Proxy

- ¿Recuerdan la maquina de chicles?

- El dueno nos contacto de nuevo y ahora nos pregunta “¿Puedes encontrar una forma de enviarme un informe de inventario y estado de las máquinas de manera remota?”

- ¿Puedes encontrar una forma de obtener un informe de inventario y estado de la máquina?

- Ya tenemos métodos en el código máquina de chicles para obtener el conteo de chicles (getCount ()) y obtener el estado actual de la máquina (getState ()).

- Todo lo que tenemos que hacer es crear un informe que pueda imprimirse y enviarse al Dueño.
- Probablemente deberíamos agregar un campo de ubicación a cada máquina de chicles también; de esa forma, el Dueño puede mantener las máquinas en orden.

```
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.

- Ahora vamos a crear otra clase, GumballMonitor, que recupere la ubicación de la máquina, el inventario de chicles y el estado actual de la máquina, y los imprime en un pequeño y agradable informe:

```
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.

Probando el cambio

```
public class GumballMachineTestDrive {
    public static void main(String[] args) {
        int count = 0;

        if (args.length < 2) {
            System.out.println("GumballMachine <name> <inventory>");
            System.exit(1);
        }

        count = Integer.parseInt(args[1]);
        GumballMachine gumballMachine = new GumballMachine(args[0], count);

        GumballMonitor monitor = new GumballMonitor(gumballMachine);


        // rest of test code here

        monitor.report();
    }
}
```

Pass in a location and initial # of gumballs on the command line.

Don't forget to give the constructor a location and count...

...and instantiate a monitor and pass it a machine to provide a report on.

When we need a report on the machine, we call the report() method.

```
File Edit Window Help FlyingFish
%java GumballMachineTestDrive Seattle 112

Gumball Machine: Seattle
Current Inventory: 112 gumballs
Current State: waiting for quarter
```

Y fin, acabamos :)


TE LA CREISTE
WE XD
memegenerator.net

- La salida del monitor se ve genial, pero creo que no estaba claro. Necesito monitorear las máquinas de chicles ¡A DISTANCIA!
- De hecho, ya tenemos las redes establecidas para el monitoreo.

- ¿qué pasa si dejamos nuestra clase GumballMonitor como está, pero le entregamos un proxy, es decir, a un objeto remoto?

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
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.
Gumball Monitor
Proxy
Gumball Machine
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.

El papel del 'proxy remoto'

- Un proxy remoto actúa como un representante local en un objeto remoto.
- ¿Qué es un "objeto remoto"? Es un objeto que vive en el heap de una Máquina Virtual Java diferente (o más generalmente, un objeto remoto que se ejecuta en un espacio de direcciones diferente).
- ¿Qué es un "representante local"? Es un objeto que puede invocar a los métodos locales y reenviarlos al objeto remoto.

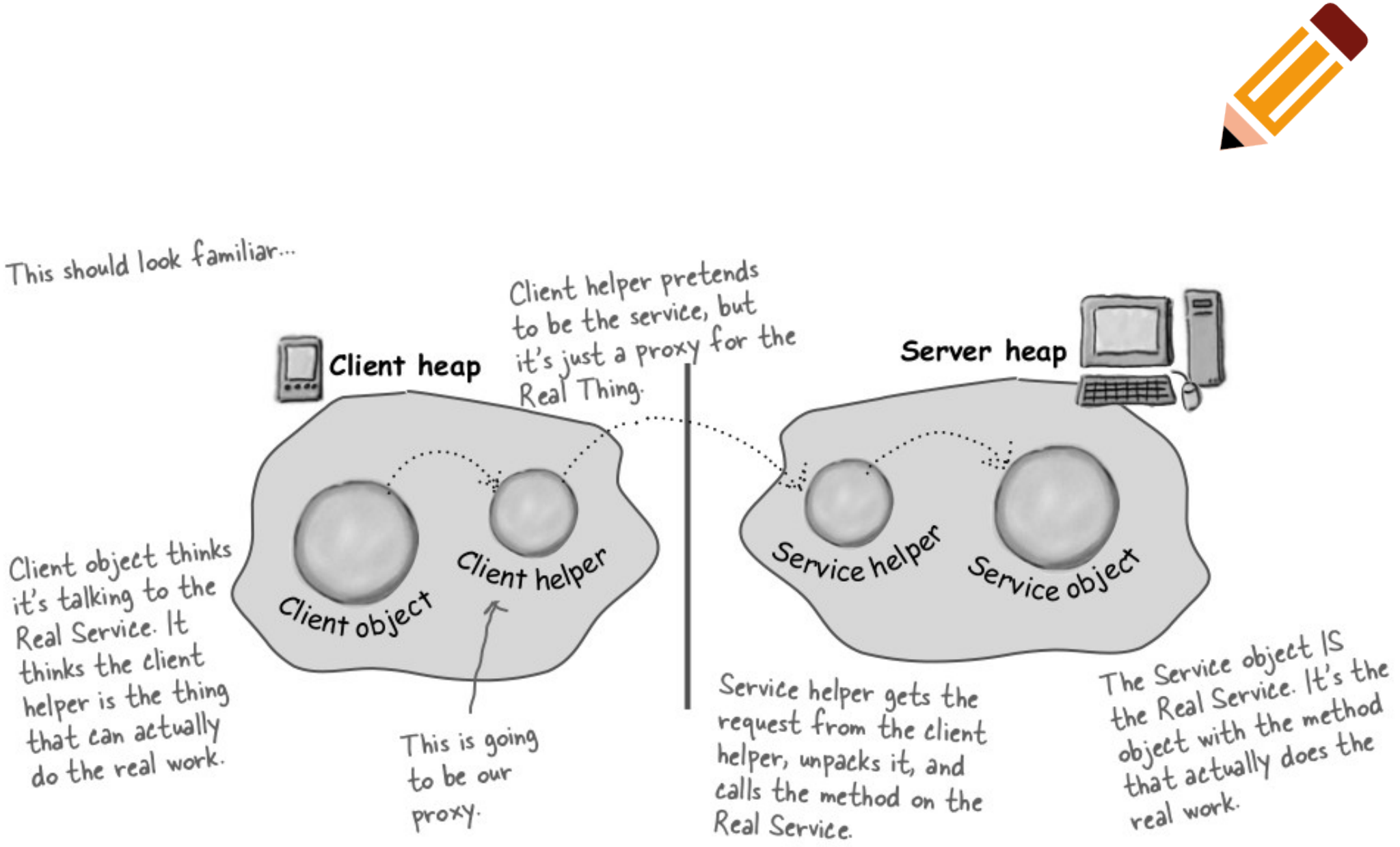
El papel del 'proxy remoto'

- Su objeto cliente actúa como si estuviera realizando llamadas a métodos remotos.
- Pero lo que realmente está haciendo es llamar a métodos en un objeto de 'proxy' local que maneja todos los detalles de bajo nivel de la comunicación de red.

Agregar un proxy remoto al código de monitoreo de la máquina de Chicles

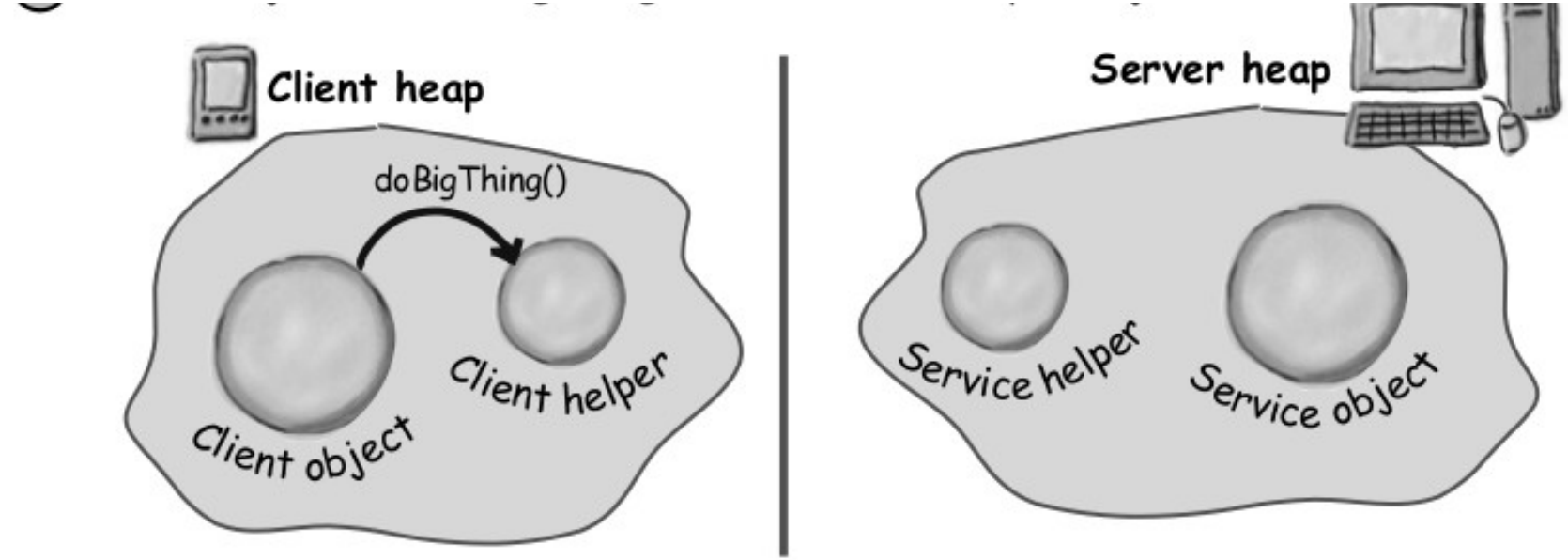
- Invocación de Método Remoto de Java RMI Remote Method Invocation
- 1. Primero,revisaremos la RMI.
- 2. Luego, tomaremos nuestro GumballMachine y lo convertiremos en un servicio remoto que brinda un conjunto de llamadas a métodos que se pueden invocar de manera remota.

- 3. Luego, vamos a crear un proxy que pueda comunicarse con un GumballMachine remoto, nuevamente usando RMI, y volver a unir el sistema de monitoreo para que el Dueño pueda monitorear cualquier cantidad de máquinas remotas.

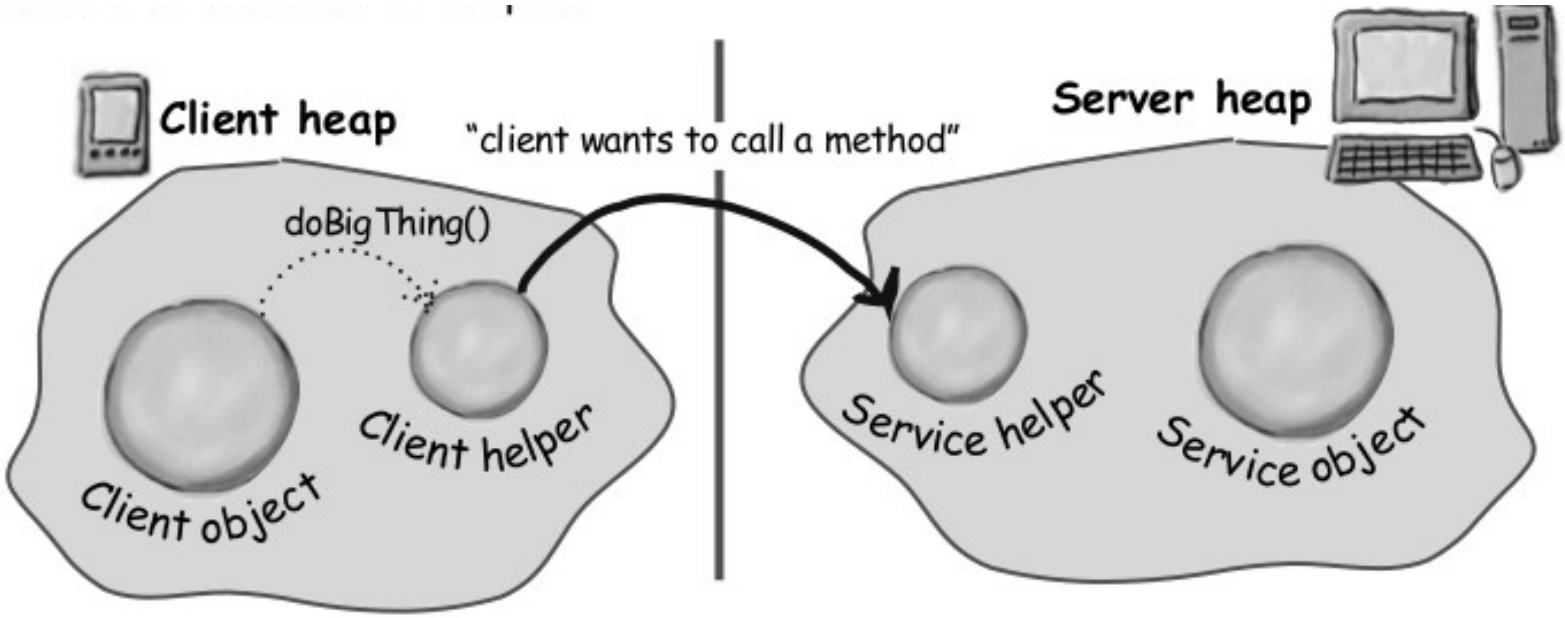

This should look familiar...
Client heap
Client helper pretends to be the service, but it's just a proxy for the Real Thing.
Server heap
Client object thinks it's talking to the Real Service. It thinks the client helper is the thing that can actually do the real work.
Client object
Client helper
This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

Cómo ocurre la llamada al método

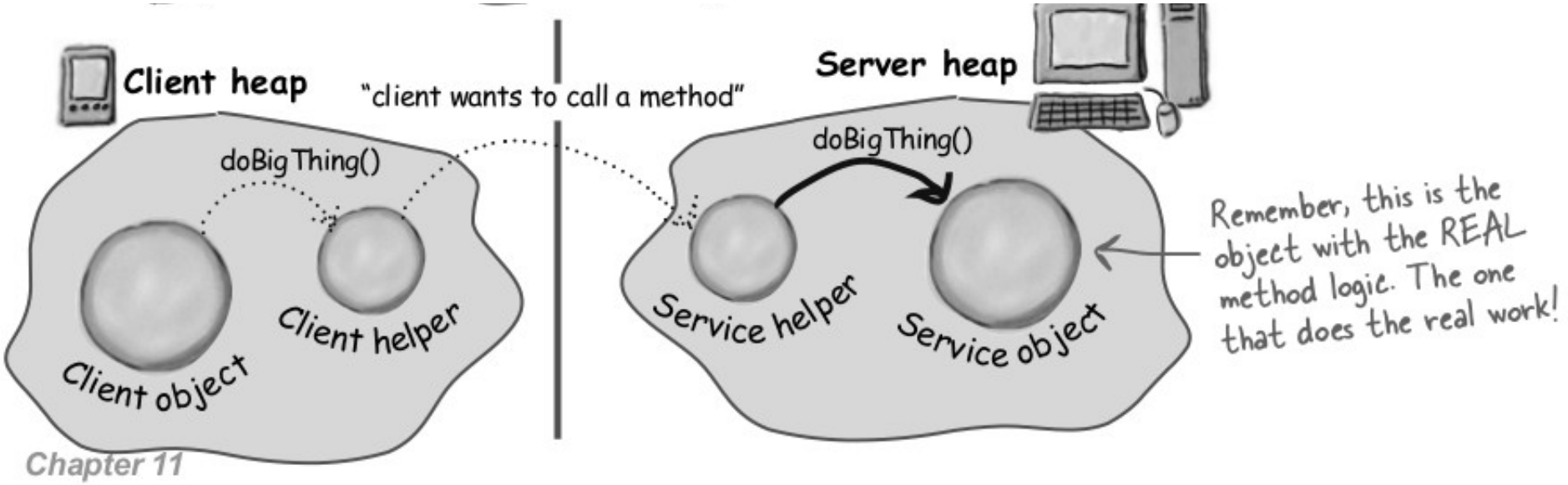
- 1 El objeto del cliente(Client object) llama a doBigThing() en el objeto auxiliar del cliente (client helper object).



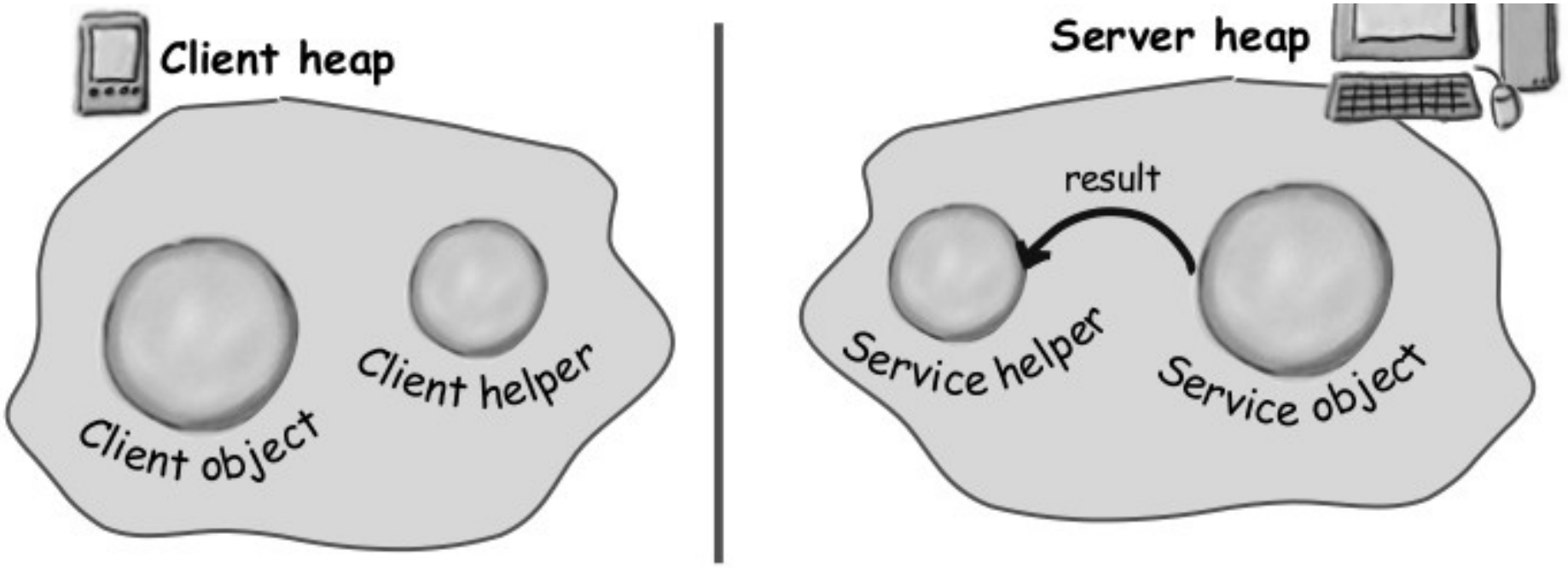
- 2.Client helper empaqueta información sobre la llamada (argumentos, nombre del método, etc.) y la envía a través de la red al ayudante del servicio.



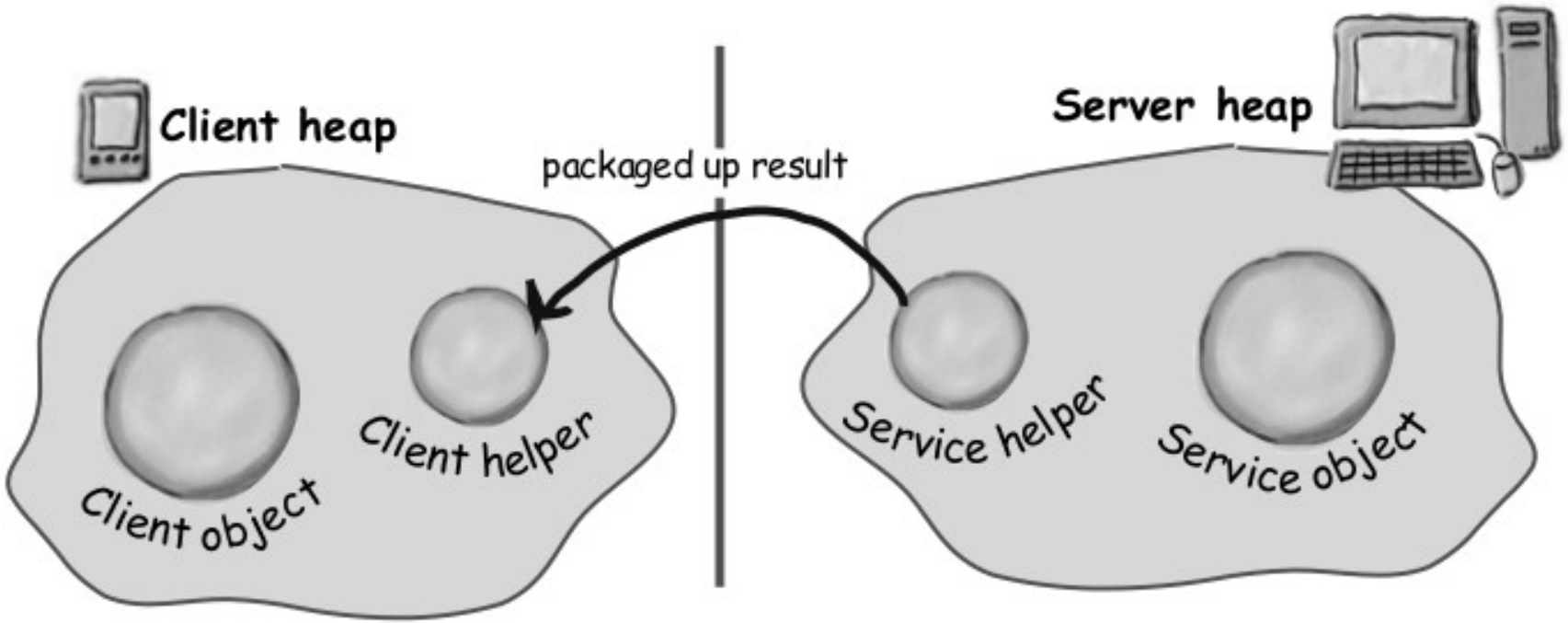
- 3. El ayudante de servicio descomprime la información del cliente auxiliar, descubre qué método llamar (y en qué objeto) e invoca el método real en el objeto de servicio real.



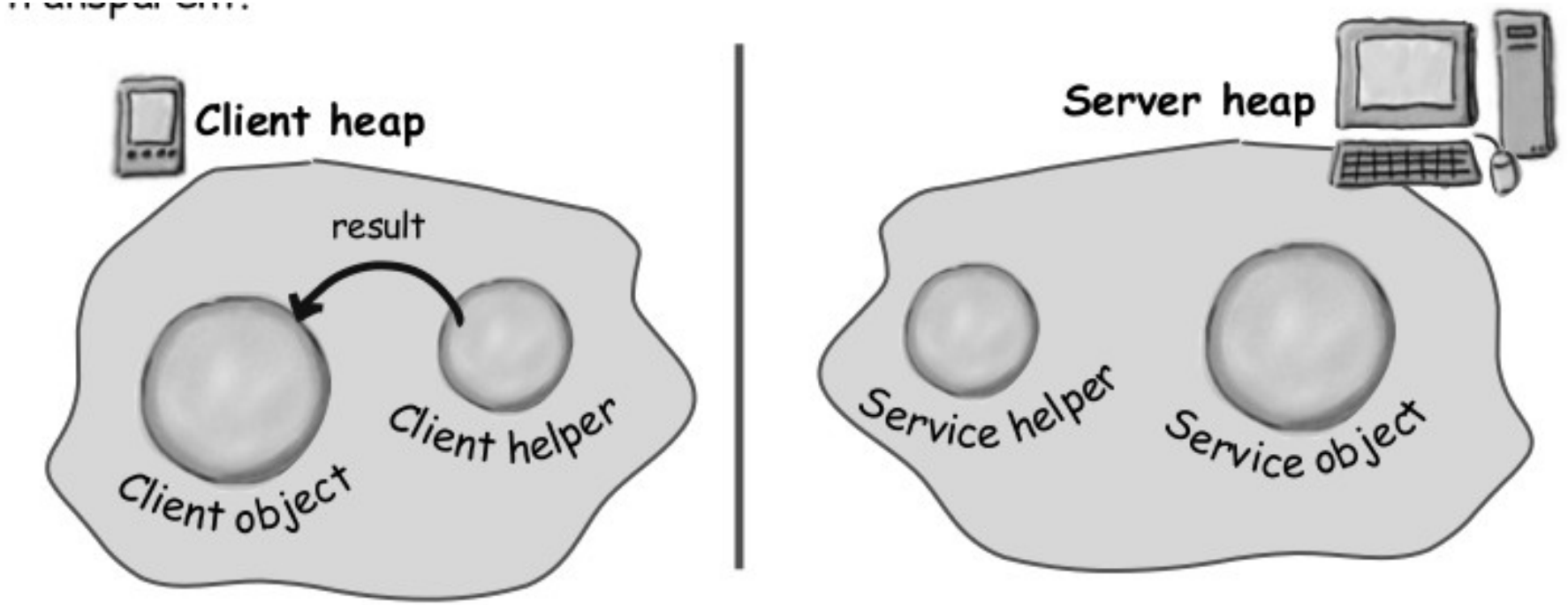
- 4. El método se invoca en el objeto de servicio, que devuelve algunos resultados al servicio auxiliar.



- 5. Service helper empaqueta la información devuelta por la llamada y la envía de vuelta a través de la red al asistente del cliente.



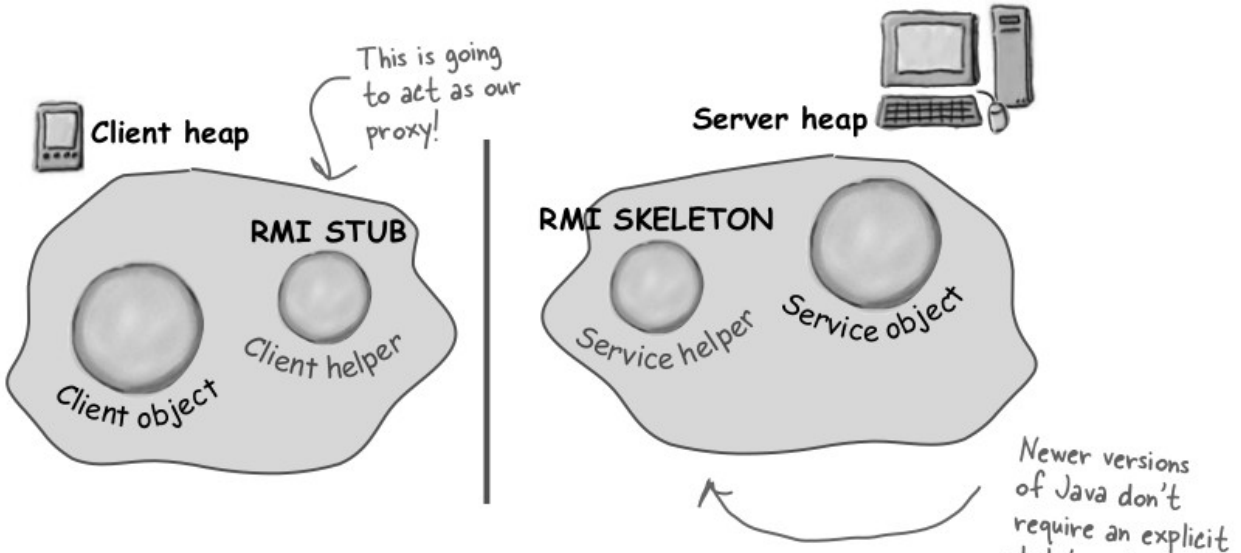
- 6. Client helper desempaqueta los valores resultantes y los devuelve al objeto del cliente. El objeto ve la información como si estuviera con el Service object



- Lo que RMI hace por ti es **construir los objetos auxiliares del cliente y del servicio**, hasta crear un objeto auxiliar para el cliente con los mismos métodos que el servicio remoto. Lo bueno de RMI es que no tienes que escribir tú mismo ninguna de las redes o códigos de E / S. Con tú cliente, llamas a métodos remotos (es decir, los que tiene el Servicio Real) al igual que las llamadas a métodos normales en objetos que se ejecutan en la propia JVM local del cliente.
- RMI también proporciona toda la infraestructura de tiempo de ejecución para que todo funcione, incluido un servicio de búsqueda que el cliente puede usar para buscar y acceder a los objetos remotos.

- Existe una diferencia entre las llamadas RMI y las llamadas a métodos locales (normales). Recuerde que aunque para el cliente parece que la llamada al método es local, el asistente del cliente envía la llamada al método a través de la red.

- Nomenclatura de RMI: en RMI, el cliente auxiliar es un 'stub' y el ayudante de servicio es un '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.

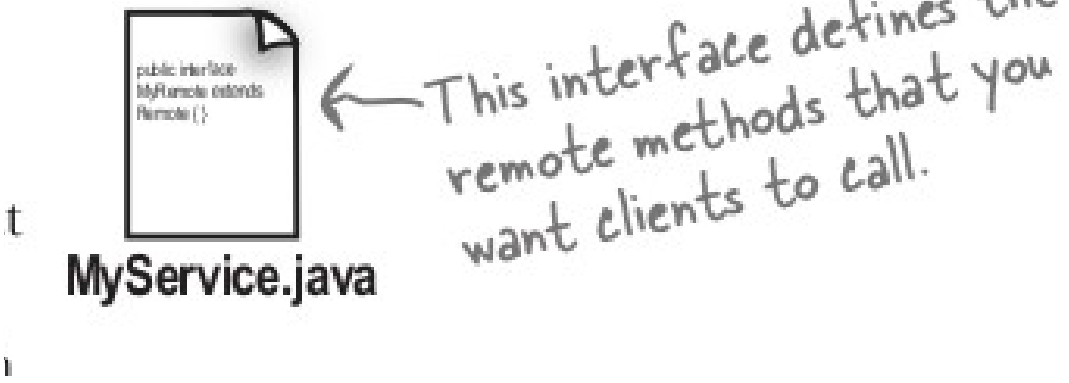
Haciendo el servicio remoto

- Paso uno:

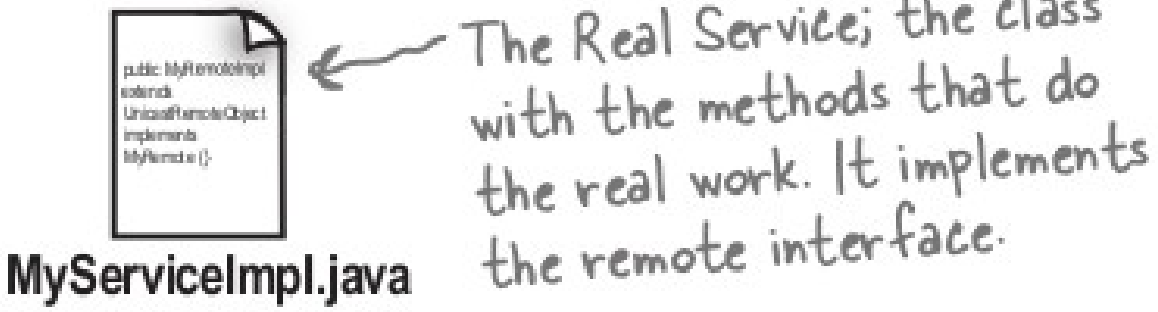
  Hacer una interfaz remota

  La interfaz remota define los métodos que un cliente puede llamar de forma remota. Es lo que el cliente MyService.java usará como el tipo de clase para su servicio. ¡Tanto el Stub como el servicio real implementarán esto!

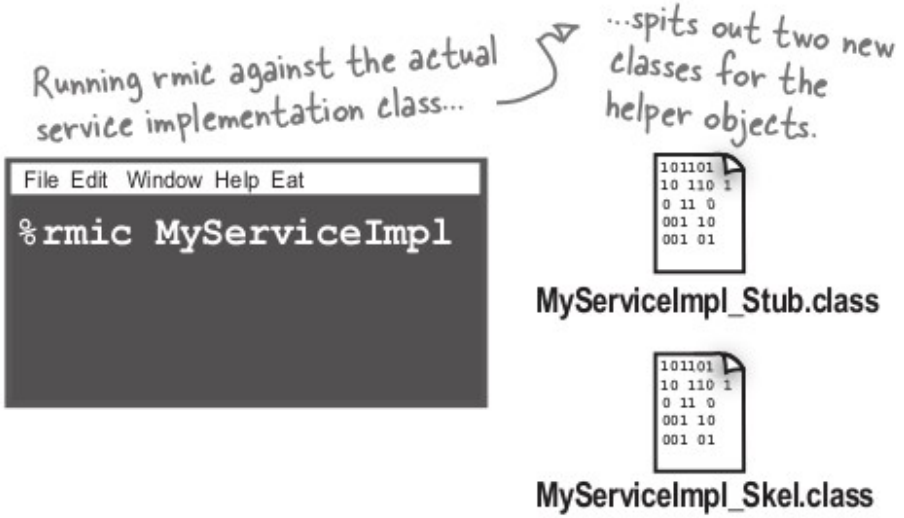
- public interface
- MyRemote extends
- Remote {



- Segundo paso:
- Hacer una implementación remota
- Esta es la clase que hace el trabajo real. Tiene la implementación real de los métodos remotos definidos en la interfaz remota. Es el objeto sobre el que el cliente quiere llamar a los métodos (por ejemplo, nuestro GumballMachine).
- public MyRemoteImpl
- extends
- UnicastRemoteObject
- implements
- MyRemote { }



- Paso tres:
- Genere los stubs y skeletons usando rmic Estos son los 'ayudantes' del cliente y del servidor. No tiene que crear estas clases ni mirar el código fuente que las genera. Todo se maneja automáticamente cuando ejecuta la herramienta rmic que viene con su kit de desarrollo Java.



- Paso cuatro:
- Inicie el registro RMI (rmiregistry)
- El rmiregistry es como las páginas blancas de una guía telefónica. Es donde el cliente va a obtener el proxy (el objeto stub / helper del cliente)

Run this in a separate terminal.

- Paso cinco:
- Comience el servicio remoto
- Tienes que poner el objeto de servicio en funcionamiento. Su clase de implementación de servicio crea una instancia del servicio y lo registra con el registro RMI. Al registrarlo, el servicio está disponible para los clientes.

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.*;

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 means 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.

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();
      }
   }
}
```

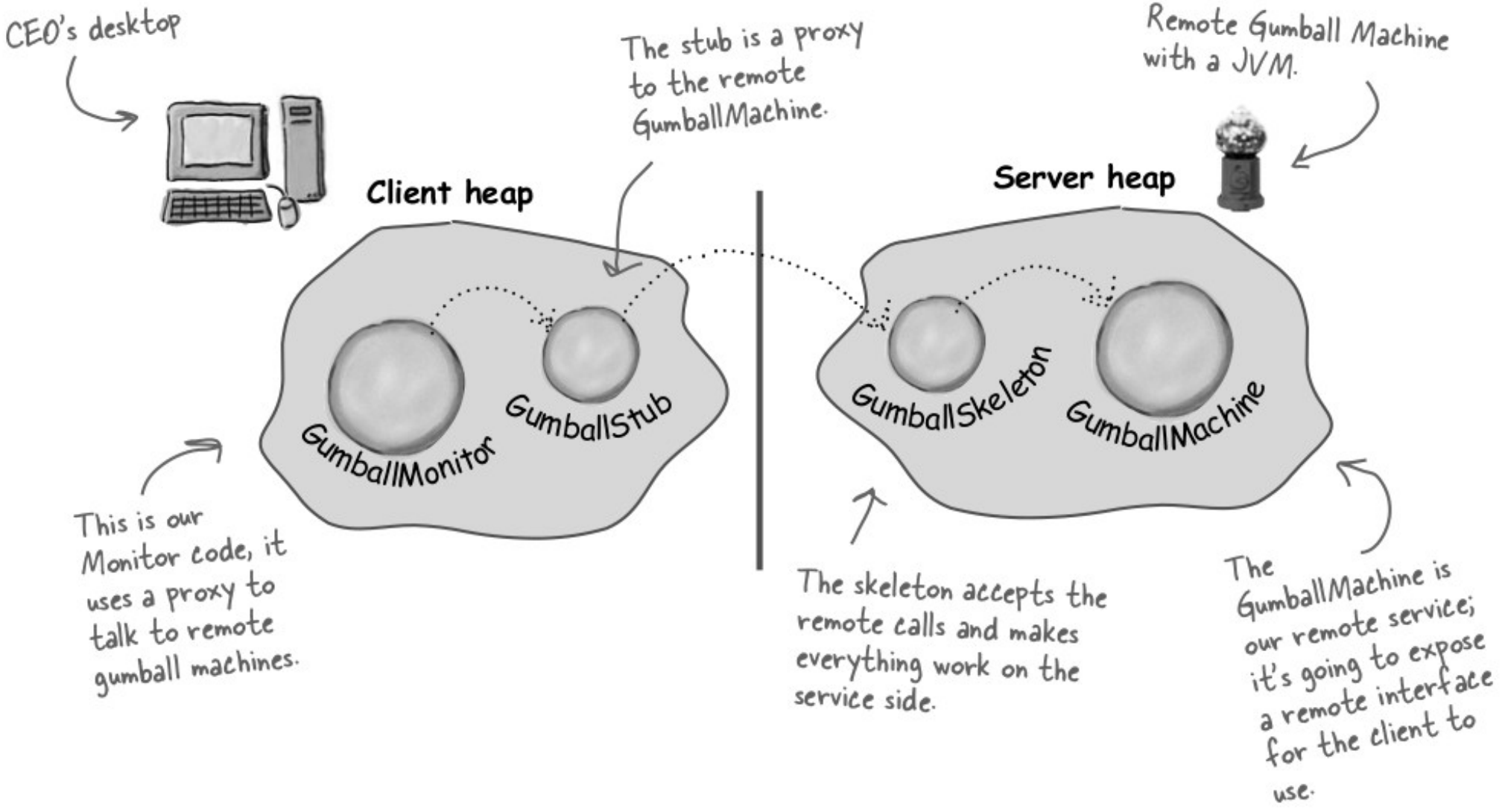
The Naming class (for doing the rmiregistry lookup) is in the java.rmi package.

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.)


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.

Don't forget to import java.rmi.*

This is the 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;
}
```

All return types need to be primitive or Serializable...

Here are the methods were going to support. Each one throws RemoteException.

```
import java.io.*;
```

Serializable is in the java.io package.

```
public interface State extends Serializable {
    public void insertQuarter();
    public void ejectQuarter();
    public void turnCrank();
    public void dispense();
}
```

Then we just extend the Serializable interface (which has no methods in it). And now State in all the subclasses can be transferred over the network.

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

    // all other methods here
}
```

In each implementation of State, we add the transient keyword to the GumballMachine instance variable. This tells the JVM not to serialize this field. Note that this can be slightly dangerous if you try to access this field once its been serialized and transferred.

First, we need to import the rmi packages.

GumballMachine is going to subclass the UnicastRemoteObject; this gives it the ability to act as a remote service.

GumballMachine also needs to implement the remote interface...

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

public class GumballMachine
        extends UnicastRemoteObject implements GumballMachineRemote
{
    // instance variables here

    public GumballMachine(String location, int numberGumballs) throws RemoteException {
        // code here
    }

    public int getCount() {
        return count;
    }

    public State getState() {
        return state;
    }

    public String getLocation() {
        return location;
    }

    // other methods here
}
```

...and the constructor needs to throw a remote exception, because the superclass does.

That's it! Nothing changes here at all!

```
public class GumballMachineTestDrive {

    public static void main(String[] args) {
        GumballMachineRemote gumballMachine = null;
        int count;
        if (args.length < 2) {
            System.out.println("GumballMachine <name> <inventory>");
            System.exit(1);
        }

        try {
            count = Integer.parseInt(args[1]);

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

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

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

Let's go ahead and get this running...

Run this first.

This gets the RMI registry service up and running.

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

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

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

Run this second.

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

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

```
import java.rmi.*;

public class GumballMonitor {
    GumballMachineRemote machine;

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

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

Now we're going to rely on the remote interface rather than the concrete GumballMachine class.

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

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

```
import java.rmi.*;

public class GumballMonitorTestDrive {

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

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

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

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

Here's all the locations were going to monitor.

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

We also create an array of monitors.

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

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

On each machine, run rmiregistry in the background or from a separate terminal window...

...and then run the GumballMachine, giving it a location and an initial gumball count.

```
File Edit Window Help Huh?
% rmiregistry &
% java GumballMachine santafe.mightygumball.com 100
```

```
File Edit Window Help Huh?
% rmiregistry &
% java GumballMachine boulder.mightygumball.com 100
```

```
File Edit Window Help Huh?
% rmiregistry &
% java GumballMachine seattle.mightygumball.com 250
```

popular machine!

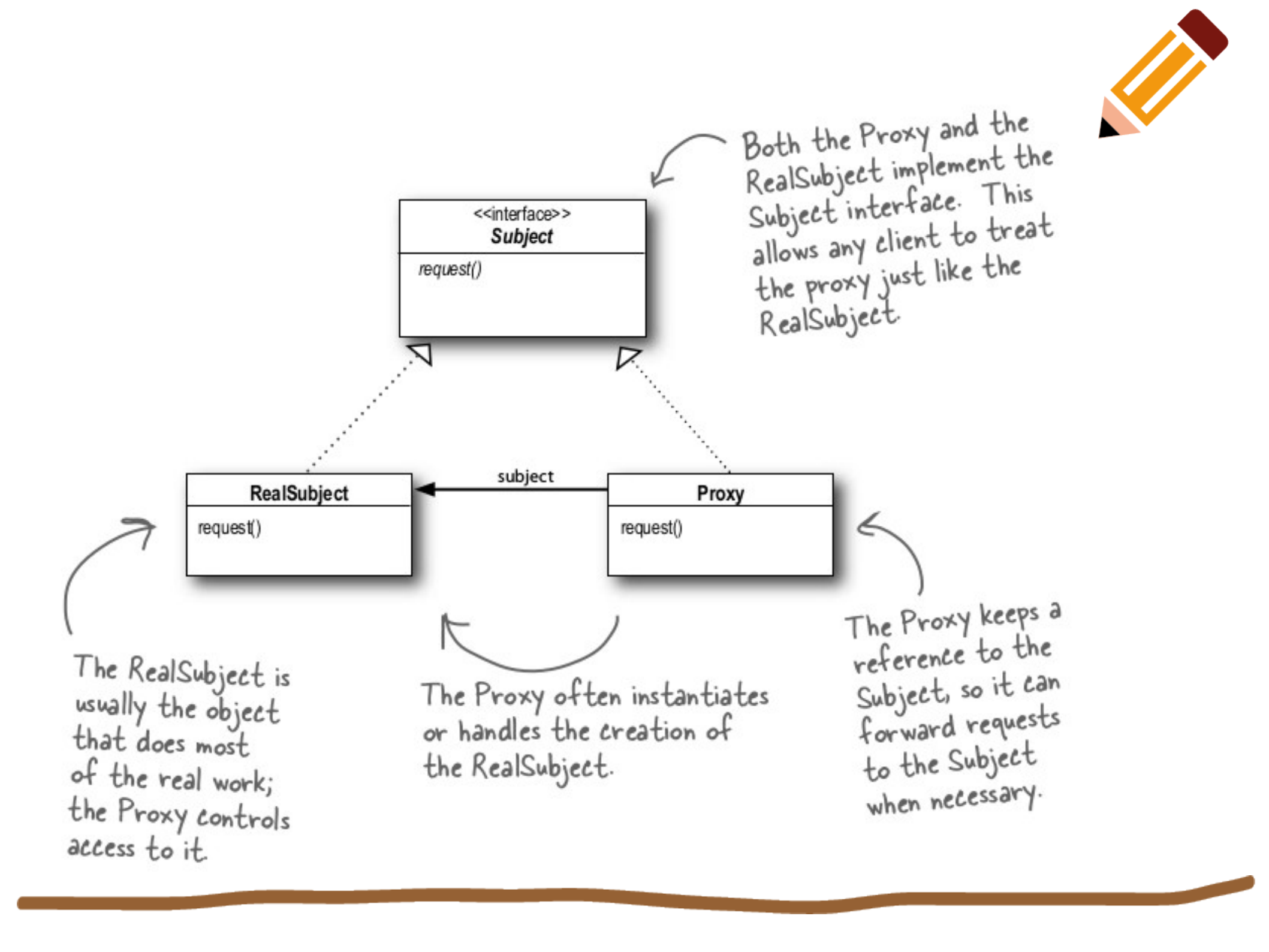
```
File Edit Window Help GumballsAndBeyond
% java GumballMonitor
Gumball Machine: santafe.mightygumball.com
Current inventory: 99 gumballs
Current state: waiting for quarter

Gumball Machine: boulder.mightygumball.com
Current inventory: 44 gumballs
Current state: waiting for turn of crank

Gumball Machine: seattle.mightygumball.com
Current inventory: 187 gumballs
Current state: waiting for quarter
%
```

The monitor iterates over each remote machine and calls its getLocation(), getCount() and getState() methods.

El patrón de proxy proporciona un sustituto o marcador de posición para otro objeto para controlar el acceso a él.


<<interface>>
Subject
request()
Both the Proxy and the RealSubject implement the Subject interface. This allows any client to treat the proxy just like the RealSubject.
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.