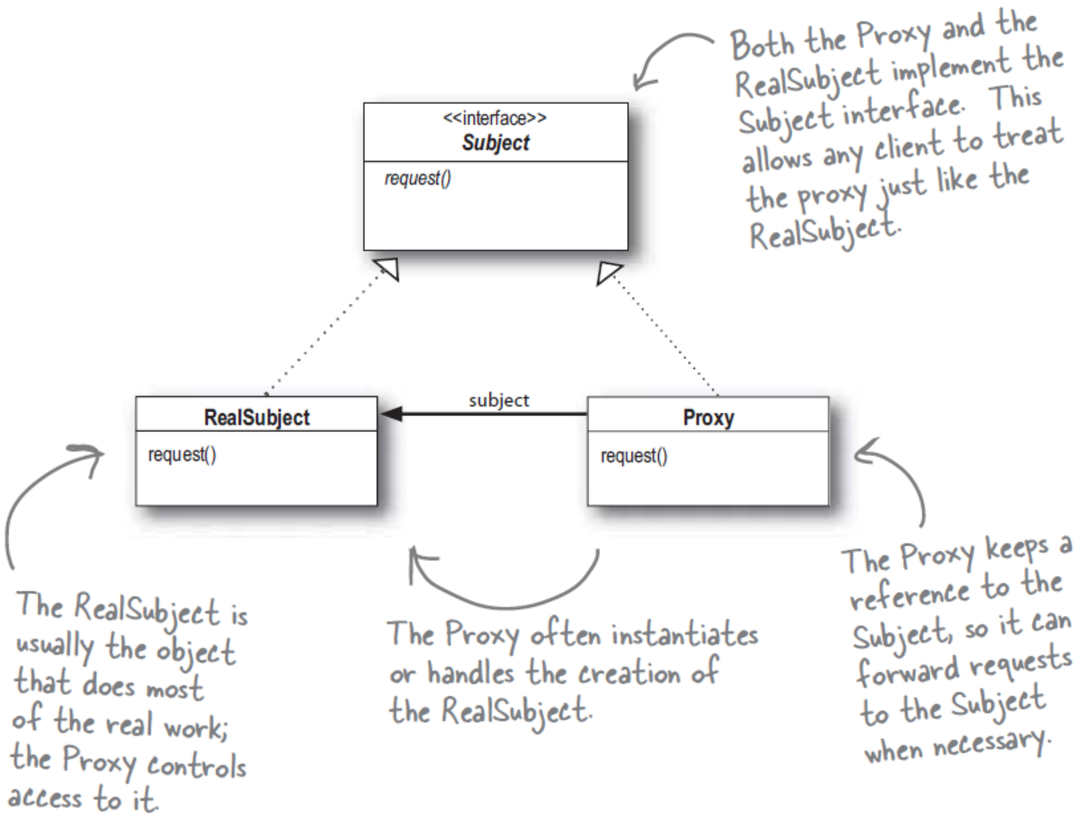
Proxy Pattern

Proxy Pattern

- **Purpose**
  - Allows for object level access control by acting as a pass through entity or a placeholder object.
- **Use When**
  - Access control for the original object is required.
  - Added functionality is required when an object is accessed.

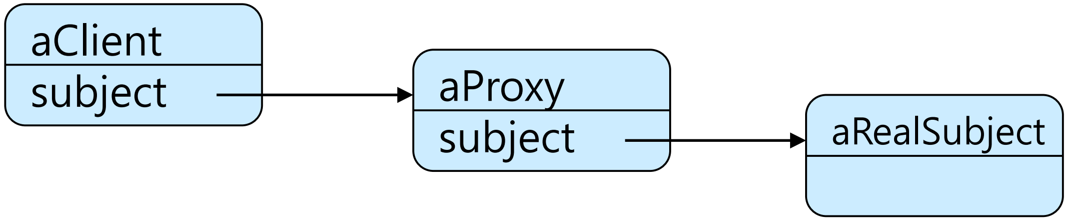
The Proxy Pattern

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



Proxy Pattern Collaborations

- Subject
  - defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected
- RealSubject
  - defines the real object that the proxy represents
- Proxy
  - maintains a reference that lets the proxy access the real subject
  - provides an interface identical to Subject's so that a proxy can by substituted for the real subject
  - controls access to the real subject and may be responsible for creating and deleting it

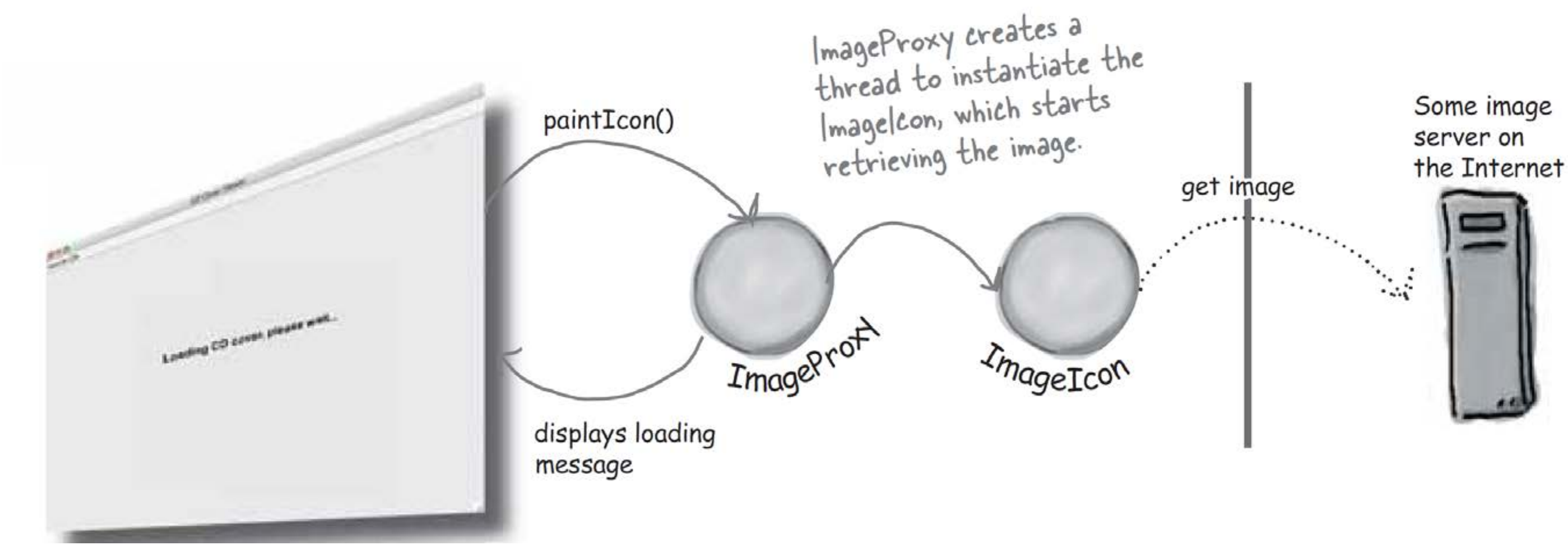


Applicability

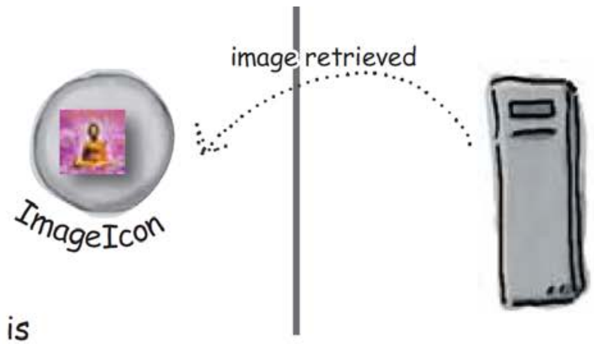
- whenever there is a need for a **more versatile** or **sophisticated reference** to an object than a simple pointer
  - *remote proxy*
    - responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space
  - *virtual proxy*
    - may cache additional information about the real subject so that they can postpone accessing it
  - *protection proxy*
    - checks that the caller has the access permissions required to perform a request

Virtual Proxy in Action

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.

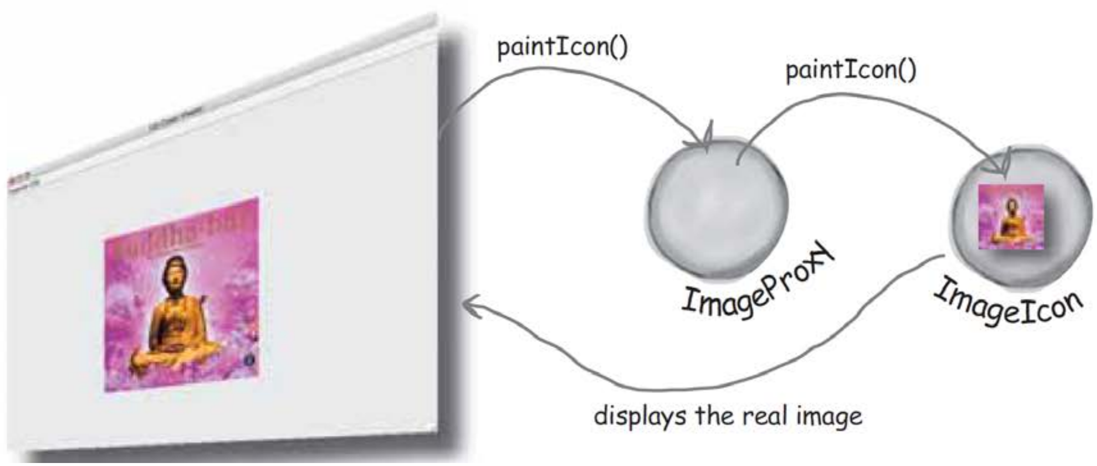


Virtual Proxy in Action



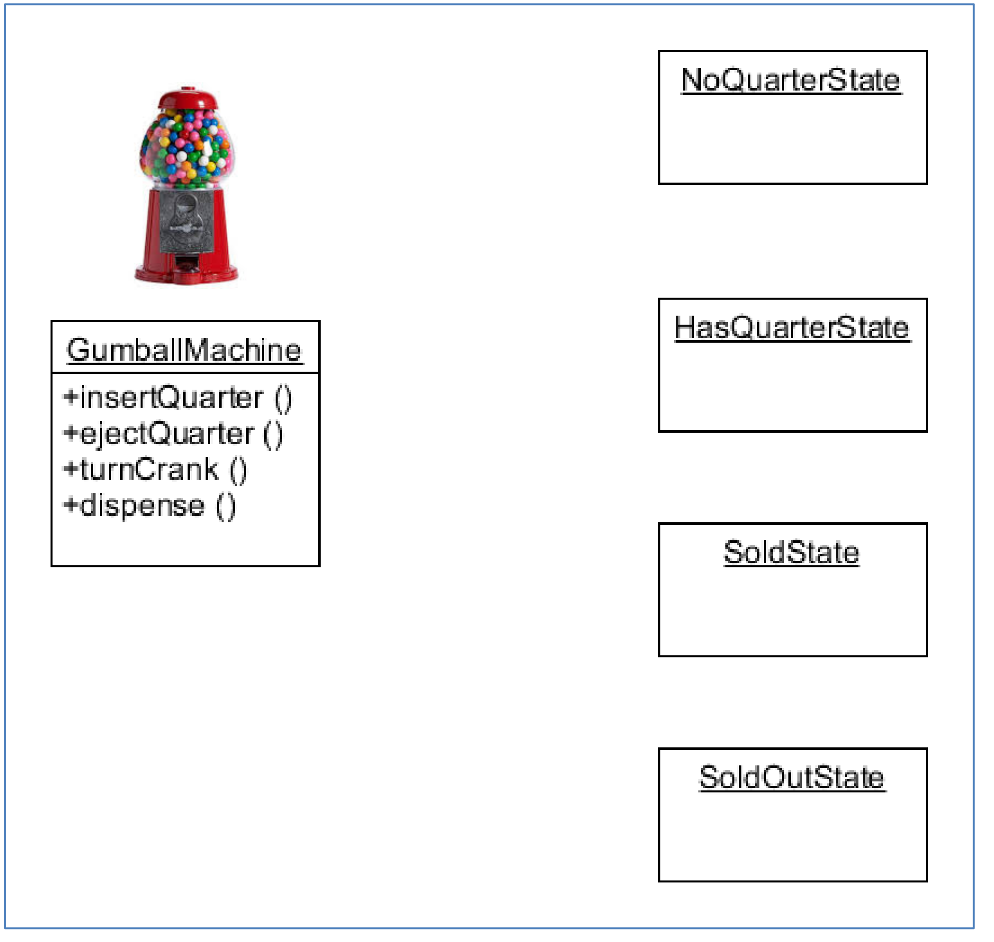
2. At some point the image is returned and the ImageIcon fully instantiated.

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



Request From the CEO of Mighty Gumball

- He wants to have some monitoring program for his gumball machines.
  - getCount()
  - getState()
  - getLocation()



Coding the Monitor

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

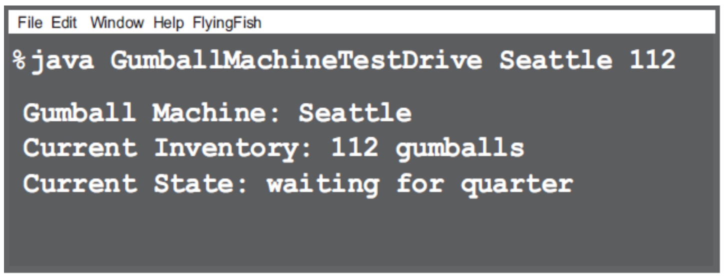
Coding the Monitor

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

Testing the Monitor

```
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);
        }
        try {
            count = Integer.parseInt(args[1]);
        } catch (Exception e) {
             e.printStackTrace();
             System.exit(1);
        }
        GumballMachine gumballMachine = new GumballMachine(args[0], count);
        GumballMonitor monitor = new GumballMonitor(gumballMachine);
        // rest of test code here

        monitor.report();
    }
}
```

Was the CEO Satisfied?



Well, what the CEO really wanted was to monitor the gumball machines REMOTELY!

Distributed Computing

- Distributed Computing
  - involves the design and implementation of applications as a set of cooperating software entities (processes, threads, objects) that are distributed across a network of machines
- Advantages to Distributed Computing
  - Performance
  - Scalability
  - Resource Sharing
  - Fault Tolerance
- Difficulties in developing Distributed Computing systems
  - Latency
  - Synchronization
  - Partial Failure

Client-Server Model and Programming

- Client-Server Model
  - Client - entity that makes a request for a service
  - Server - entity that responds to a request and provides a service
  - The predominant networking protocol in use today is the Internet Protocol (IP). The main API for writing client-server programs using IP is the Berkeley socket API.
- Programming
  - Dealing with all of the details of the socket library calls can be tedious. (See, for example, Stevens' *Unix Network Programming.)*
  - The java.net package provides classes to abstract away many of the details of socket-level programming, making it simple to write client-server applications

Remote Procedure Call (RPC)

- Disadvantage of Client-Server model
  - Both the client and server had to be aware of the socket level details
- Wouldn’t it be nice if even these details were abstracted away and the request to the server looked like a local procedure call from the viewpoint of the client?
- That’s the idea behind a Remote Procedure Call (RPC), a technology introduced in the late 1970’s
- Two RPC specifications:
  - SUN’s Open Network Computing (ONC) RPC
  - OSF’s Distributed Computing Environment (DCE) RPC

Distributed Object Technology

- But RPC is not object-oriented. In the OO world, we'd like to have distributed objects and remote method calls.
- While there are many Distributed Object Technologies available today, three are widely available:
  - RMI
  - CORBA
  - SOAP
- Remote Method Invocation (RMI)
  - Developed by SUN
  - Available as part of the core Java API
  - Java-centric
  - Object interfaces defined as Java interfaces
  - Uses object serialization

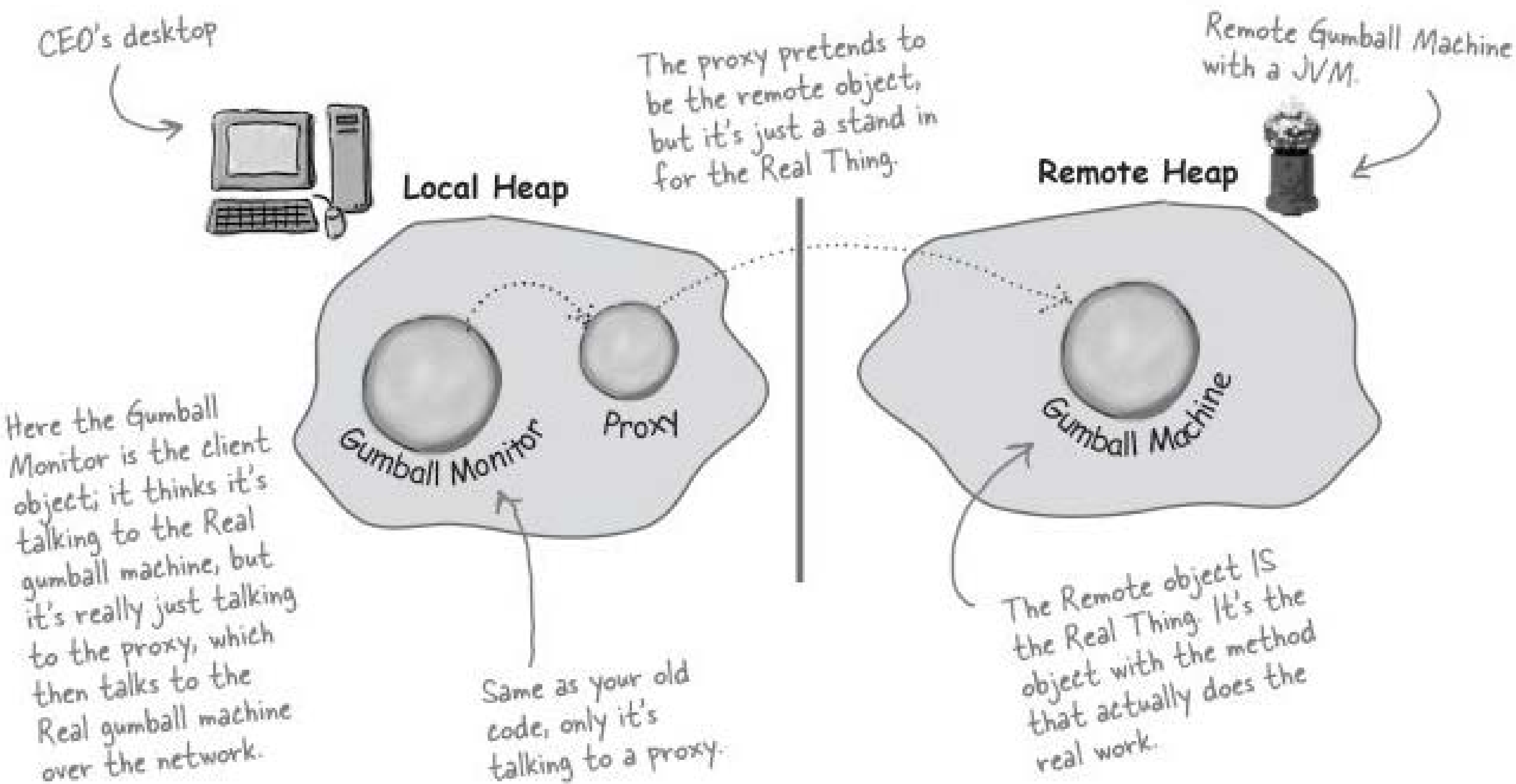
Remote Method Invocation

- Provides a distributed object capability for Java applications
- Allows a Java method to obtain a reference to a remote object and invoke methods of the remote object nearly as easily as if the remote object existed locally
- The remote object can be in another JVM on the same host or on different hosts across the network
- Uses object serialization to marshal and unmarshal method arguments
- Supports the dynamic downloading of required class files across the network

RMI Stubs and Skeletons

- RMI uses **stub** and **skeleton** objects to provide the connection between the client and the remote object
- A ***stub*** *is a* ***proxy for a remote object*** *which is responsible for* forwarding method invocations from the client to the server where the actual remote object implementation resides
- A **client's reference** to a remote object, therefore, is **actually** a **reference to a local stub**. The client has a local copy of the stub object.
- A ***skeleton*** *is a* ***server-side object*** *which contains a method that* dispatches calls to the actual remote object implementation
- A **remote object** has an **associated local skeleton object** to dispatch remote calls to it

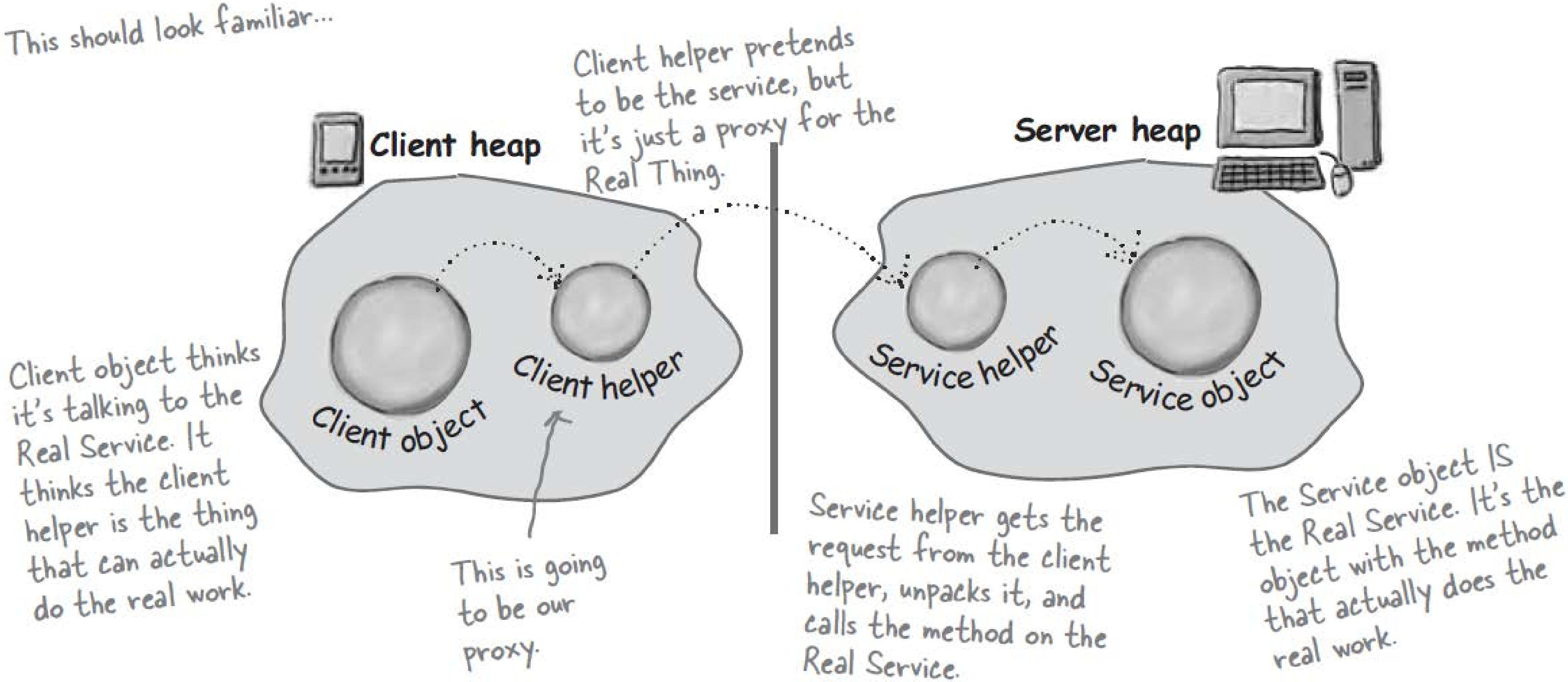
Using Remote Proxy



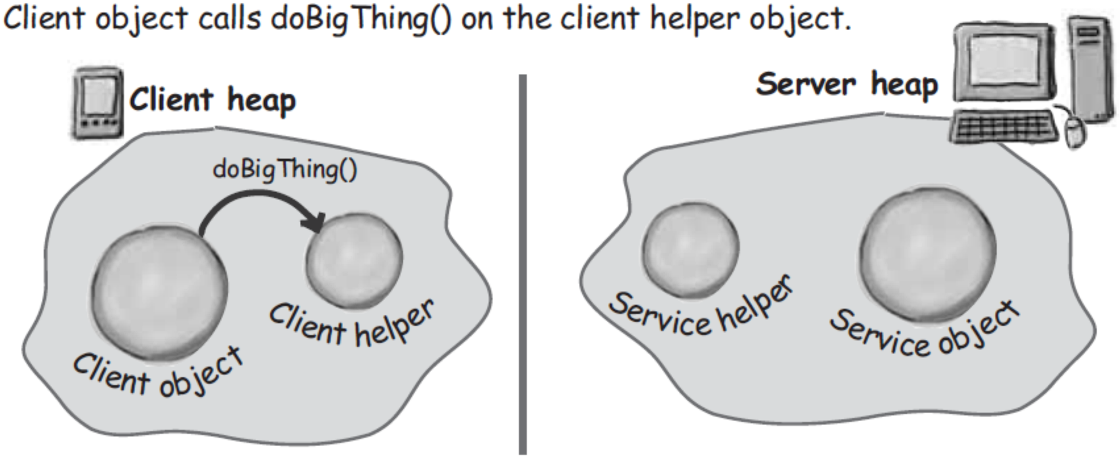
The Roles of Helper Objects

- Request From Client to Server
  - Client Side: The client calls a method on the client helper, as if the client helper were the actual service. Then the client helper takes care of forwarding that request to the server.
  - Server Side: The service helper receives the request from the client helper, via a Socket connection, unpacks the information about the call and then invokes the real method on the real service object. (local call in the server)
- Reply From Server to Client
  - Server Side: The service helper gets the return value from the service, packs it, and ships it back to the client helper.
  - Client Side: The client helper unpacks the information and returns the value to the client object.

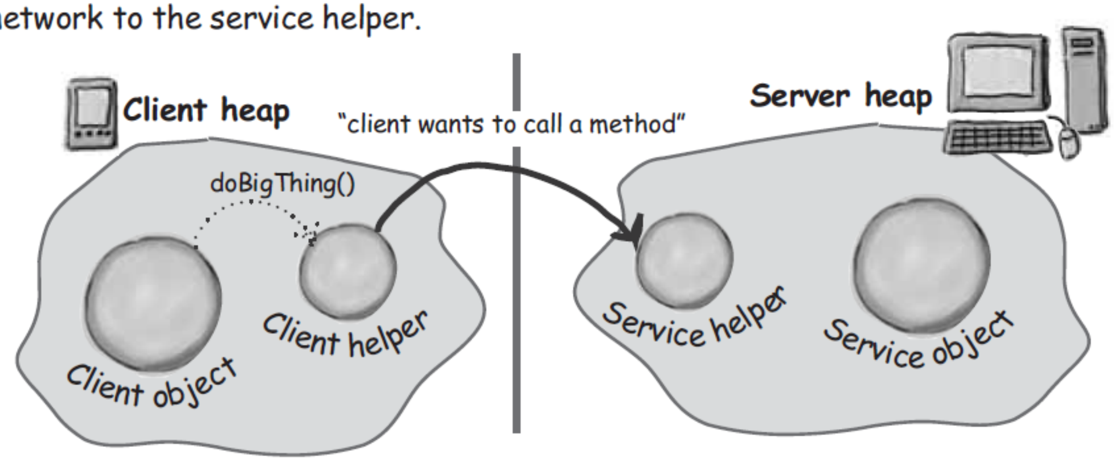
Remote Method Invocation



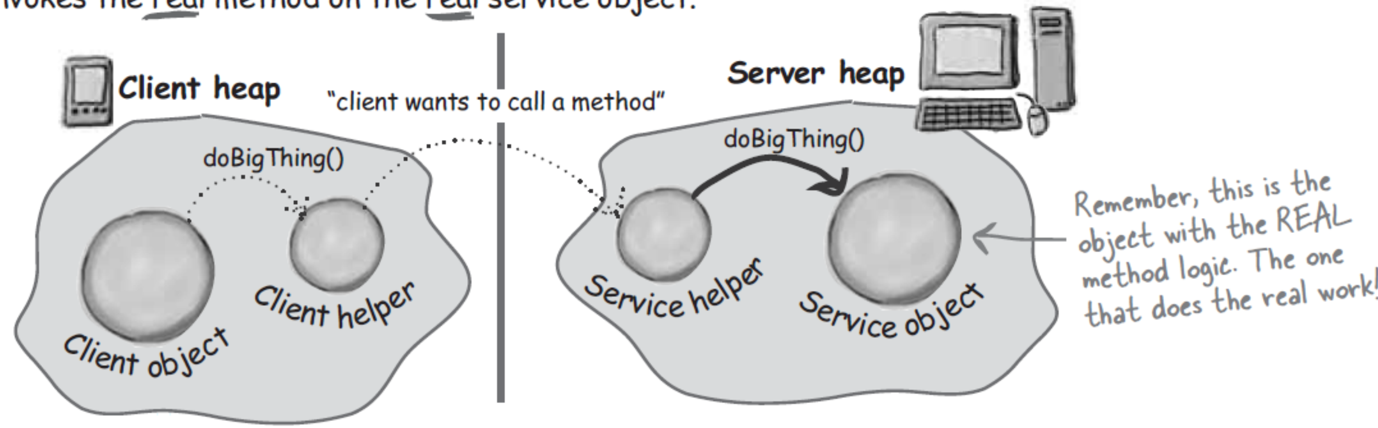
(1) Client object calls doBigThing() on the client helper object.



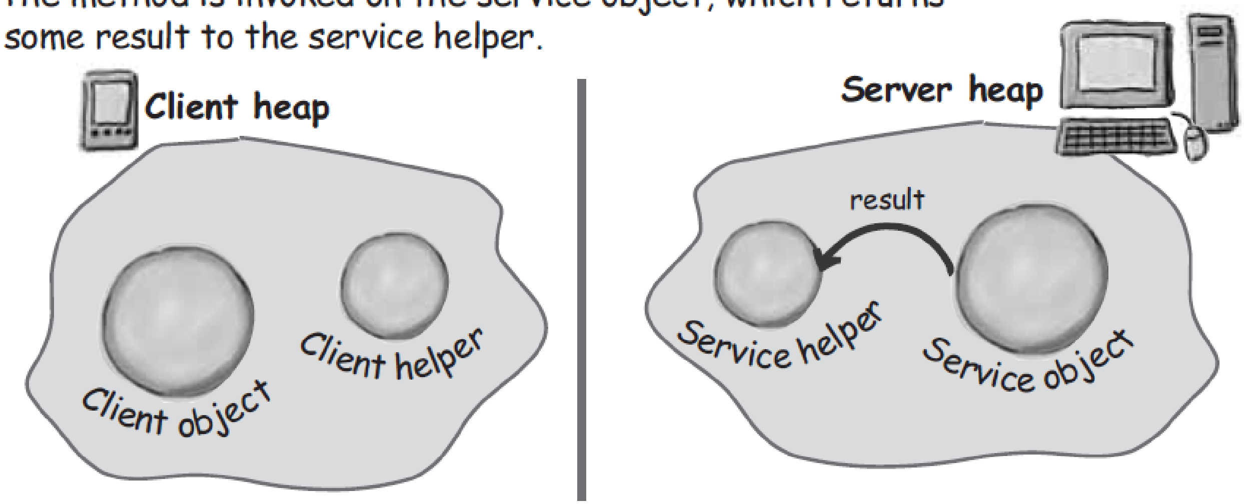
(2) Client helper packages up information about the call (arguments, method name, etc.) and ships it over the network to the service helper.



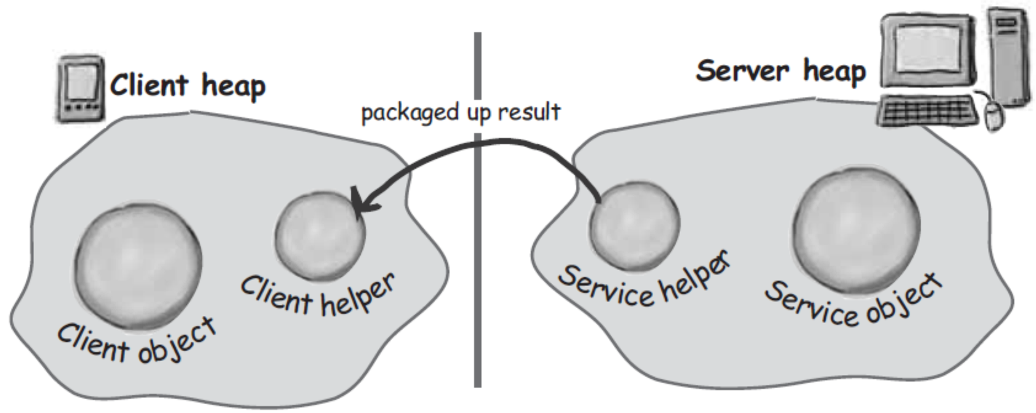
3. Service helper unpacks the information from the client helper, finds out which method to call (and on which object) and invokes the <u>real</u> method on the <u>real</u> service object.



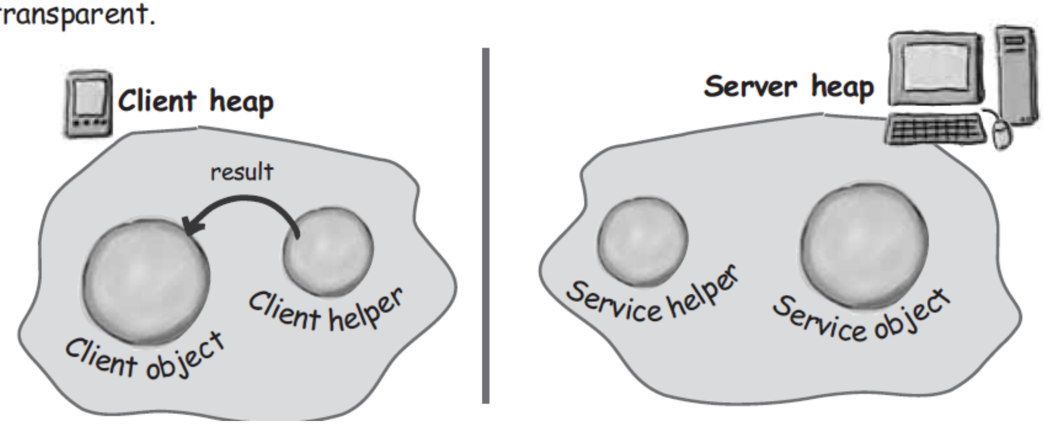
4. The method is invoked on the service object, which returns some result to the service helper.



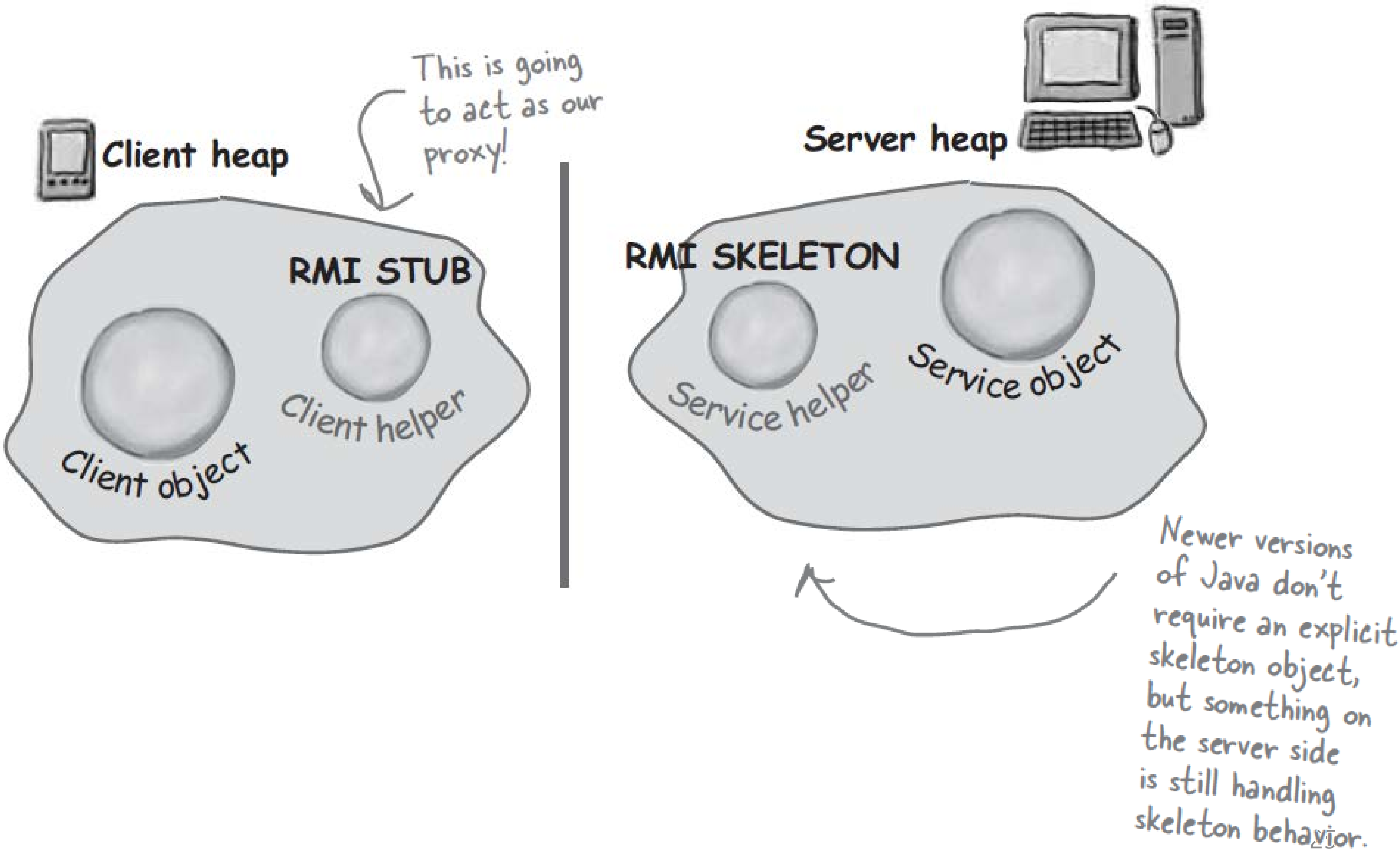
⑤ Service helper packages up information returned from the call and ships it back over the network to the client helper.



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



Java RMI the Big Picture



Making the Remote Service

1. Make a Remote Interface (MyService.java)
   - The remote interface defines the methods that a client can call remotely
2. Make a Remote Implementation (MyServiceImpl.java)
   - It has the real implementation of the remote methods defined in the remote interface
3. Generate the stubs and skeletons using rmic
   - Helpers <- they are automatically generated from the remote implementation, MyServiceImpl.java.
4. Start the RMI Registry
5. Start the remote service

Step One: Make a Remote Interface

- Extend java.rmi.Remote
- Declare the all methods throw a RemoteException
- Be sure arguments and return values are primitives or Serializable

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

Step Two: Make a Remote Implementation

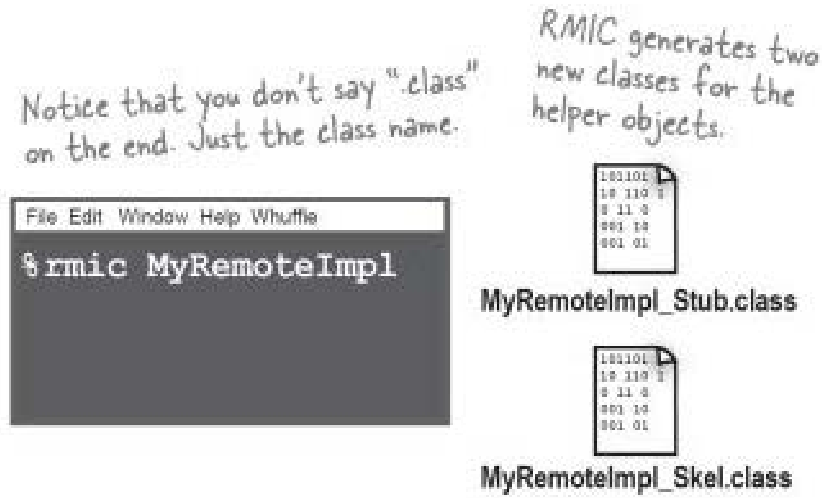
- Implement the Remote Interface
- Extend UnicastRemoteObject
- Write a no-arg constructor that declares a RemoteException
- Register the service with the RMI registry

Remote Service 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();
         }
    }
}
```

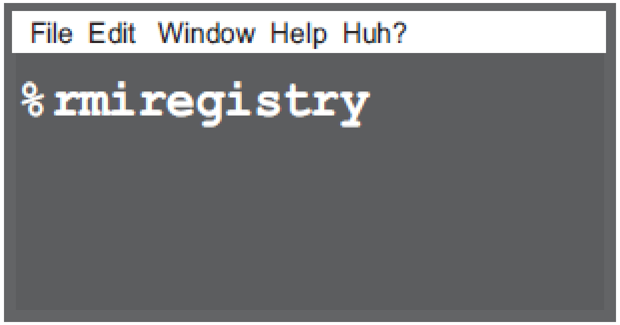
Step Three: Generates Stubs and Skeletons

- Run *rmic* on the remote implementation class (not the remote interface)

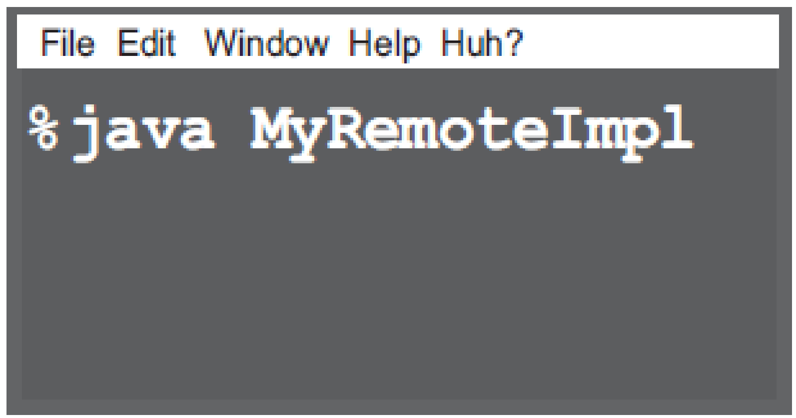


Step Four: Run rmiregistry

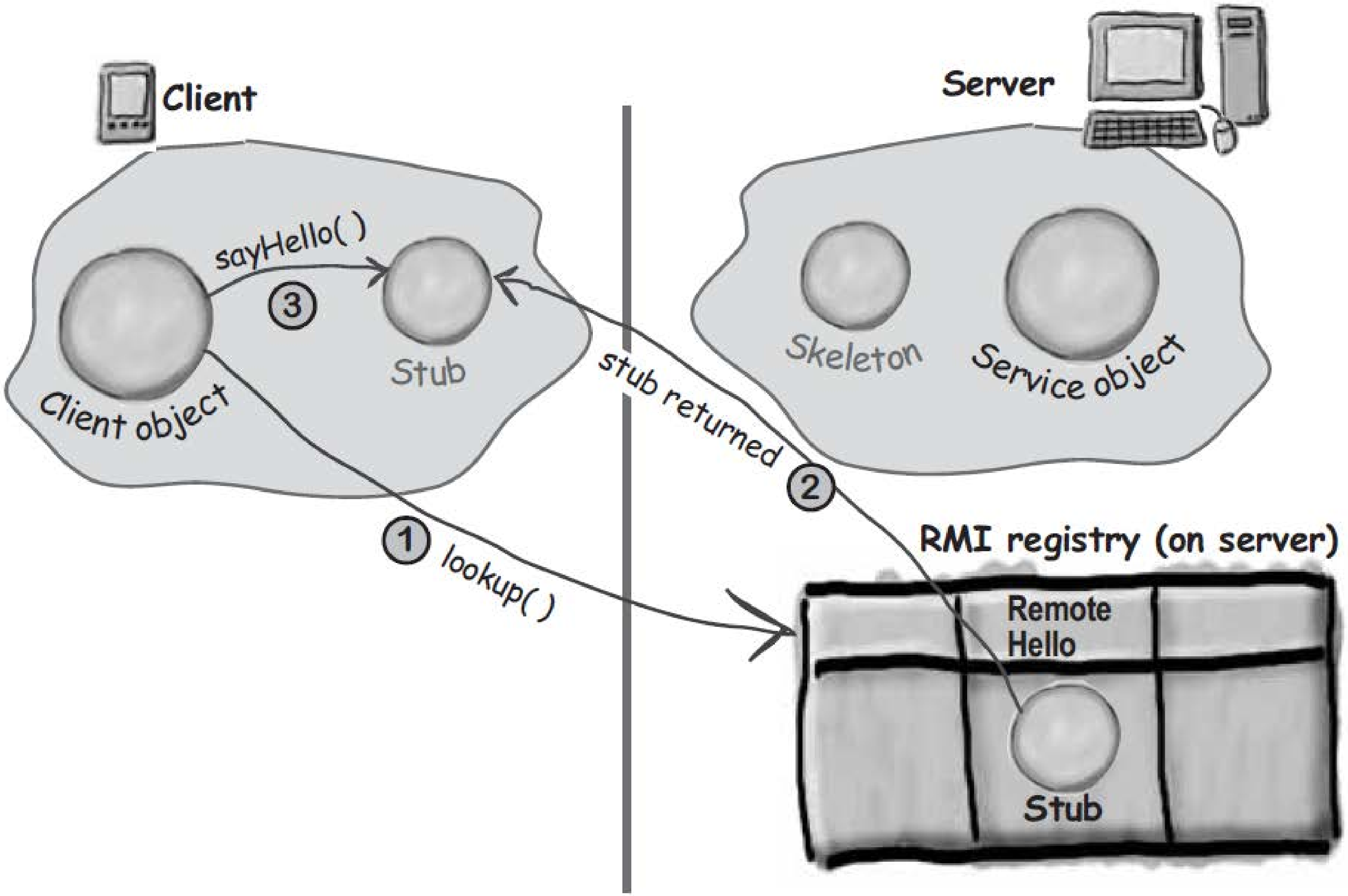
- Be sure you start it from a directory that has access to your classes.



Step Five: Start the Service



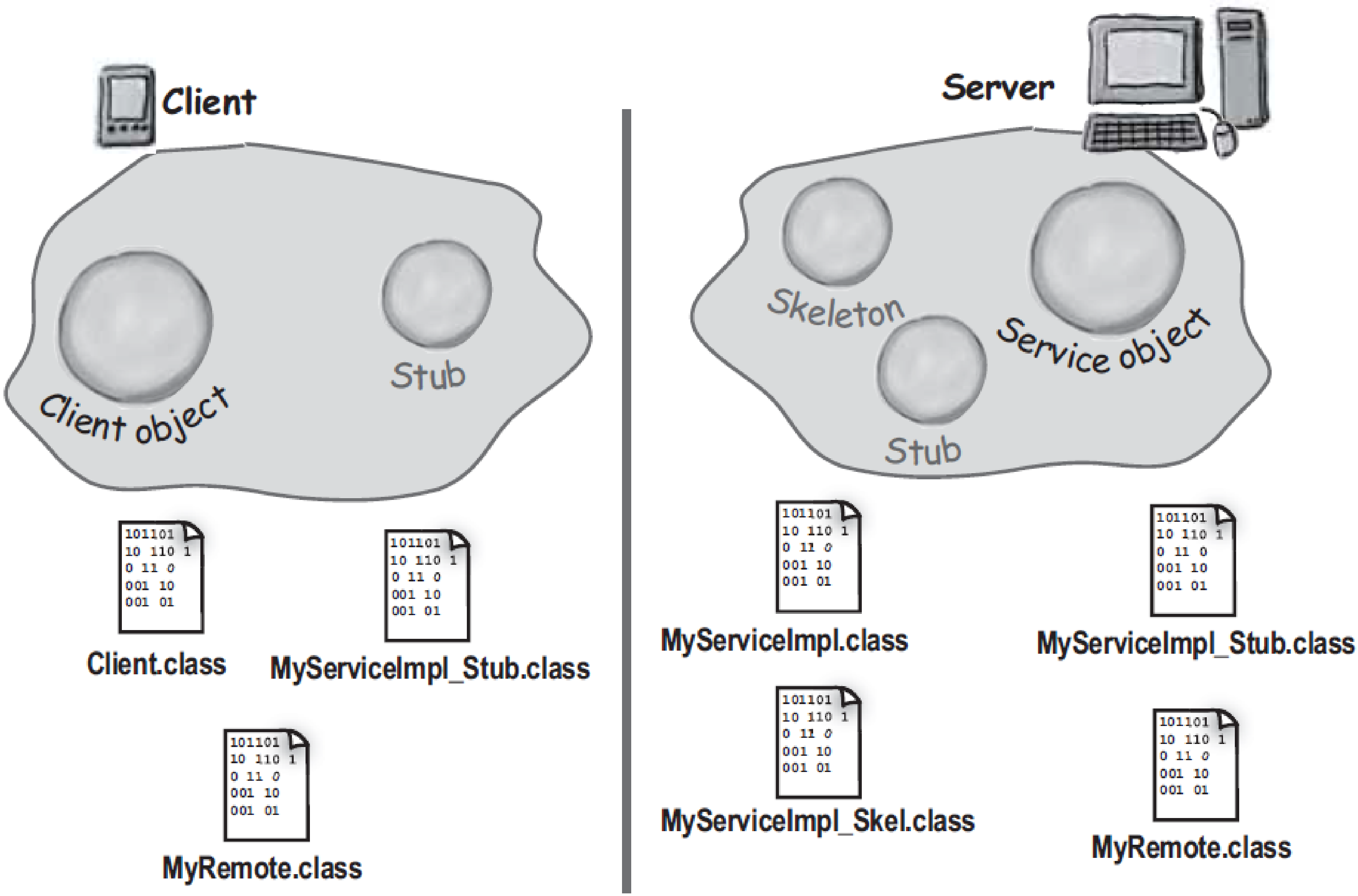
How Does Client Get the Stub Object?



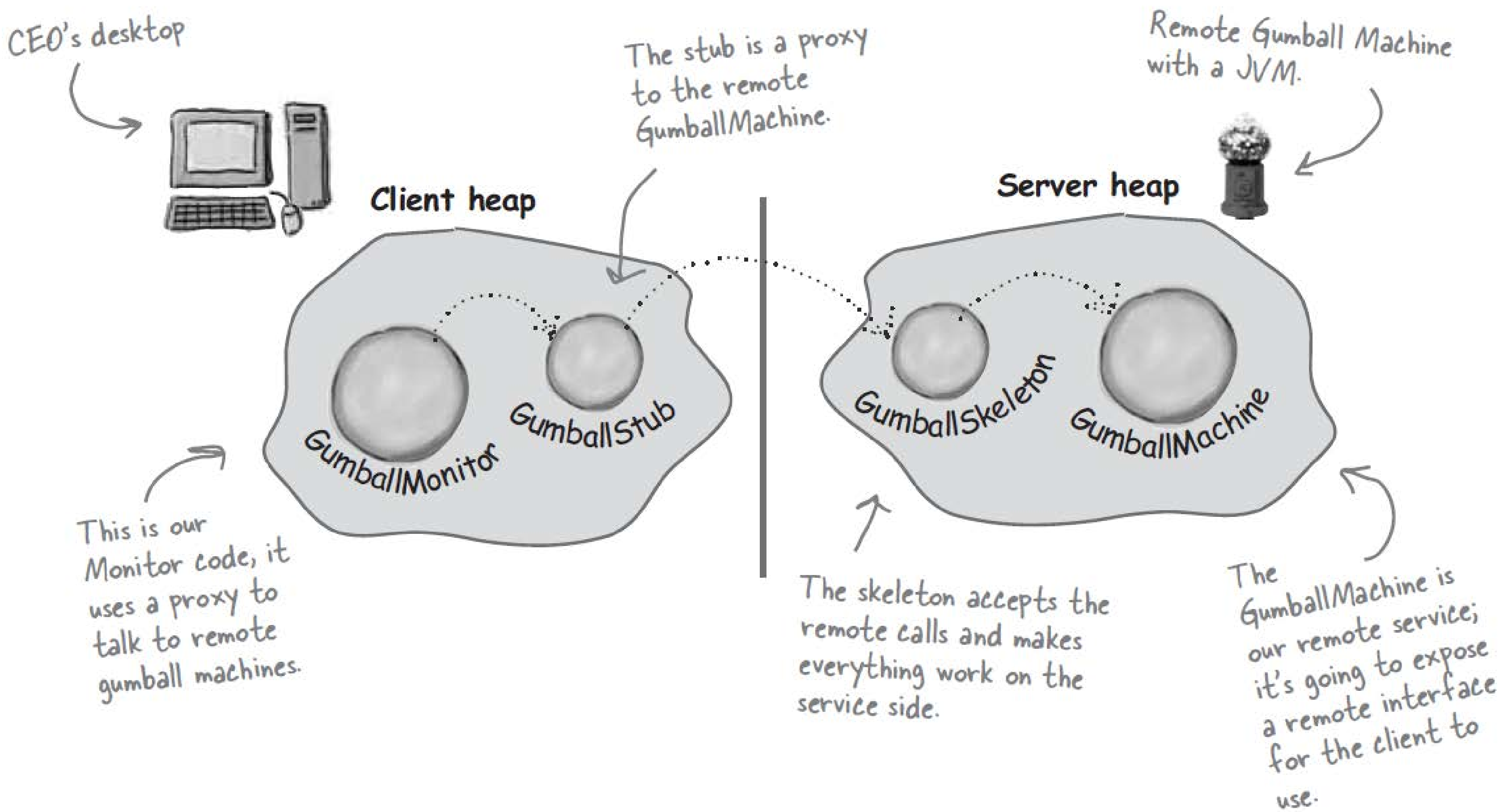
MyRemoteClient

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

After All, We Have These…

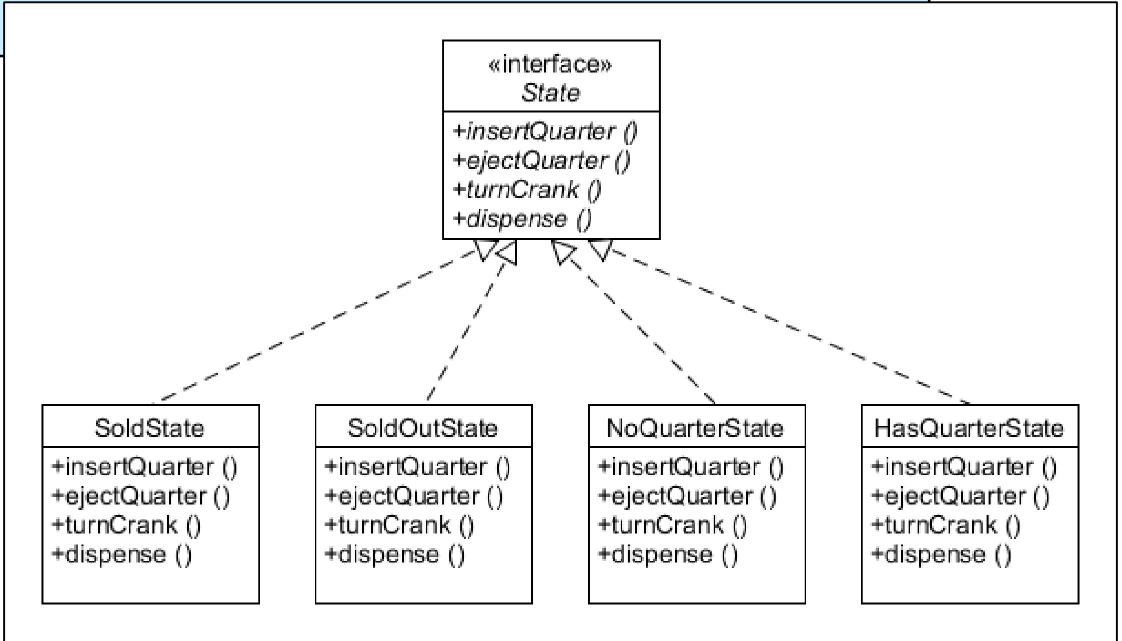


Distributed Gumball Machines



Code for Distributed Gumball Machines

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



Changes to the Concrete States

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

    public HasQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }
    public void insertQuarter() {
        System.out.println("You can't insert another quarter");
    }
    public void ejectQuarter() {
        System.out.println("Quarter returned");
        gumballMachine.setState(gumballMachine.getNoQuarterState());
    }
    public void turnCrank() {
        System.out.println("You turned...");
        gumballMachine.setState(gumballMachine.getSoldState());
        gumballMachine.dispense();
    }
    public void dispense() {
        System.out.println(“No Gumball dispensed");
    }

}
```

Code for Distributed Gumball Machines

```
Gumball Machine (Service Implementation)
import java.rmi.*;
import java.rmi.server.*;
public class GumballMachine
        extends UnicastRemoteObject implements GumballMachineRemote
{
    // instance variables
    public GumballMachine(String location, int numberGumballs)
        throws RemoteException {
        // constructor body
    }
    public int getCount() { return count; }
    public State getState() { return state; }
    public String getLocation() { return location; }
    public void insertQuarter() { state.insertQuarter(); }
    // other methods
}
```

Serialization basics

- **Serialization**
  - the process of **transforming** an **in-memory** object to a **byte stream**.
- **Deserialization**
  - the inverse process of reconstructing an object from a byte stream to the same state in which the object was previously serialized.
- For an object to be serializable
  - its class or some ancestor must implement the *empty* **`Serializable`** interface.
- An empty interface is called a *marker interface*.

Object graphs and transient fields

- If an object has references to other objects or arrays, the entire *object graph* is serialized when the object is serialized.
  - The object graph consists of the **object directly serialized and any other objects or arrays to which the object has** direct or indirect **references**.

- A field marked as **`transient`** is not impacted by serialization.
  - During deserialization, **`transient`** fields are restored to their *default* values (e.g., transient numeric fields are restored to zero).

Registering with the RMI Registry

```
import java.rmi.*;
public class GumballMachineTestDrive {
  public static void main(String[] args) {
    if (args.length < 2) {
      System.out.println("GumballMachine <name> <inventory>");
      System.exit(1);
    }
    GumballMachineRemote gumballMachine = null;
    int count;
    try {
      count = Integer.parseInt(args[1]);
      gumballMachine = new GumballMachine(args[0], count);
      Naming.rebind("//"+ args[0]+"/gumballmachine", gumballMachine);
    } catch (Exception e) { e.printStackTrace(); }
  }
}
```

Gumball Monitor Client

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

Gumball Monitor TestDrive

```
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();
  }
}
```

Starting Gumball Machines

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!

Results

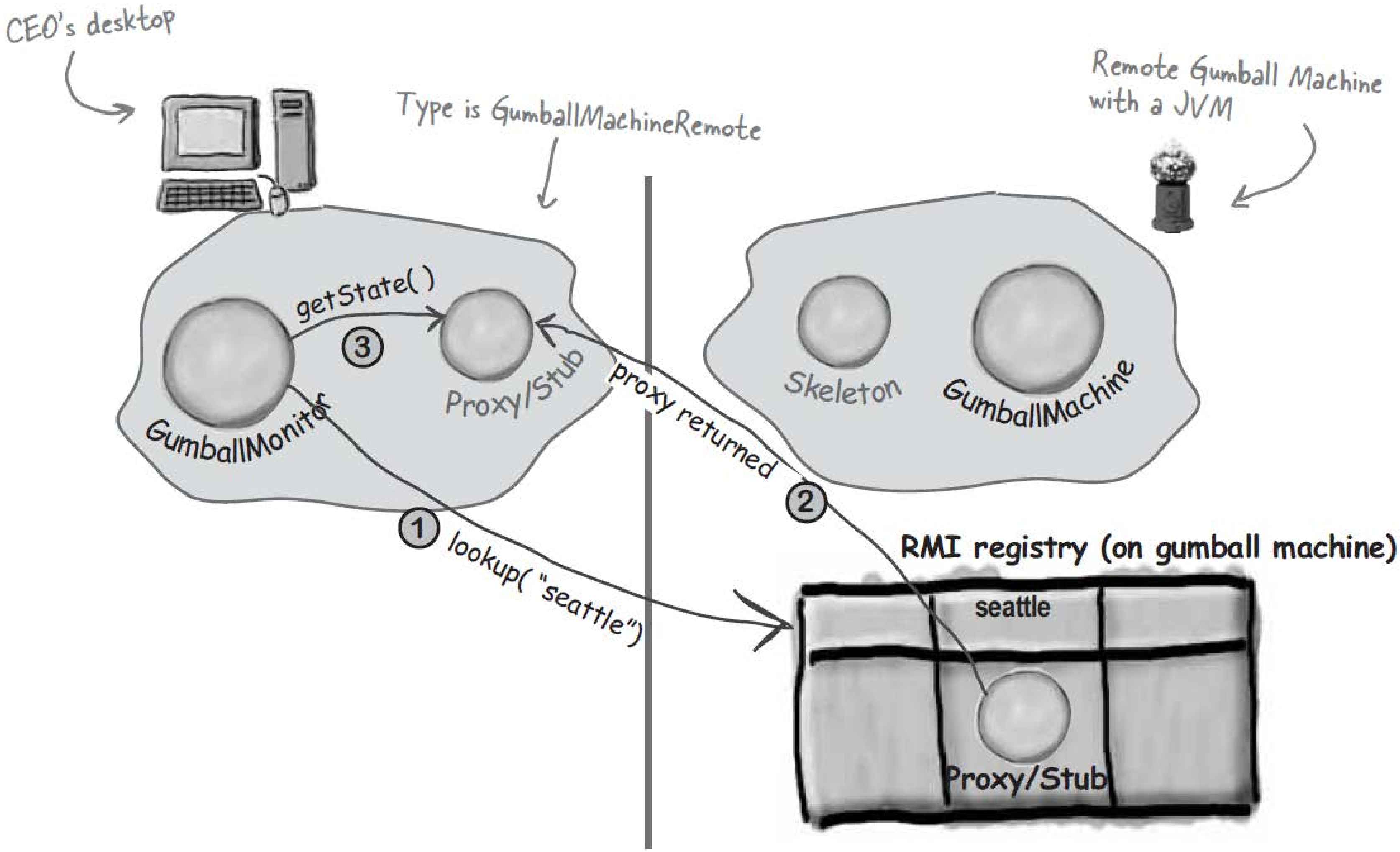
File Edit Window Help GumballsAndBeyond

```
% 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.

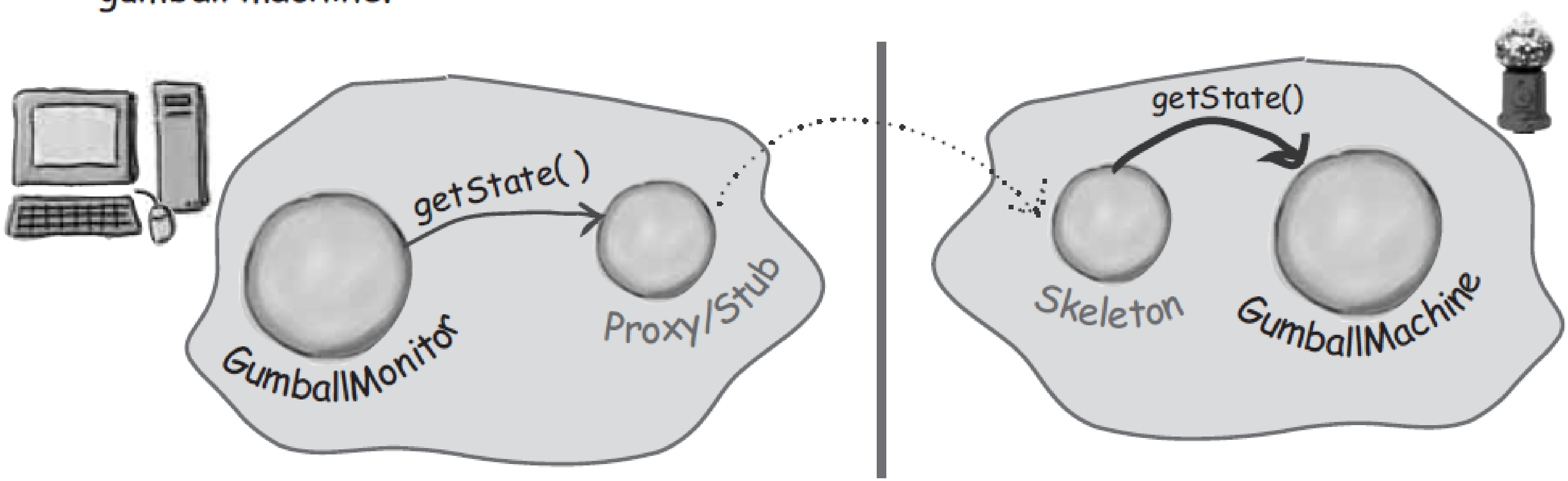
This is amazing; it's going to revolutionize my business and blow away the competition!

Behind the Scenes



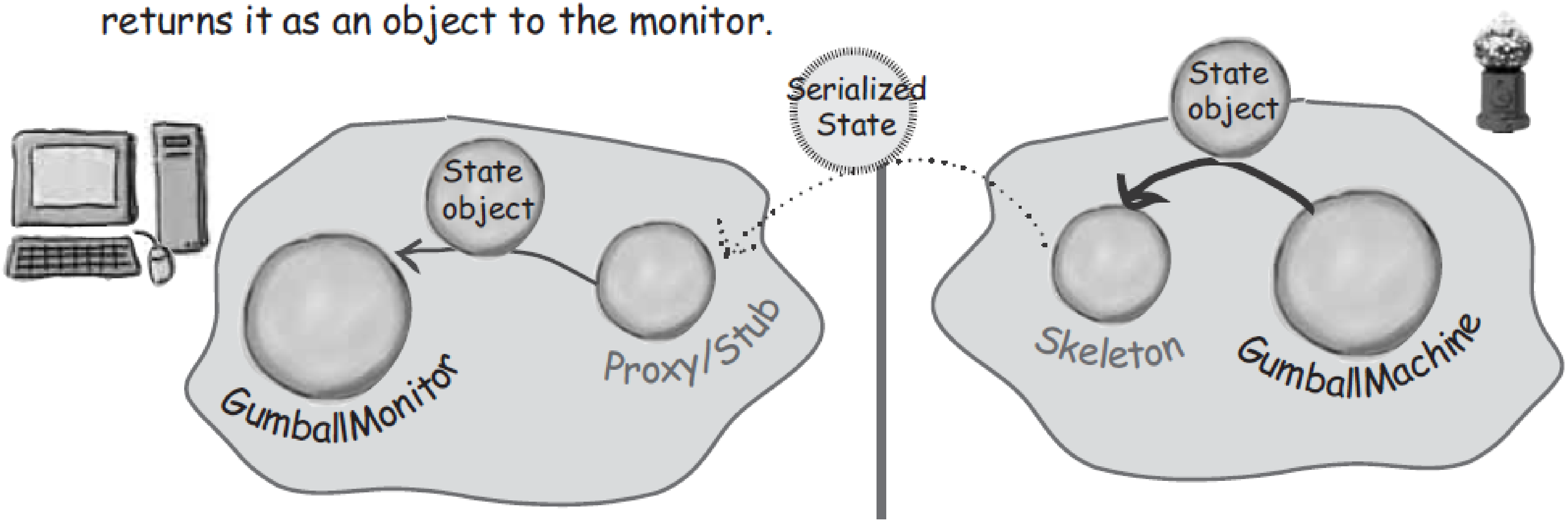
Behind the Scenes

❷ 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.



Behind the Scenes

❸ 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.



Related Patterns

- w.r.t. Interface
  - Adapter provides a different interface to its subject
  - Proxy provides the same interface
  - Decorator provides an enhanced interface

- w.r.t. Structure
  - Decorator and Proxy have similar structures
  - Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests