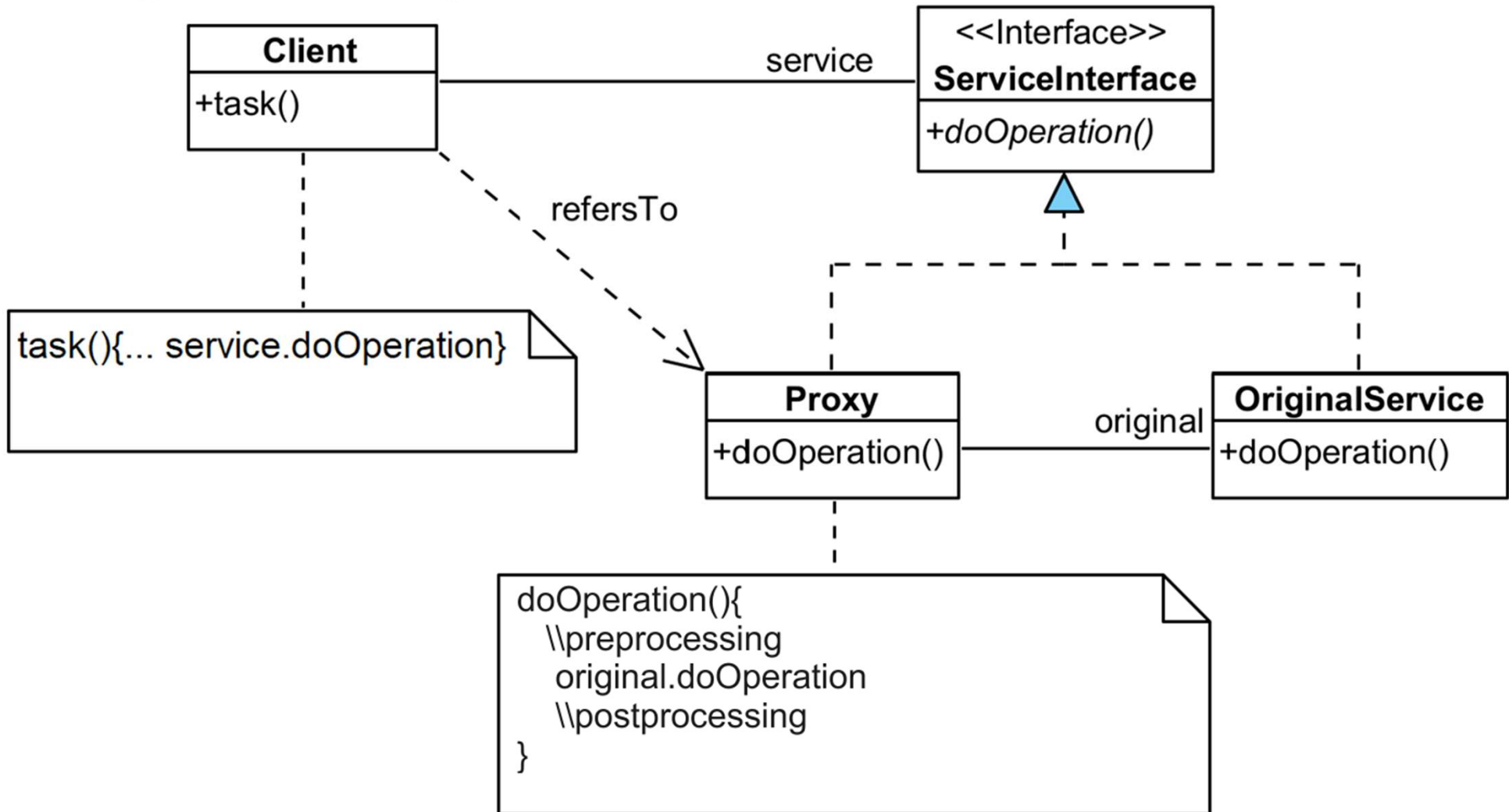
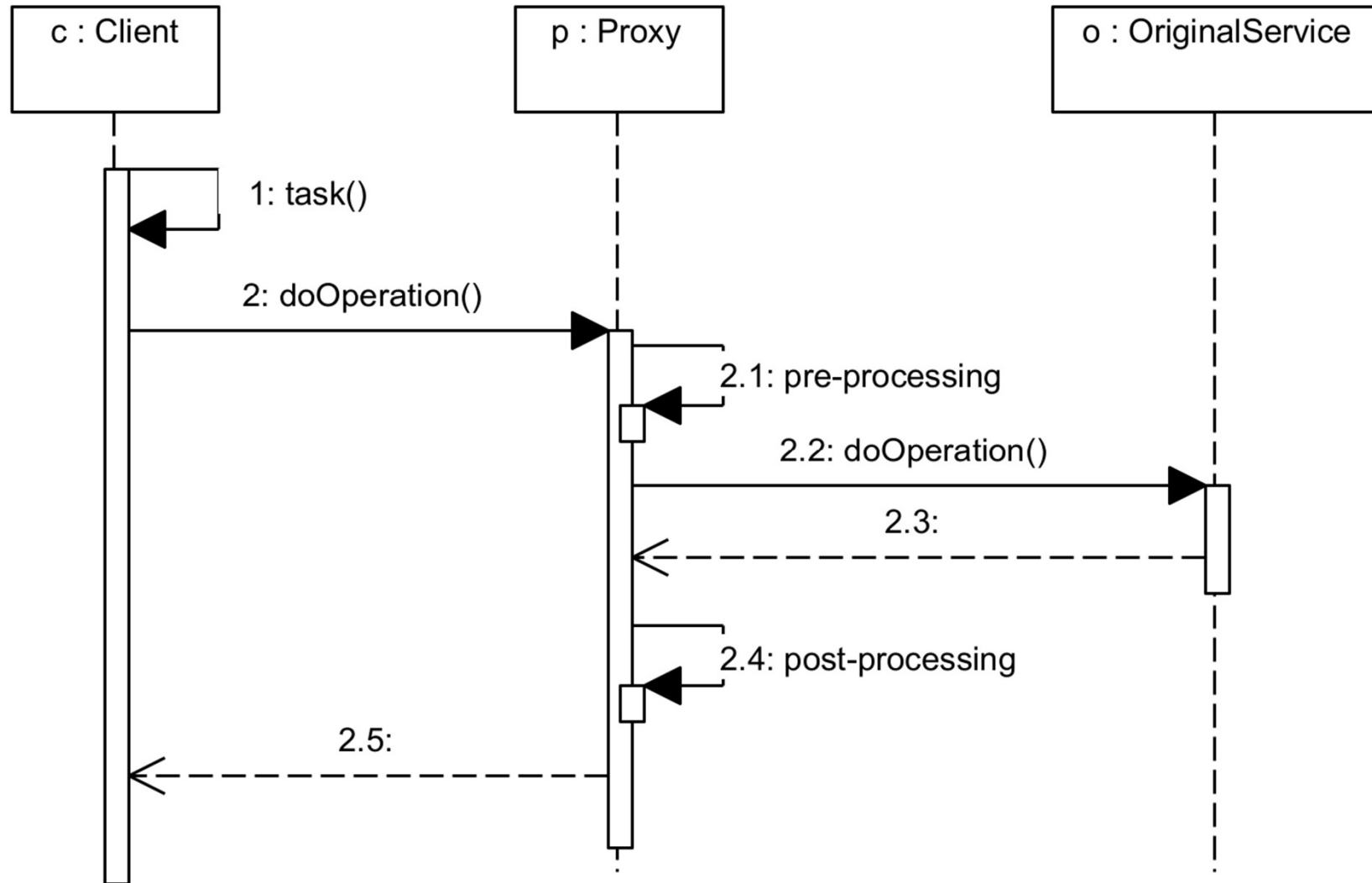

Design Patterns: Proxy

Proxy: modello statico



Proxy: modello dinamico



Osservazioni

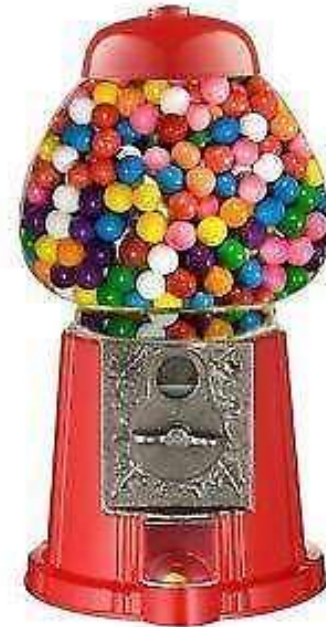
- Proxy fornisce un **surrogato** (o segnaposto) per un altro oggetto, per **controllarne l'accesso**
- **Simile ad Adapter** in quanto si introduce un **intermediario**.
- Sono **diversi** perché
 1. Proxy e oggetto originale hanno la stessa **interfaccia**, Adapter e Adaptee no
 2. Il Proxy può eseguire pre- e posr-elaborazioni

Diversi usi/tipi di Proxy

- Remote Proxy
 - Il Proxy permette l'accesso a un oggetto remoto
 - Usato per esempio in RMI e in Corba
- Protection Proxy
 - Il Proxy implementa un controllo sugli accessi
- Cache Proxy
 - Il Proxy mantiene coppie richiesta-risposta sgravando il server
- Synchronization Proxy
 - Gestisce gli accessi concorrenti a un servizio
- Virtual Proxy
 - Il proxy si comporta come l'originale mentre l'originale viene costruito, ppi passa le richieste a quest'ultimo

Case study: Gumball machine example

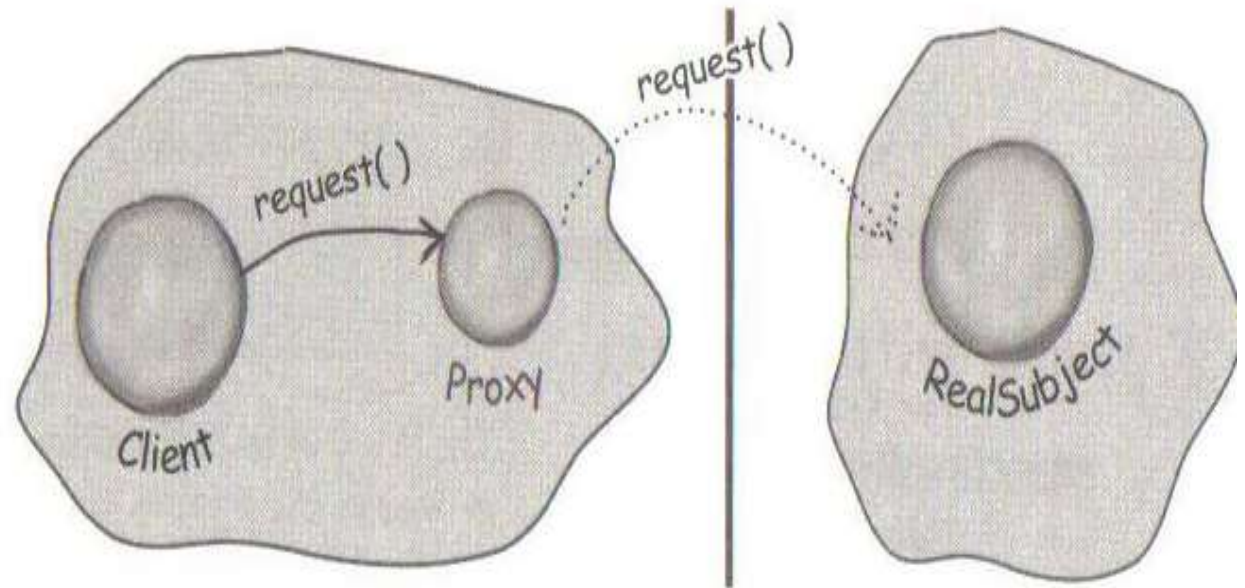
We want to add some monitor to a collection of Gumball machines



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...

Gumball Class


```
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

```
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());
    }
}
```

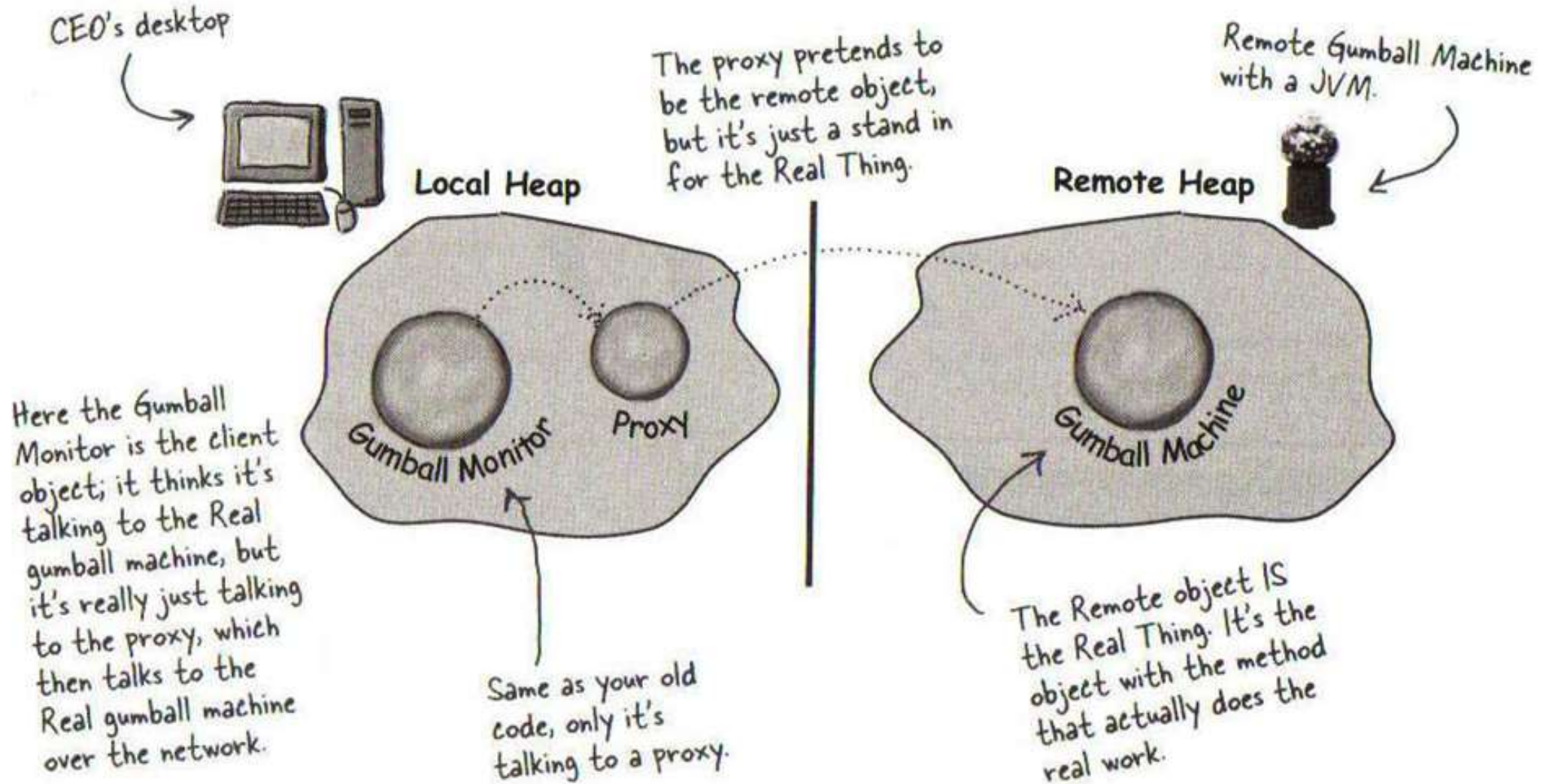
The monitor takes the machine in its constructor and assigns it to the machine instance variable.



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

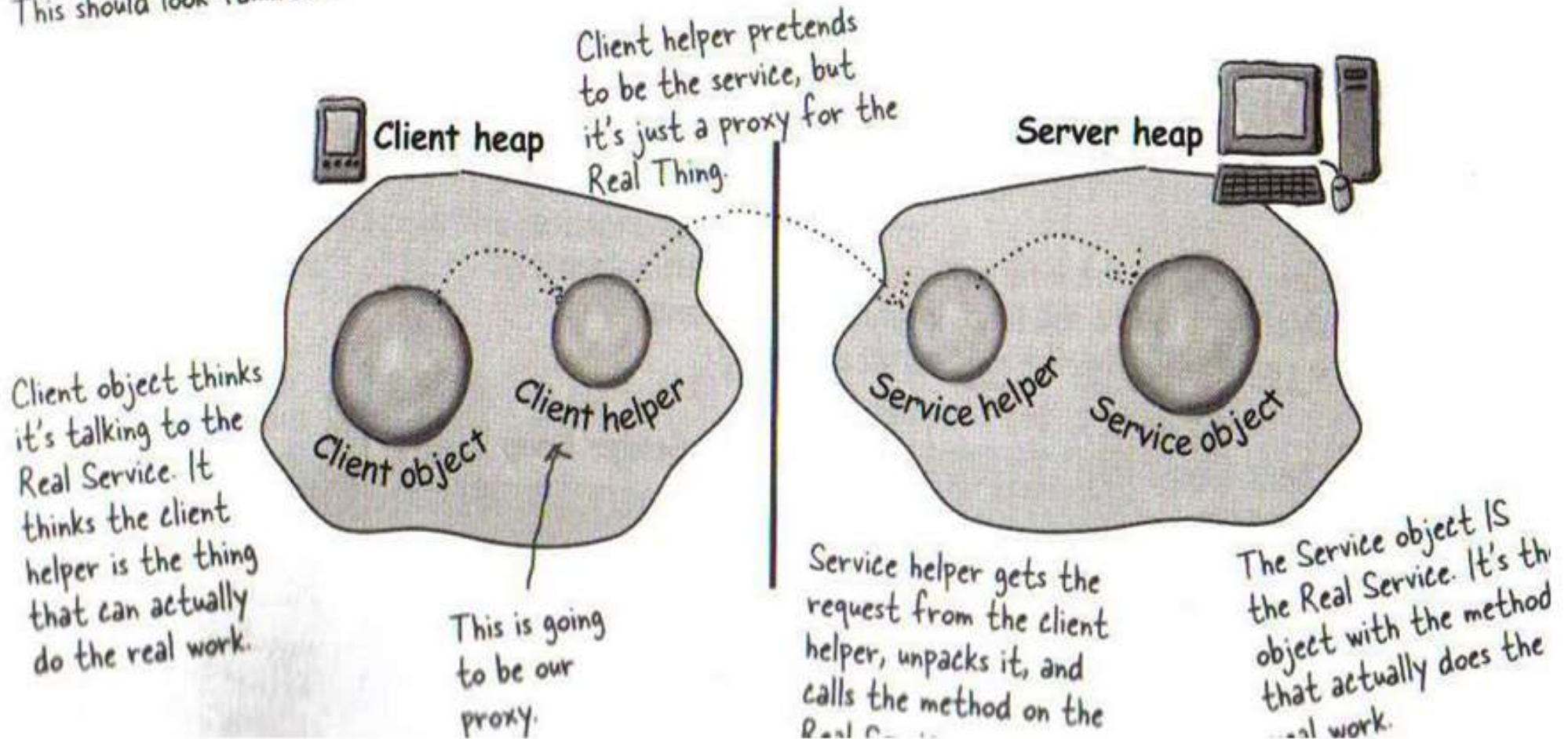


Role of the remote Proxy



Remote Methods

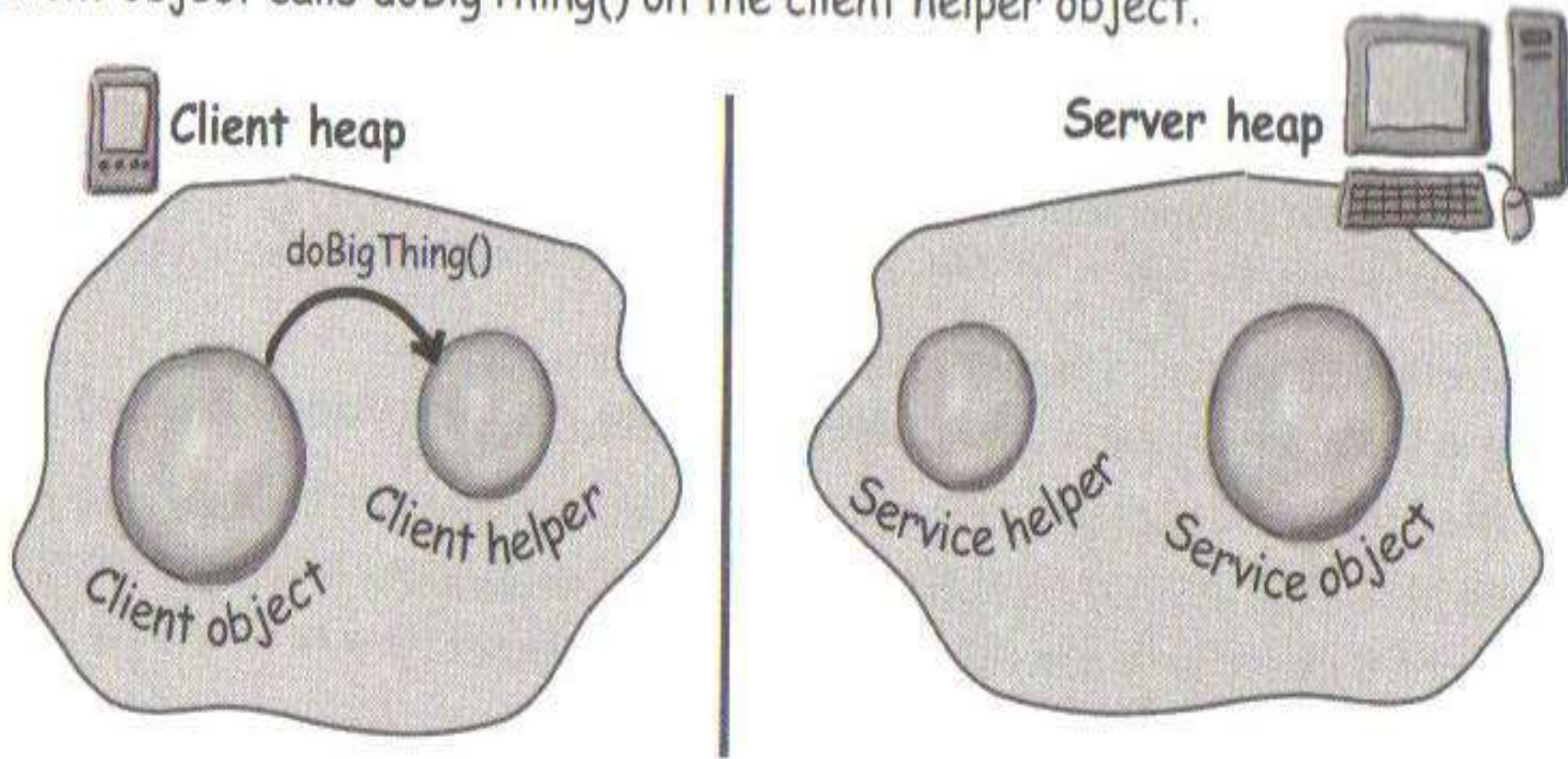
This should look familiar...



How the method call happens

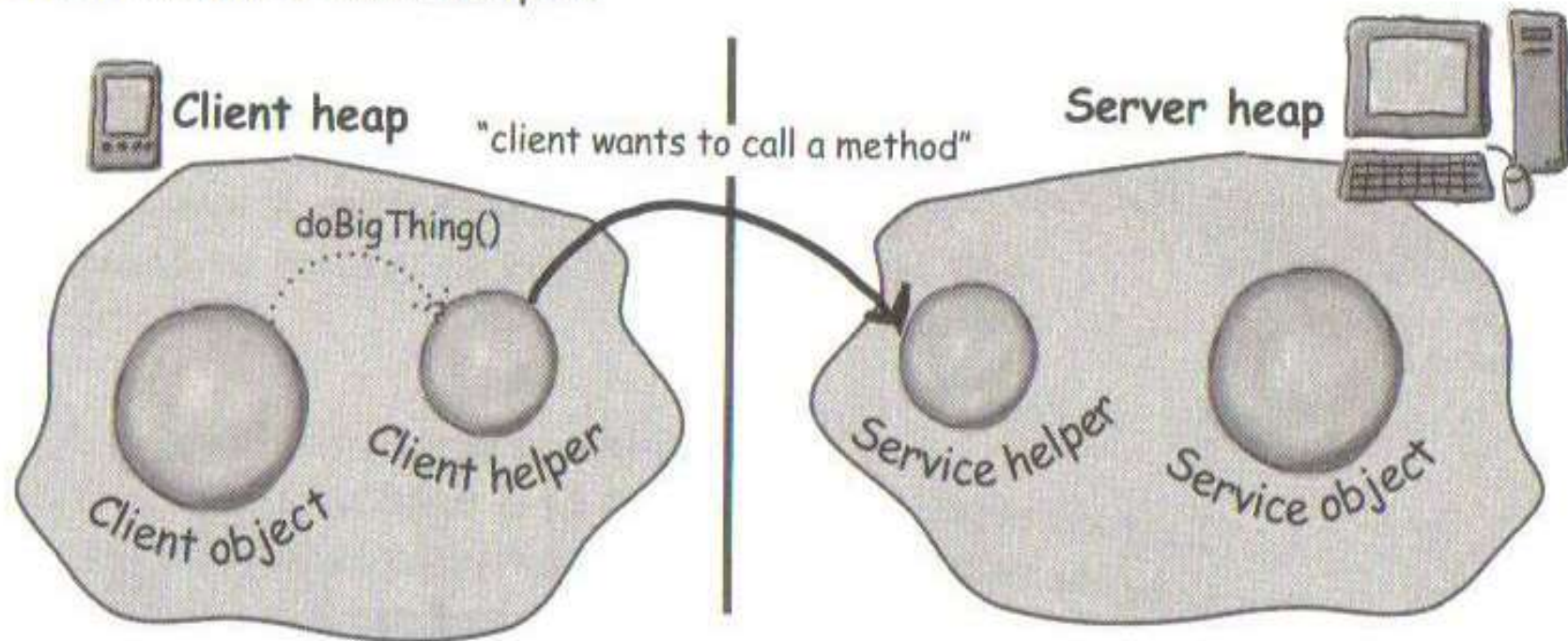
1) Client calls method doBigThing()

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



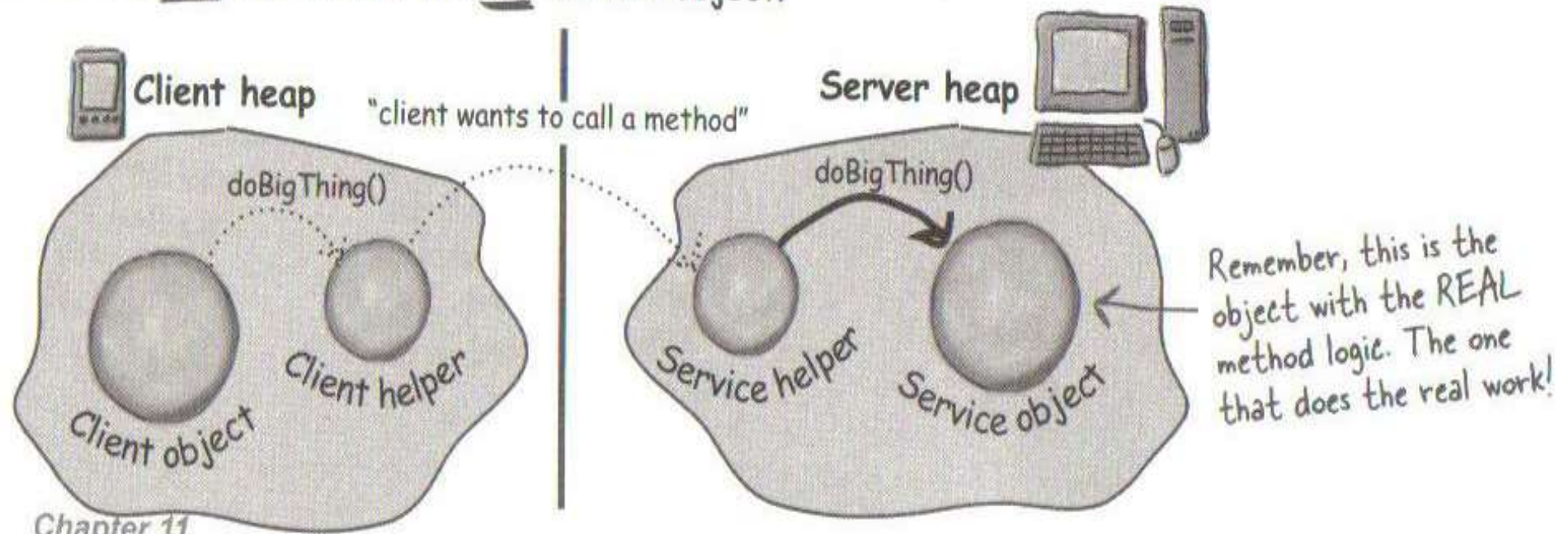
2) Client Helper forwards to service helper

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



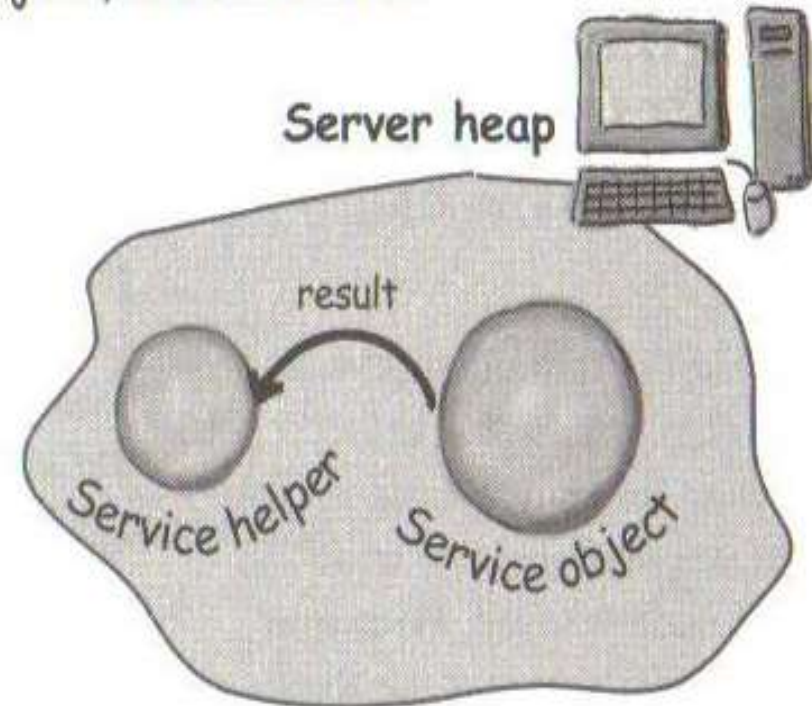
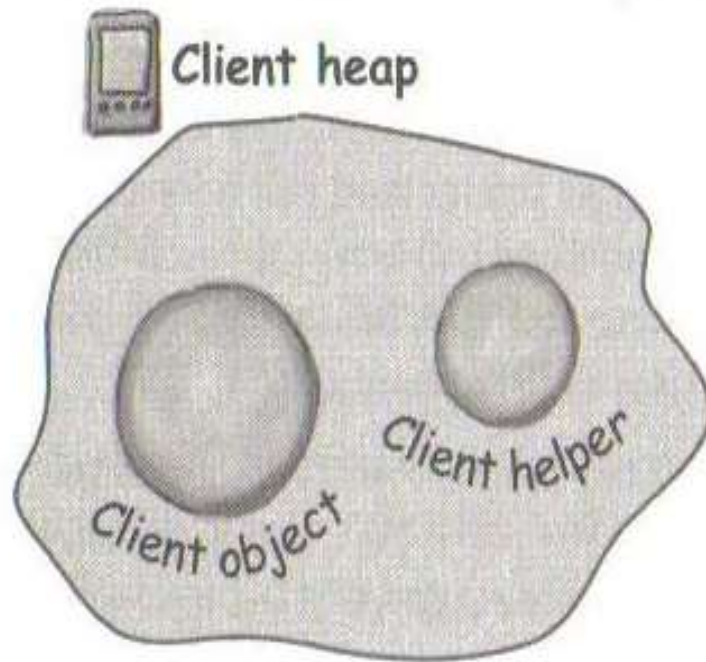
3) Service helper calls the real object

- 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.



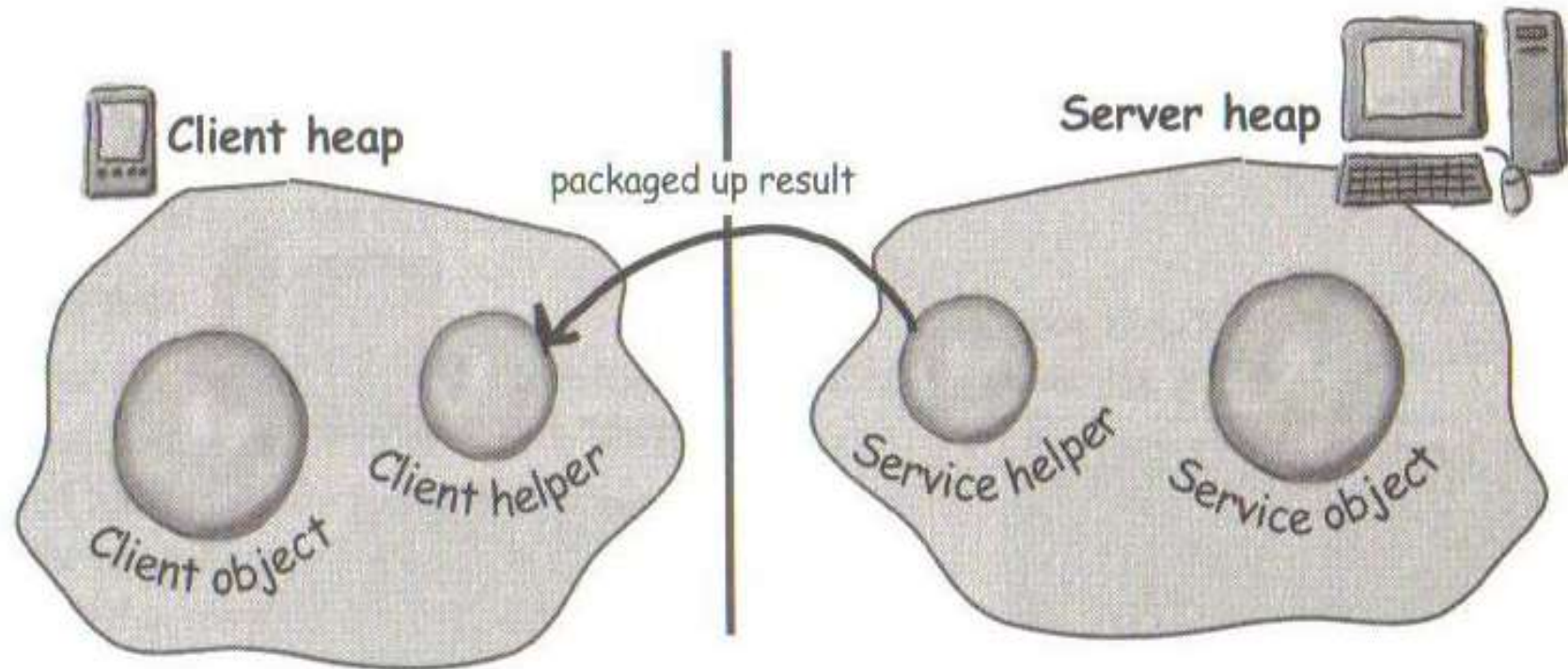
4) Real object returns result

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



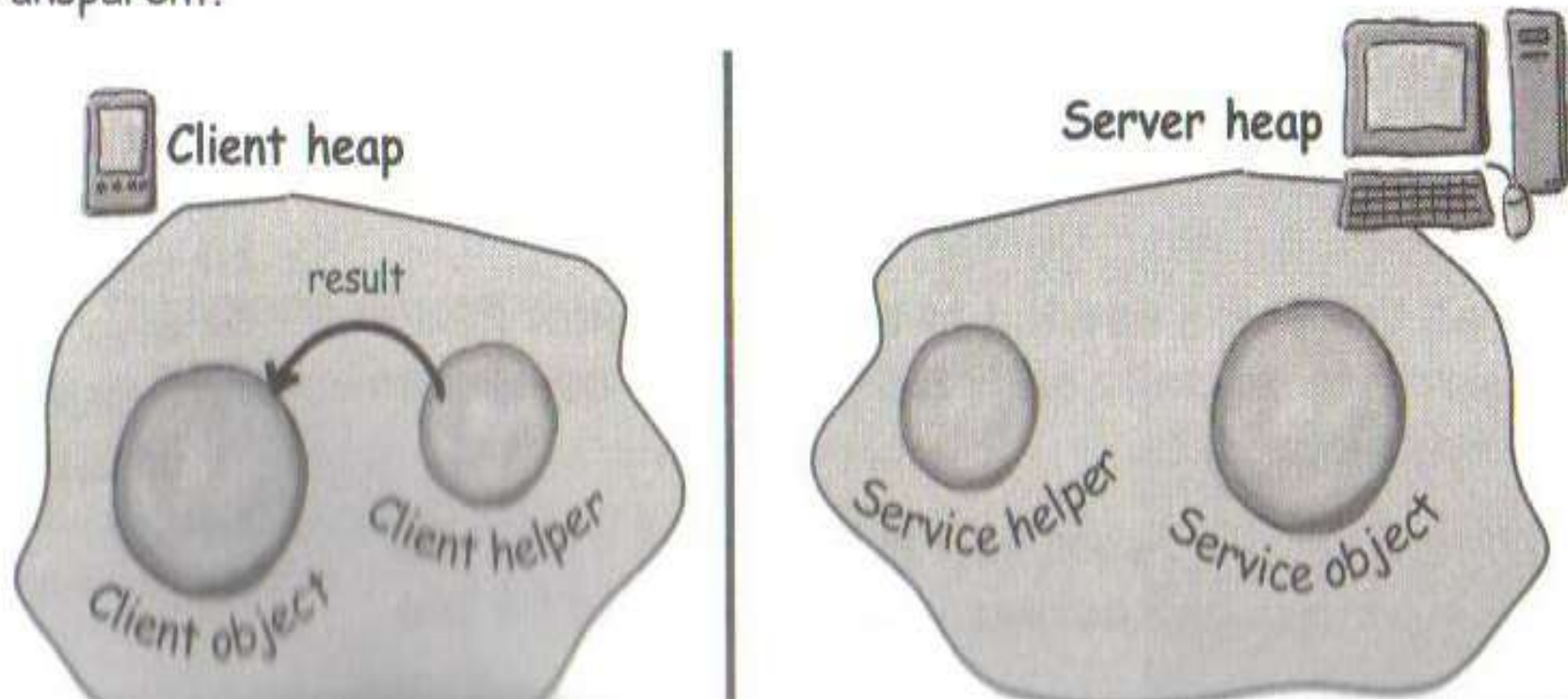
5) Service helper forwards result to client helper

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



6) Client helper returns result to client

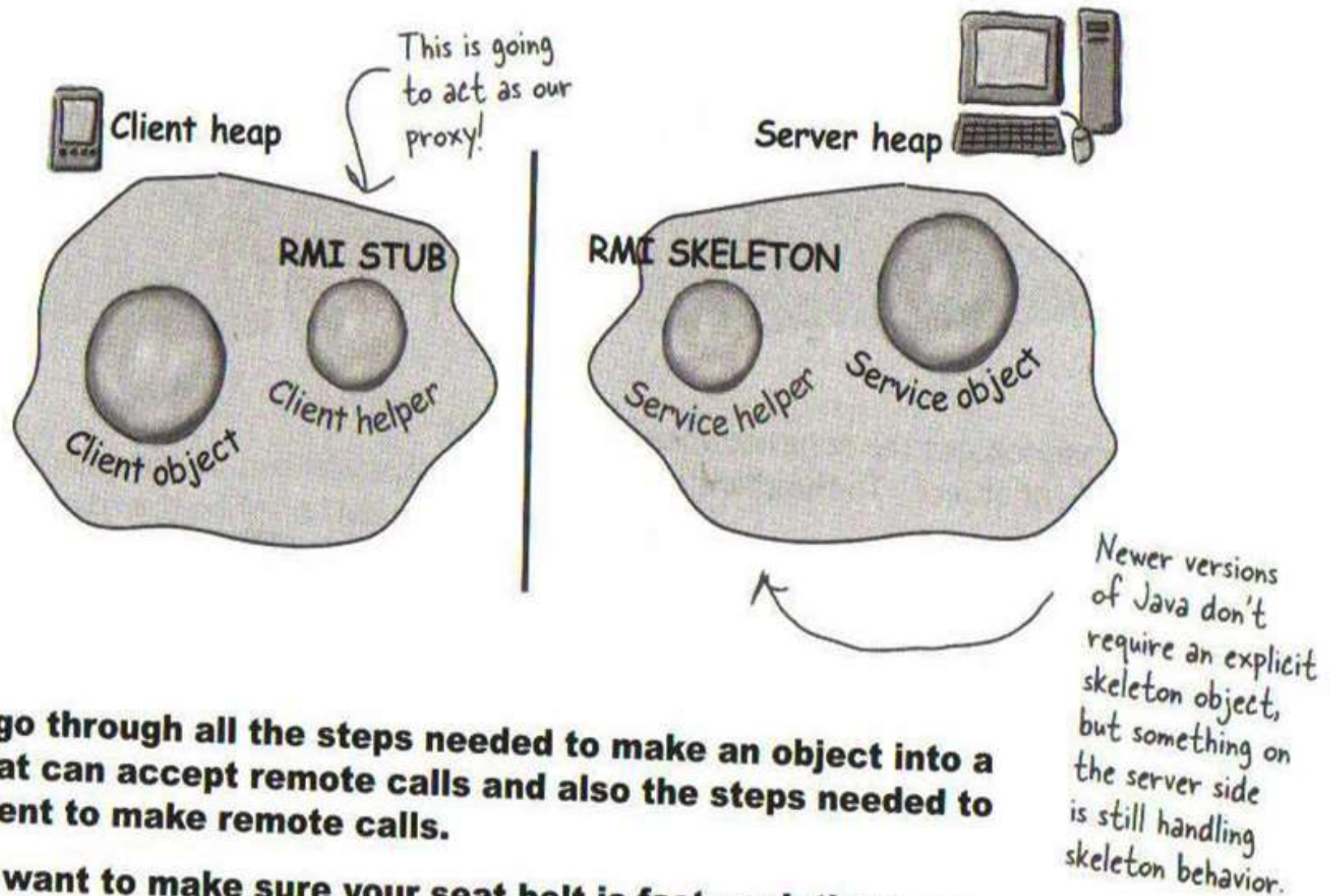
- ⑥ Client helper unpackages the returned values and returns them to the client object. To the client object, this was all transparent.



Remote Proxy with RMI

The client helper is called stub

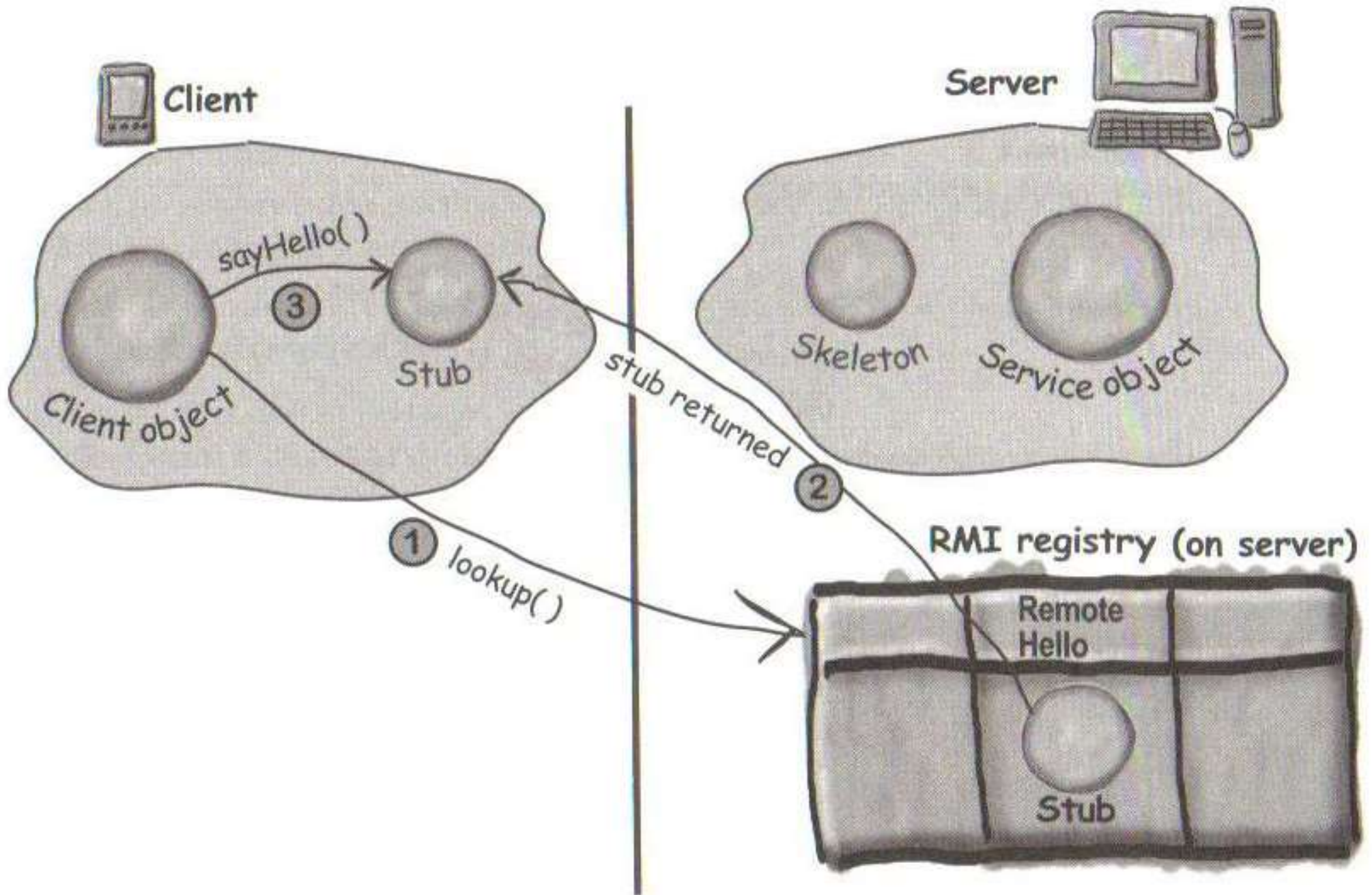
The server helper is called skeleton



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.

Hooking up client and server objects



How it works...

- ① Client does a lookup on the RMI registry

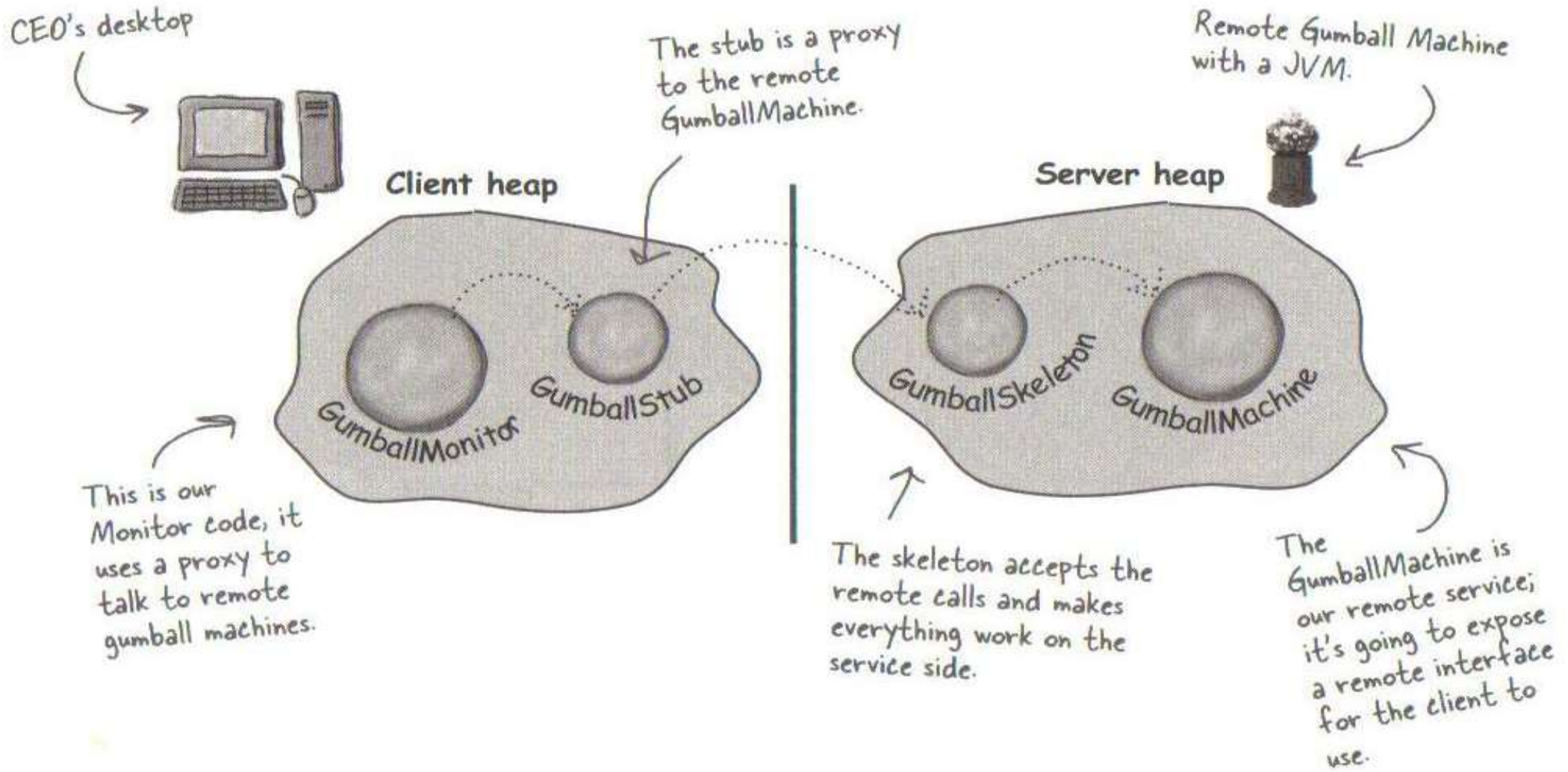
```
Naming.lookup("rmi://127.0.0.1/RemoteHello");
```

- ② RMI registry returns the stub object

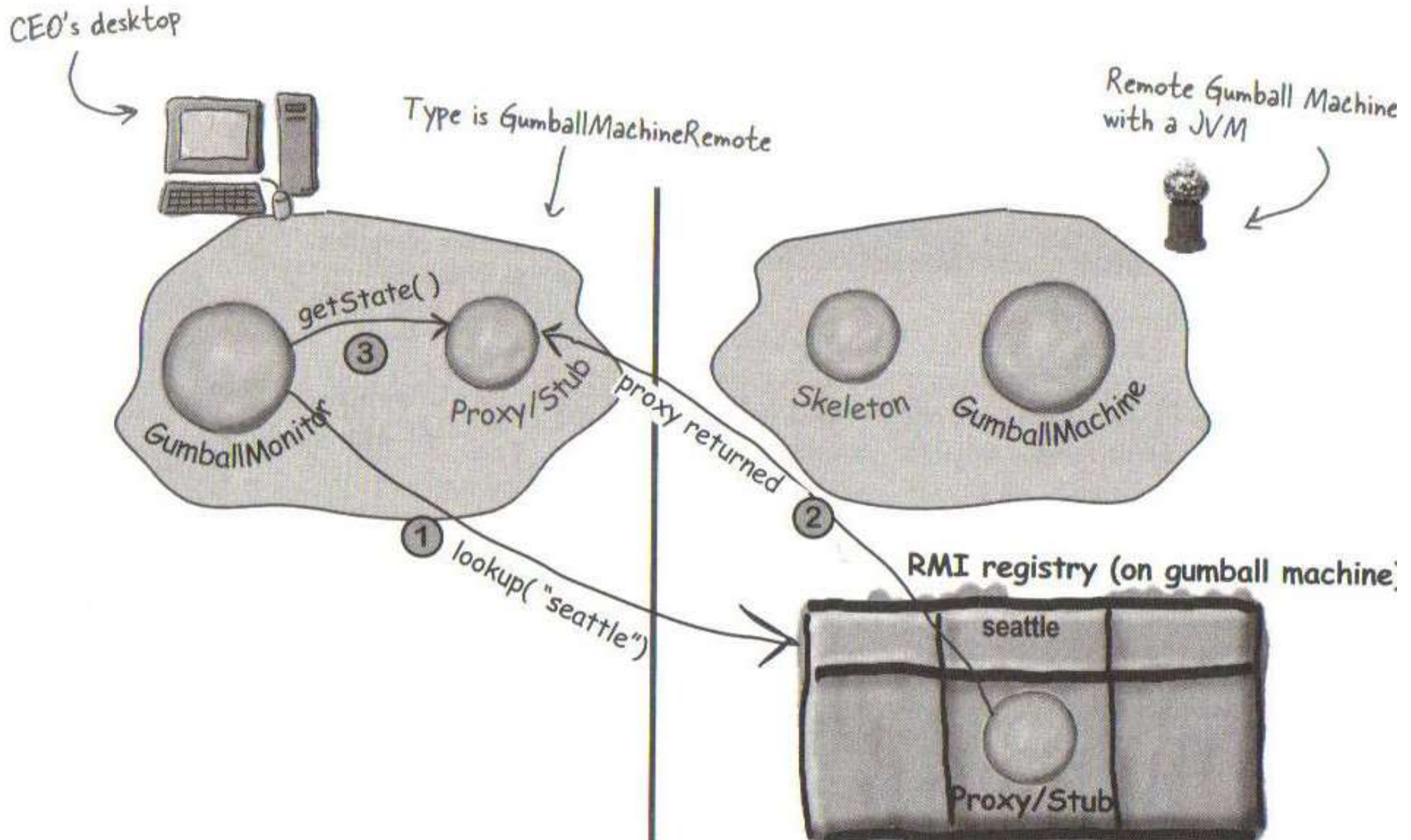
(as the return value of the lookup method) and RMI deserializes the stub automatically. You **MUST** have the stub class (that `rmic` generated for you) on the client or the stub won't be deserialized.

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

Back to Gumball machine problem

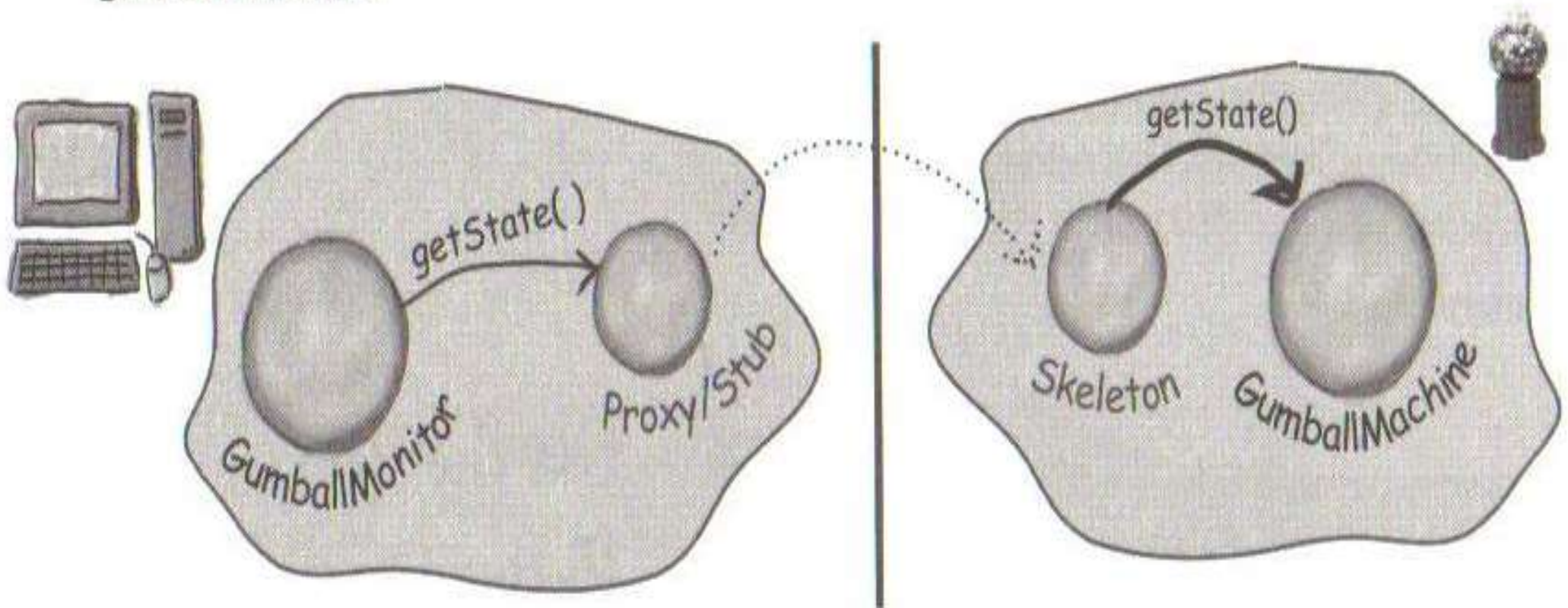


- 1 The CEO runs the monitor, which first grabs the proxies to the remote gumball machines and then calls `getState()` on each one (along with `getCount()` and `getLocation()`).

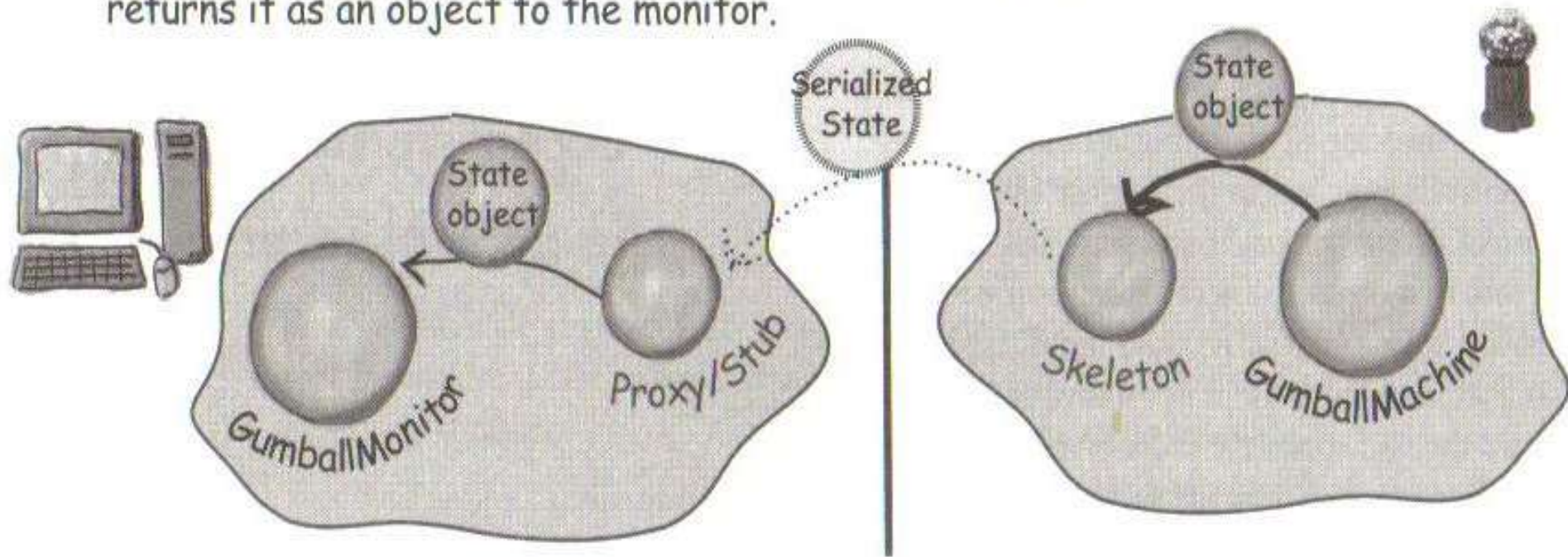


Making the call

- 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 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.

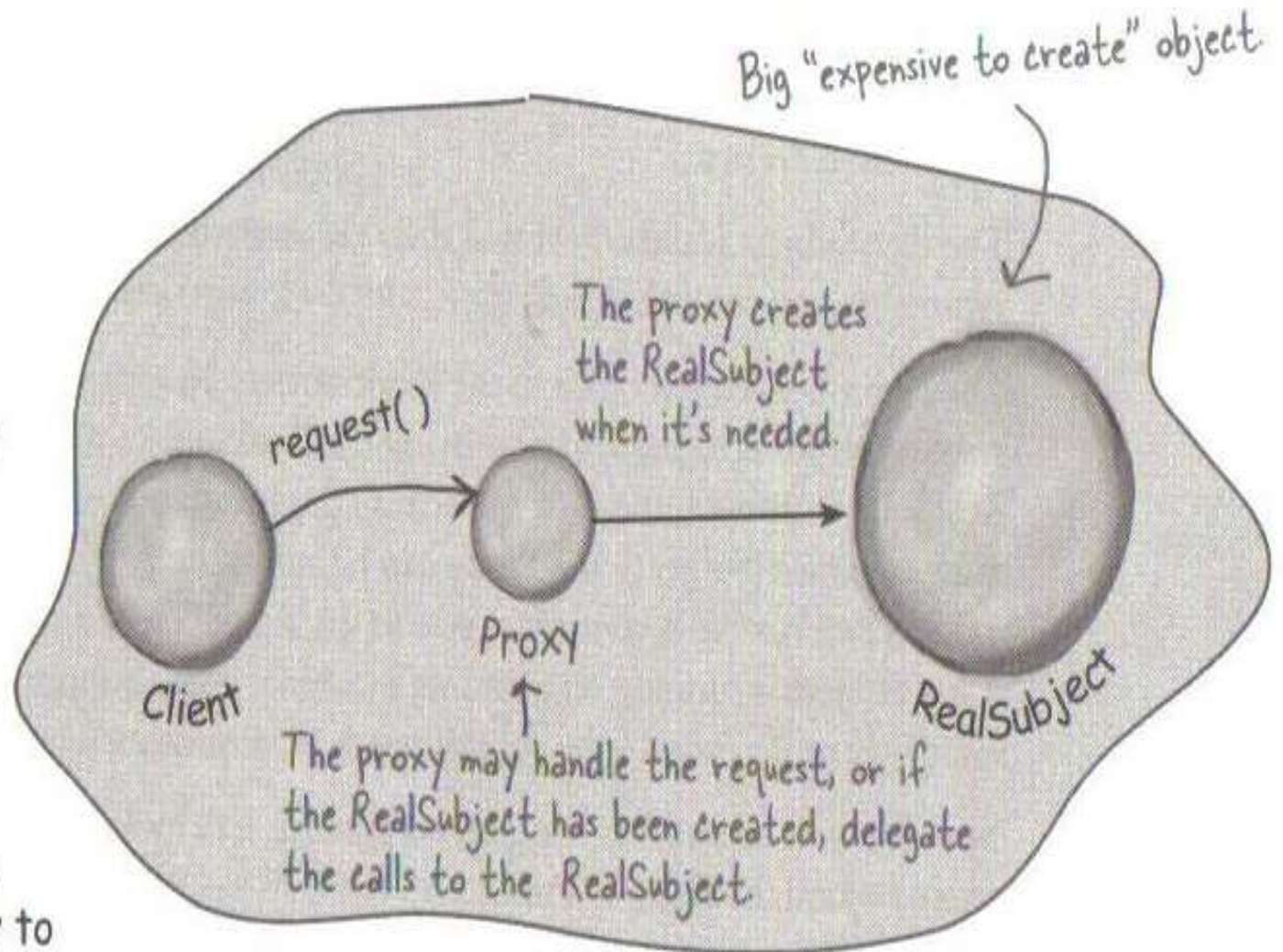
Likewise, the GumballMachine implements another interface and may throw a remote exception in its constructor, but other than that, the code hasn't changed.

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.

Virtual Proxy

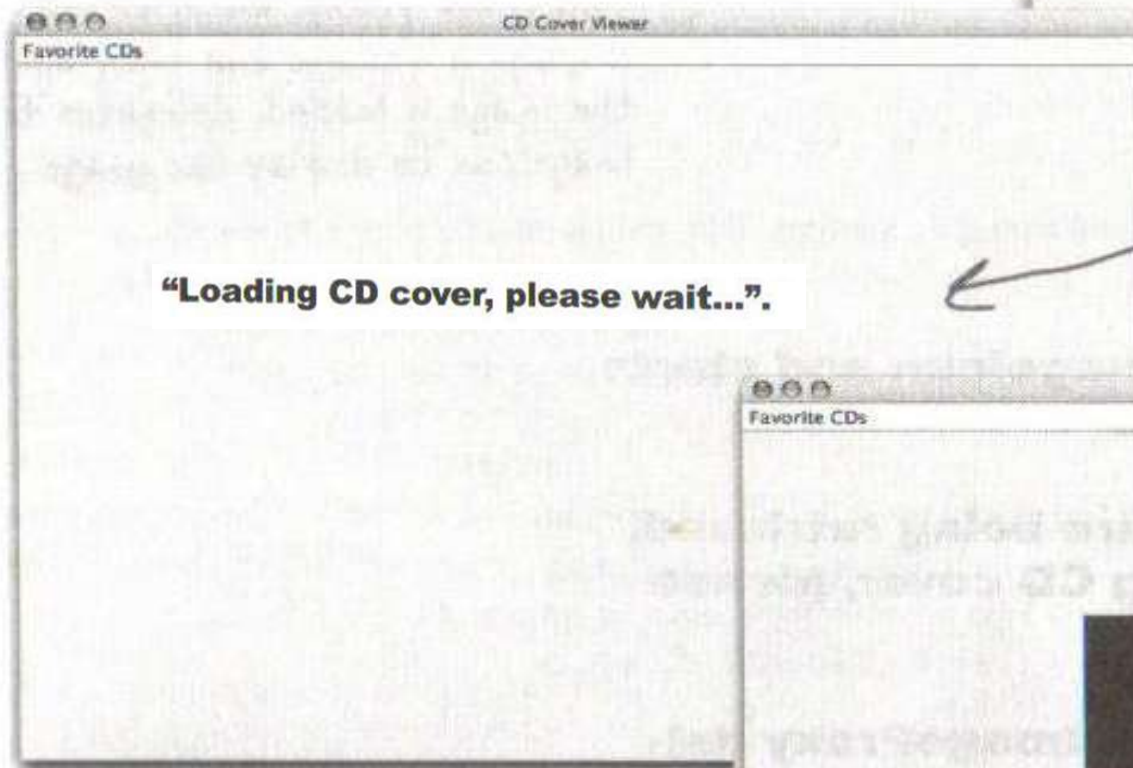
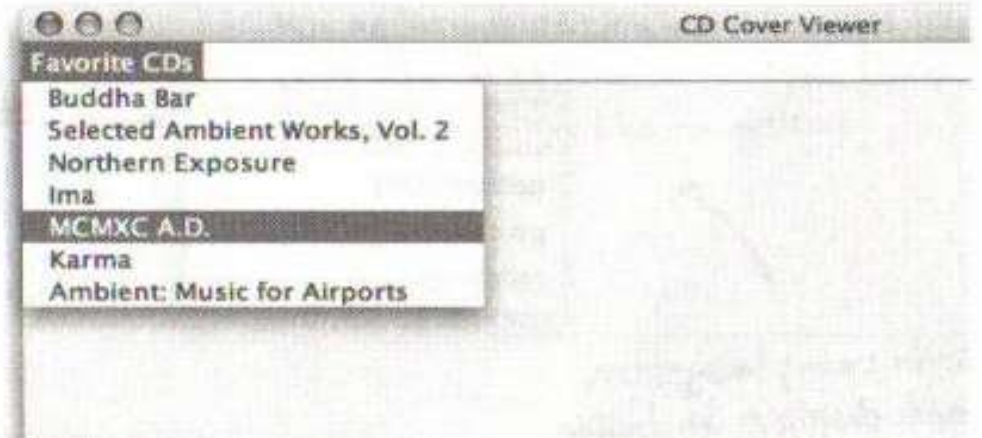
Virtual Proxy

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.



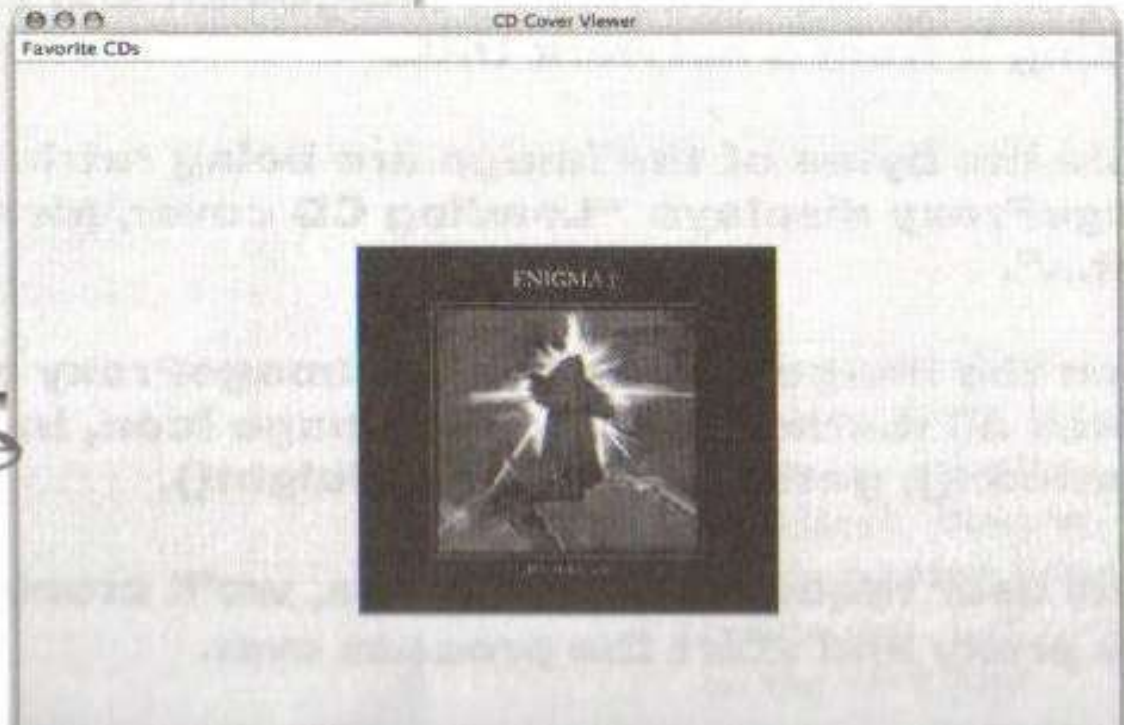
Choose the album cover of your liking here.

Playing CD Covers



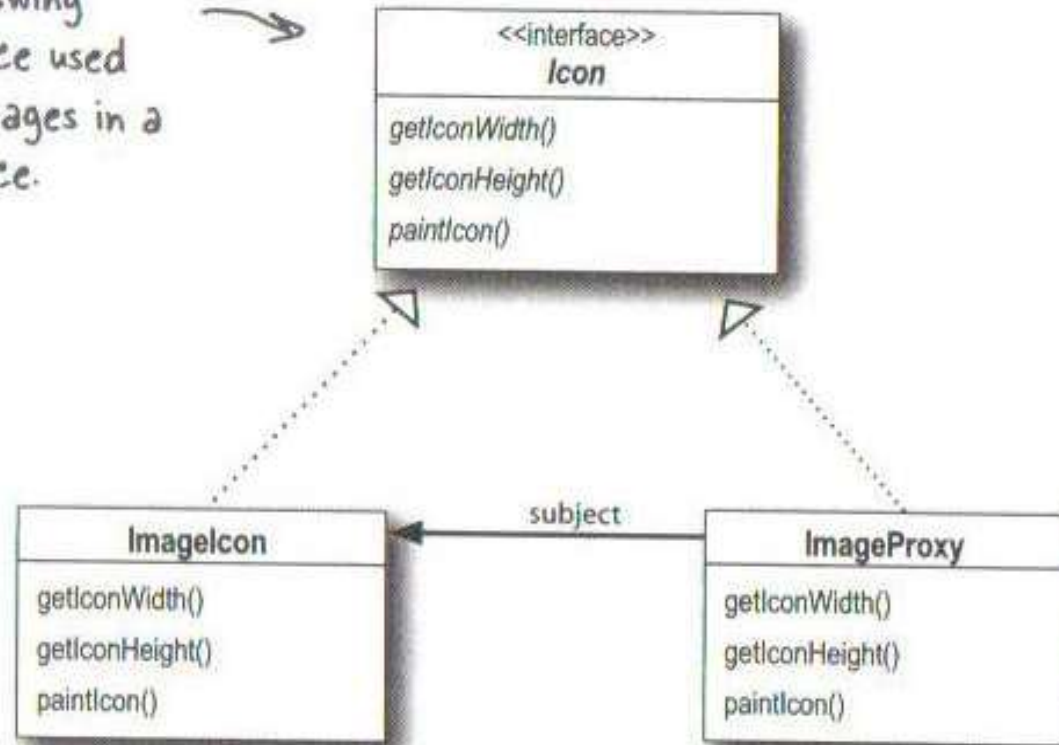
While the CD cover is loading, the proxy displays a message.

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

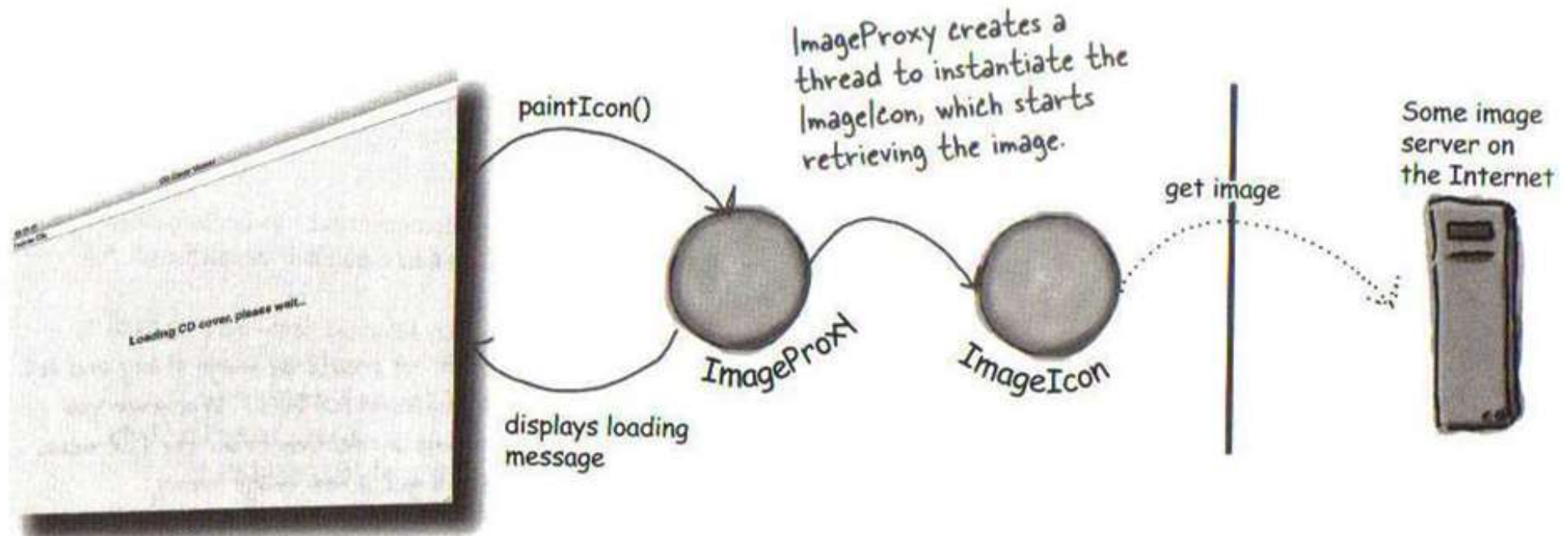
- 1 ImageProxy first creates an ImageIcon and starts loading it from a network URL.**
- 2 While the bytes of the image are being retrieved, ImageProxy displays “Loading CD cover, please wait...”.**
- 3 When the image is fully loaded, ImageProxy delegates all method calls to the image icon, including 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

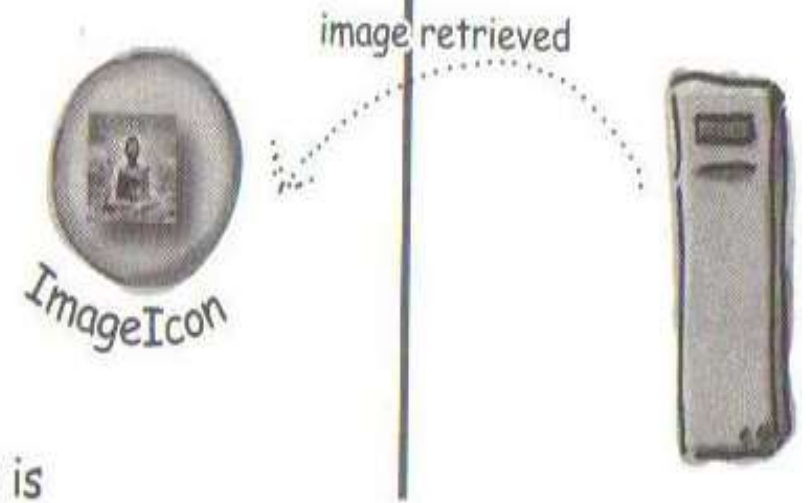
What did we do?

- 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.

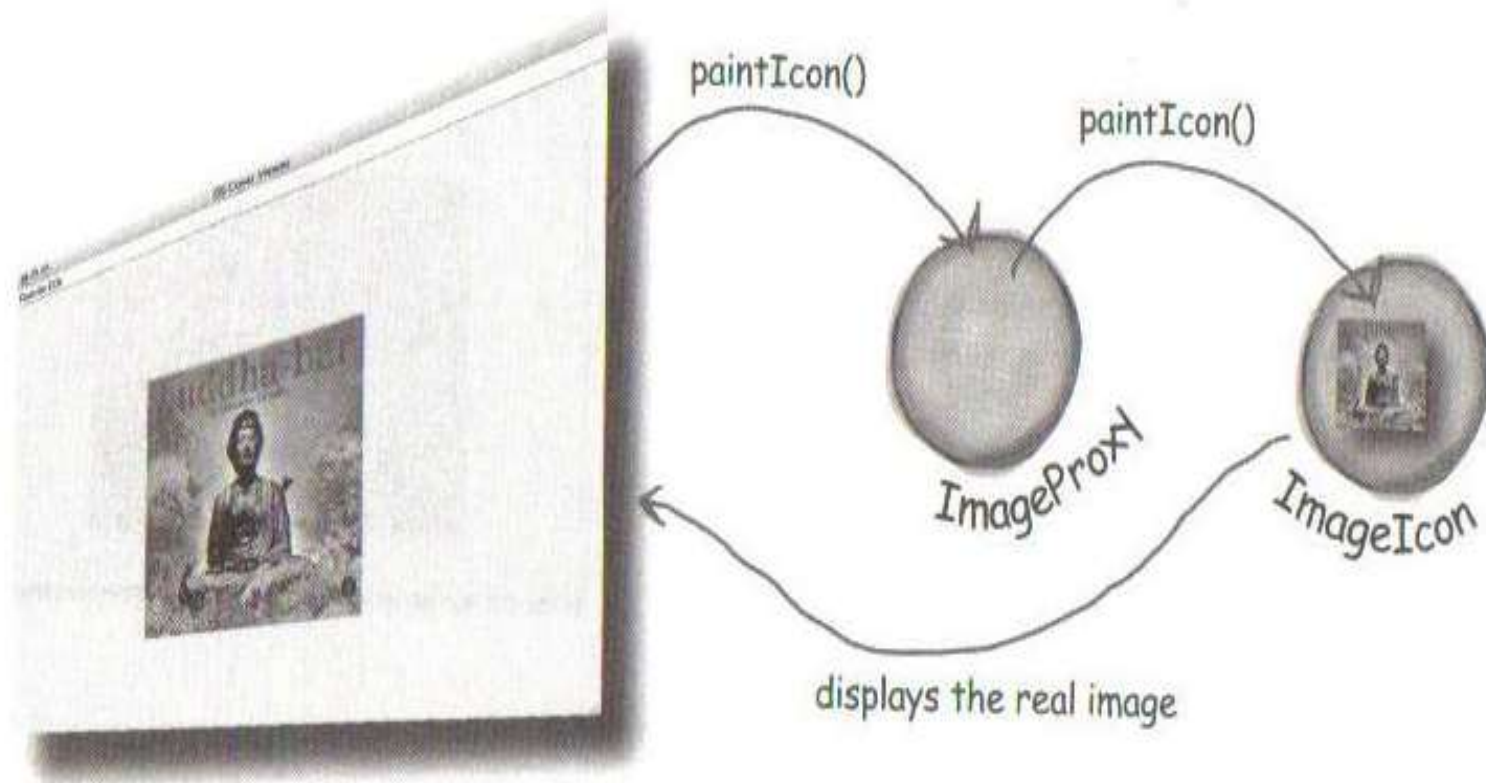
Behind
the Scenes



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.



```

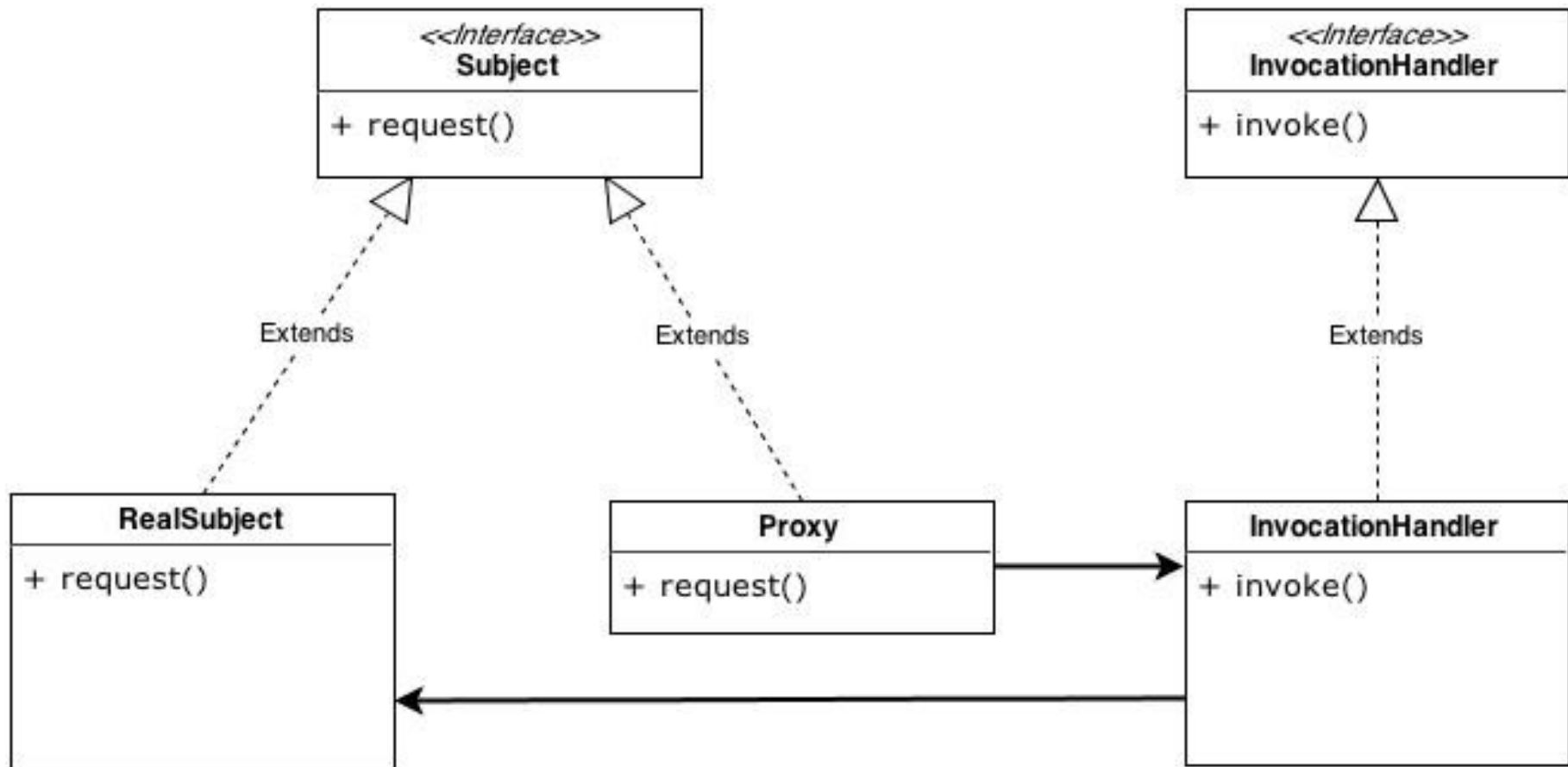
class ImageProxy implements Icon {
    ImageIcon imageIcon;
    URL imageURL;
    Thread retrievalThread;
    boolean retrieving = false;

    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();
            }
        }
    }
}

```


java.lang.reflect package can be used to create a proxy class dynamically.



Homework

Consider your phone being the subject.

Build a firewall proxy that filters sms and phone calls to block those of stalkers .

- The blacklist must be updatable