Proxy

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 27

Acknowledgement & Materials Copyright

- I’d like to start by acknowledging Dr. Ken Anderson
- Ken is a Professor and the Chair of the Department of Computer Science
- Ken taught OOAD on several occasions, and has graciously allowed me to use his copyrighted material for this instance of the class

Head First Design Patterns

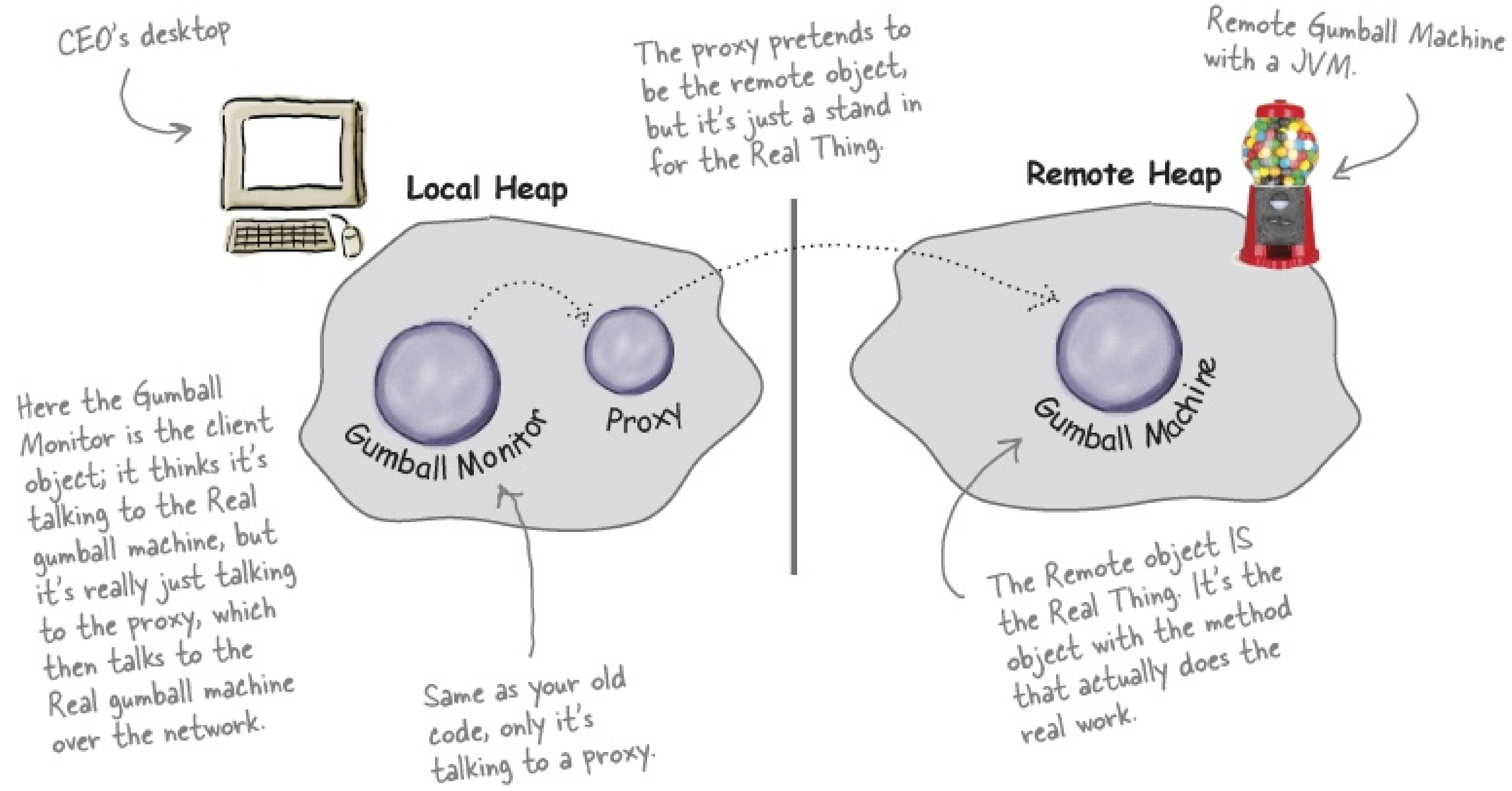
- This material is from Chapter 11 of the Head First book, and it continues using the Gumball Machines from the Chapter 10 State examples

Local monitoring approach

- The initial request is to monitor the state of Gumball Machines (for count and current state)
- The initial code pass creates
    - An updated GumballMachine class that knows its location, its gumball count, and the current state
        - Public methods are provided for those elements
    - A new GumballMonitor class that is given a reference to a machine, and is then able to report() on the values of location, count, and state for that machine instance
    - A test frame that creates both a GumballMachine and a GumballMonitor object, and has the monitor call its report() method

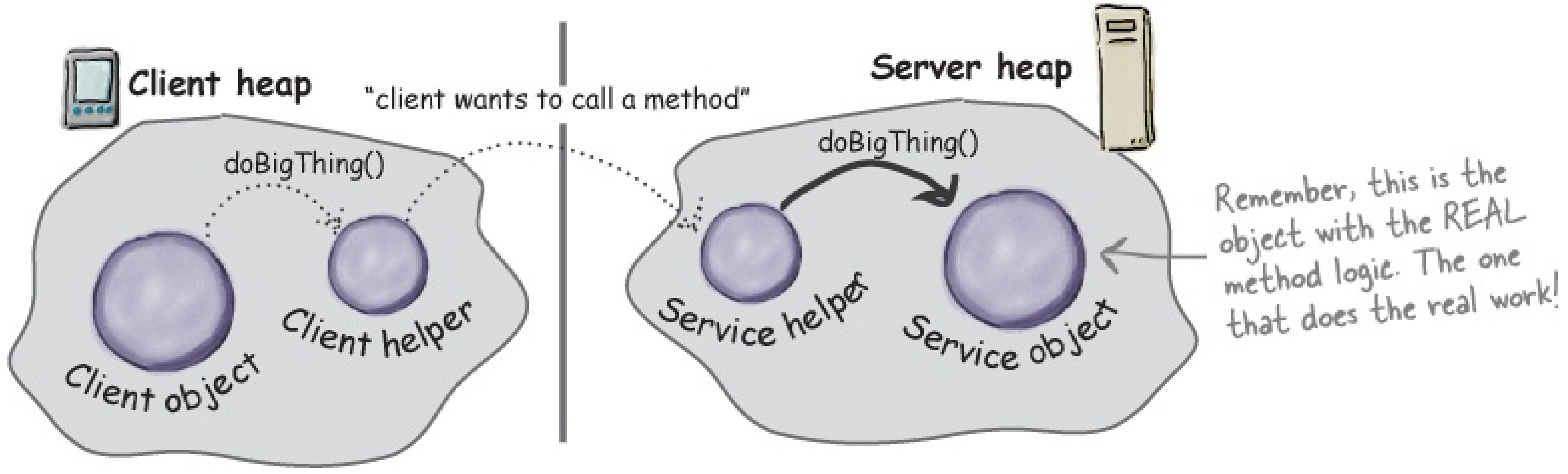
Extension to "Remote" Proxy

- In this case, the GumballMachine is a remote object, living in the heap of a different JVM somewhere else
- To talk to that remote object, we want to interact with a proxy


CEO's desktop
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.
Local Heap
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.

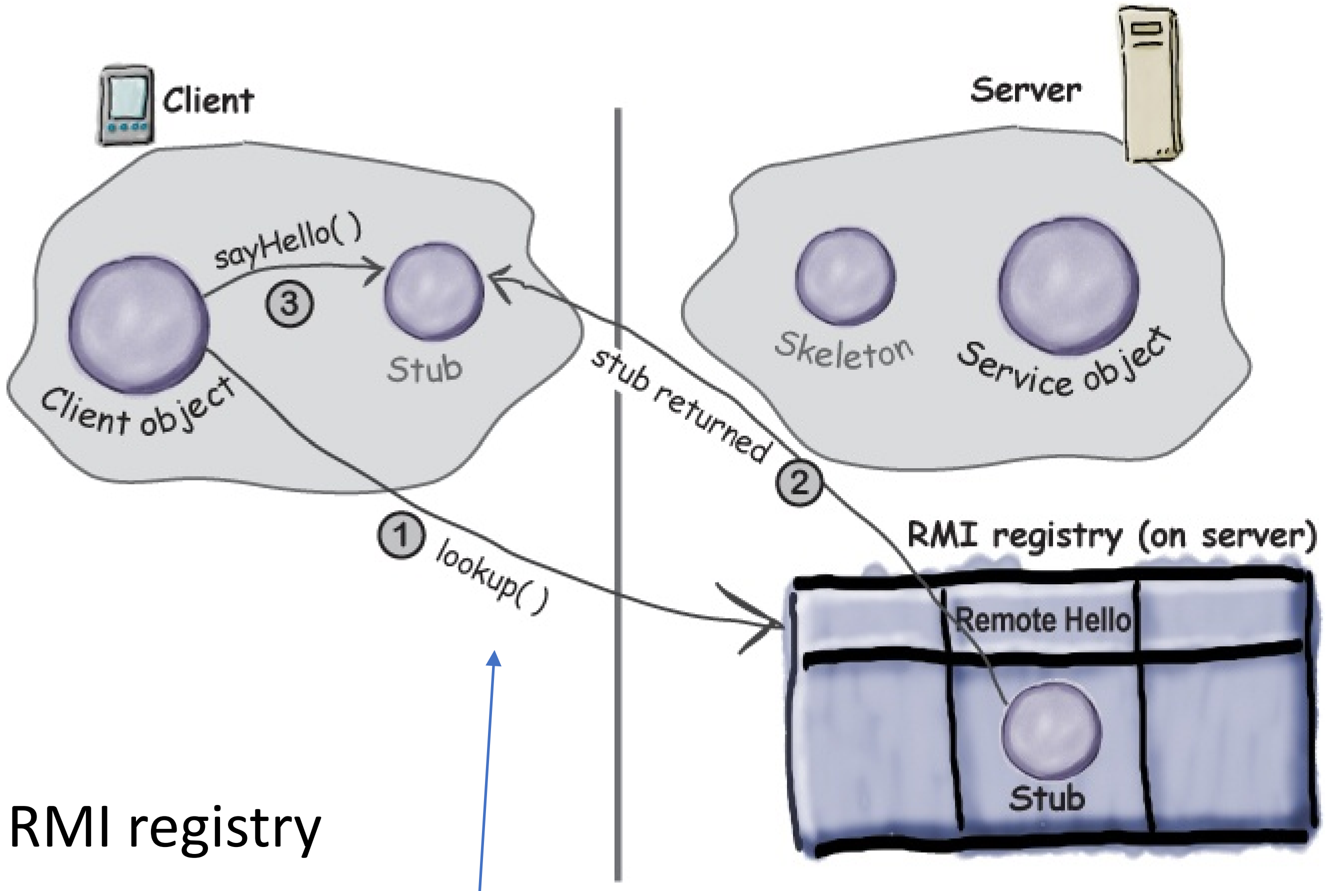
Java RMI – Remote Method Invocation

- RMI provides a client helper and a service helper object, connected via a Socket connection to allow a client object to call methods on a remote service object in a remote memory space



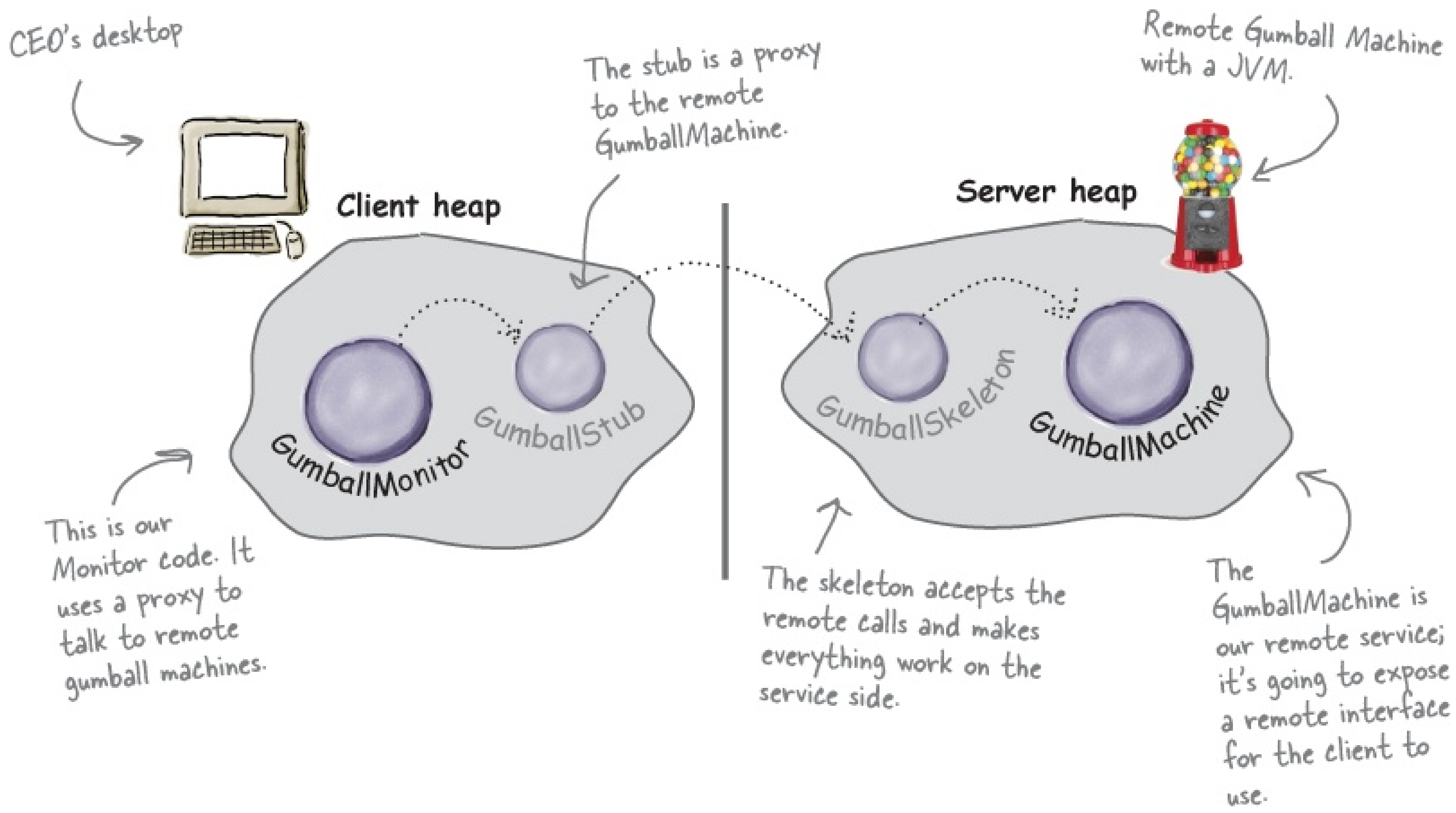
Client heap
"client wants to call a method"
Server heap
doBigThing()
doBigThing()
Client object
Client helper
Service helper
Service object
Remember, this is the object with the REAL method logic. The one that does the real work!

Java RMI Steps

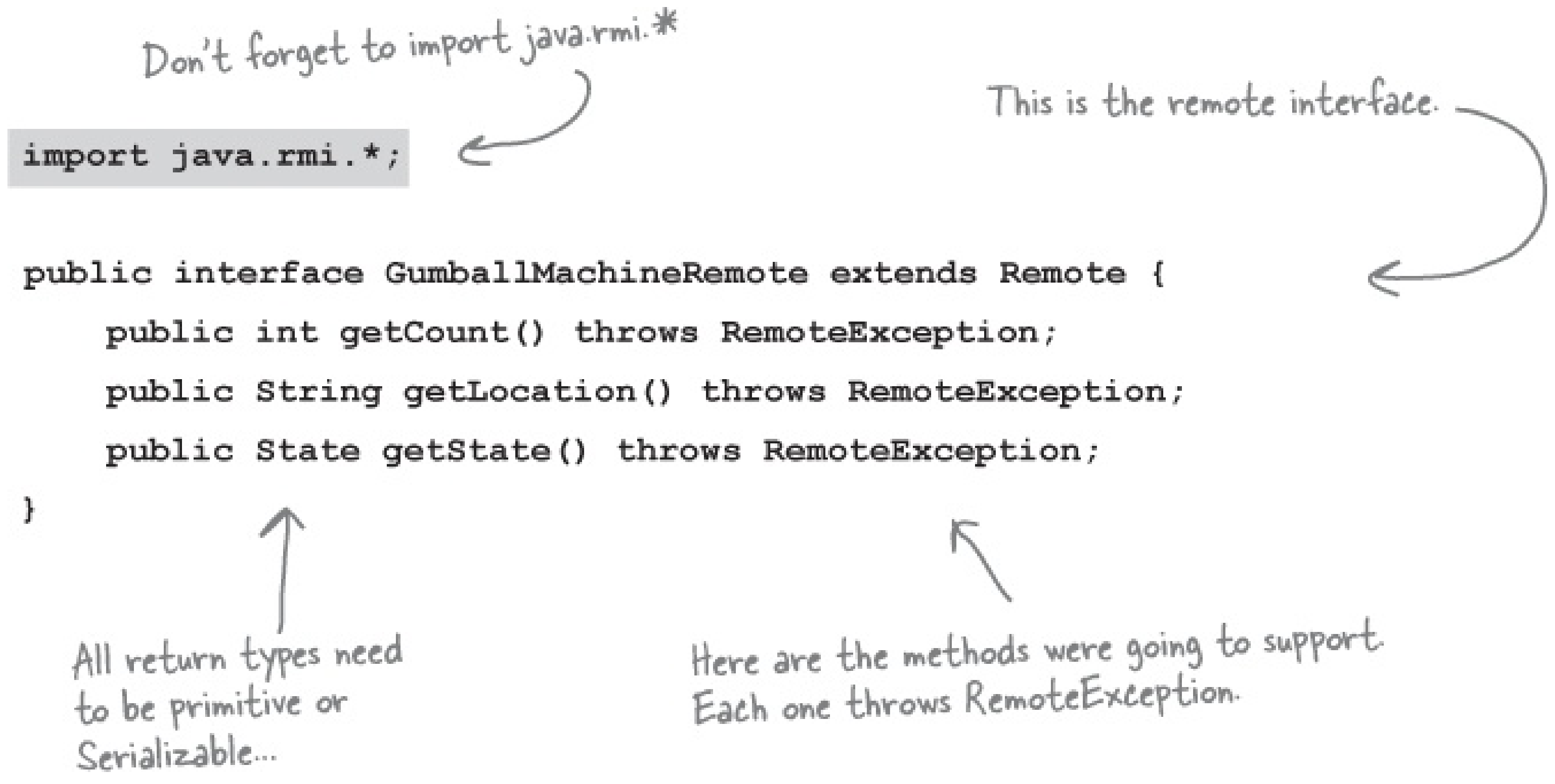


- Make remote interface
- Implement the remote server
  - Create and export a remote object
  - Extend UnicastRemoteObject
  - Register the remote object with a Java RMI registry
- Run rmiregistry on the server's host
- Start remote server
- Implement the client
  - Obtain stub for registry on server host
  - Look up the remote object's stub by name in the registry
  - Invoke remote method on remote object via the stub
- Network communication is subject to failure and exceptions
- Remote methods should throw RemoteException
- Data being returned or passed should be primitives or Serializable objects
- https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/hello/hello-world.html

After RMI


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.

Careful…



Don't forget to import java.rmi.*

This is the remote interface.

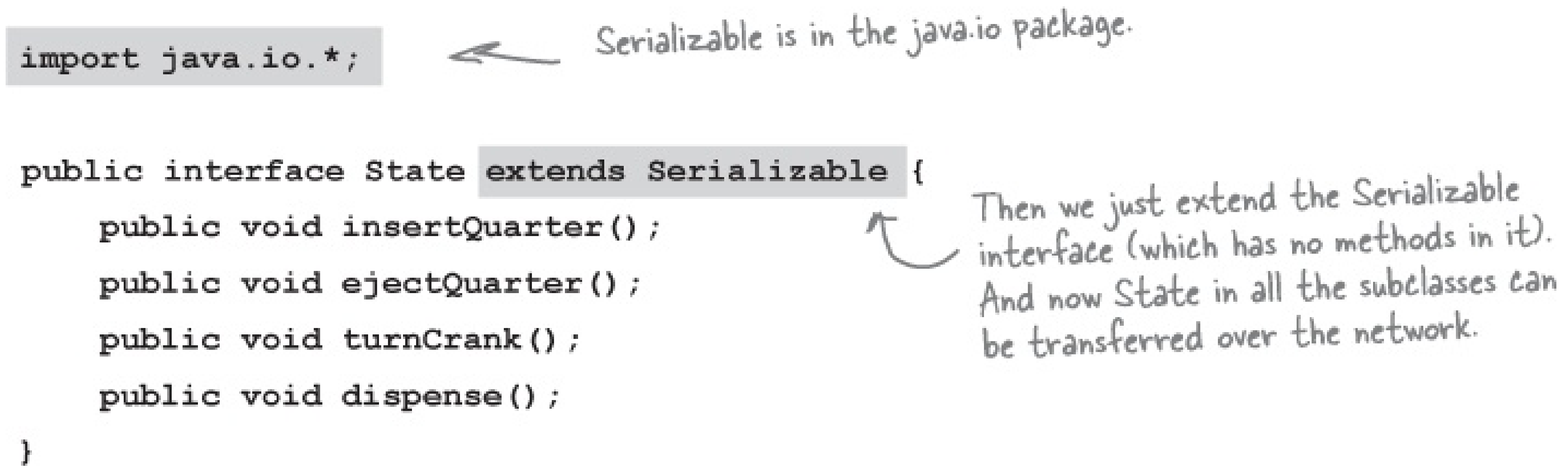
```
import java.rmi.*;

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.

- We want to pass state from the remote server, but there are two issues. First, State must be Serializable if we are to pass it from a remote method, so…



Serializable is in the java.io package.

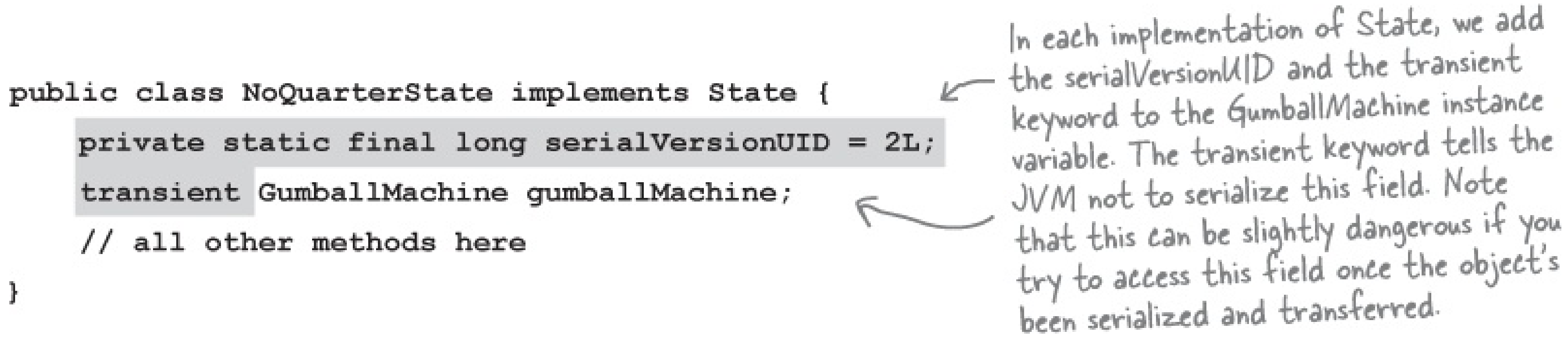
```
import java.io.*;

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.

One more issue…

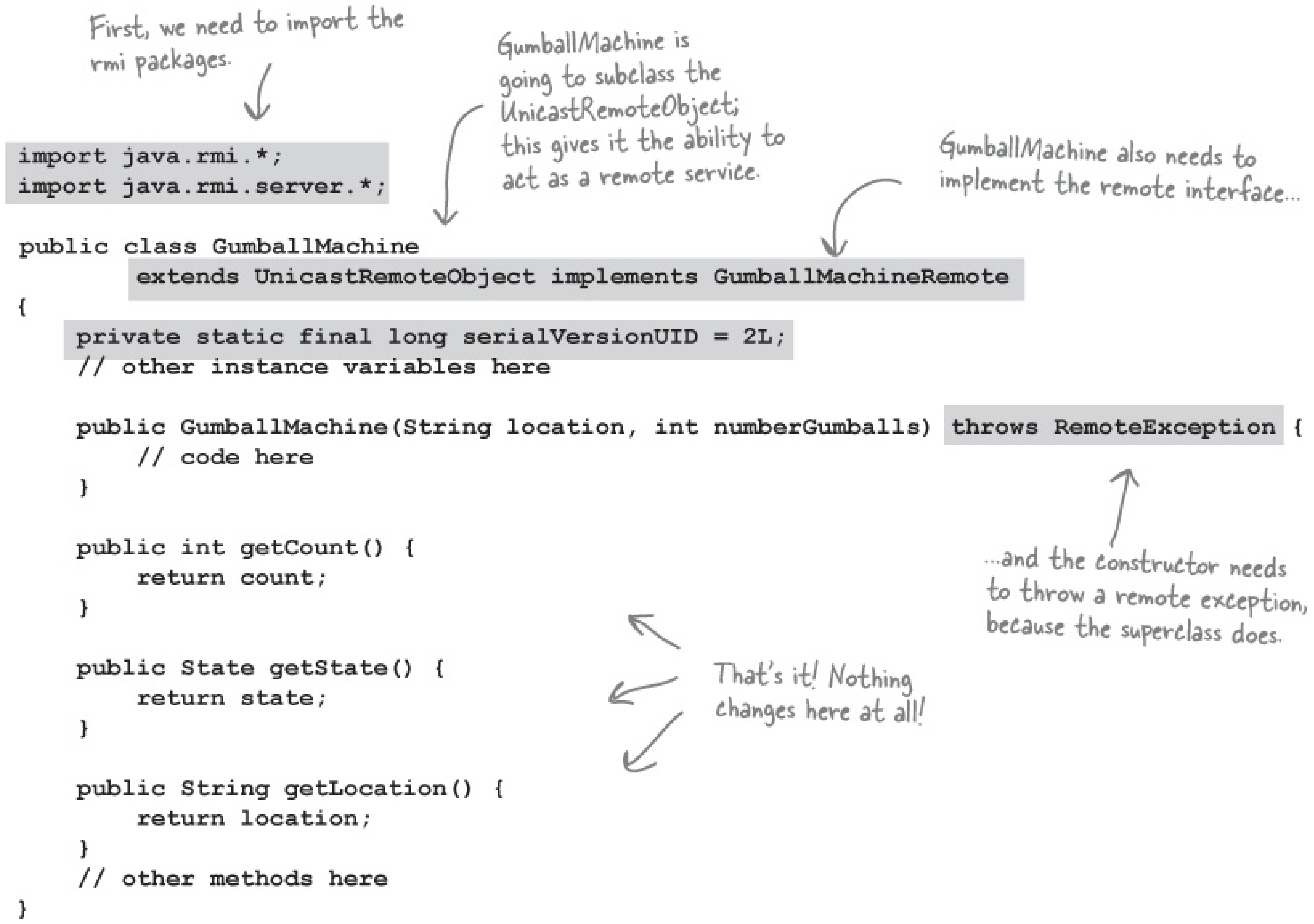
- State objects have a reference to a GumballMachine so it can call that machine’s methods and change it’s state
- We don’t want the entire gumball machine serialized…



```
public class NoQuarterState implements State {
    private static final long serialVersionUID = 2L;
    transient GumballMachine gumballMachine;
    // all other methods here
}
```

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

A remoteable GumballMachine class…



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
{
    private static final long serialVersionUID = 2L;
    // other 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!

Registering the remote service with RMI

- When the remote GumballMachine starts, it has to register with the RMI Registry

```
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.

Minor client-side changes for Monitor

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

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

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.

Now we can connect to several proxies…

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

Here's all the locations were going to monitor.

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

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

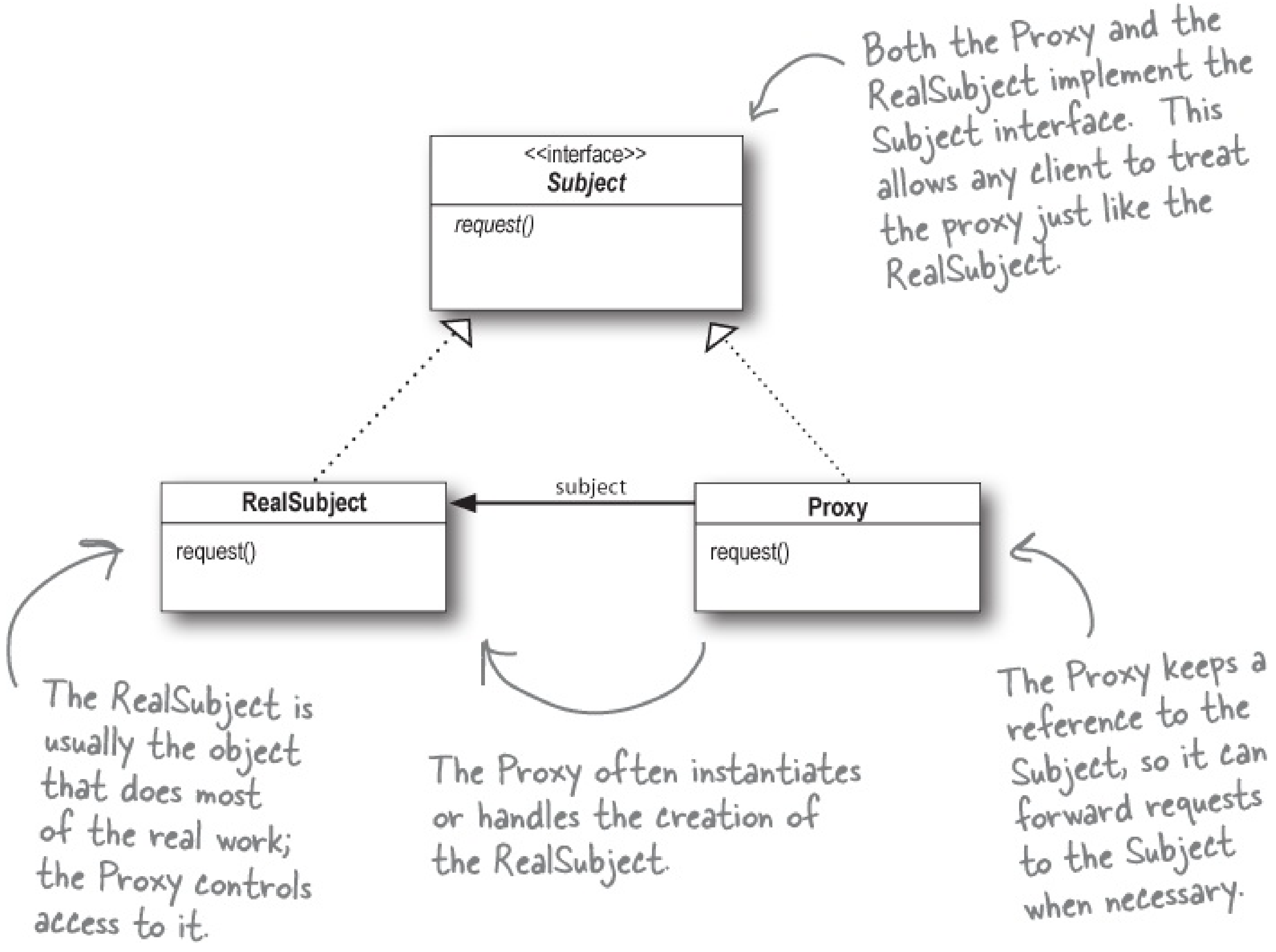
We also create an array of monitors.

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

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

The Proxy Pattern

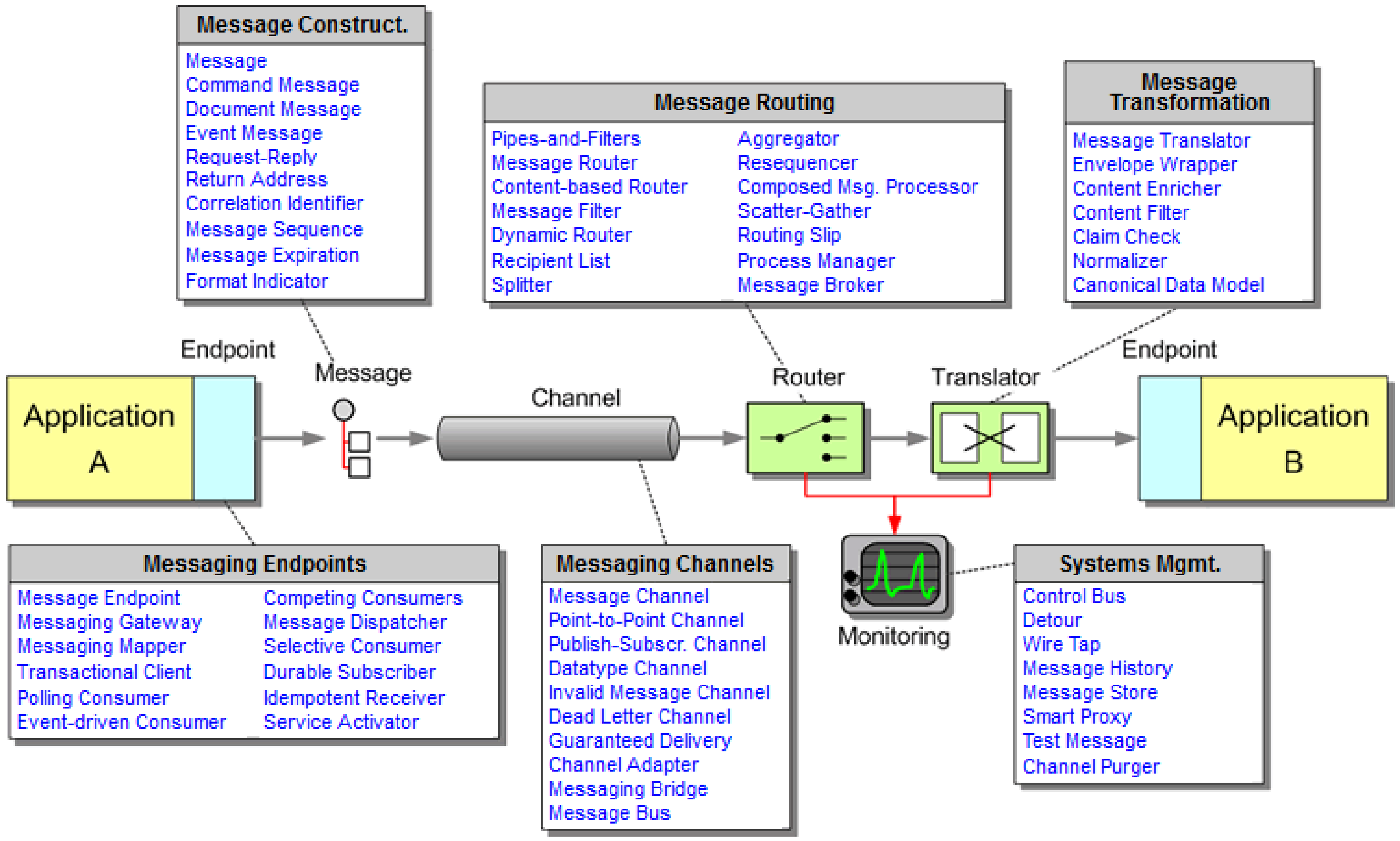
- The Proxy Pattern provides a surrogate or placeholder for another object to control access to it.
- Use the Proxy Pattern to create a representative object that controls access to another object
  - which may be remote (remote proxy) – what we just did
  - expensive to create (virtual proxy)
  - or in need of securing (protection proxy)



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

Alternatives to RMI – Messaging Patterns

- Messaging – publish/subscribe, point to point, request/response
- The Enterprise Integration Patterns book has defined 65 messaging patterns
- http://www.enterpriseintegrationpatterns.com/patterns/messaging/index.html
- Typical Message Queuing Tools
  - ZeroMQ
  - RabbitMQ
  - AWS SQS



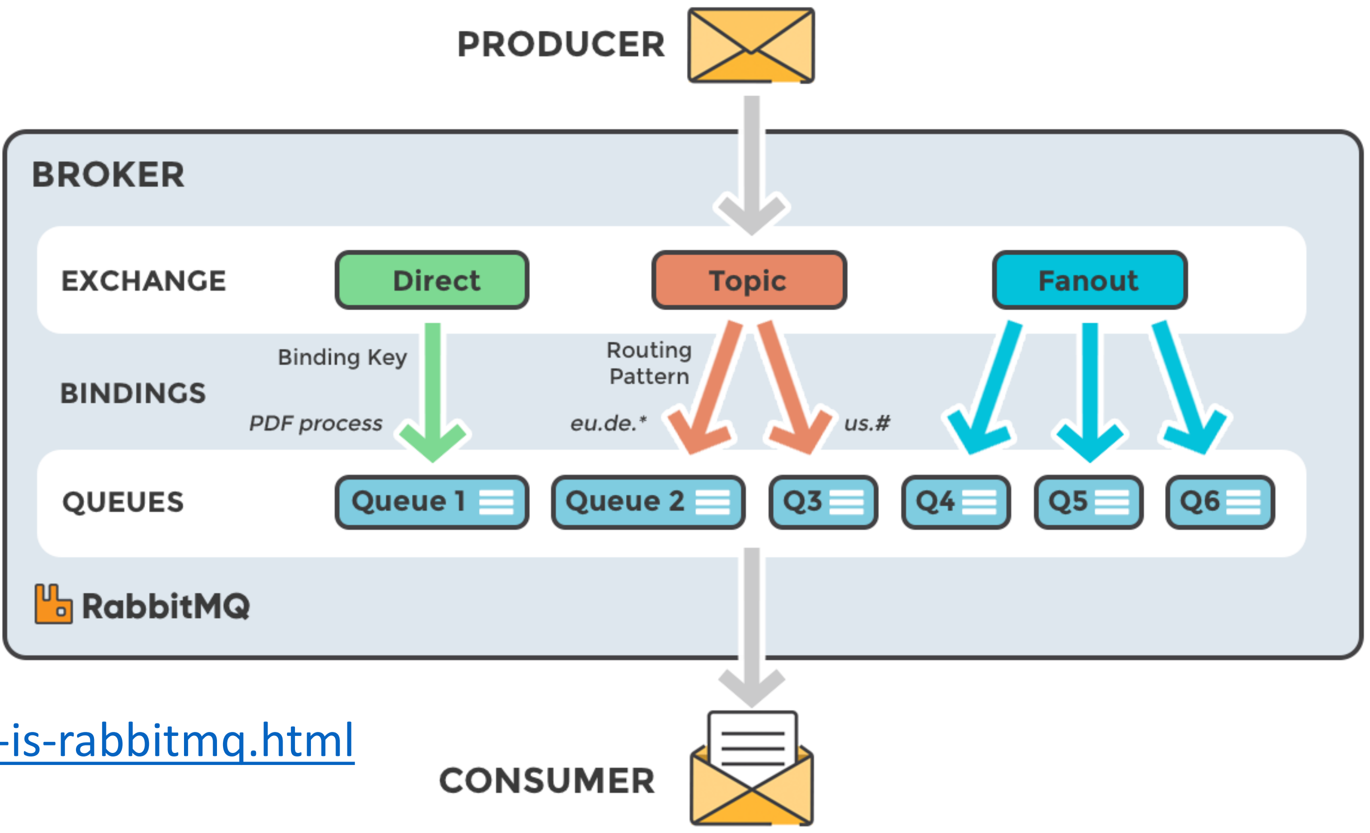
ZeroMQ

ØMQ

- 0MQ was initially developed in 2007, and is now an open-source project [5]
- Provides **sockets** that carry atomic messages across various transports (in-process, inter-process, TCP, and multicast)
- Designed to be an ultra-simple API (Application Programming Interface) based on BSD sockets.
- Implements real messaging patterns like topic pub-sub, fan-out, and request-response.
- Available for most programming languages, operating systems, and hardware. It provides a single consistent model for all language APIs; It is simple to learn and use, with a learning curve of roughly one hour
- Designed as a library linked with applications - **no brokers** to start and manage, fewer moving pieces - less to break and go wrong - low CPU footprint
- Licensed as LGPL code
- https://zeromq.org/

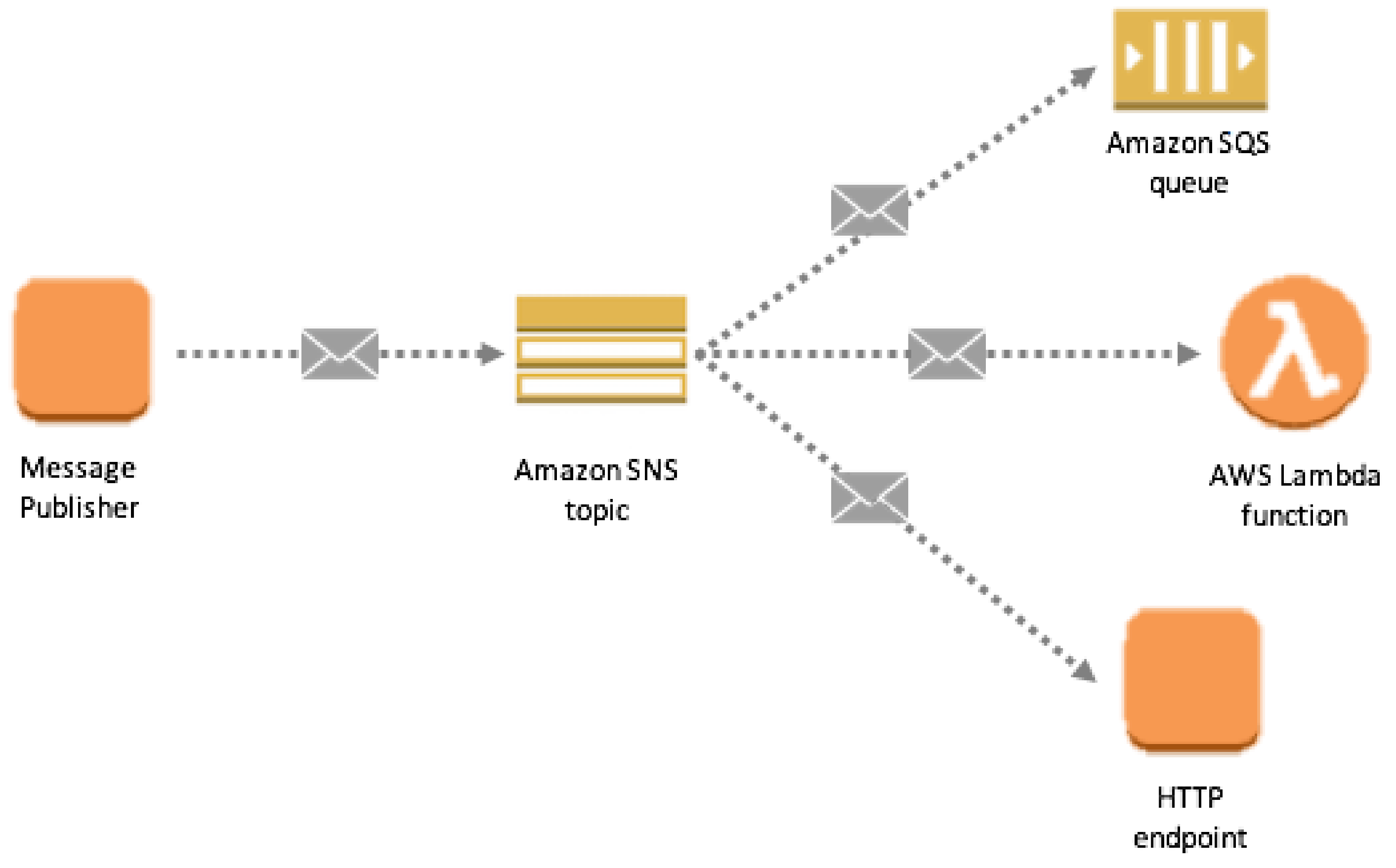
RabbitMQ

- “Most Widely Deployed Open Source Message Broker”
- RabbitMQ is **a message broker**, containing and managing queues for messages, producers, and consumers
- Base RabbitMQ uses asynchronous messaging with AMQP 0.9.1, but it supports multiple alternative protocols including STOMP, MQTT, AMQP 1.0, and HTTP
- Support for most programming languages
- Supports work queues, pub/sub, request/reply, and other messaging architectures
- Supports TLS & LDAP for authentication and authorization
- Has a management and monitoring infrastructure – HTTP-API
- Reference https://www.rabbitmq.com/ , https://www.cloudamqp.com/blog/2015-05-18-part1-rabbitmq-for-beginners-what-is-rabbitmq.html



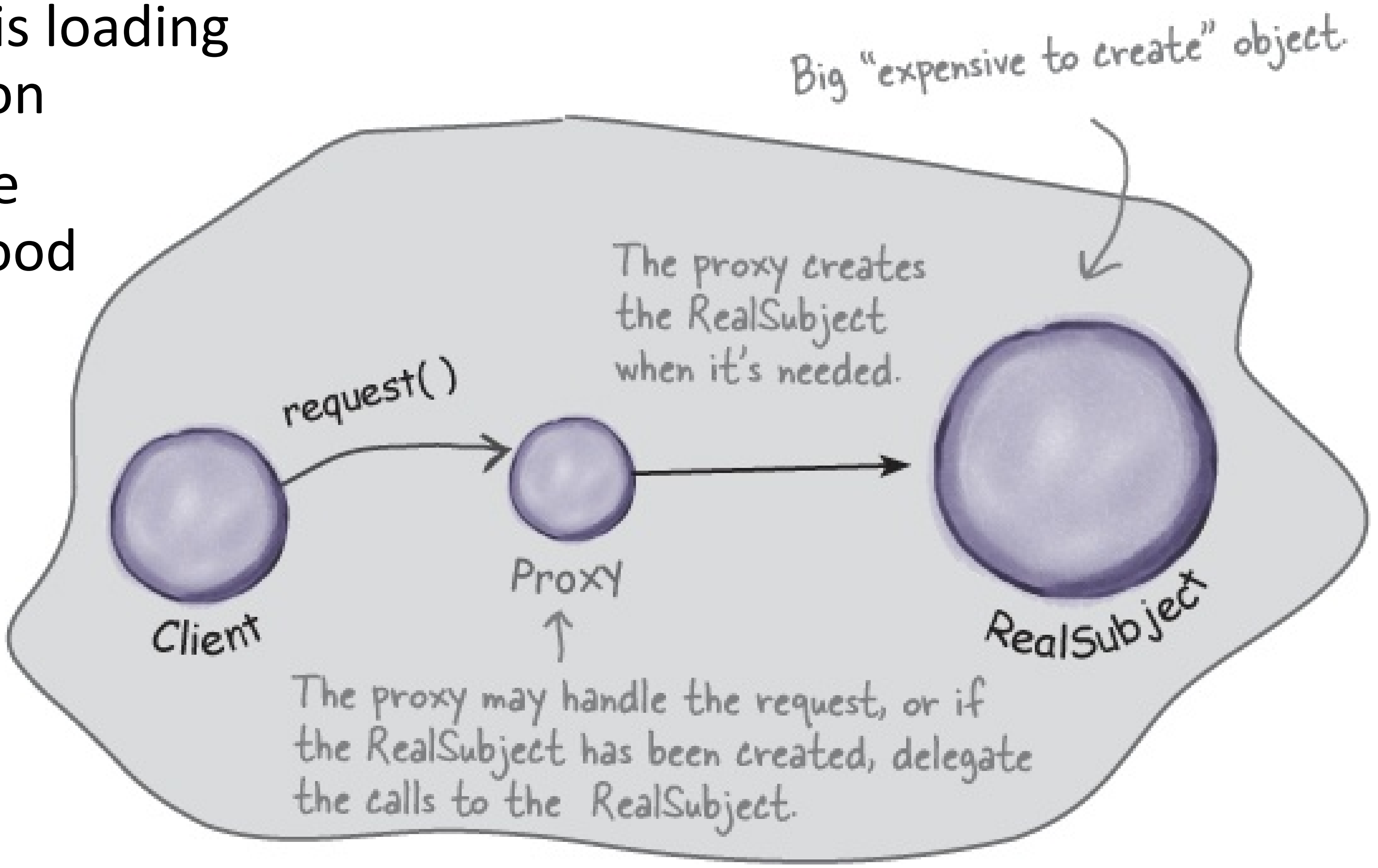
Cloud Based Message Queues – AWS SQS

- Fully managed cloud-based message queuing service
- Two types of message queues
    - Standard queues offer maximum throughput, best-effort ordering, and at-least-once delivery
    - SQS FIFO queues are designed to guarantee that messages are processed exactly once, in the exact order that they are sent
- Easily administrated, reliable
- Easy to integrate with IoT tools, AWS Lambda Serverless processes and other AWS tools
- Can integrate with server-side encryption
- Scale easily
- Reference https://aws.amazon.com/sqs/



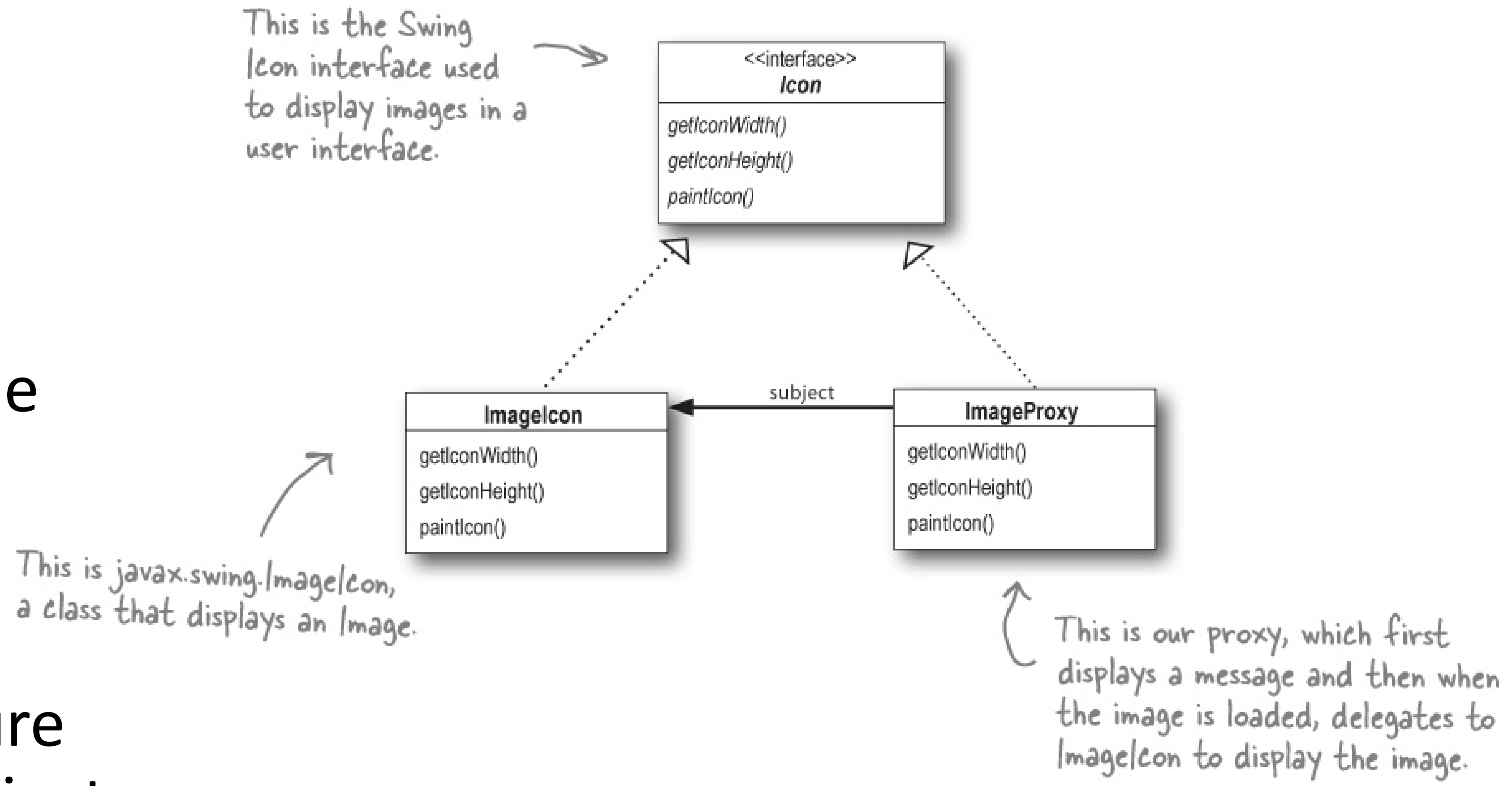
Virtual Proxy

- Virtual Proxy – why? Creating the RealSubject is “expensive”; if the Proxy can handle the issue, it will
- Example in book is loading a music album icon
- If we already have the icon, we’re good
- We only do the icon request if we need to
- And the proxy provides a temporary icon until the real one loads


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.

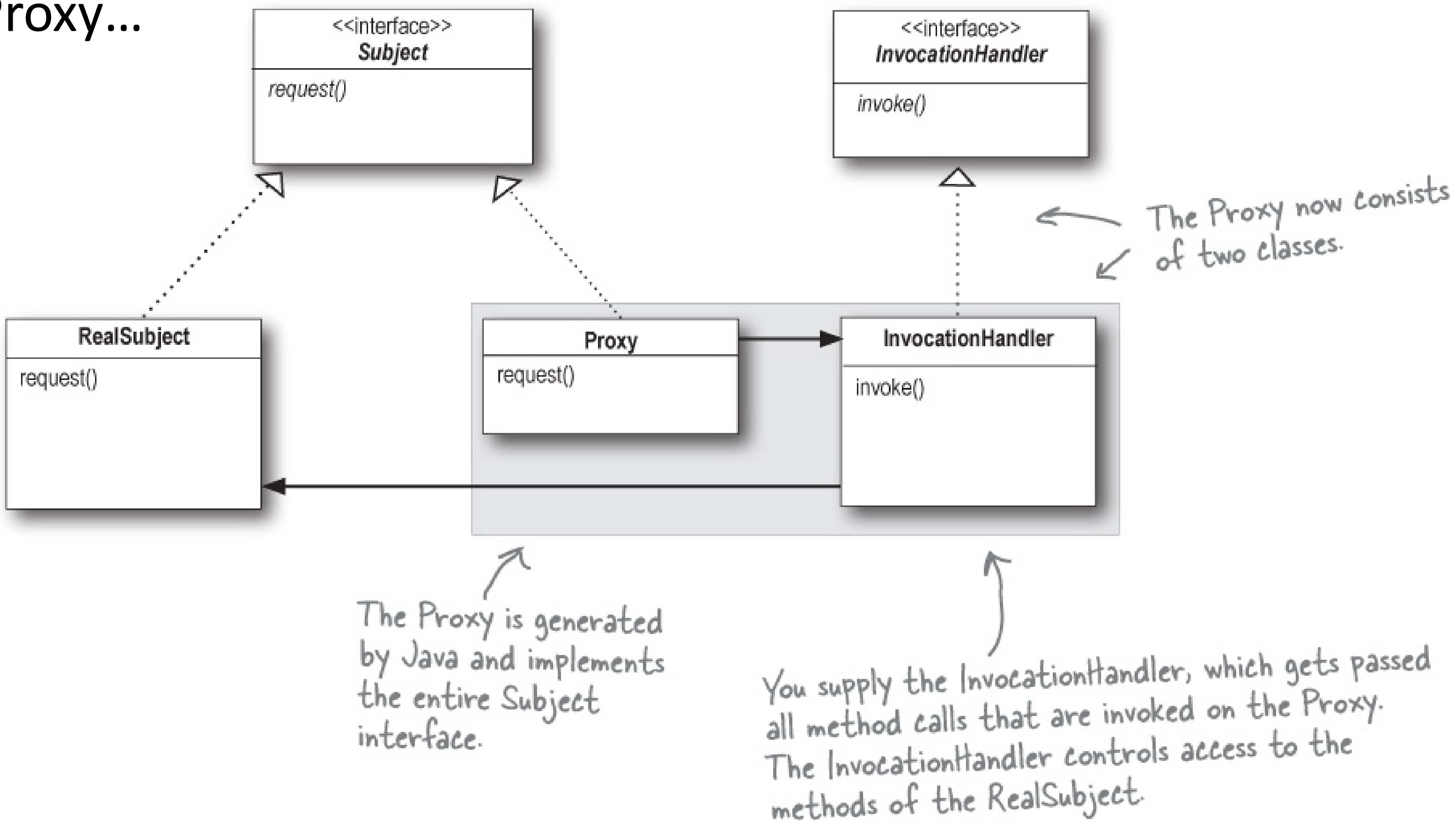
Looks a bit like Decorator…

- They do look a bit alike…
- The difference is, Decorator is extending behavior of a class, while Proxy is controlling access to the class
- You can combine Proxy with Factory to make sure clients use the Proxy object instead of instantiating the RealSubject you're controlling access to…
- Proxy also looks a bit like Adapter, since it sits in front of another object, but Proxy implements the exact same interface, Adapter changes the interface of objects it adapts…


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.

Protection Proxy

- If we use the RMI approach, Java is creating the Proxy class for you, so you need a way to tell Proxy what to do.
- For this, we create an InvocationHandler that responds to any calls on the Proxy…



<<interface>>
Subject
request()
<<interface>>
InvocationHandler
invoke()
The Proxy now consists of two classes.
RealSubject
request()
Proxy
request()
InvocationHandler
invoke()
The Proxy is generated by Java and implements the entire Subject interface.
You supply the InvocationHandler, which gets passed all method calls that are invoked on the Proxy. The InvocationHandler controls access to the methods of the RealSubject.

Creating the Protection Proxy

- The example is for a service that sets attributes on something “owned” by an individual
  - But there are some methods the individual shouldn’t be allowed to use
  - And there are some methods others accessing that individual shouldn’t be allowed to use
- Create two InvocationHandlers – one for invocations by the individual owner, one for invocations by others

Other Proxy Types

- Firewall Proxy – controls access to a set of network resources, protecting subject from bad clients
- Smart Reference Proxy – provides additional actions when a subject is referenced, such as maintaining a reference count
- Caching Proxy – a version of Virtual Proxy to maintain temporary storage that can be referenced rather than asking the subject to do an expensive task
- Synchronization Proxy – provide safe access to a subject from multiple threads
- Complexity Hiding (or Façade) Proxy – Unlike Façade, this proxy provides both an alternative interface and access control
- Copy-on-Write Proxy – Controls copying an object by deferring copy until required by a client (another Virtual Proxy variant)

Python Proxy?

A Smart Reference Proxy implementation…

Subjects…

```
from abc import ABC, abstractmethod

class Subject(ABC):
    # The Subject interface declares common operations for both RealSubject and
    # the Proxy. As long as the client works with RealSubject using this
    # interface, you'll be able to pass it a proxy instead of a real subject.

    @abstractmethod
    def request(self) -> None:
        pass

class RealSubject(Subject):
    # The RealSubject contains some core business logic. Usually, RealSubjects are
    # capable of doing some useful work which may also be very slow or sensitive -
    # e.g. correcting input data. A Proxy can solve these issues without any
    # changes to the RealSubject's code.

    def request(self) -> None:
        print("RealSubject: Handling request.")
```

Python Proxy?

Proxy and Client…

```
class Proxy(Subject):
    # The Proxy has an interface identical to the RealSubject.

    def __init__(self, real_subject: RealSubject) -> None:
        self._real_subject = real_subject

    def request(self) -> None:
        # The most common applications of the Proxy pattern are lazy loading,
        caching, controlling the access, logging, etc.

        if self.check_access():
            self._real_subject.request()
            self.log_access()

    def check_access(self) -> bool:
        print("Proxy: Checking access prior to firing a real request.")
        return True

    def log_access(self) -> None:
        print("Proxy: Logging the time of request.", end="")

def client_code(subject: Subject) -> None:
    # The client code is supposed to work with all objects (both subjects and
    # proxies) via the Subject interface in order to support both real subjects
    # and proxies. In real life, however, clients mostly work with their real
    # subjects directly. In this case, to implement the pattern more easily, you
    # can extend your proxy from the real subject's class.

    subject.request()
```

Python Proxy!

Main and Runtime

Example from:

https://refactoring.guru/design-patterns/proxy/python/example

```
if __name__ == "__main__":
   print("Client: Executing the client code with a real subject:")
   real_subject = RealSubject()
   client_code(real_subject)

   print("")

   print("Client: Executing the same client code with a proxy:")
   proxy = Proxy(real_subject)
   client_code(proxy)
```

Output:

```
Client: Executing the client code with a real subject:
RealSubject: Handling request.

Client: Executing the same client code with a proxy:
Proxy: Checking access prior to firing a real request.
RealSubject: Handling request.
Proxy: Logging the time of request.
```

Key Points

- The Proxy Pattern provides a representative for another object in order to control the client’s access to it. There are a number of ways it can manage that access.
- A Remote Proxy manages interaction between a client and a remote object.
- A Virtual Proxy controls access to an object that is expensive to instantiate.
- A Protection Proxy controls access to the methods of an object based on the caller.
- Many other variants of the Proxy Pattern exist including caching proxies, synchronization proxies, firewall proxies, copy-on-write proxies, and so on.
- Proxy is structurally similar to Decorator, but the two differ in their purpose.
- The Decorator Pattern adds behavior to an object, while a Proxy controls access.
- Java’s built-in support for Proxy can build a dynamic proxy class on demand and dispatch all calls on it to a handler of your choosing.
- Like any wrapper, proxies will increase the number of classes and objects in your designs.