Tecniche di Progettazione: Design Patterns

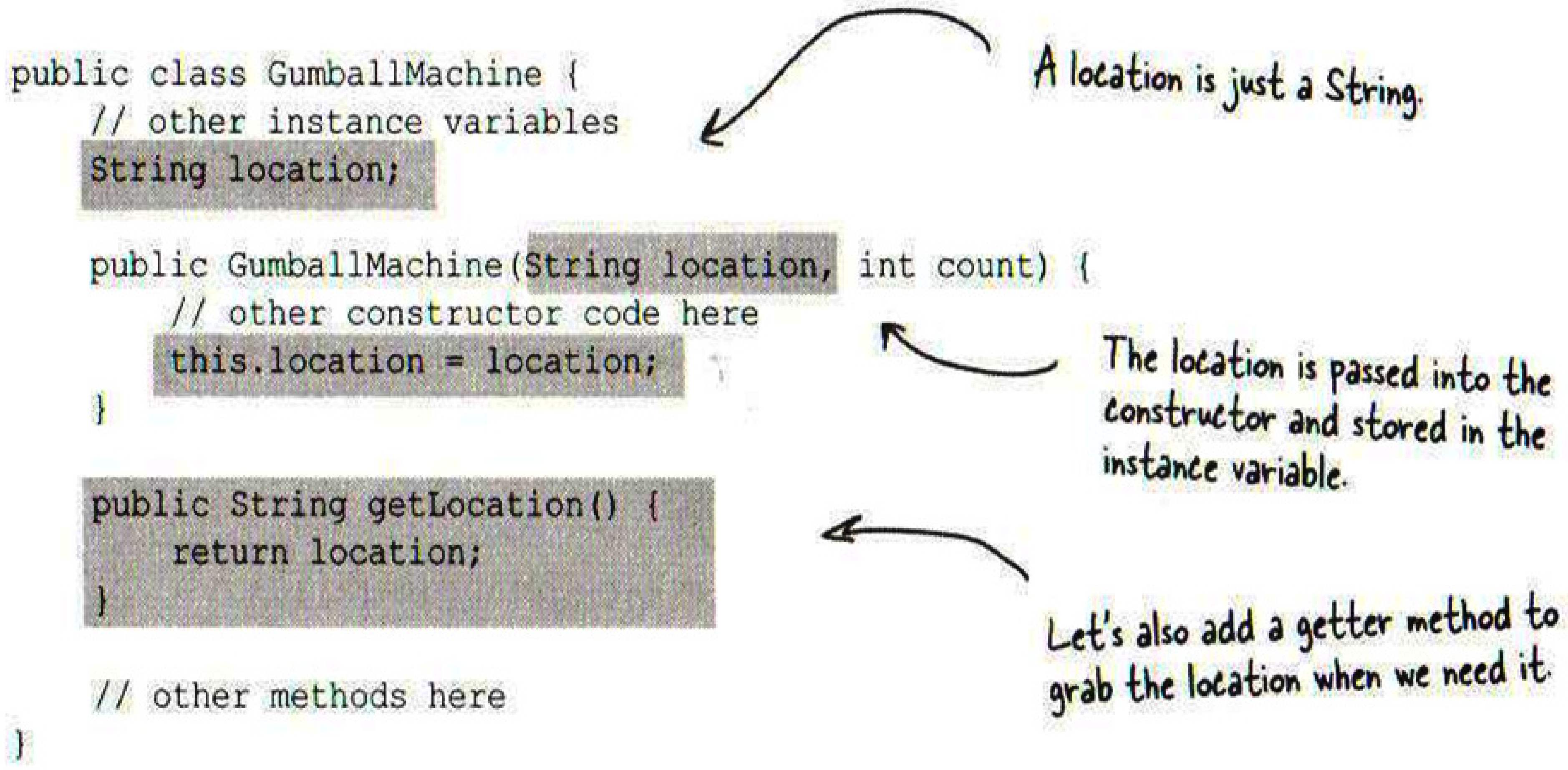
GoF: Proxy

Case study: Gumball machine example

- The same example covered in Head First for the State pattern

- Now we want to add some monitor to a collection of Gumball machines

Gumball Class



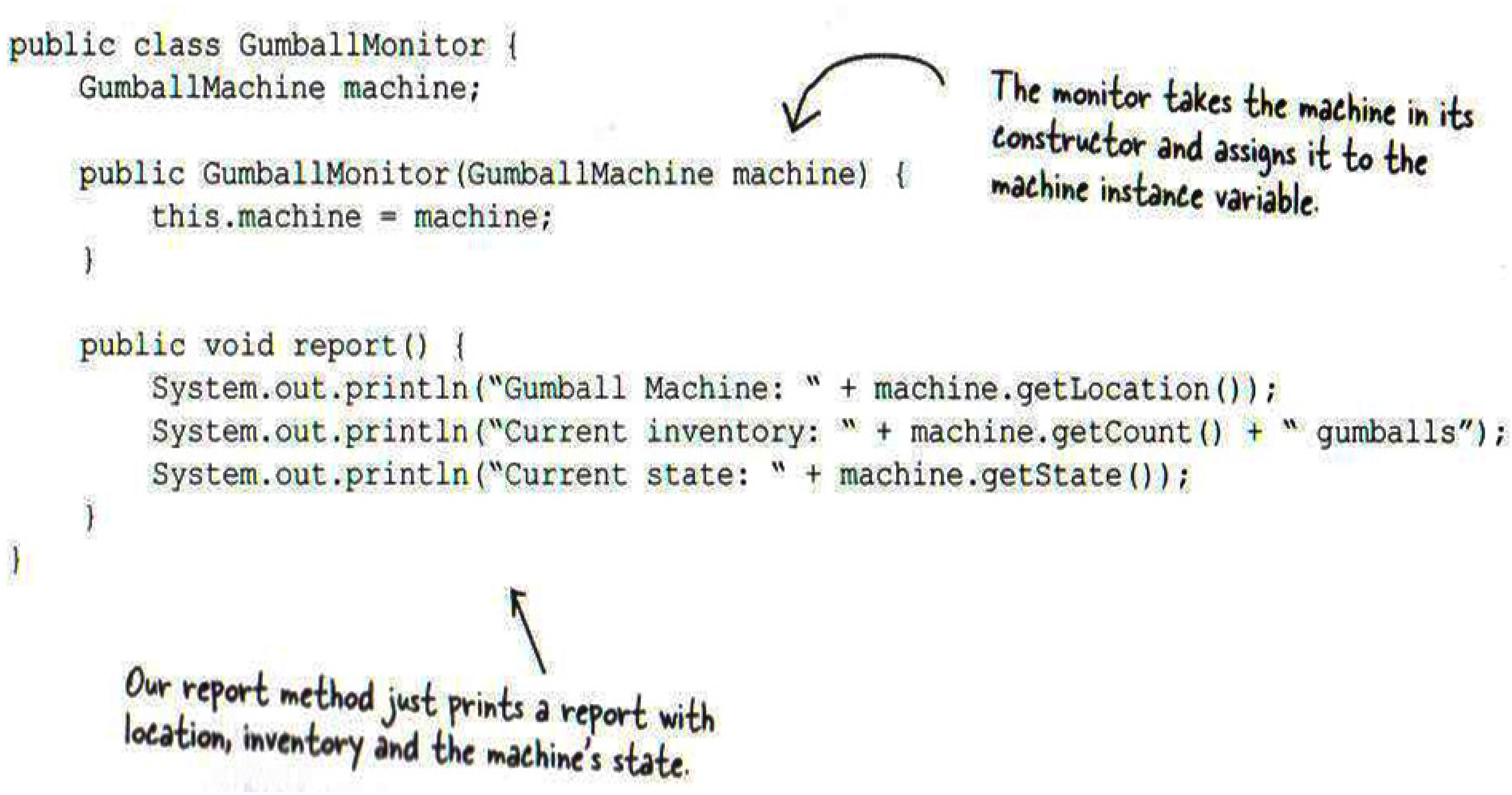
```
public class GumballMachine {
    // other instance variables
    String location;

    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

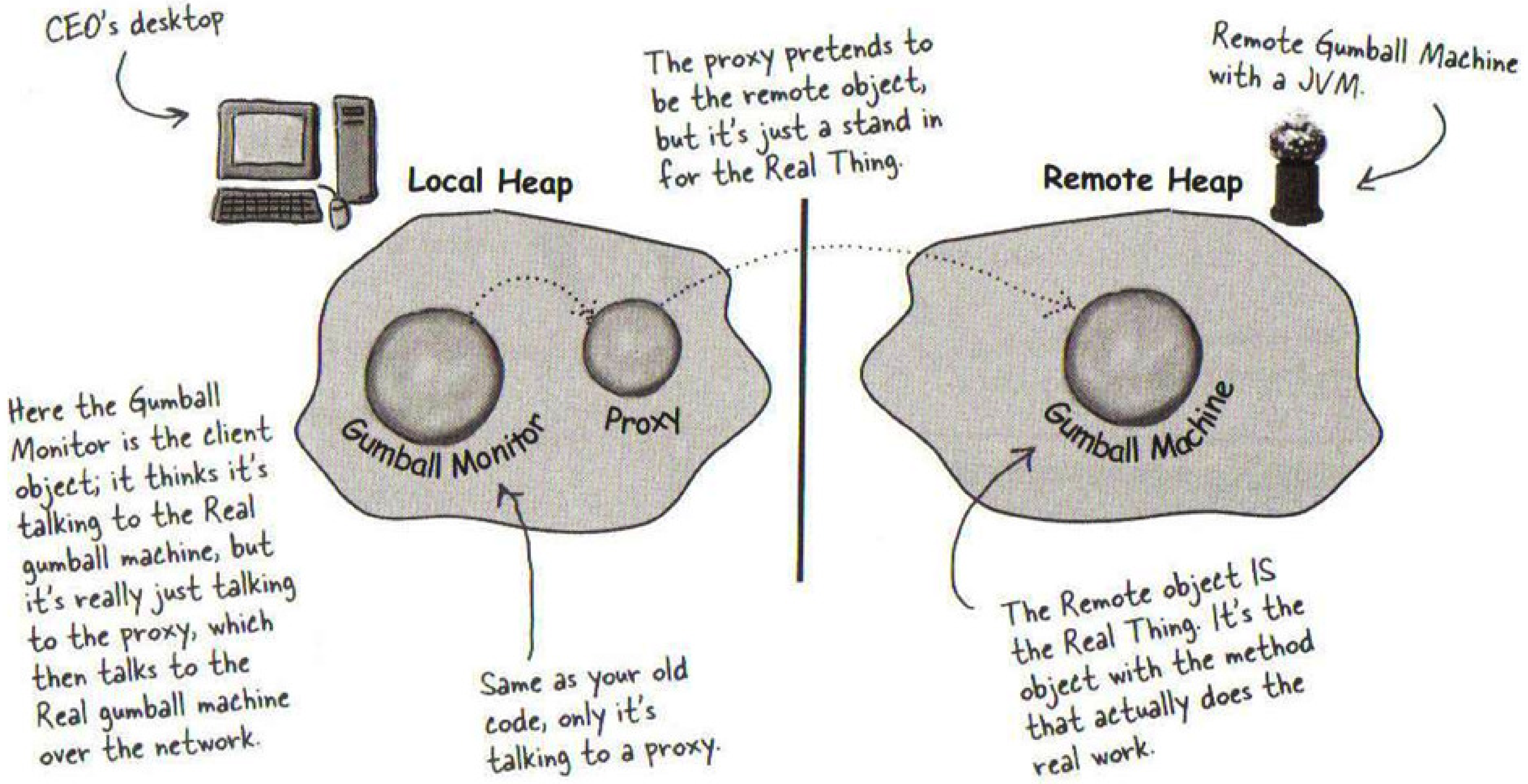


```
public class GumballMonitor {
    GumballMachine machine;

    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.

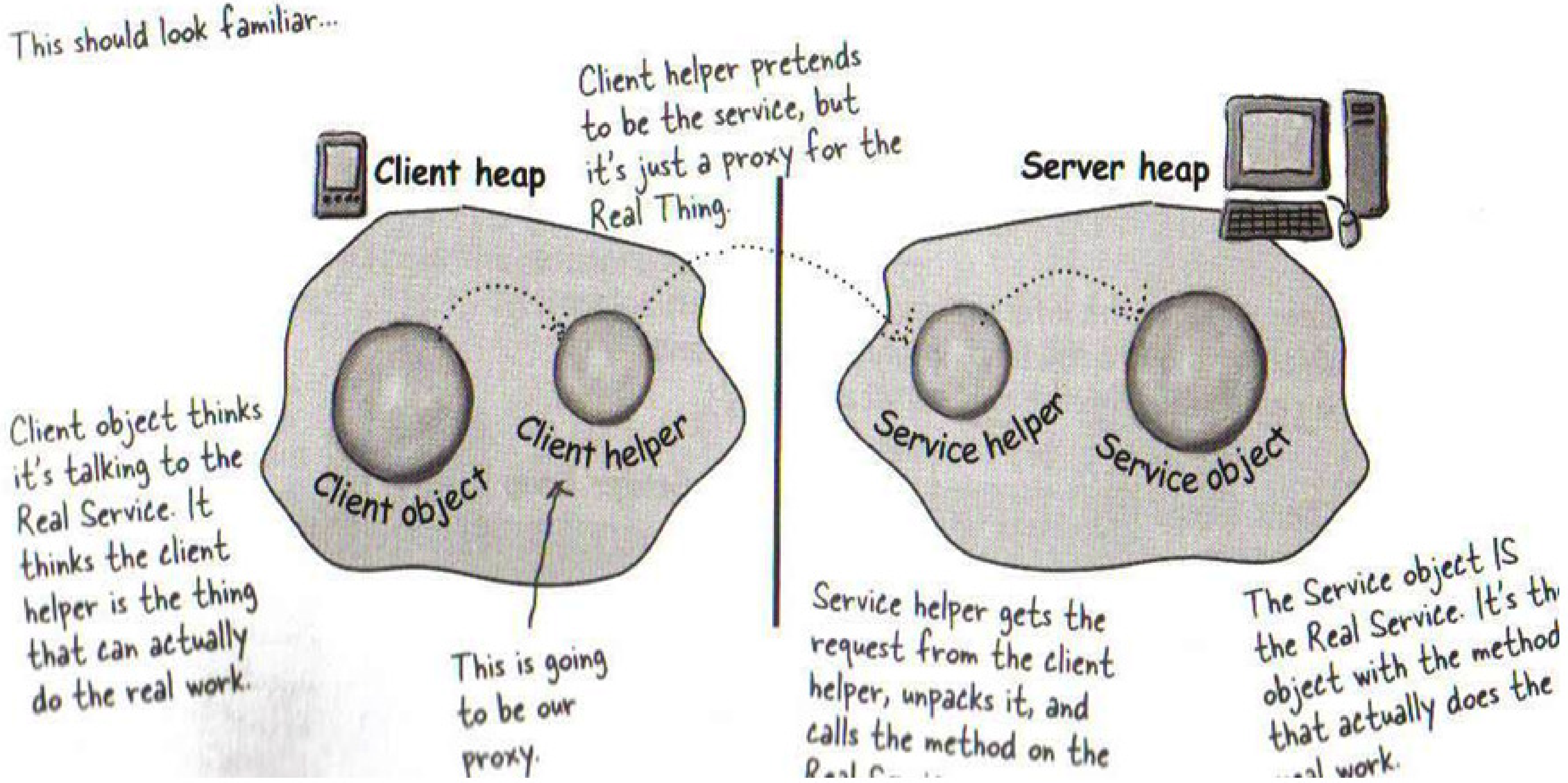
Our report method just prints a report with location, inventory and the machine's state.

Role of the remote Proxy



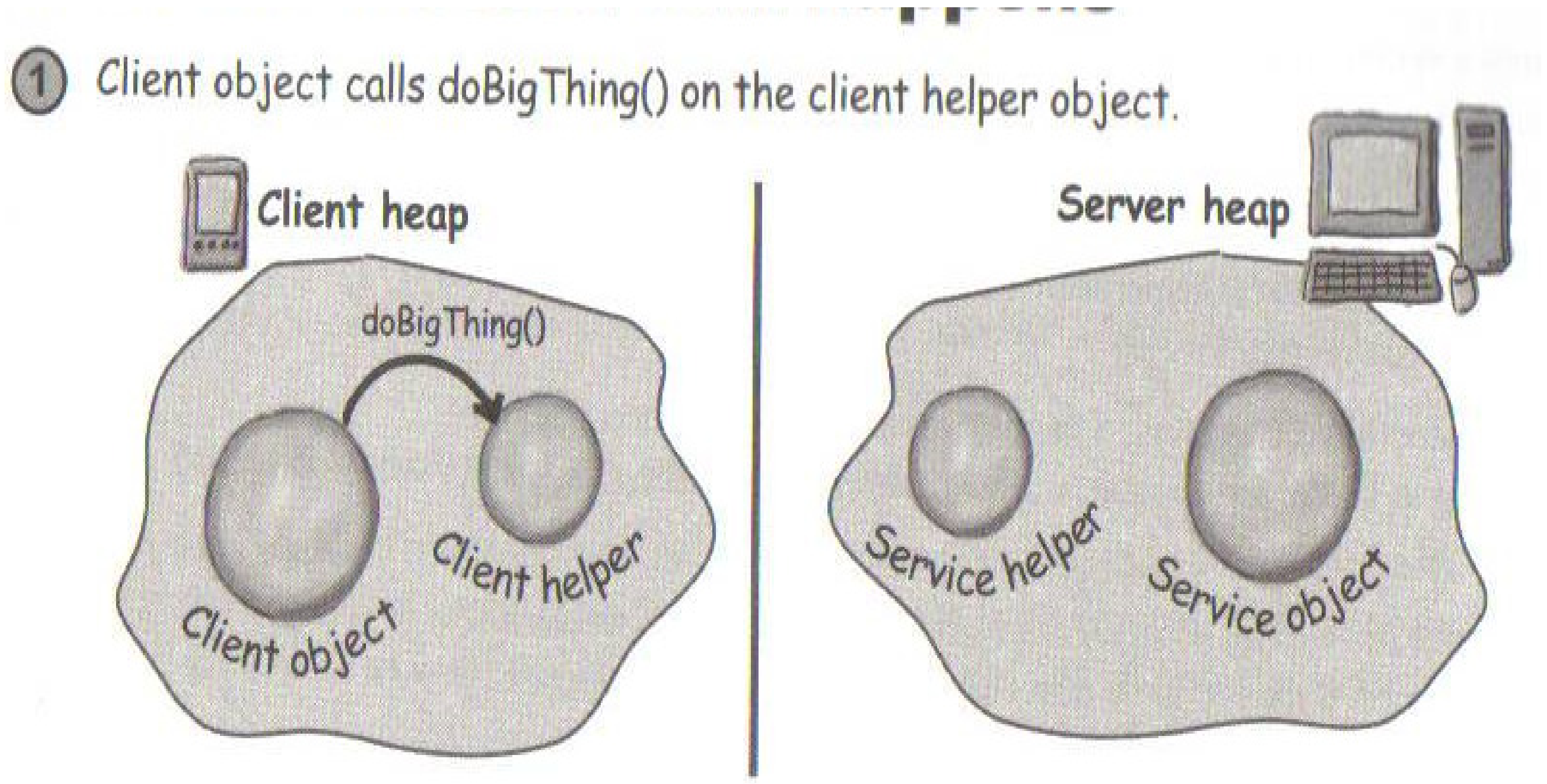
CEO's desktop
Local Heap
The proxy pretends to be the remote object, but it's just a stand in for the Real Thing.
Remote Gumball Machine with a JVM.
Remote Heap
Gumball Monitor
Proxy
Gumball Machine
Here the Gumball Monitor is the client object; it thinks it's talking to the Real gumball machine, but 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.

Remote Methods



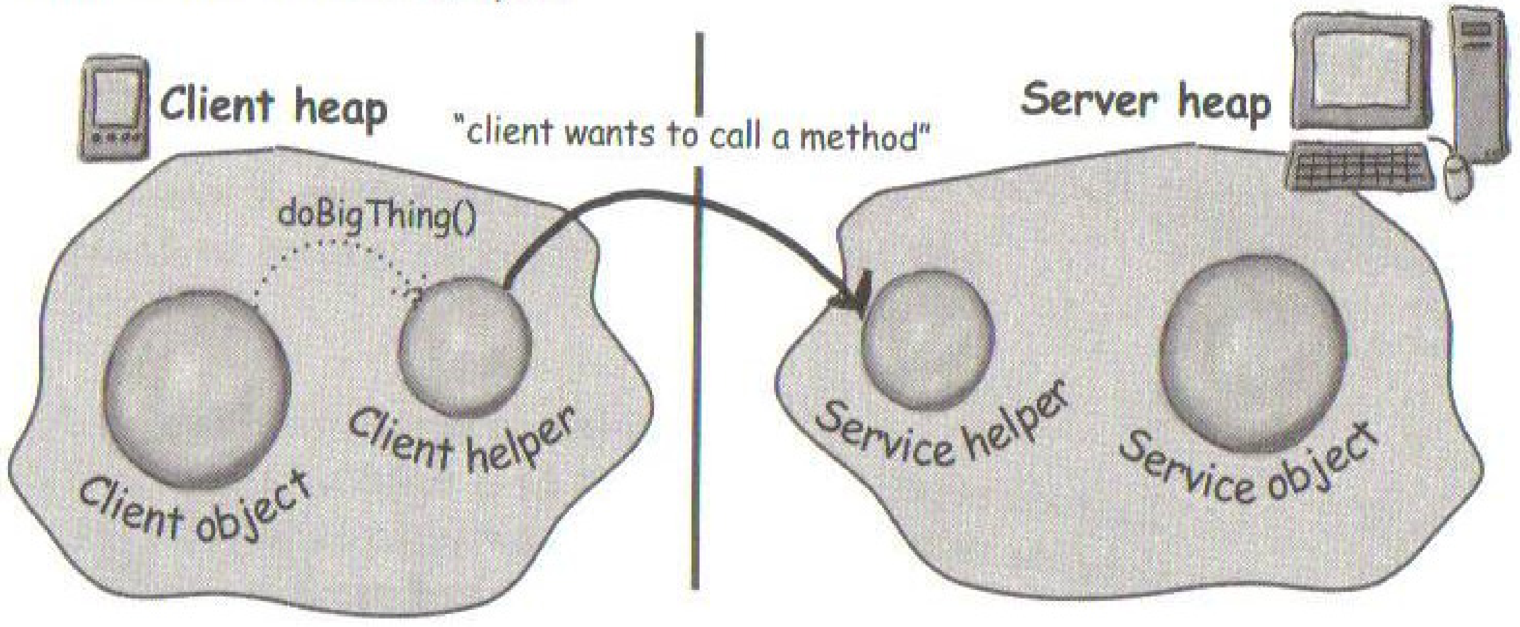
This should look familiar...
Client heap
Client helper pretends to be the service, but it's just a proxy for the Real Thing.
Server heap
Client object
Client helper
Service helper
Service object
Client object thinks it's talking to the Real Service. It thinks the client helper is the thing that can actually do the real work.
This is going to be our proxy.
Service helper gets the request from the client helper, unpacks it, and calls the method on the
The Service object IS the Real Service. It's th object with the method that actually does the
work.

How the method call happens
Client calls method



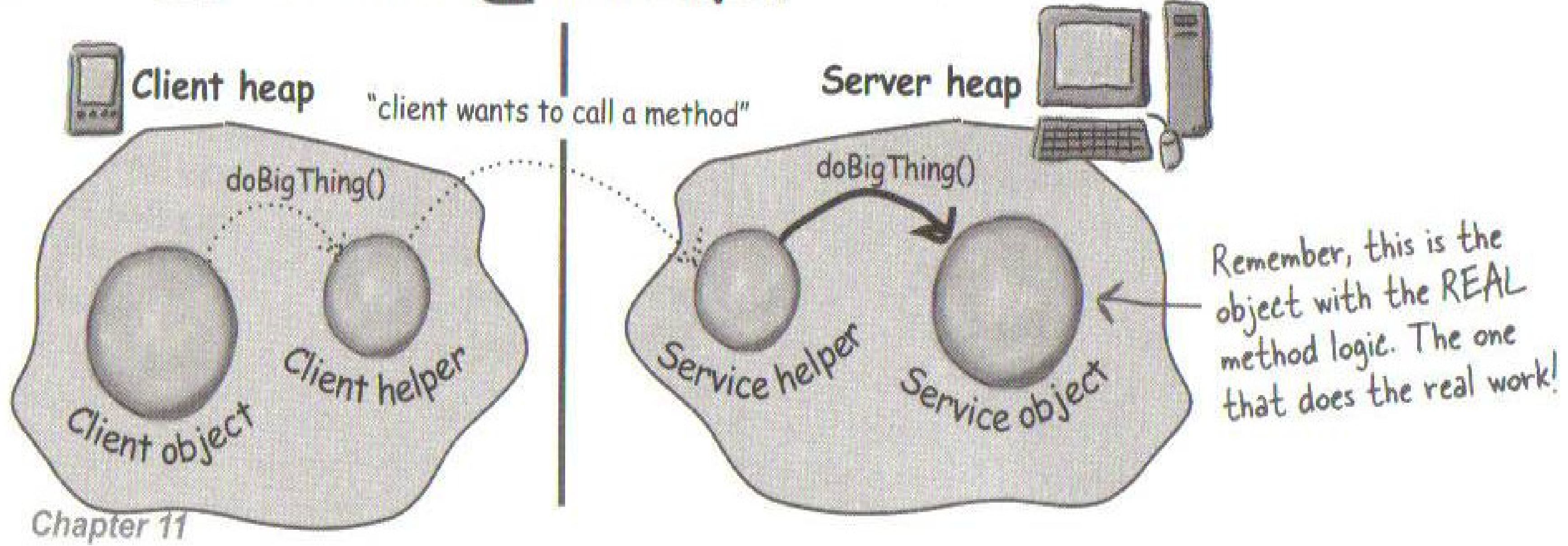
Client Helper forwards to service helper

(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

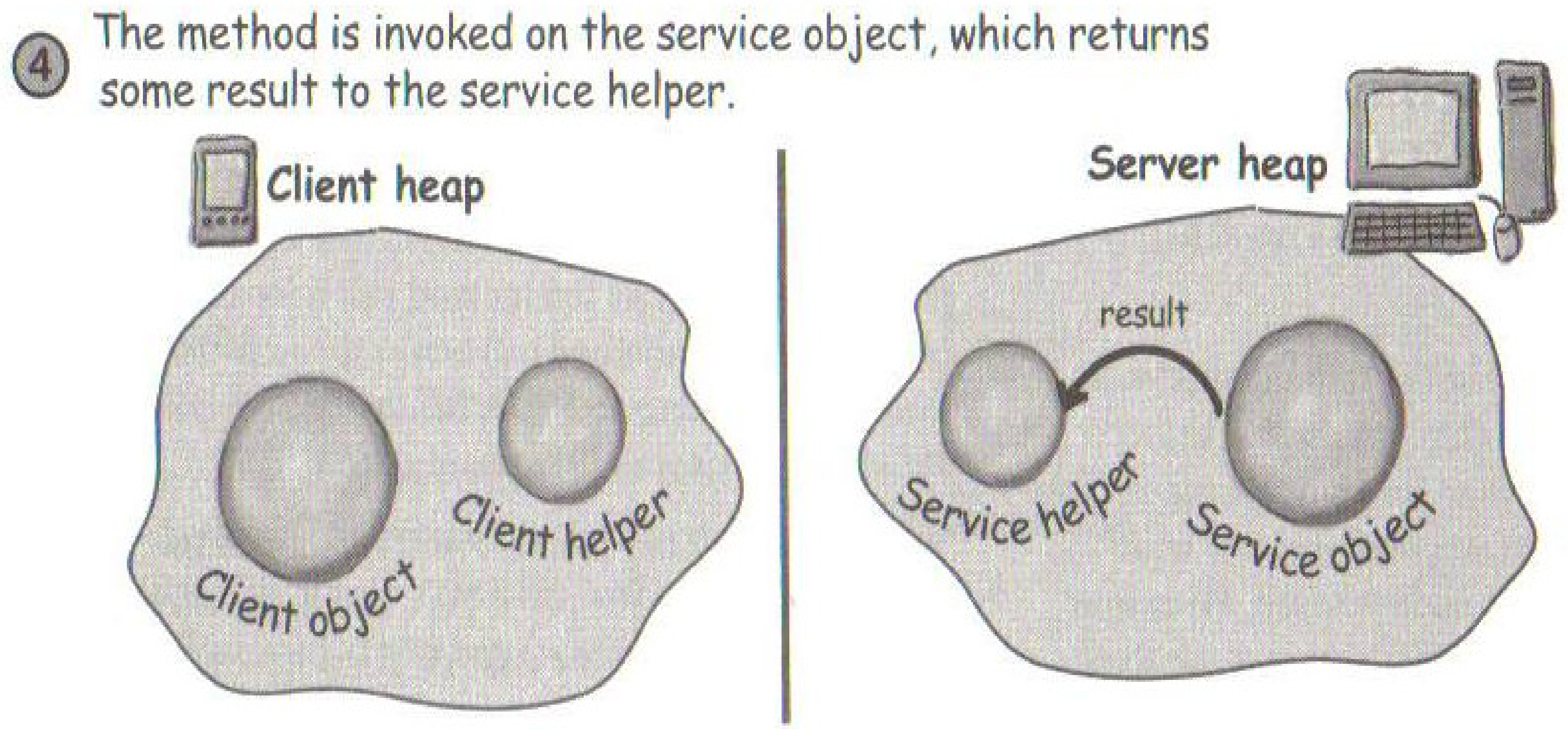
3 Service helper unpacks the information from the client helper, finds out which method to call (and on which object) and invokes the real method on the real service object.



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!

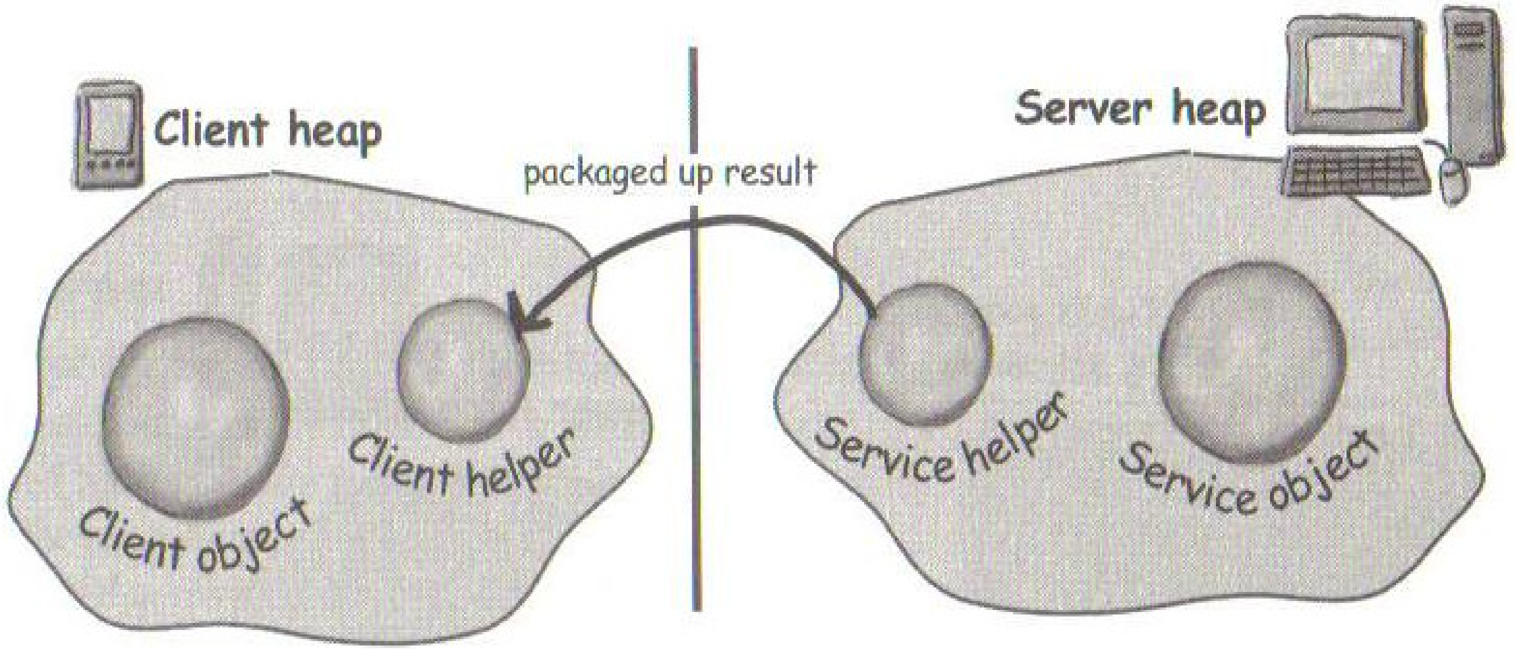
438 Chapter 11

Real object returns result

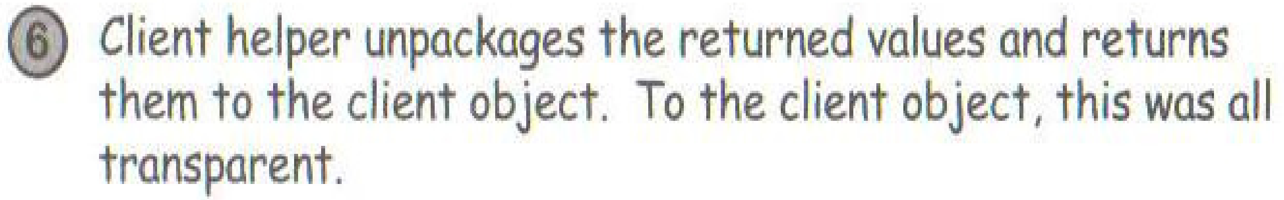

4
The method is invoked on the service object, which returns some result to the service helper.
Client heap
Client object
Client helper
Server heap
result
Service helper
Service object

Service helper forwards result to client helper

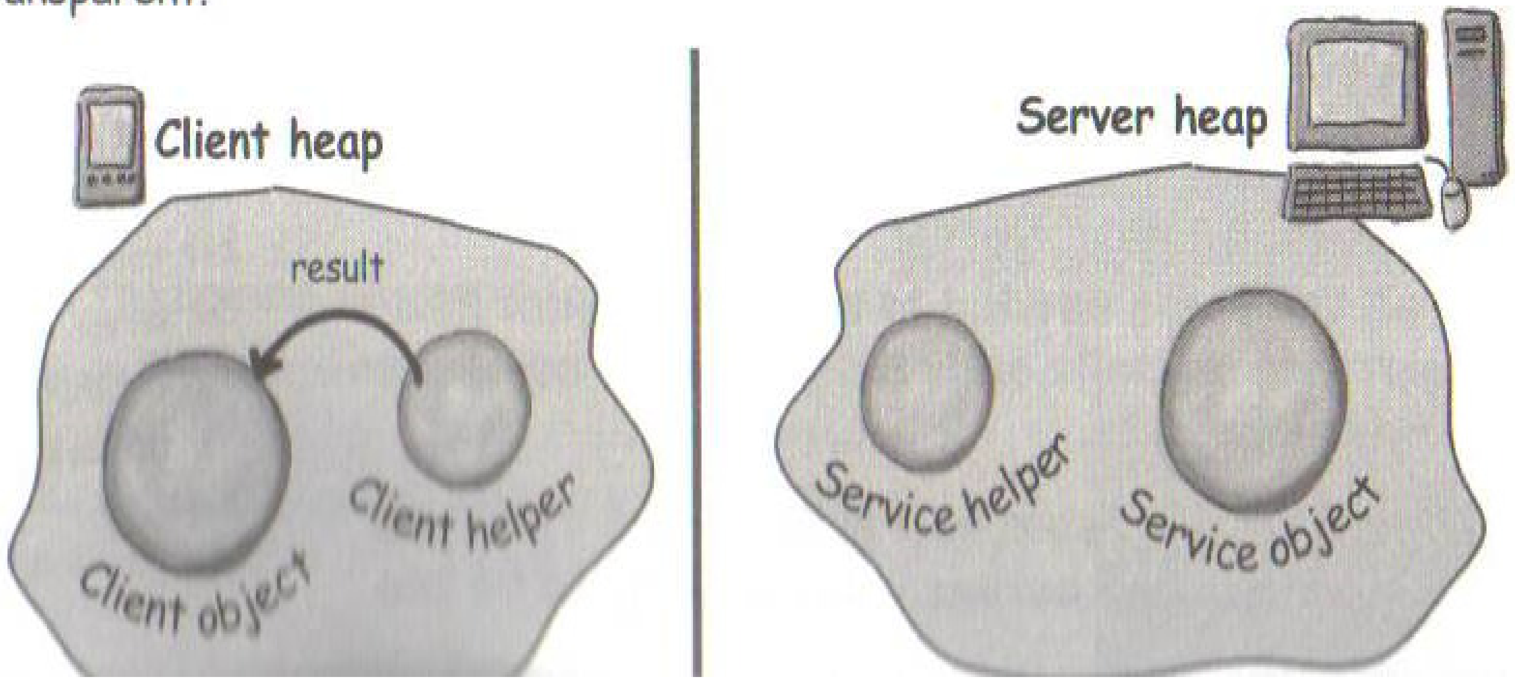
(5) Service helper packages up information returned from the call and ships it back over the network to the client helper.



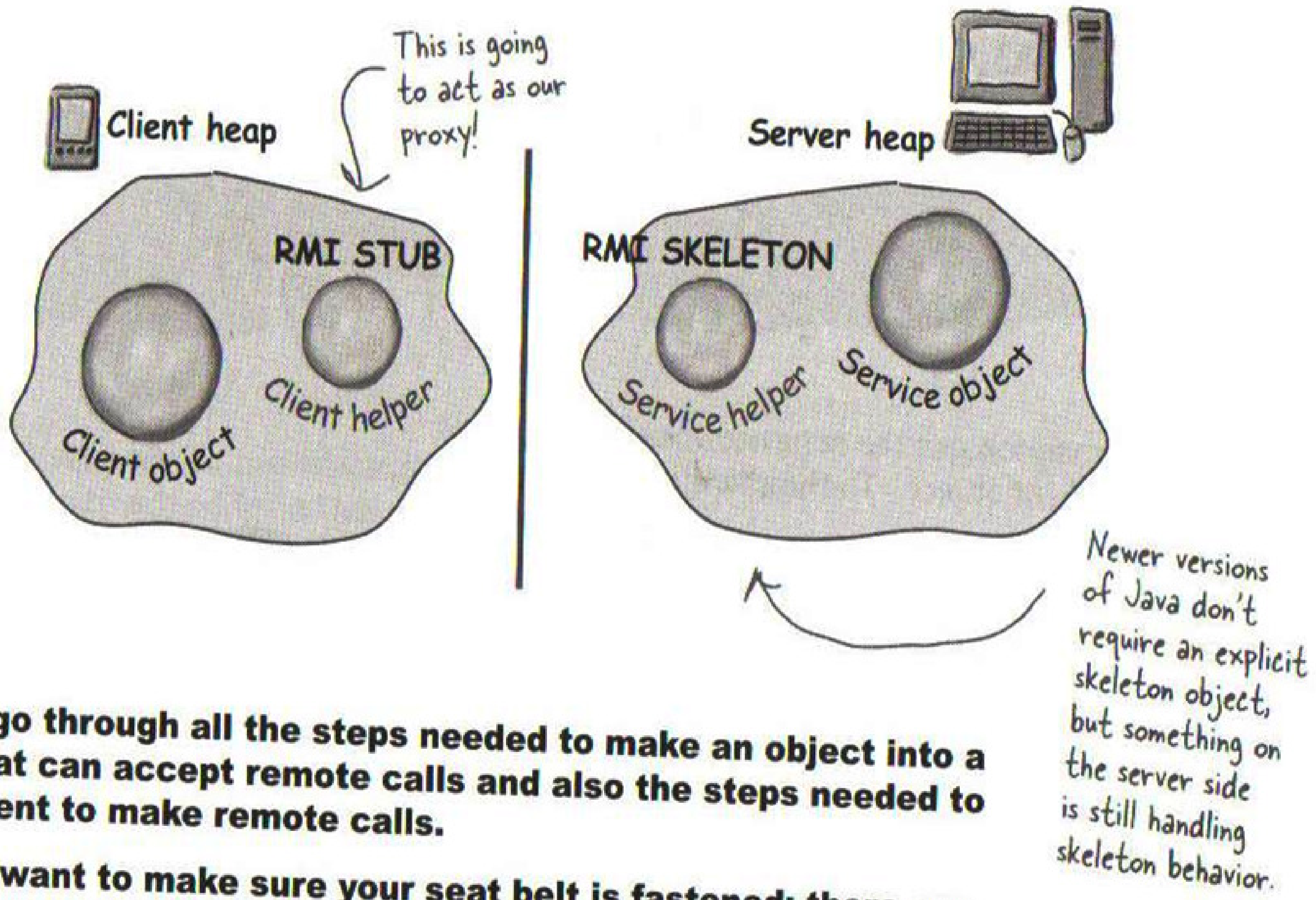
Client helper returns result to client



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



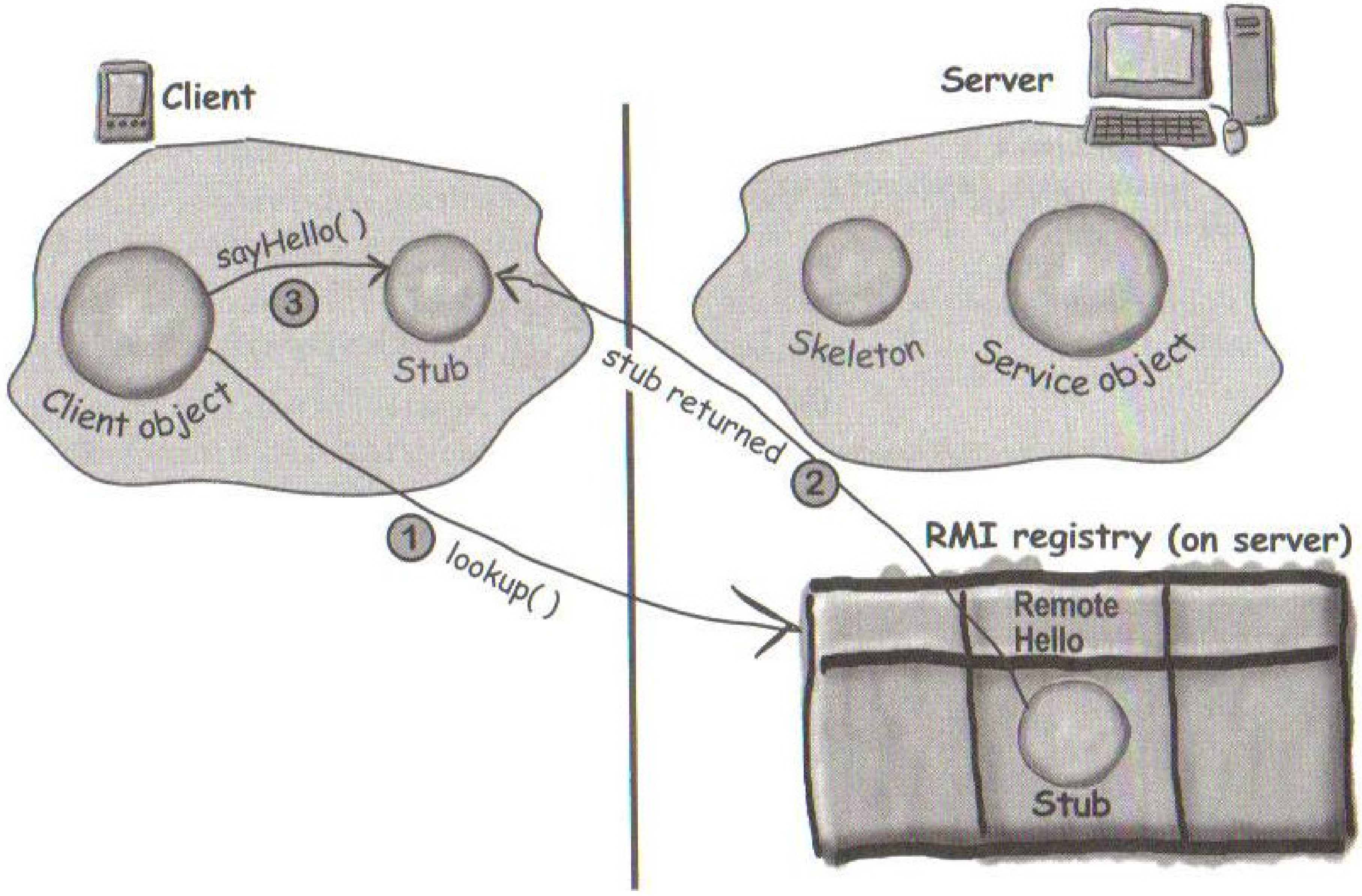
RMI Nomenclature: in RMI, the client helper is a 'stub' and the service helper is a 'skeleton'.


This is going to act as our proxy!
Client heap
RMI STUB
Client object
Client helper
Server heap
RMI SKELETON
Service helper
Service object
Newer versions of Java don't require an explicit skeleton object, but something on the server side is still handling skeleton behavior.

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...

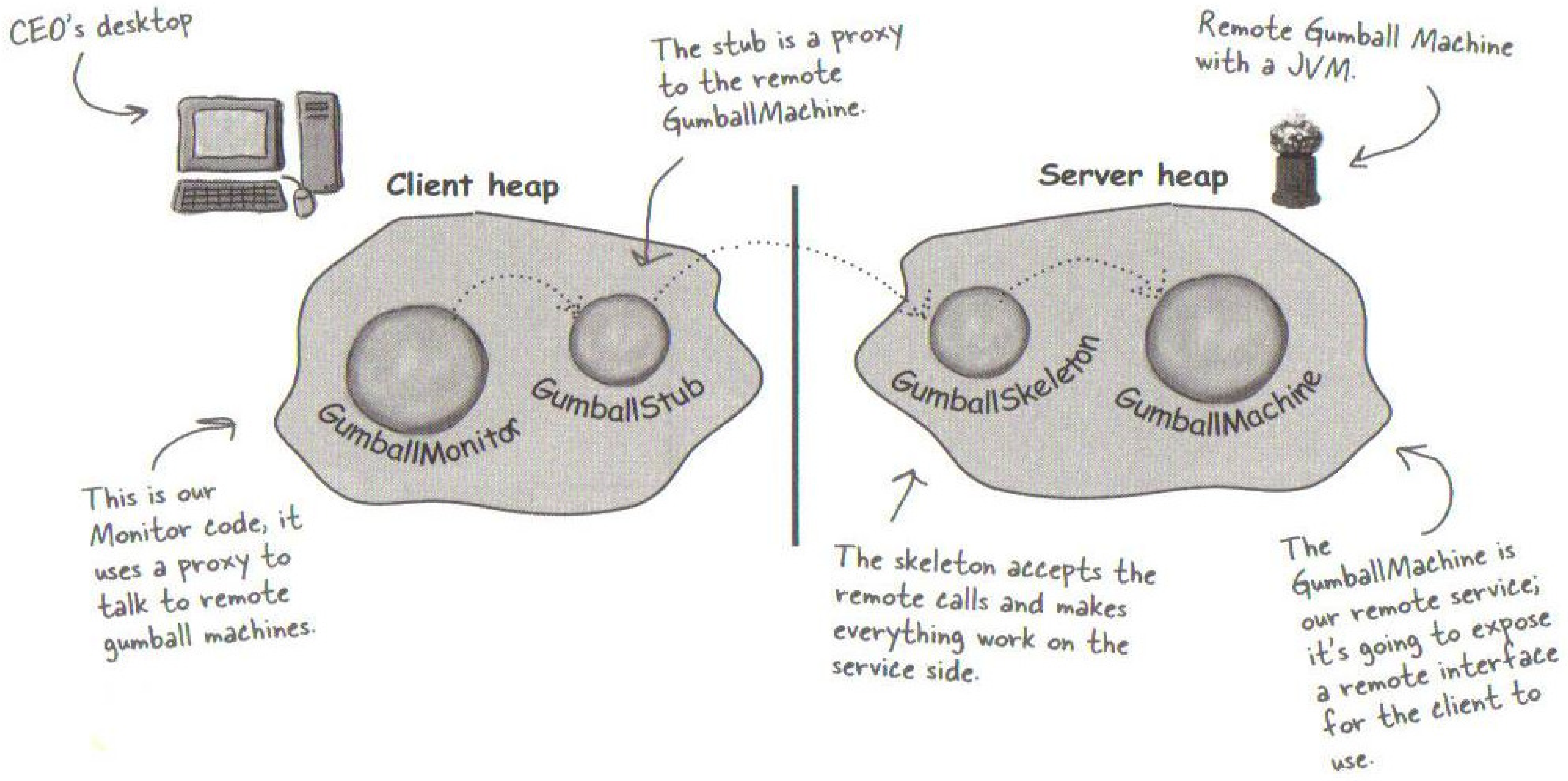
1. **Client does a lookup on the RMI registry**

```
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

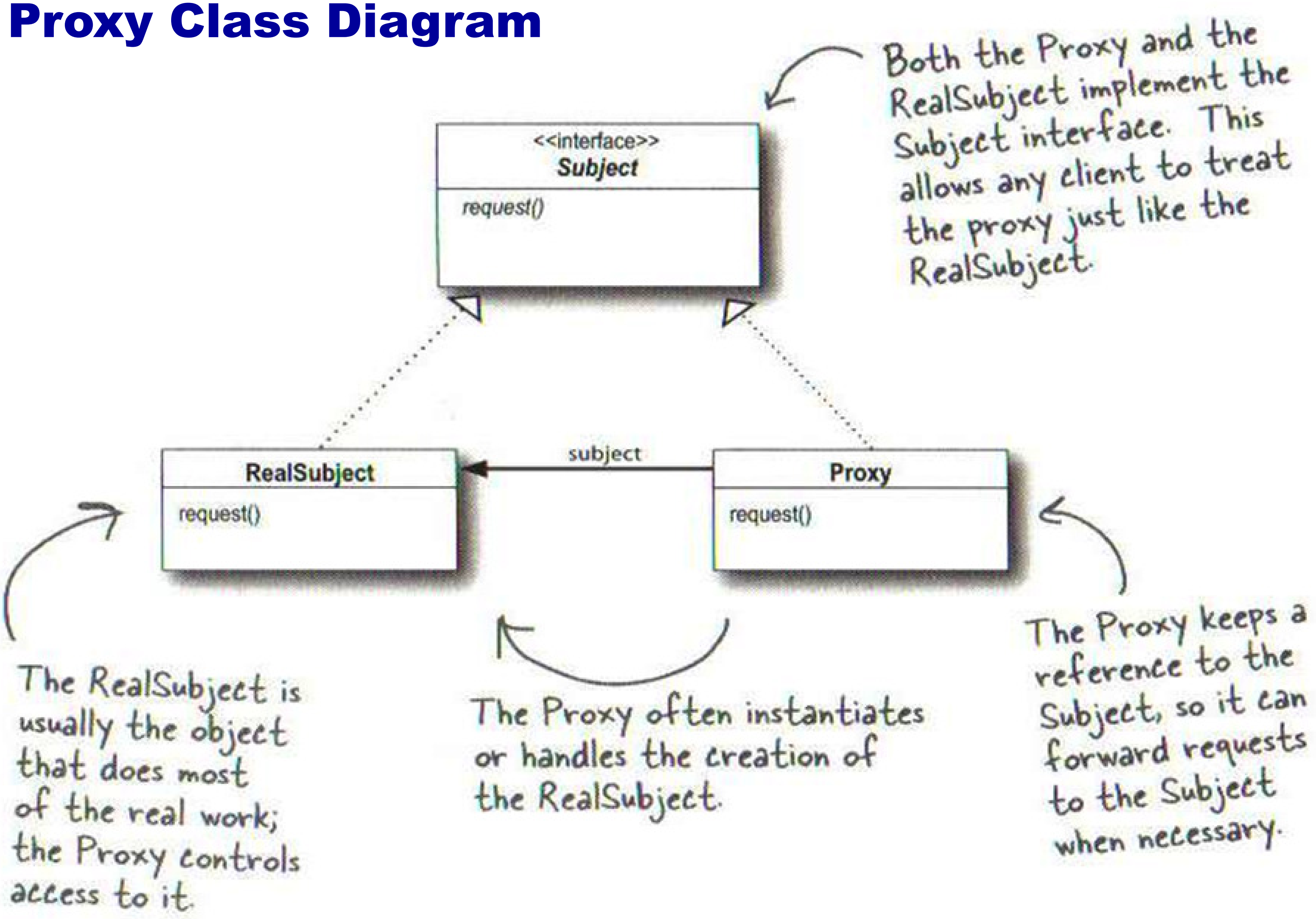

CEO's desktop
The stub is a proxy to the remote GumballMachine.
Remote Gumball Machine with a JVM.
Client heap
Server heap
GumballMonitor
GumballStub
GumballSkeleton
GumballMachine
This is our Monitor code, it uses a proxy to talk to remote gumball machines.
The skeleton accepts the remote calls and 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.

Proxy Pattern defined

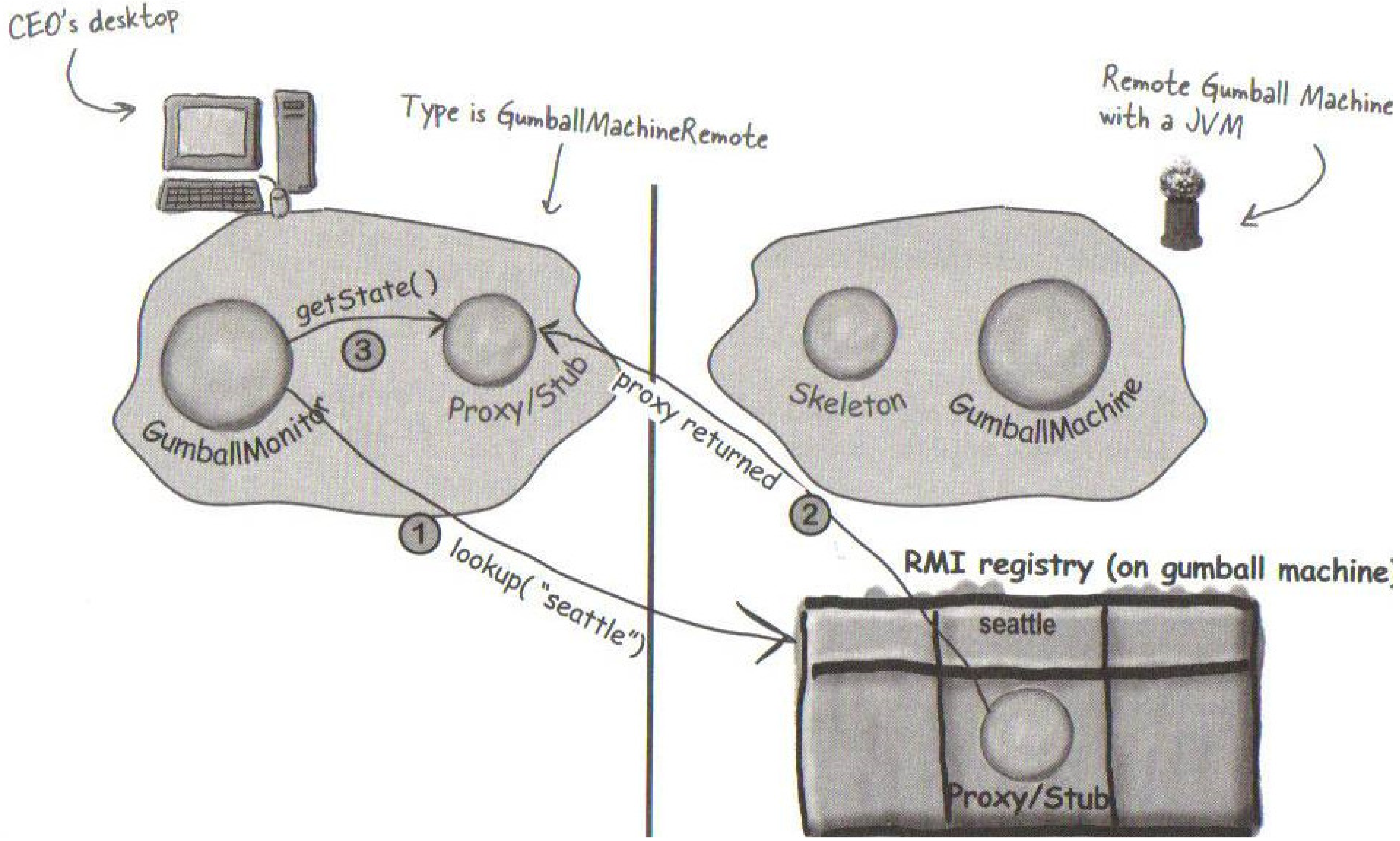
The Proxy Pattern provides a surrogate or placeholder for another object to control access to it.

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

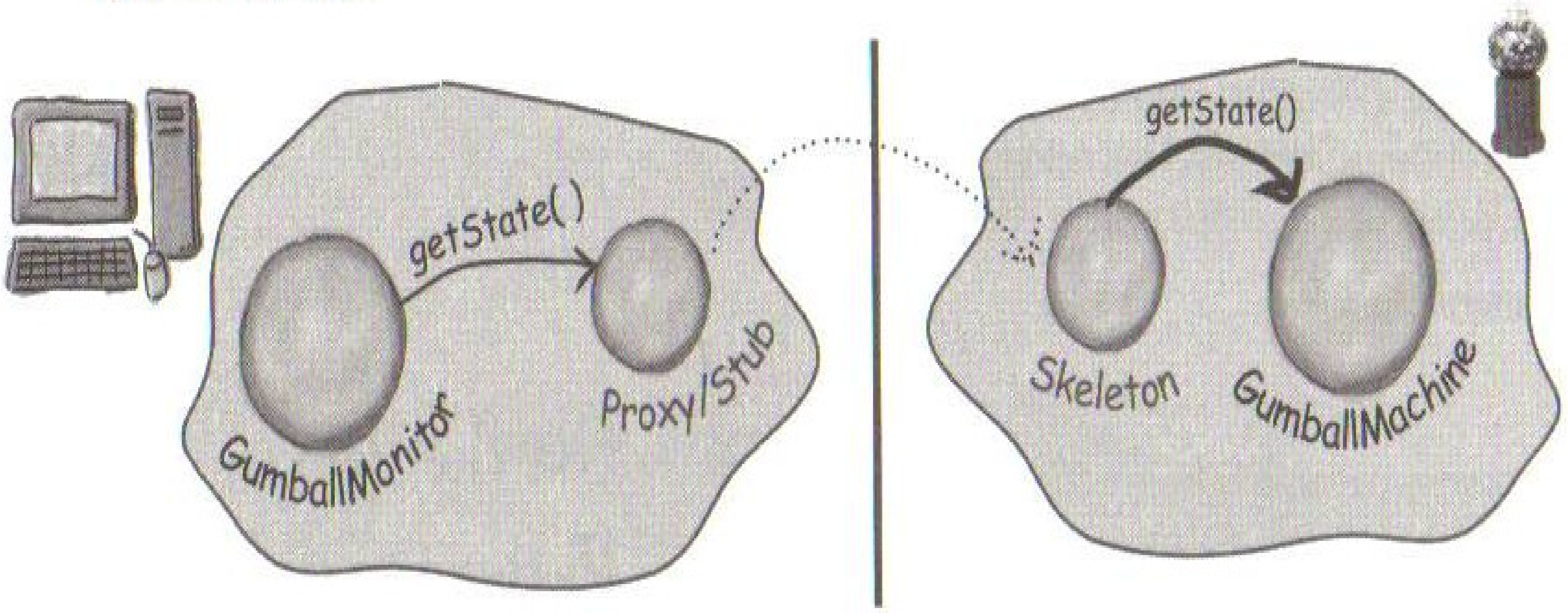

Both the Proxy and the RealSubject implement the Subject interface. This allows any client to treat the proxy just like the RealSubject.
<<interface>>
Subject
request()
RealSubject
request()
subject
Proxy
request()
The RealSubject is usually the object that does most of the real work; the Proxy controls 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()).

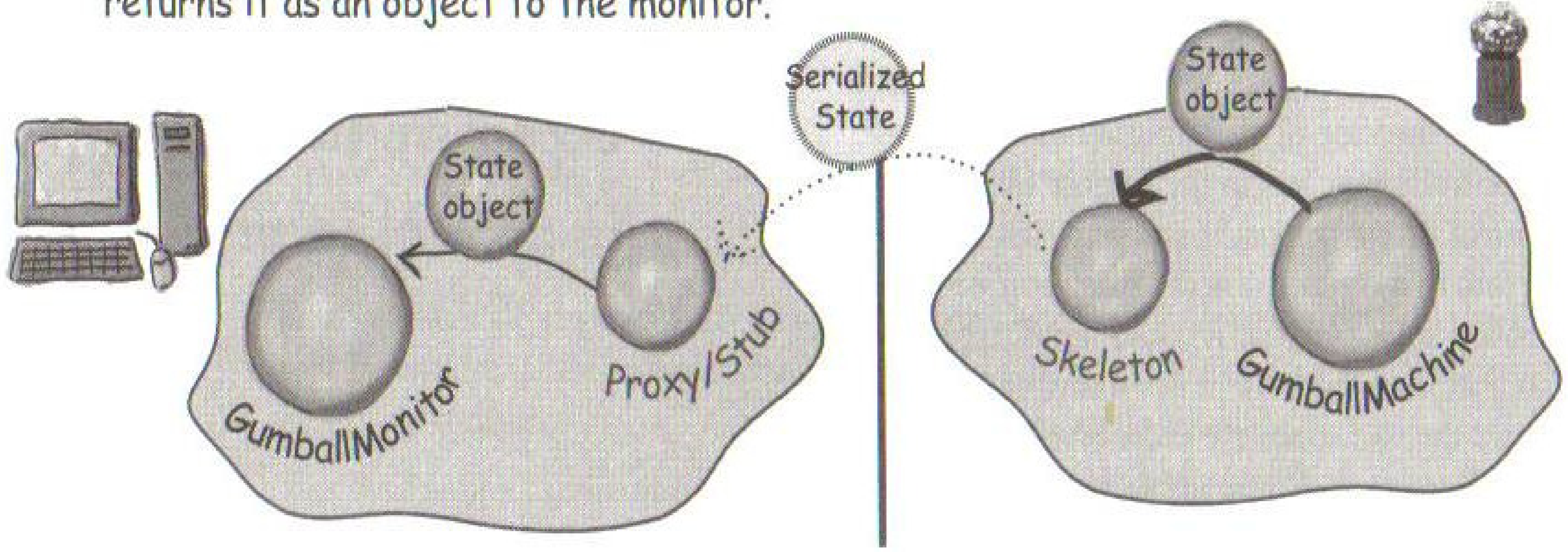


Making the call

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.

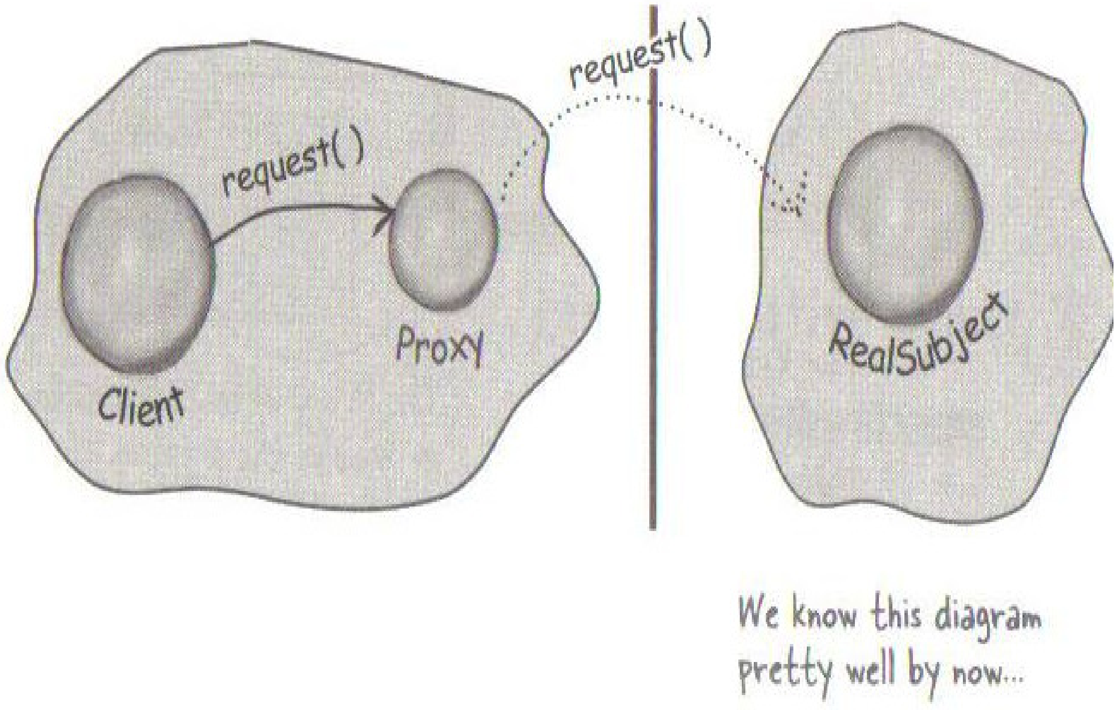
Likewise, the GumballMachine implements another interface and may throw a remote exception in its constructor, but other than that, the code hasn't changed.

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

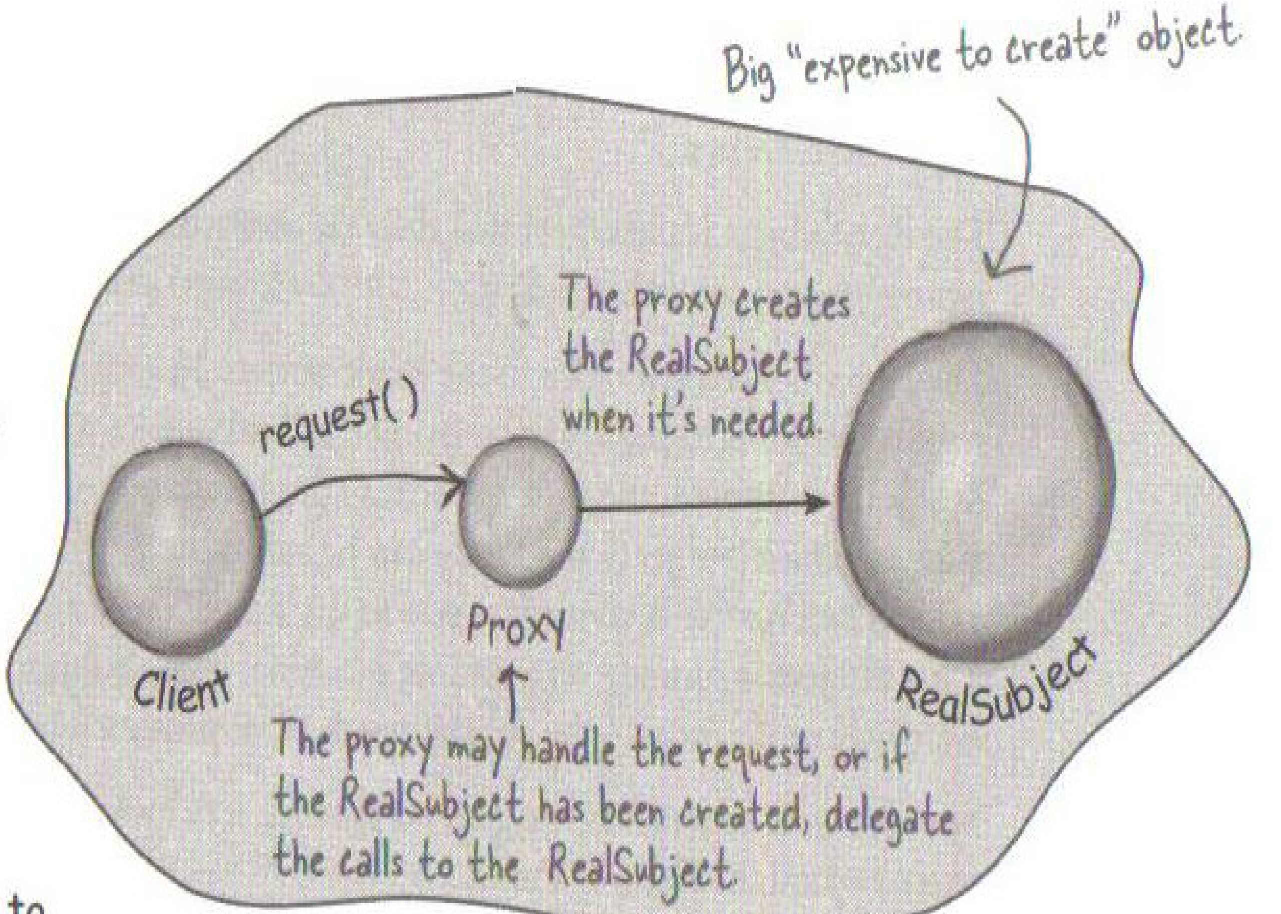
With Remote Proxy, the proxy acts as a local representative for an object that lives in a different JVM. A method call on the proxy results in the call being transferred over the wire, invoked remotely, and the result being returned back to the proxy and then to the Client.



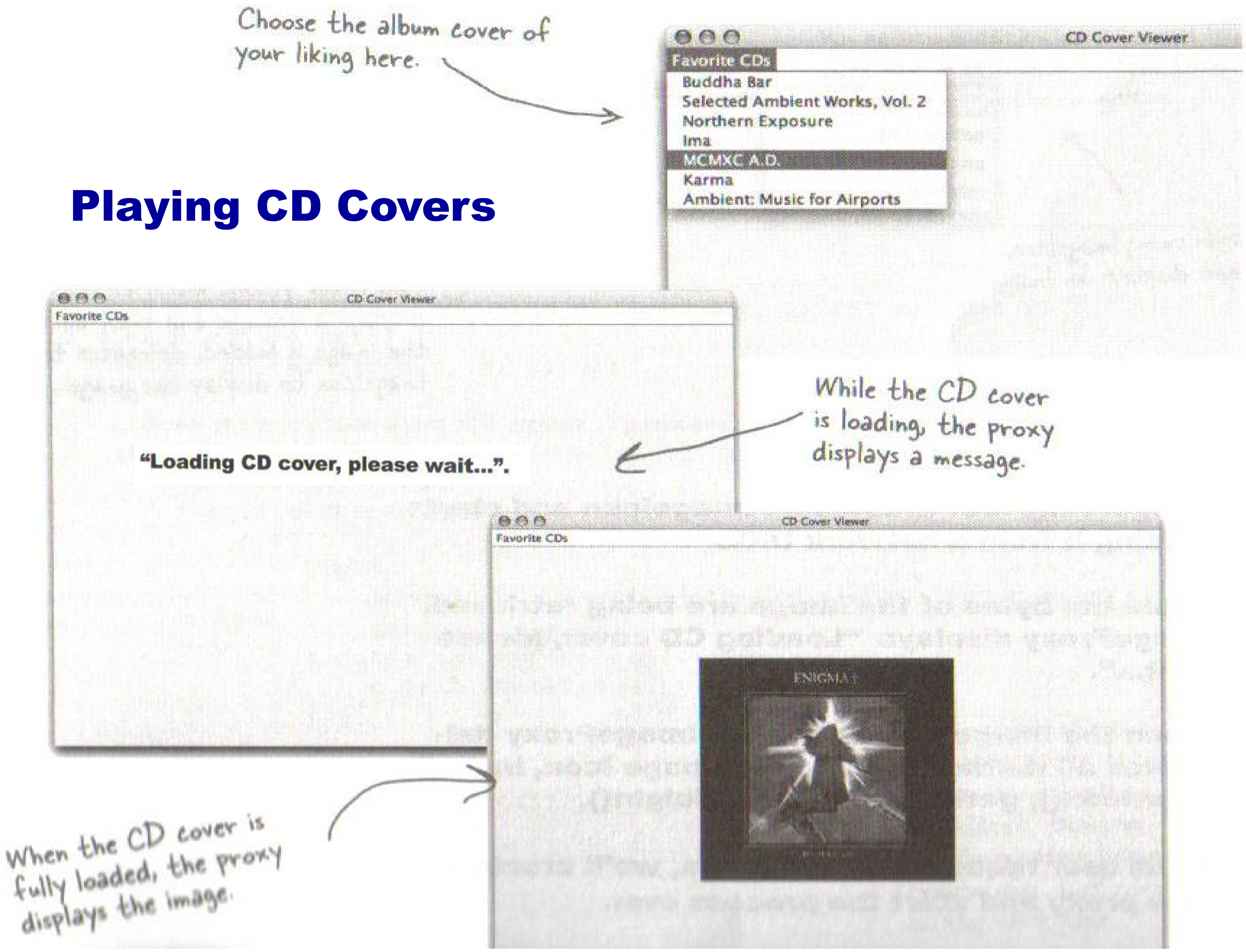
Virtual Proxy

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.

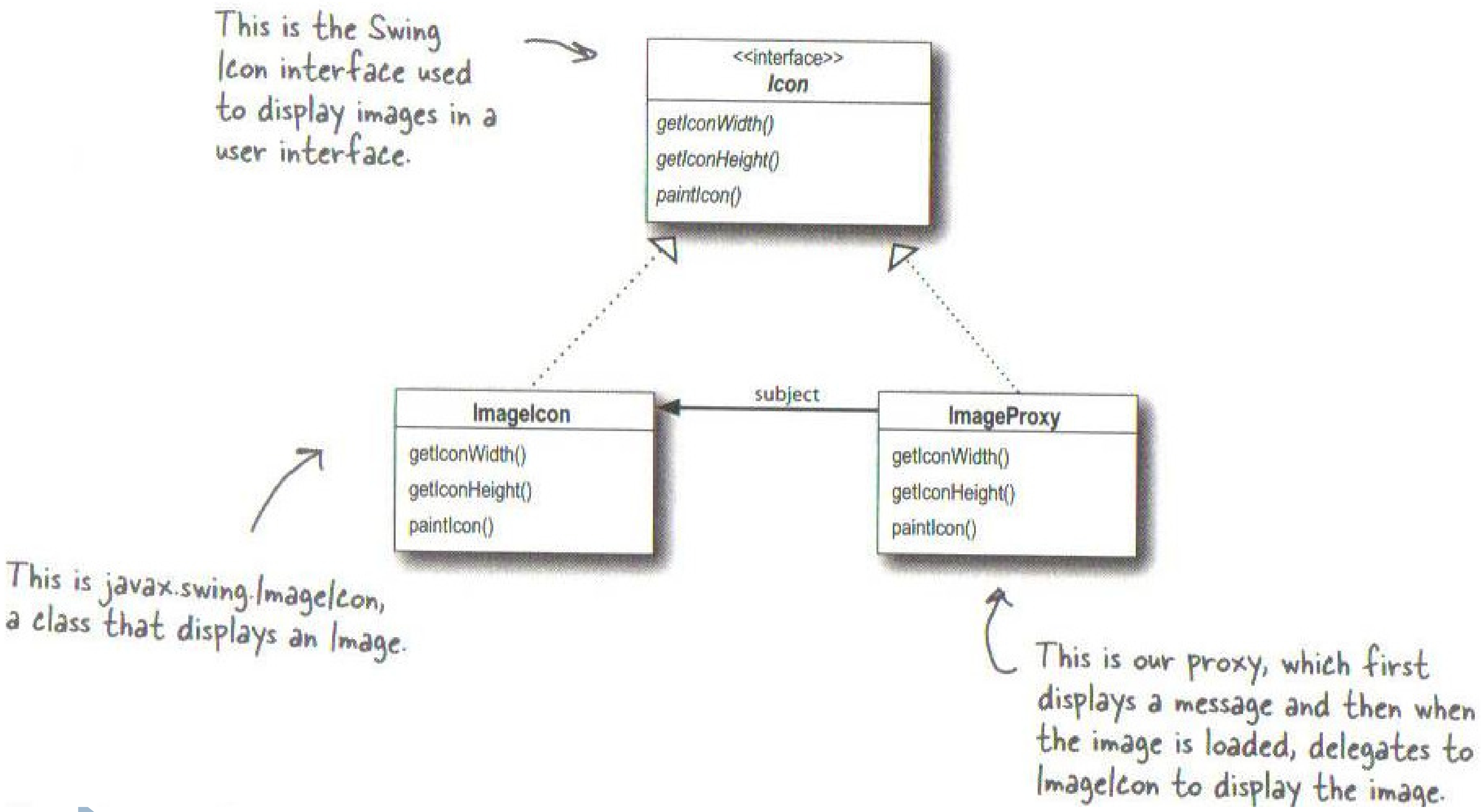

Big "expensive to create" object.
The proxy creates the RealSubject when it's needed.
request()
Client
Proxy
RealSubject
The proxy may handle the request, or if the RealSubject has been created, delegate the calls to the RealSubject.

Playing CD Covers


“Loading CD cover, please wait...”.

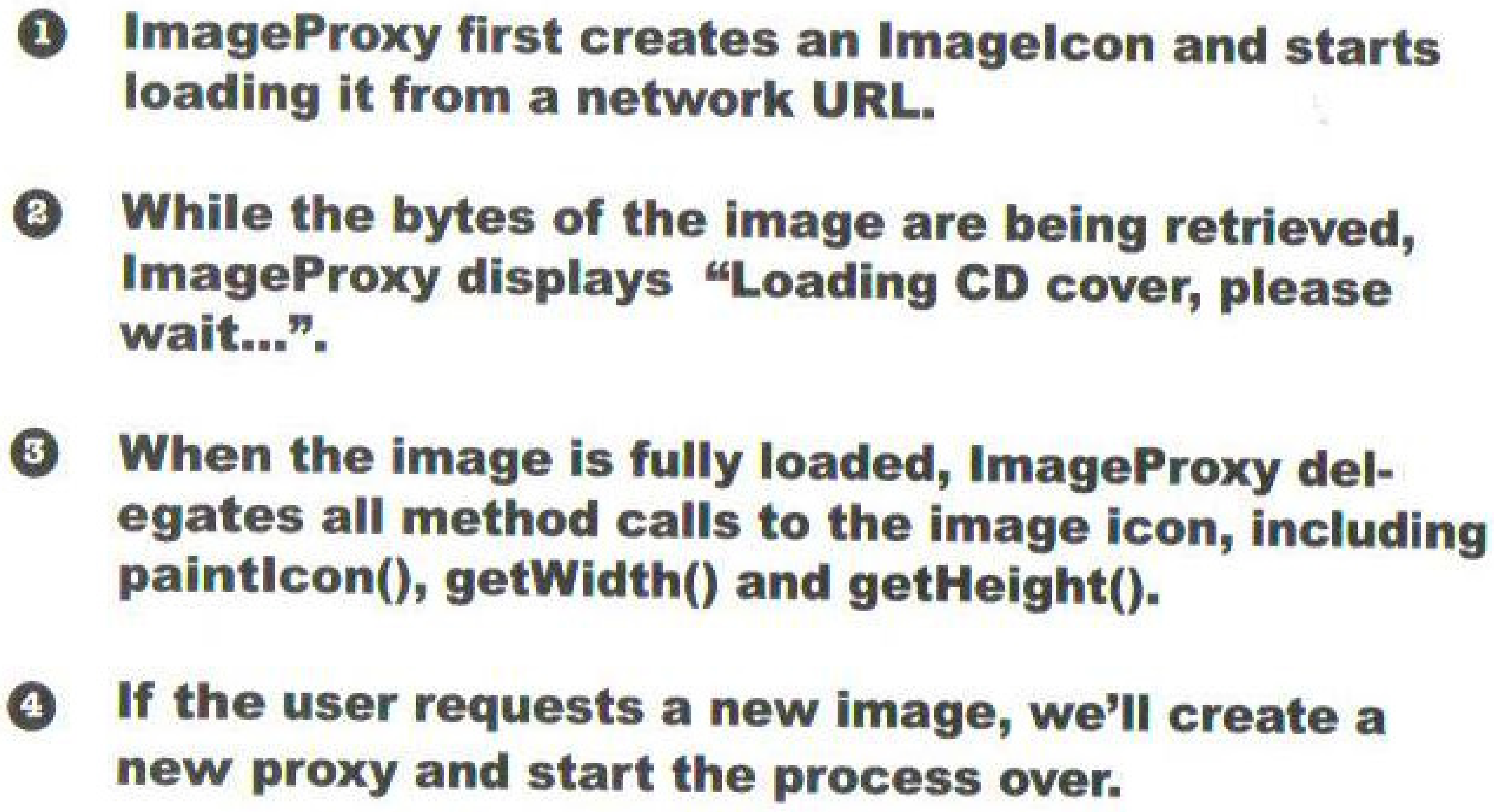
Choose the album cover of your liking here.
CD Cover Viewer
Favorite CDs
Buddha Bar
Selected Ambient Works, Vol. 2
Northern Exposure
Ima
MCMXC A.D.
Karma
Ambient: Music for Airports
While the CD cover is loading, the proxy displays a message.
ENIGMA
When the CD cover is fully loaded, the proxy displays 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()
subject
ImageIcon
getIconWidth()
getIconHeight()
paintIcon()
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



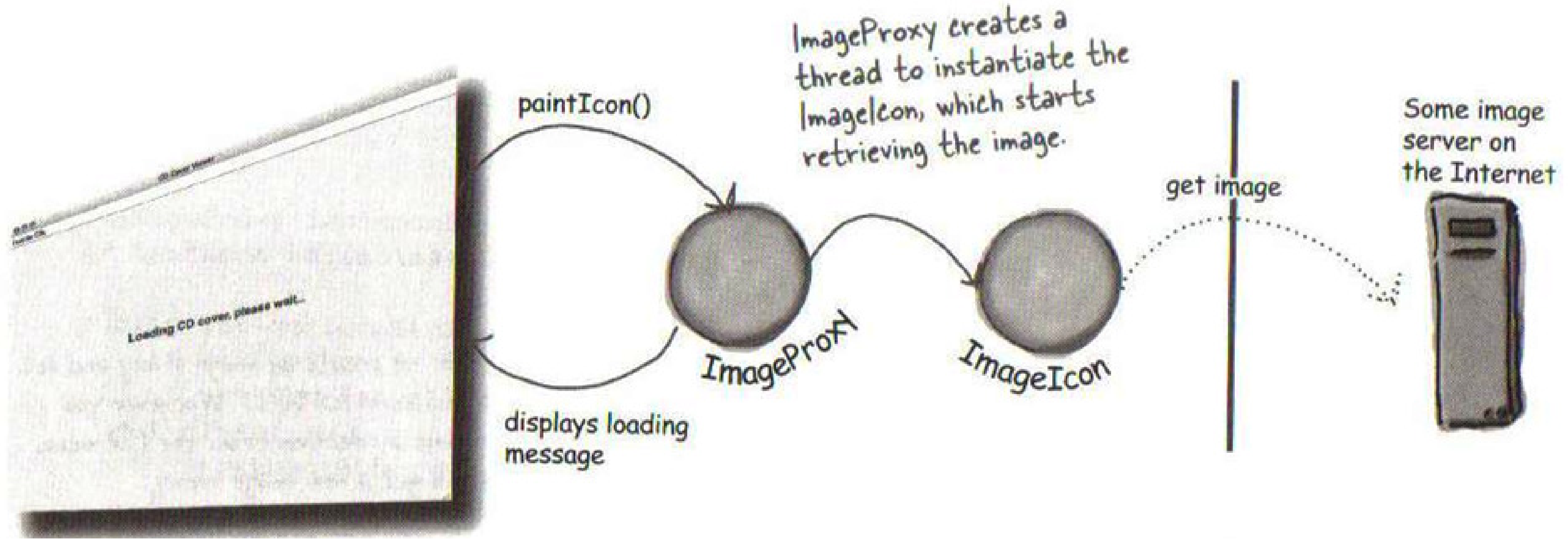
1. ImageProxy first creates an ImageIcon and starts loading it from a network URL.
2. While the bytes of the image are being retrieved, ImageProxy displays "Loading CD cover, please wait...".
3. When the image is fully loaded, ImageProxy delegates all method calls to the image icon, including paintIcon(),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

ImageProxy process

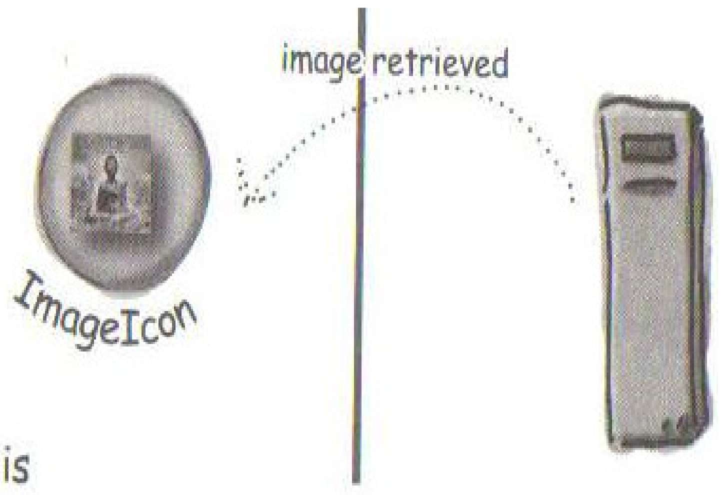
What did we do?

Behind the Scenes

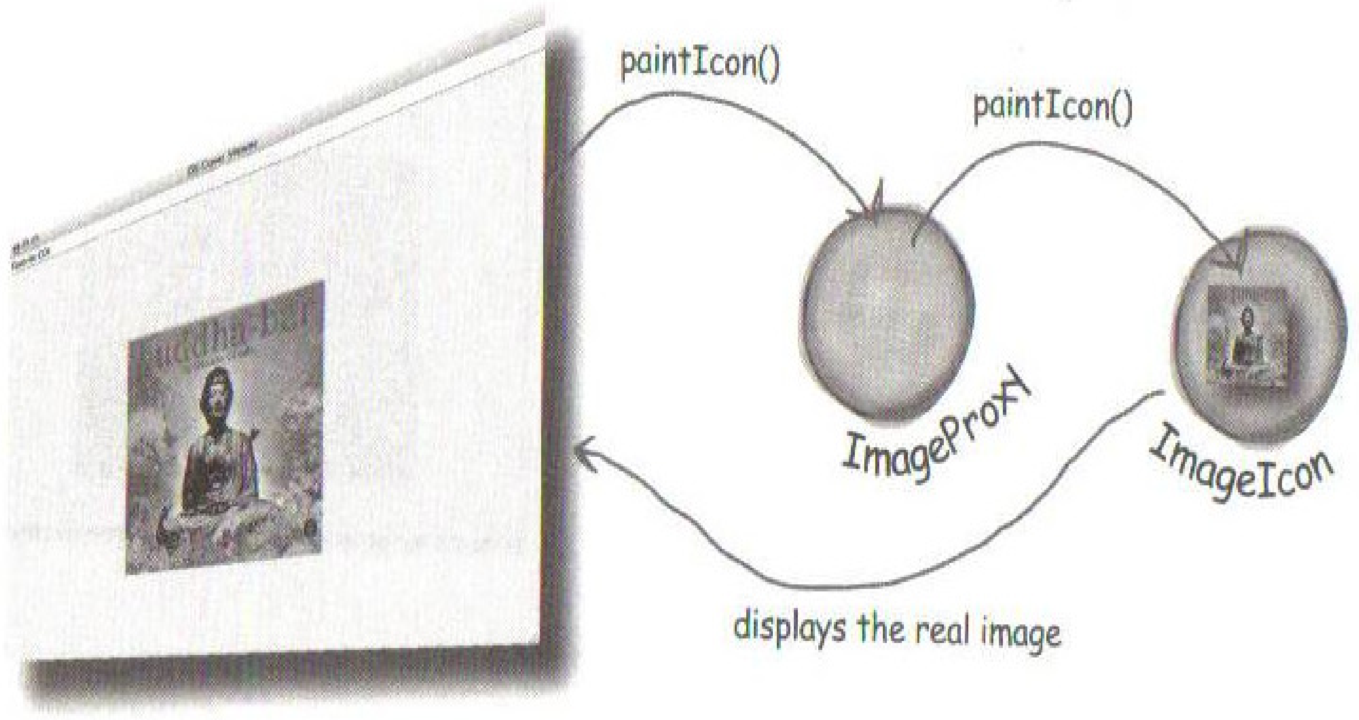
1. We created an ImageProxy for the display. The paintIcon() method is called and ImageProxy fires off a thread to retrieve the image and create the ImageIcon.


paintIcon()
ImageProxy creates a thread to instantiate the ImageIcon, which starts retrieving the image.
Some image server on the Internet
get image
Loading CD cover, please wait...
ImageProxy
ImageIcon
displays loading message

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



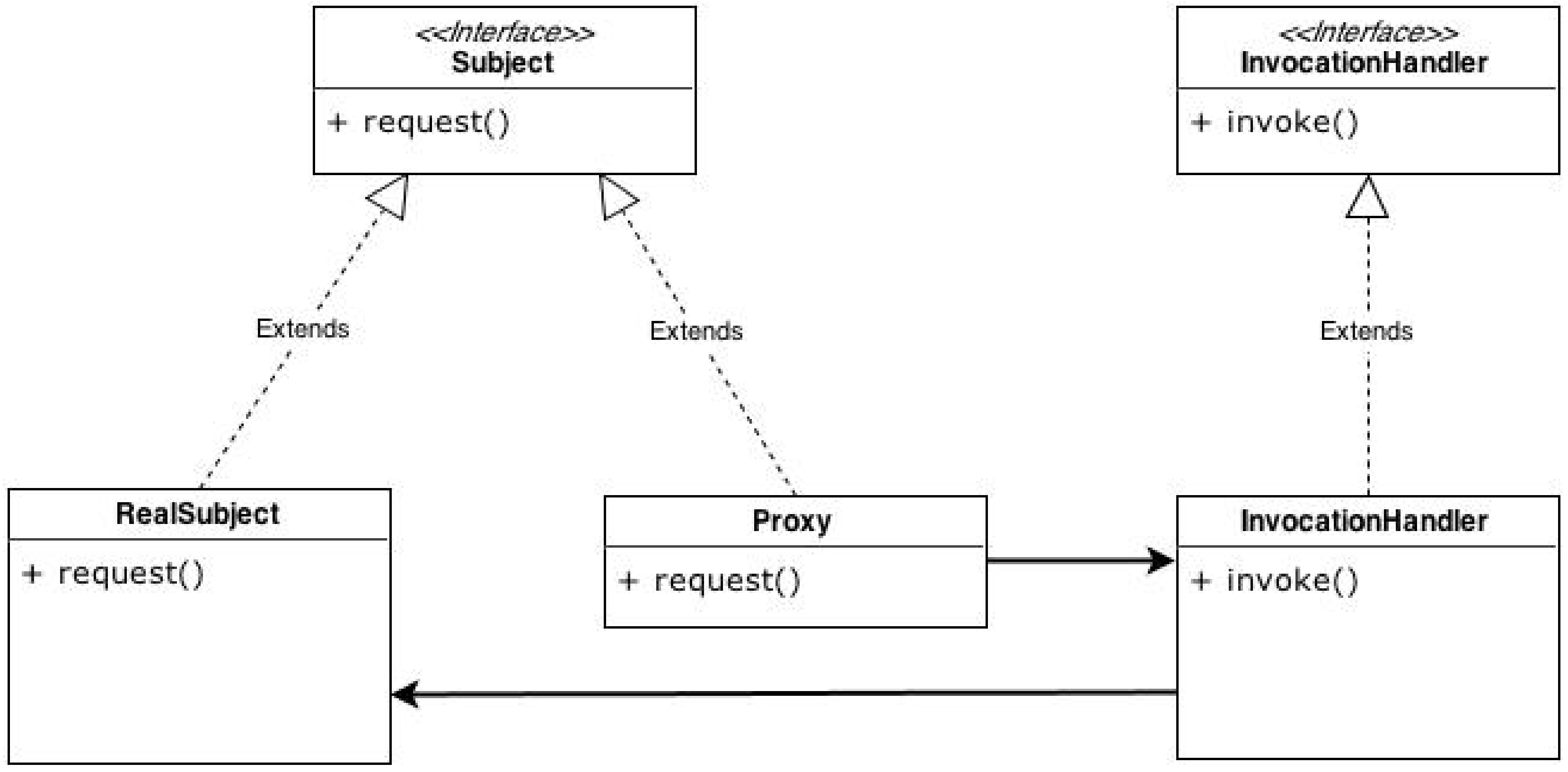
3. After the ImageIcon is created, the next time paintIcon() is called, the proxy delegates to the ImageIcon.



```
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 on the fly.

A proxy controls the access to the real object applying protection to the method calls in a transparent way. The client will invoke methods against the proxy thinking it is the real object.



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

Homework

- Consider your phone being the subject.
- Build a firewall proxy that filters sms and phone calls to block those of stalkers (e.g. your former boy/girlfriends).
  - The blacklist must be updateble

Appendix

Copy-on-write Proxy

Copy-On-Write Proxy Example

- Scenario: Suppose we have a large collection object, such as a hash table, which multiple clients want to access concurrently. One of the clients wants to perform a series of consecutive fetch operations while not letting any other client add or remove elements.

- Solution 1: Use the collection's lock object. Have the client implement a method which obtains the lock, performs its fetches and then releases the lock.
- For example:
  - public void doFetches(Hashtable ht) { synchronized(ht) {
    - // Do fetches using ht reference. } }
- But this method may require holding the collection object's lock for a long period of time, thus preventing other threads from accessing the collection

Copy-On-Write Proxy Example (Continued)

- Solution 2: Have the client clone the collection prior to performing its fetch operations. It is assumed that the collection object is cloneable and provides a clone method that performs a sufficiently deep copy
  - For example, java.util.Hashtable provides a clone method that makes a copy of the hash table itself, but not the key and value objects

```
void doFetches(Hashtable ht) {
    Hashtable newht = (Hashtable) ht.clone();
    // Do fetches using newht reference.   } l
```

- □ The collection lock is held while the clone is being created. But once the clone is created, the fetch operations are done on the cloned copy, without holding the original collection lock. l But if no other client modifies the collection while the fetch operations are being done, the expensive clone operation was a wasted effort!

Copy-On-Write Proxy Example (Continued)

- Solution 3: It would be nice if we could actually clone the collection only when we need to, that is when some other client has modified the collection. For example, it would be great if the client that wants to do a series of fetches could invoke the clone() method, but no actual copy of the collection would be made until some other client modifies the collection. This is a copy-on-write cloning operation.
- We can implement this solution using proxies
- Here is an example implementation of such a proxy for a hash table written by Mark Grand from the book Patterns in Java.

Copy-On-Write Proxy Example (Continued)

- The proxy is the class LargeHashtable. When the proxy's clone() method is invoked, it returns a copy of the proxy and both proxies refer to the same hash table. When one of the proxies modifies the hash table, the hash table itself is cloned.

- The ReferenceCountedHashTable class is used to let the proxies know they are working with a shared hash table . This class keeps track of the number of proxies using the shared hash table.

Copy-On-Write Proxy Example (Continued)

```
// The proxy.
public class LargeHashtable extends Hashtable {
        // The ReferenceCountedHashTable that this is a proxy for.
        private ReferenceCountedHashTable theHashTable;
        // Constructor
        public LargeHashtable() {
            theHashTable = new ReferenceCountedHashTable();  }
        // Return the number of key-value pairs in this hashtable.
        public int size() {    return theHashTable.size();  }
```

Copy-On-Write Proxy Example (Continued)

```
// Return the value associated with the specified key.
public synchronized Object get(Object key) {
        return theHashTable.get(key);  }
// Add the given key-value pair to this Hashtable.
public synchronized Object put(Object key, Object value) {
        copyOnWrite();
        return theHashTable.put(key, value);  }
// Return a copy of this proxy that accesses the same Hashtable.
public synchronized Object clone() {
        Object copy = super.clone();
        theHashTable.addProxy();
        return copy;  }
```

Copy-On-Write Proxy Example (Continued)

```
// This method is called before modifying the underlying
// Hashtable.  If it is being shared then this method clones it.
private void copyOnWrite() {
 if (theHashTable.getProxyCount() > 1) {
          synchronized (theHashTable) {
                    theHashTable.removeProxy();
                    try {
                    theHashTable =  (ReferenceCountedHashTable)
                    theHashTable.clone();       }
                    catch (Throwable e) {
                    theHashTable.addProxy();       }     }   } }  …
```

Copy-On-Write Proxy Example (Continued)

```
// Private class to keep track of proxies sharing the hash table.
private class ReferenceCountedHashTable extends Hashtable {
        private int proxyCount = 1;
        // Constructor
        public ReferenceCountedHashTable() {     super();    }
        // Return a copy of this object with proxyCount set back to 1.
        public synchronized Object clone() {
          ReferenceCountedHashTable copy;
          copy = (ReferenceCountedHashTable)super.clone();
          copy.proxyCount = 1;
          return copy;    }
```

Copy-On-Write Proxy Example (Continued)

```
// Return the number of proxies using this object.    synchronized
int getProxyCount() {      return proxyCount;    }

// Increment the number of proxies using this object by one.
synchronized void addProxy() {      proxyCount++;    }

// Decrement the number of proxies using this object by one.
synchronized void removeProxy() {      proxyCount--;    }  }
}
```