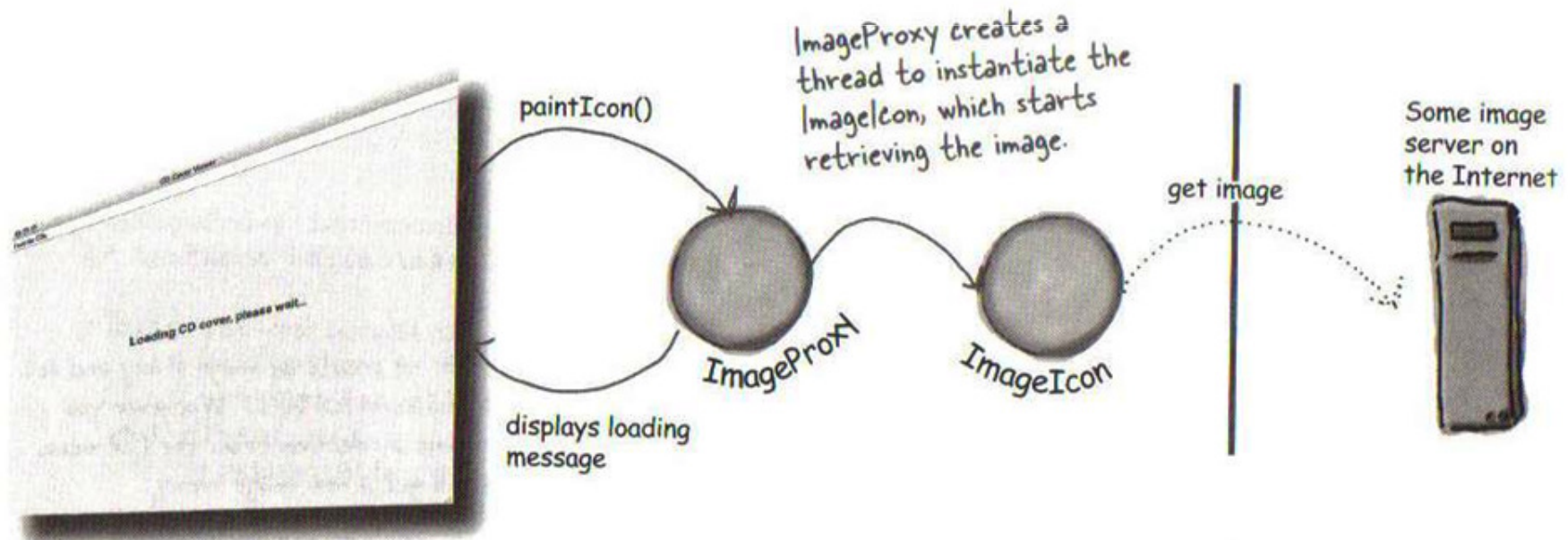
ImageProxy process

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**

ImageProxy creates a thread to instantiate the ImageIcon, which starts retrieving the image.

paintIcon()

Some image server on the Internet

get image

ImageProxy

ImageIcon

displays loading message

Loading CD cover, please wait...

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

image retrieved
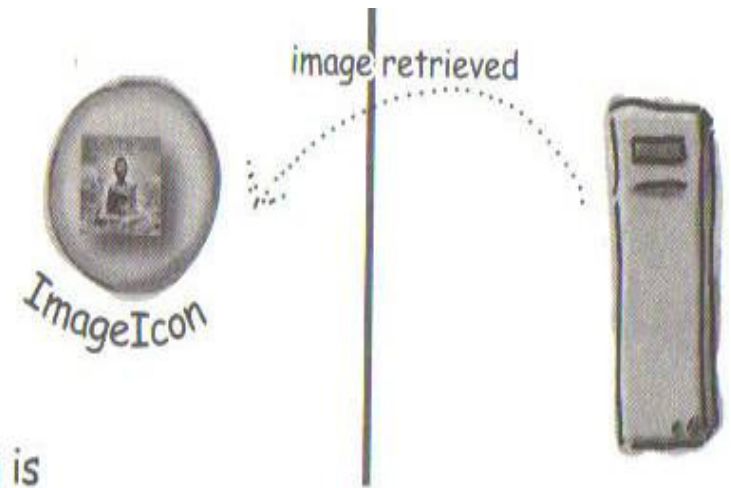
ImageIcon

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

paintIcon()

paintIcon()

ImageProxy
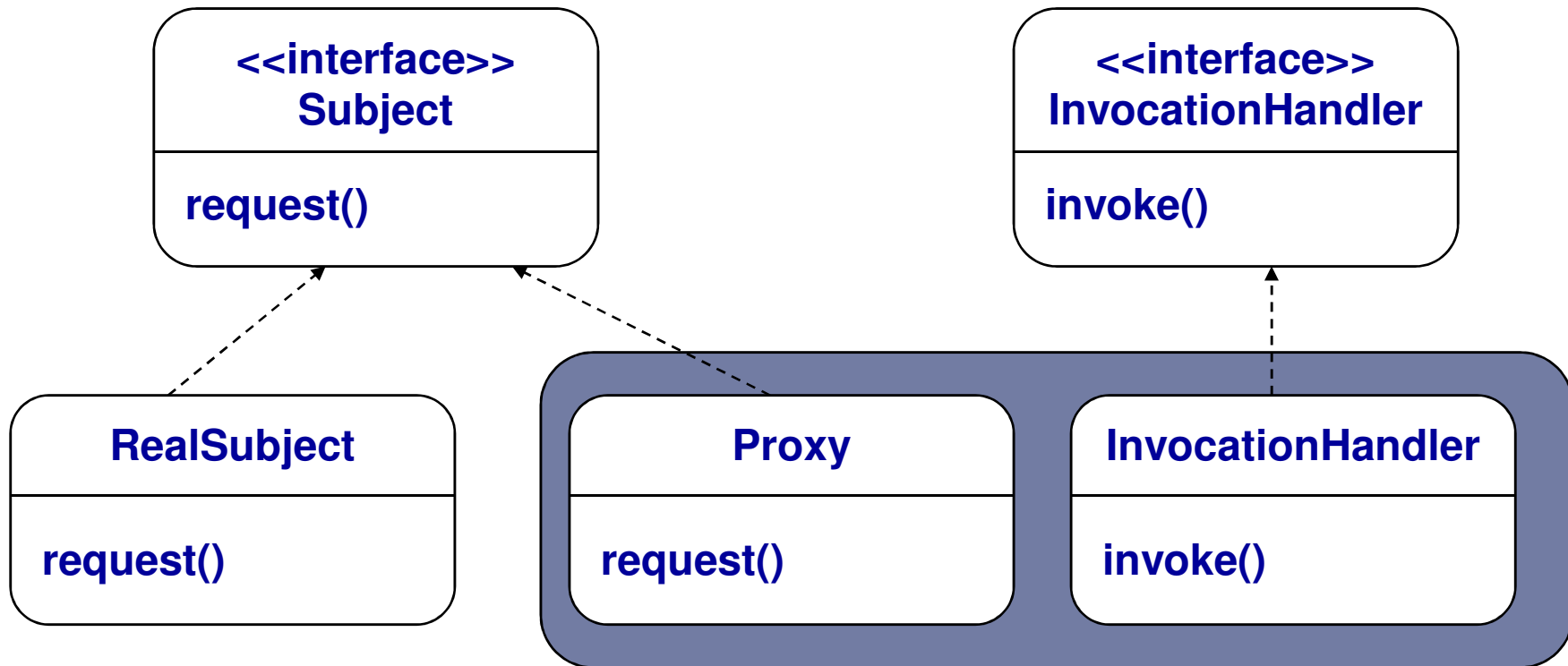
ImageIcon

displays the real image
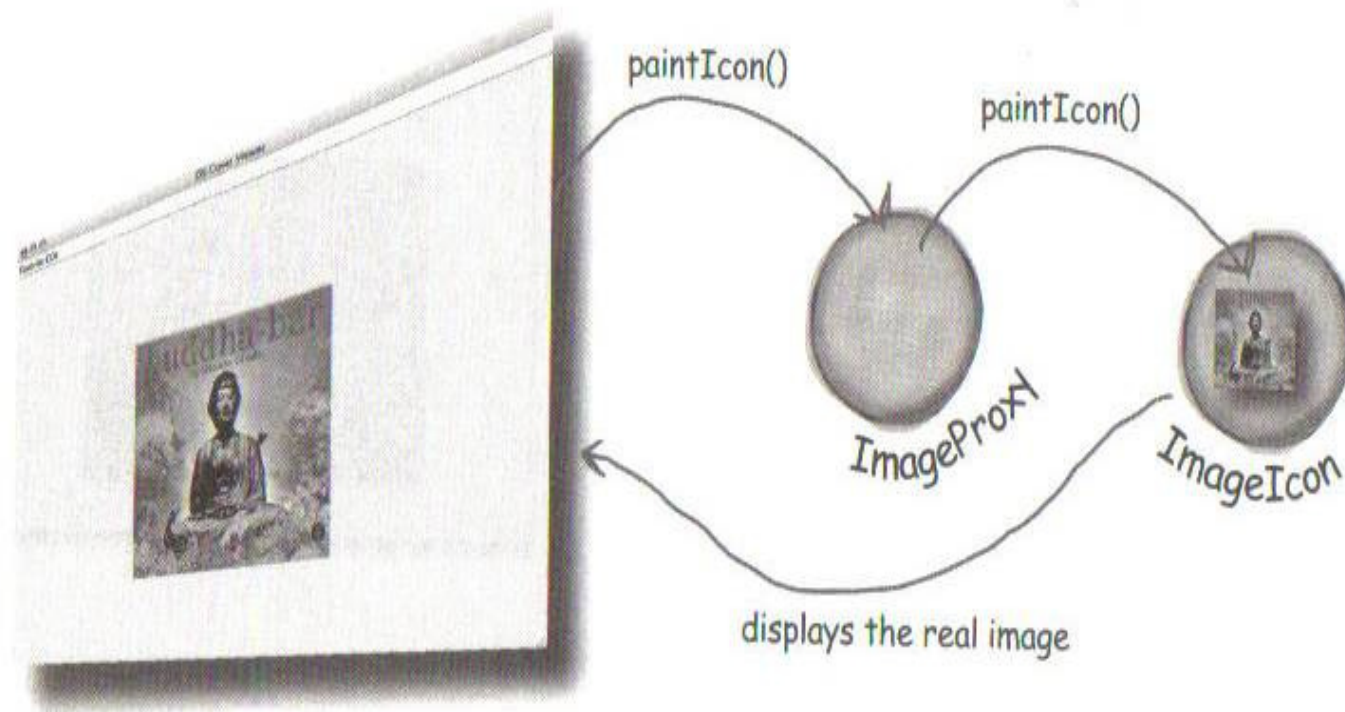
```java
class ImageProxy implements Icon {
   ImageIcon imageIcon;
   URL imageURL;
   Thread retrievalThread;
   boolean retrieving = false;

   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();
             }
      }
}
```

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

```java
}
```

Using Java API's Proxy to create a protection proxy

The proxy zoo

▸ **Firewall proxy**

▸ **Smart Reference proxy**

    ▸ E.g. counts the number of references

▸ **Caching proxy**

▸ **Synchronization Proxy**

▸ **Complexity hiding Proxy**

    ▸ Similar to façade pattern, it also controls accesses

▸ **Copy-on-write Proxy**