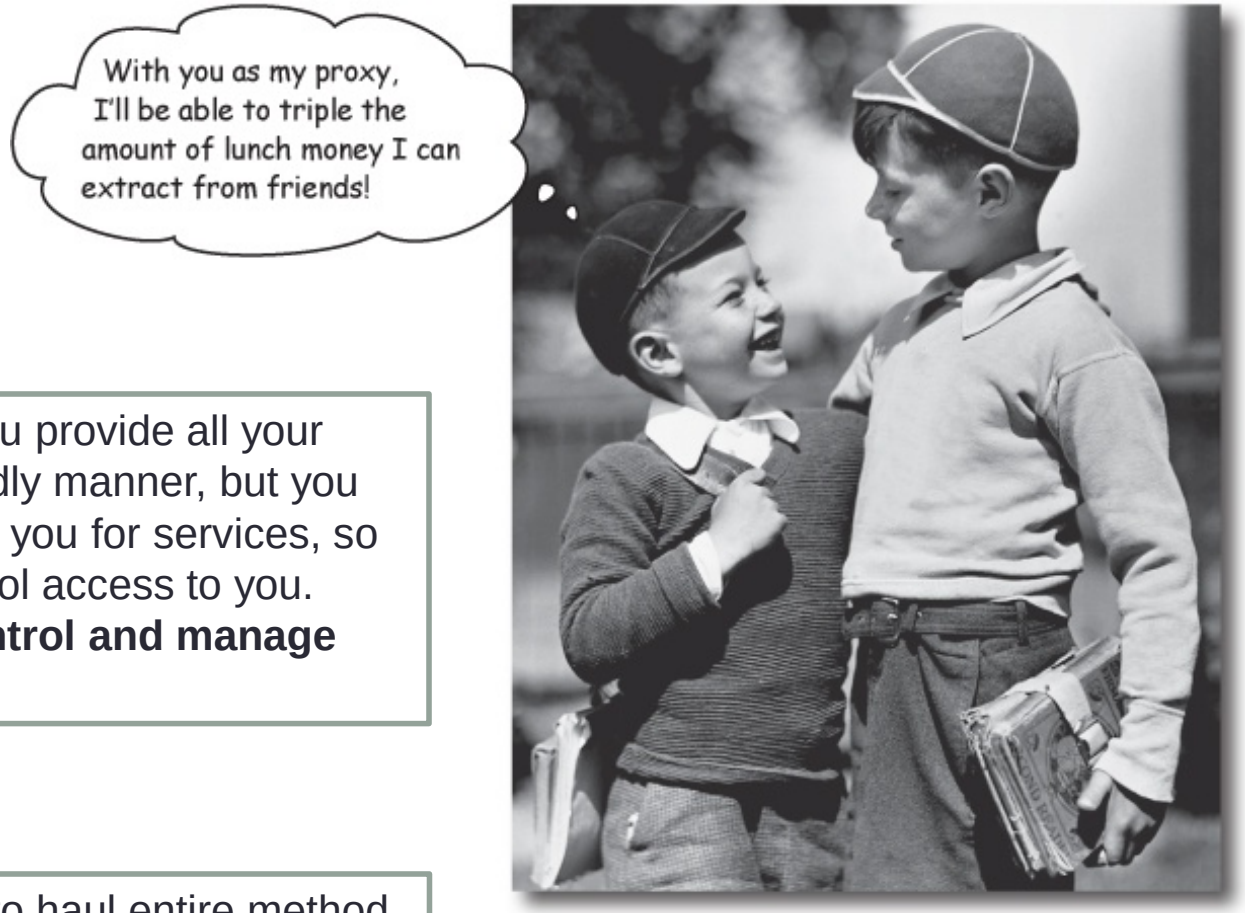
THE PROXY PATTERN

Chandan R. Rupakheti

Week 8-2

Today



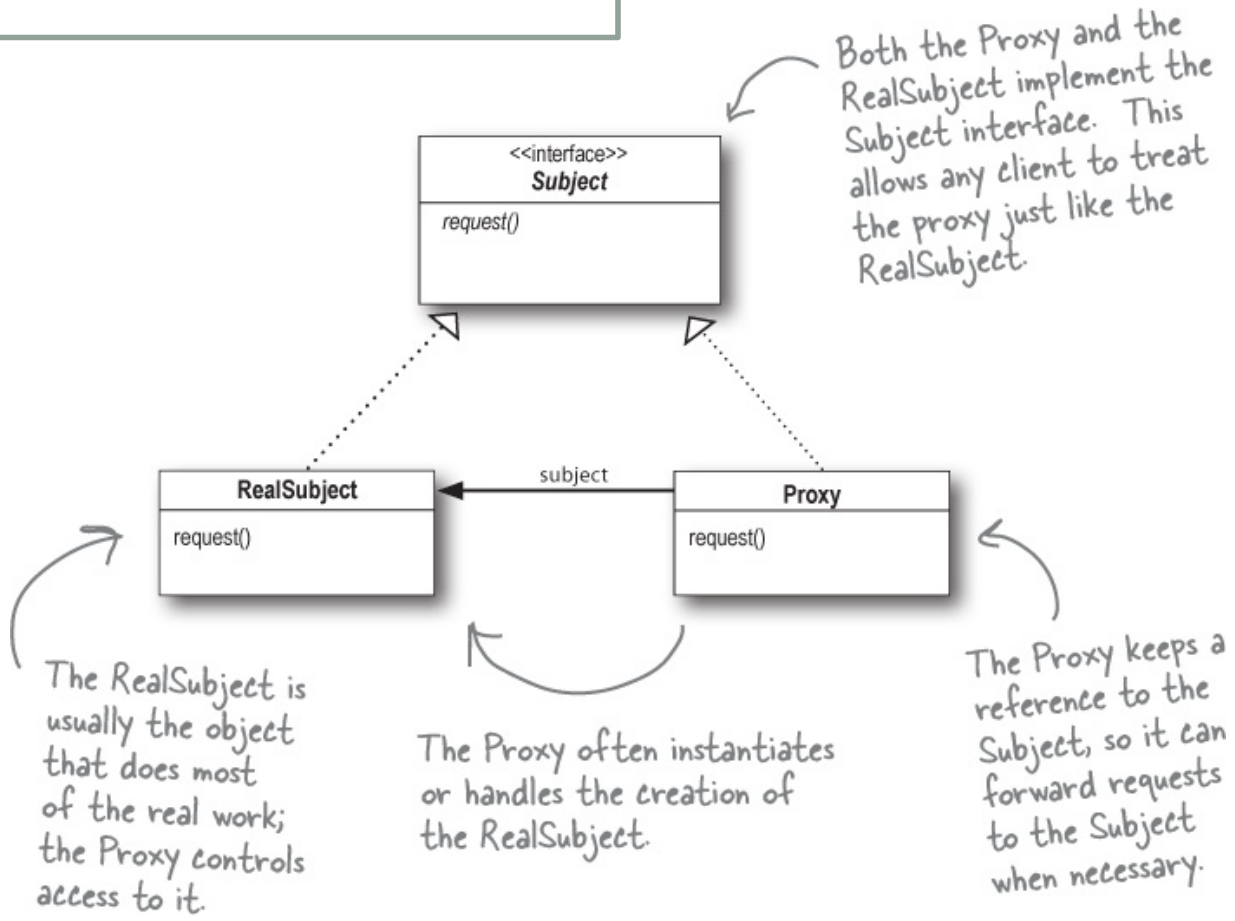
You're the good cop and you provide all your services in a nice and friendly manner, but you don't want everyone asking you for services, so you have the bad cop control access to you. That's what proxies do: **control and manage access**.

Proxies have been known to haul entire method calls over the Internet for their proxied objects; they've also been known to patiently stand in the place for some pretty lazy objects.

But first, let's prepare for next week's client meeting!

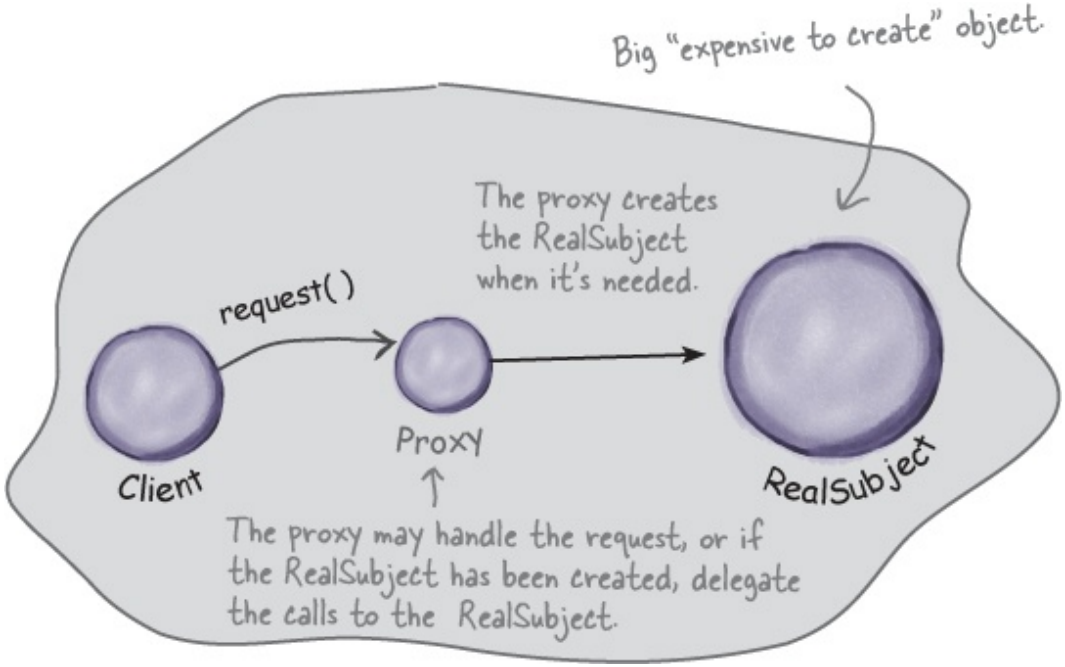
The Proxy Pattern Defined

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

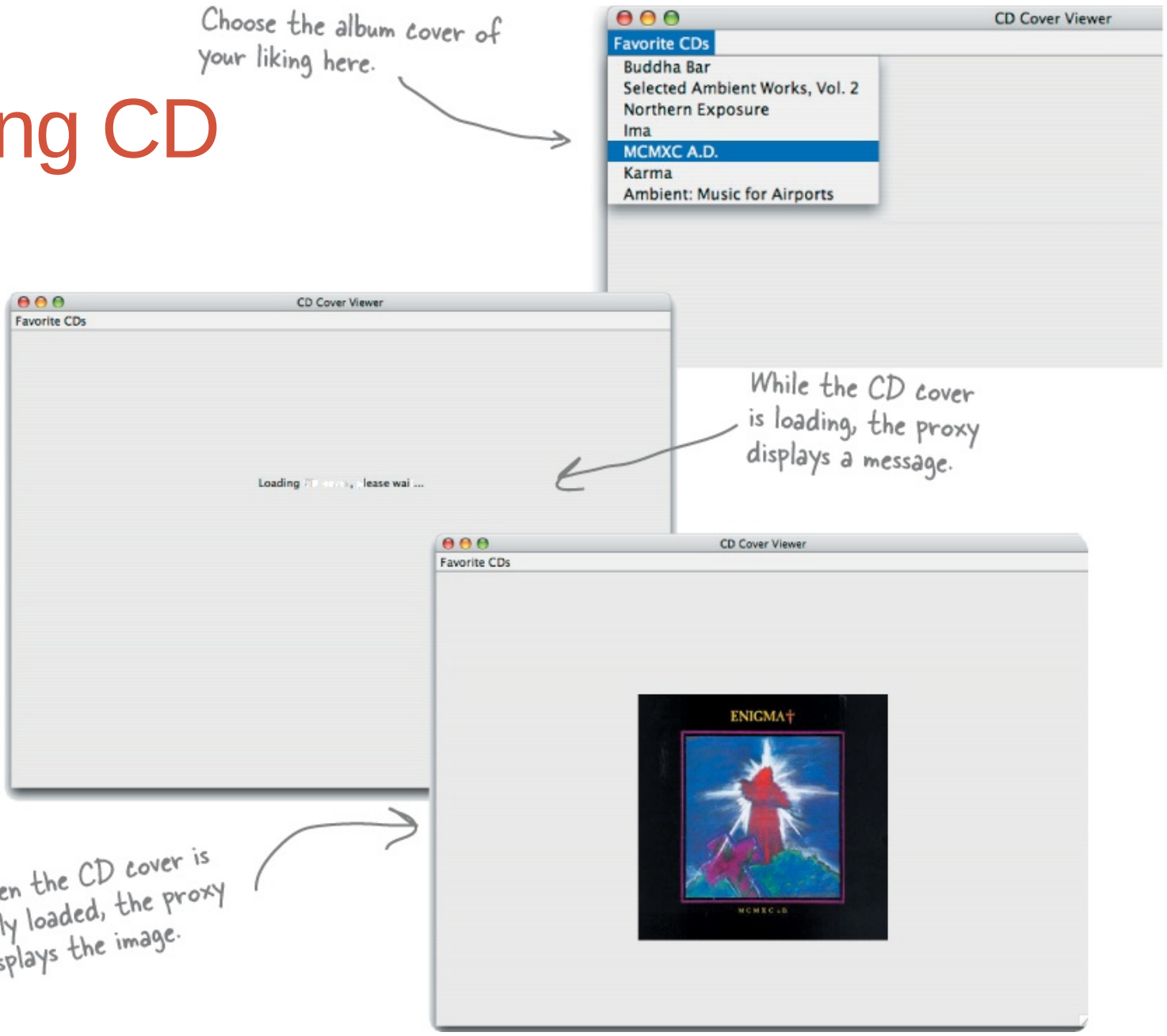


Get ready for Virtual Proxy

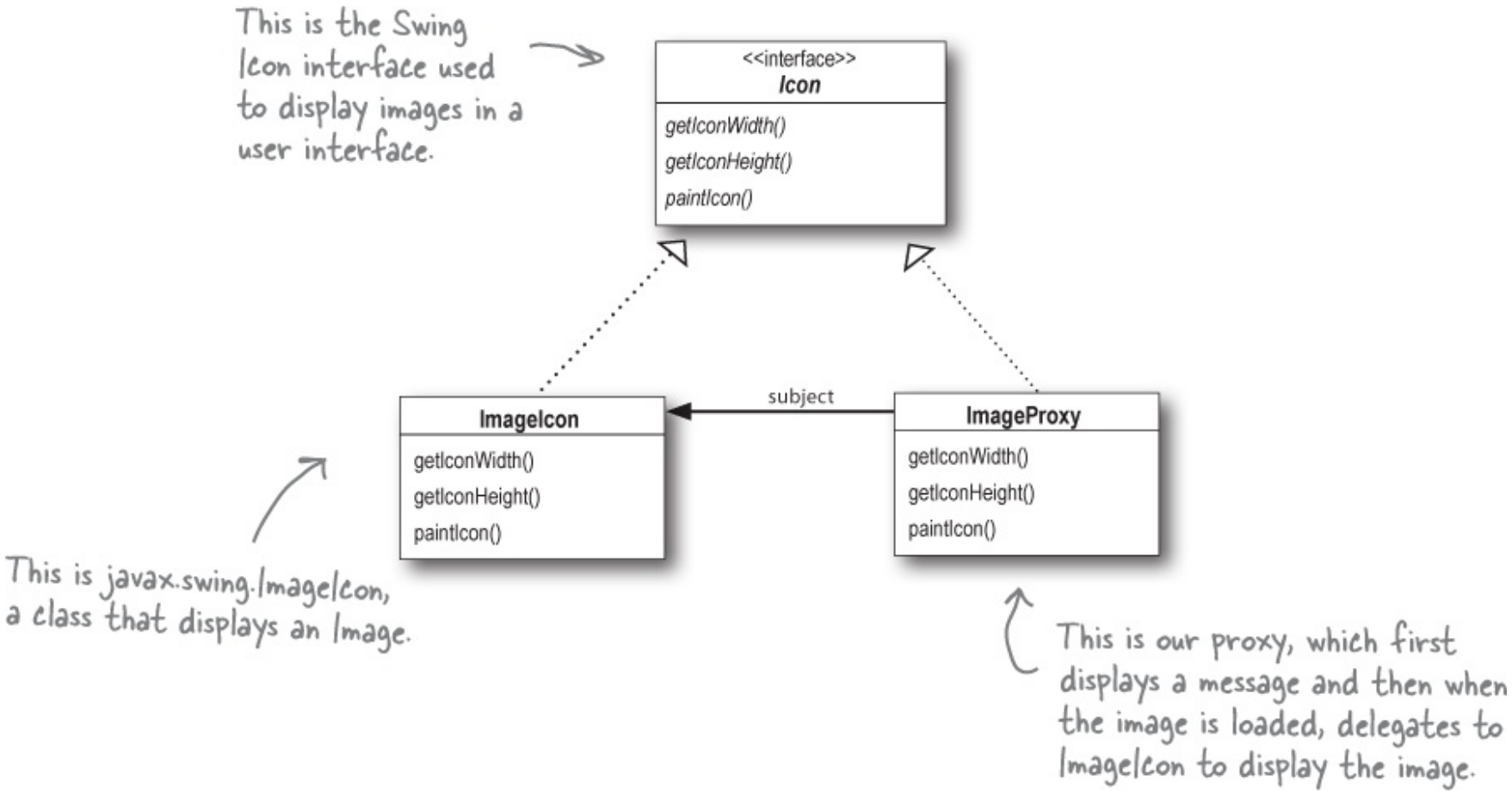
Virtual Proxy acts as a representative for an object that may be expensive to create.



Displaying CD covers



Designing the CD cover Virtual Proxy



How ImageProxy is going to work

ImageProxy first creates an ImageIcon and starts loading it from a network URL.

While the bytes of the image are being retrieved, ImageProxy displays “Loading CD cover, please wait...”.

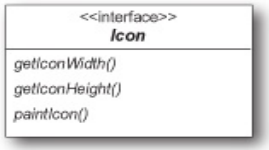
When the image is fully loaded, ImageProxy delegates all method calls to the image icon, including paintIcon(), getWidth() and getHeight().

If the user requests a new image, we’ll create a new proxy and start the process over.

Writing the Image Proxy

```
class ImageProxy implements Icon {
    volatile ImageIcon imageIcon;
    final 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;
        }
    }

    synchronized void setImageIcon(ImageIcon imageIcon) {
        this.imageIcon = imageIcon;
    }

    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 {
                            setImageIcon(new ImageIcon(imageURL, "CD Cover"));
                            c.repaint();
                        } catch (Exception e) {
                            e.printStackTrace();
                        }
                    }
                });
                retrievalThread.start();
            }
        }
    }
}
```

The ImageProxy implements the Icon interface.



The imageIcon is the REAL icon that we eventually want to display when it's loaded.

We pass the URL of the image into the constructor. This is the image we need to display once it's loaded!

We return a default width and height until the imageIcon is loaded; then we turn it over to the imageIcon.

imageIcon is used by two different threads so along with making the variable volatile (to protect reads), we use a synchronized setter (to protect writes).

Here's where things get interesting. This code paints the icon on the screen (by delegating to the imageIcon). However, if we don't have a fully created ImageIcon, then we create one. Let's look at this closer on the next page...

```
File Edit Window Help JustSomeOfTheCDsThatGotUsThroughThisBook
% java ImageProxyTestDrive
```

Testing the CD Cover Viewer

Running ImageProxyTestDrive should give you a window like this.

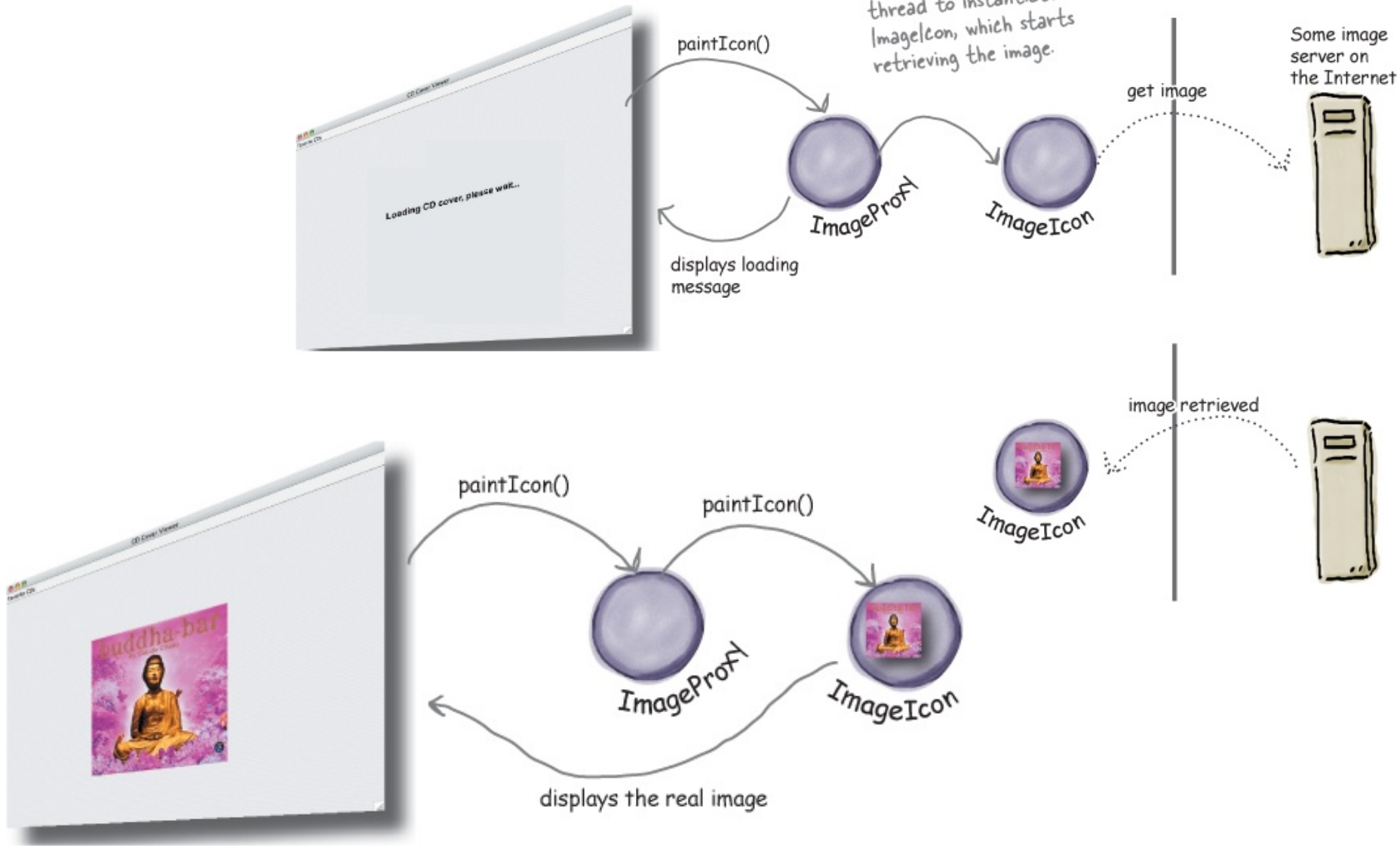
```
public class ImageProxyTestDrive {
    ImageComponent imageComponent;
    public static void main (String[] args) throws Exception {
        ImageProxyTestDrive testDrive = new ImageProxyTestDrive();
    }

    public ImageProxyTestDrive() throws Exception {

        // set up frame and menus

        Icon icon = new ImageProxy(initialURL);
        imageComponent = new ImageComponent(icon);
        frame.getContentPane().add(imageComponent);
    }
}
```

Here we create an image proxy and set it to an initial URL. Whenever you choose a selection from the CD menu, you'll get a new image proxy.

Next we wrap our proxy in a component so it can be added to the frame. The component will take care of the proxy's width, height and similar details.

Finally we add the proxy to the frame so it can be displayed.

What did we do?



Proxy vs Decorator vs Adapter

A **decorator** adds behavior to a class, while a **proxy** controls access to it

Adapter changes the interface of the objects it adapts, while the **Proxy** implements the same interface

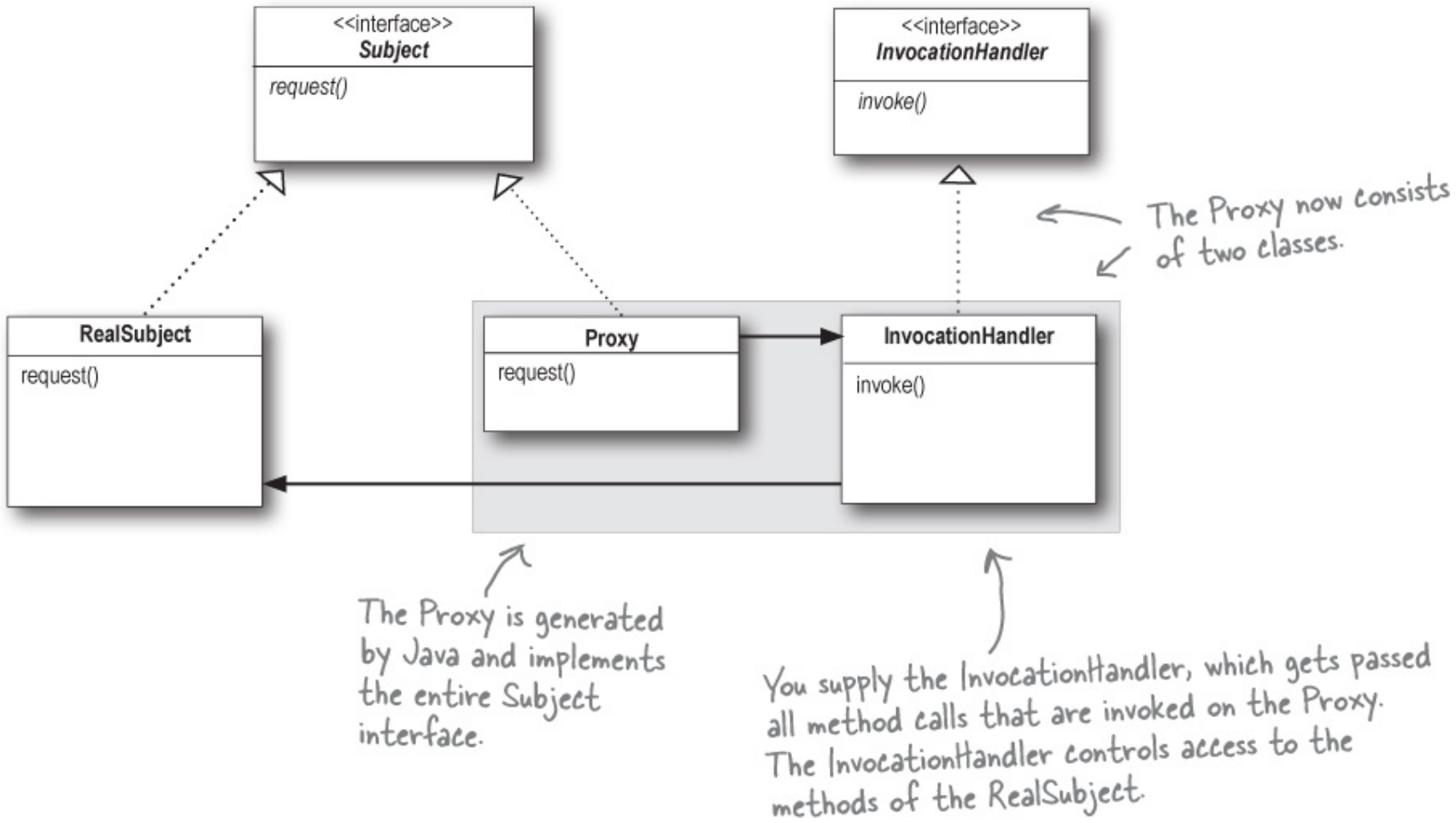
Java API's Proxy to create a protection proxy

Java's got its own proxy support right in the **java.lang.reflect** package.

Java lets you create a proxy class **on the fly** that implements one or more interfaces and forwards method invocations to a class that you specify.

Because the actual proxy class is created at runtime, we refer to this Java technology as a **dynamic proxy**.

Protection Proxy

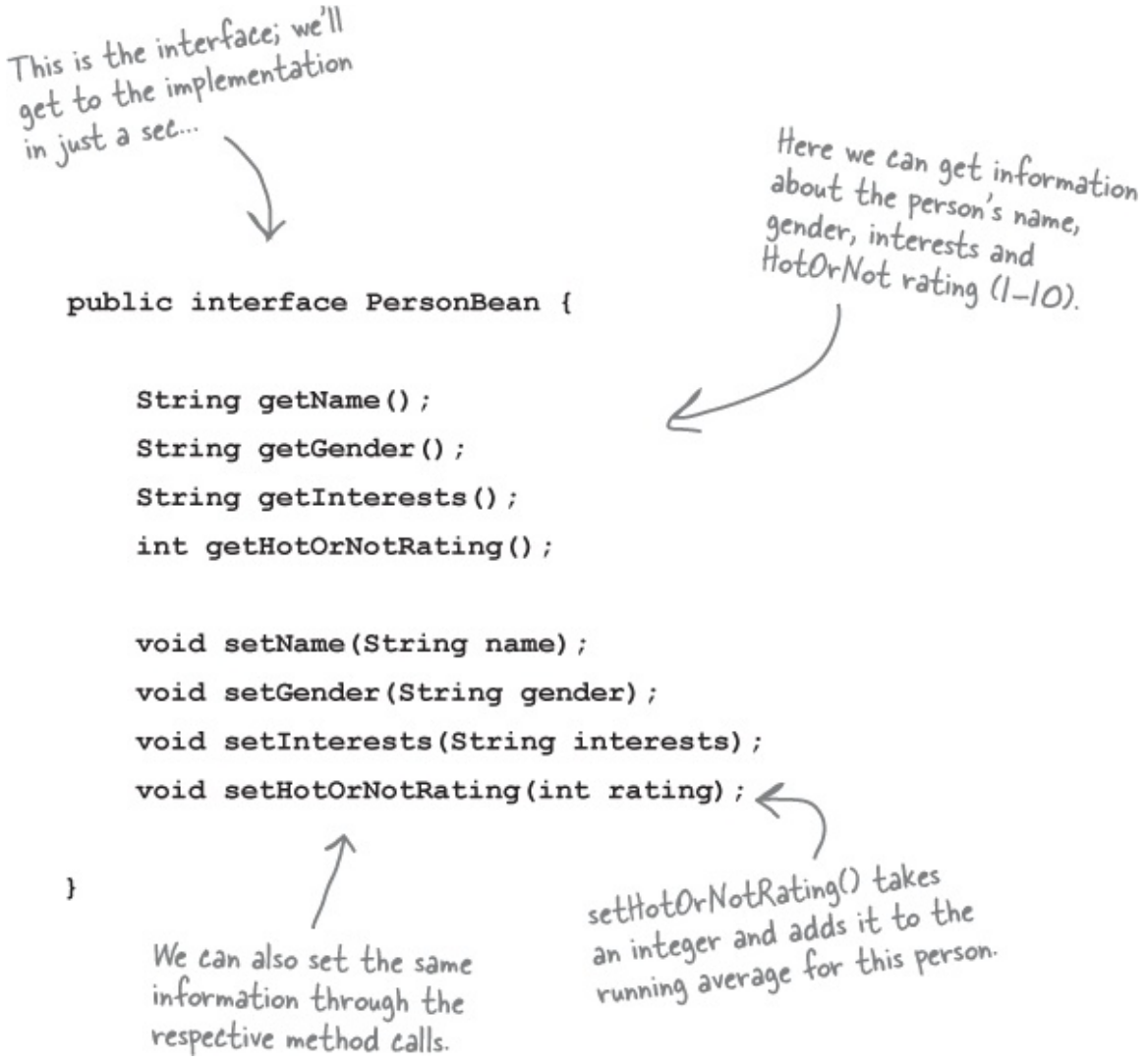


Matchmaking in Objectville

Every town needs a matchmaking service, right?

You've risen to the task and implemented a dating service for Objectville. You've also tried to be innovative by including a "Hot or Not" feature in the service where participants can rate each other—you figure this keeps your customers engaged and looking through possible matches; it also makes things a lot more fun.

Service revolves around a PersonBean



The Person Bean code

```
public class PersonBeanImpl implements PersonBean {
    String name;
    String gender;
    String interests;
    int rating;
    int ratingCount = 0;

    public String getName() {
        return name;
    }

    public String getGender() {
        return gender;
    }

    public String getInterests() {
        return interests;
    }

    public int getHotOrNotRating() {
        if (ratingCount == 0) return 0;
        return (rating/ratingCount);
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    public void setInterests(String interests) {
        this.interests = interests;
    }

    public void setHotOrNotRating(int rating) {
        this.rating += rating;
        ratingCount++;
    }
}
```

The PersonBeanImpl implements the PersonBean interface.

The instance variables.

All the getter methods; they each return the appropriate instance variable...

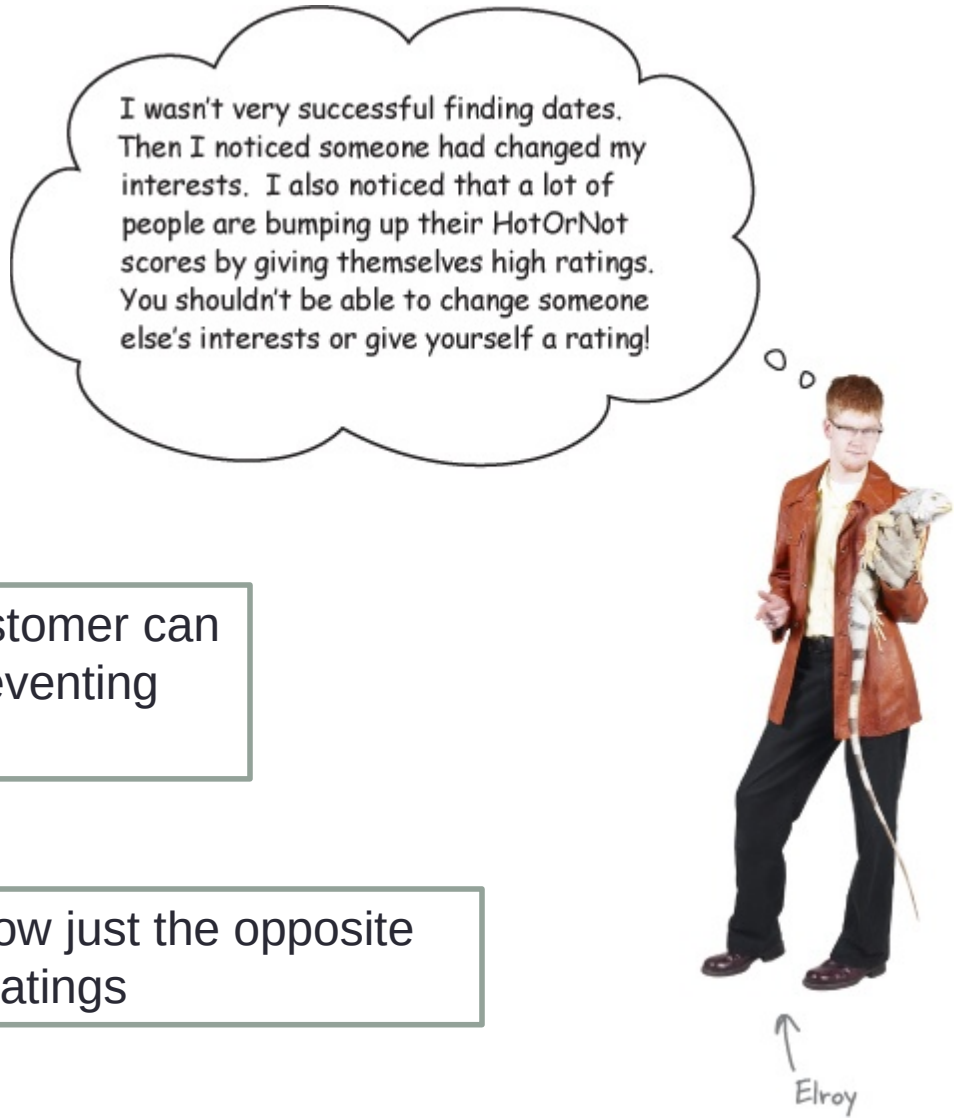
...except for getHotOrNotRating(), which computes the average of the ratings by dividing the ratings by the ratingCount.

And here's all the setter methods, which set the corresponding instance variable.

Finally, the setHotOrNotRating() method increments the total ratingCount and adds the rating to the running total.

Illegal access?

The way our PersonBean is defined, any client can call any of the methods.



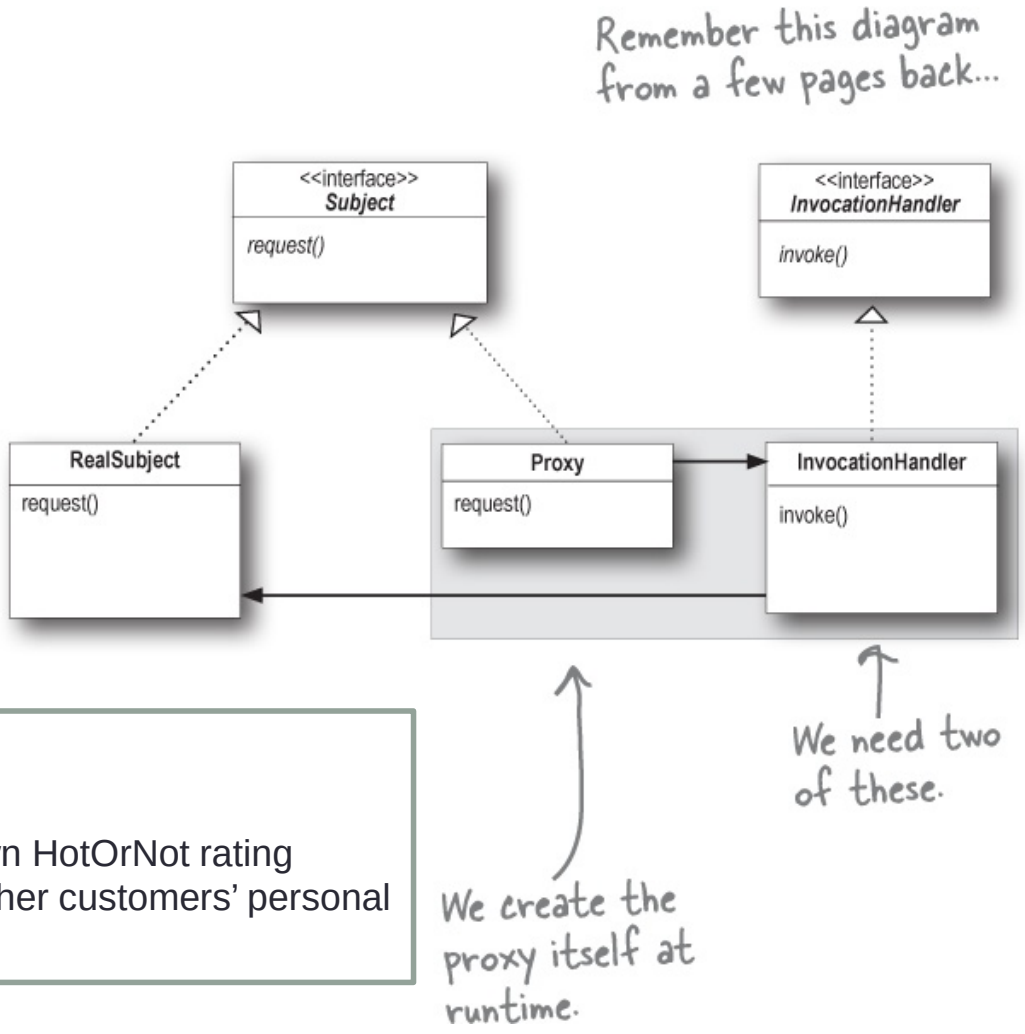
We want to make sure that a customer can set his own information while preventing others from altering it.

We also want to allow just the opposite with the HotOrNot ratings

What we need here is a Protection Proxy.

Protection Proxy

The Big Picture



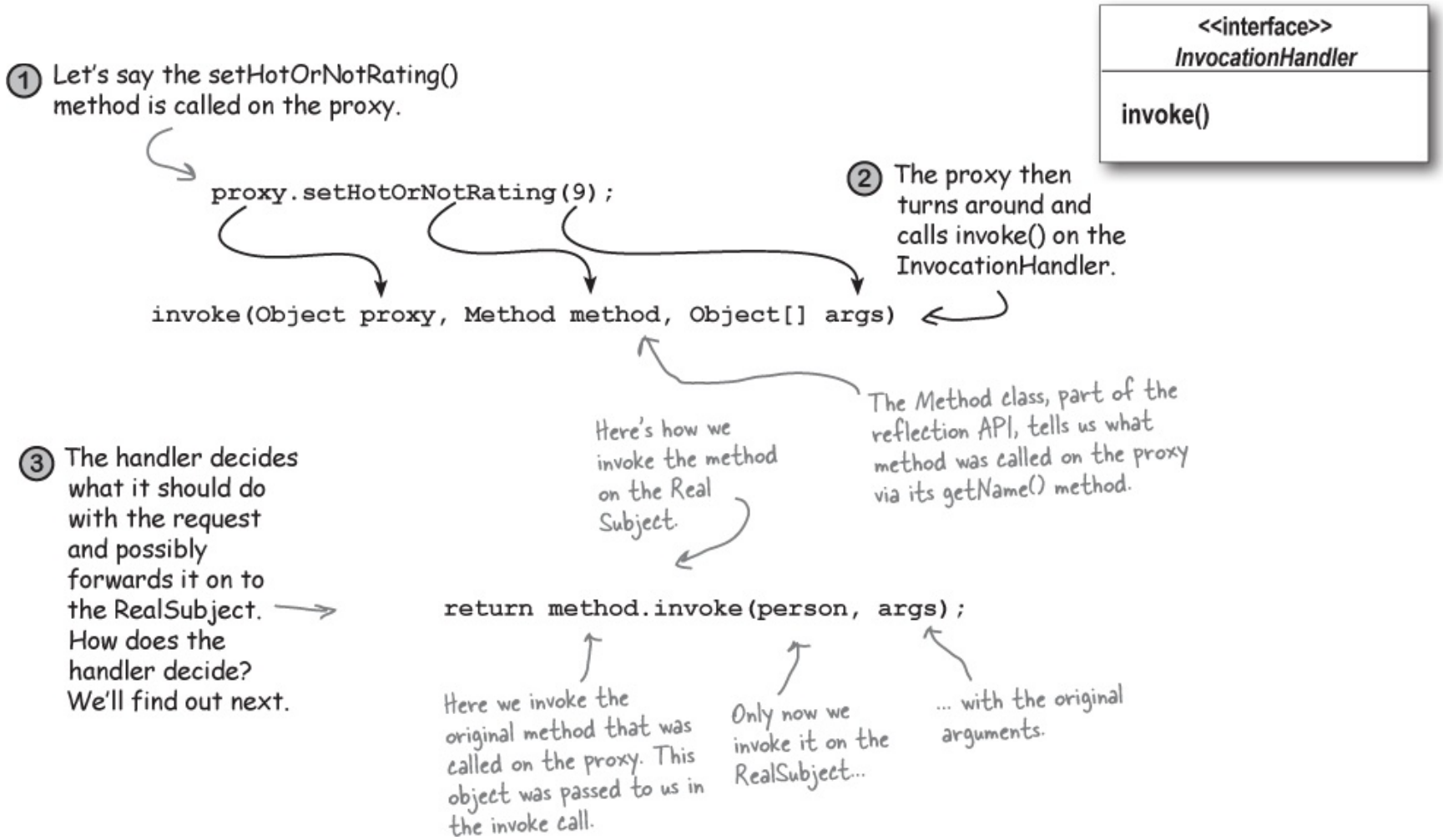
We have a couple of problems to fix:

1. Customers shouldn't be changing their own HotOrNot rating
2. Customers shouldn't be able to change other customers' personal information

We're going to create two proxies:

1. One for accessing your own PersonBean object
2. One for accessing another customer's PersonBean object

Invocation Handlers



Creating Invocation Handlers

InvocationHandler is part of the java.lang.reflect package, so we need to import it.

All invocation handlers implement the InvocationHandler interface.

```
import java.lang.reflect.*;

public class OwnerInvocationHandler implements InvocationHandler {
    PersonBean person;

    public OwnerInvocationHandler(PersonBean person) {
        this.person = person;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
            throws IllegalAccessException {

        try {
            if (method.getName().startsWith("get")) {
                return method.invoke(person, args);
            } else if (method.getName().equals("setHotOrNotRating")) {
                throw new IllegalAccessException();
            } else if (method.getName().startsWith("set")) {
                return method.invoke(person, args);
            }
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

We're passed the Real Subject in the constructor and we keep a reference to it.

Here's the invoke method that gets called every time a method is invoked on the proxy.

If the method is a getter, we go ahead and invoke it on the real subject.

Otherwise, if it is the setHotOrNotRating() method we disallow it by throwing a IllegalAccessException.

This will happen if the real subject throws an exception.

Because we are the owner any other set method is fine and we go ahead and invoke it on the real subject.

If any other method is called, we're just going to return null rather than take a chance.

Proxy Class and Proxy Object

This method takes a person object (the real subject) and returns a proxy for it. Because the proxy has the same interface as the subject, we return a PersonBean.

This code creates the proxy. Now this is some mighty ugly code, so let's step through it carefully.

To create a proxy we use the static newProxyInstance method on the Proxy class.

```
PersonBean getOwnerProxy(PersonBean person) {

    return (PersonBean) Proxy.newProxyInstance(
            person.getClass().getClassLoader(),
            person.getClass().getInterfaces(),
            new OwnerInvocationHandler(person));
}
```

We pass it the classloader for our subject...

...and the set of interfaces the proxy needs to implement...

We pass the real subject into the constructor of the invocation handler. If you look back one page you'll see this is how the handler gets access to the real subject.

...and an invocation handler, in this case our OwnerInvocationHandler.

Testing the matchmaking service

```
public class MatchMakingTestDrive {
    // instance variables here

    public static void main(String[] args) {
        MatchMakingTestDrive test = new MatchMakingTestDrive();
        test.drive();
    }

    public MatchMakingTestDrive() {
        initializeDatabase();
    }

    public void drive() {
        PersonBean joe = getPersonFromDatabase("Joe Javabean");
        PersonBean ownerProxy = getOwnerProxy(joe);
        System.out.println("Name is " + ownerProxy.getName());
        ownerProxy.setInterests("bowling, Go");
        System.out.println("Interests set from owner proxy");
        try {
            ownerProxy.setHotOrNotRating(10);
        } catch (Exception e) {
            System.out.println("Can't set rating from owner proxy");
        }
        System.out.println("Rating is " + ownerProxy.getHotOrNotRating());

        PersonBean nonOwnerProxy = getNonOwnerProxy(joe);
        System.out.println("Name is " + nonOwnerProxy.getName());
        try {
            nonOwnerProxy.setInterests("bowling, Go");
        } catch (Exception e) {
            System.out.println("Can't set interests from non owner proxy");
        }
        nonOwnerProxy.setHotOrNotRating(3);
        System.out.println("Rating set from non owner proxy");
        System.out.println("Rating is " + nonOwnerProxy.getHotOrNotRating());
    }

    // other methods like getOwnerProxy and getNonOwnerProxy here
}
```

Main just creates the test drive and calls its drive() method to get things going.

The constructor initializes our DB of people in the matchmaking service.

Let's retrieve a person from the DB...

...and create an owner proxy.

Call a getter.

And then a setter.

And then try to change the rating.

This shouldn't work!

Now create a non-owner proxy...

...and call a getter.

Followed by a setter.

This shouldn't work!

Then try to set the rating.

This should work!

Running the code …

File Edit Window Help Born2BDynamic

```
% java MatchMakingTestDrive
Name is Joe Javabean
Interests set from owner proxy
Can't set rating from owner proxy
Rating is 7

Name is Joe Javabean
Can't set interests from non owner proxy
Rating set from non owner proxy
Rating is 5
%
```

Our Owner proxy allows getting and setting, except for the HotOrNot rating.

Our NonOwner proxy allows getting only, but also allows calls to set the HotOrNot rating.

The new rating is the average of the previous rating, 7 and the value set by the nonowner proxy, 3.

The Proxy Zoo

1/2

Firewall Proxy controls access to a set of network resources, protecting the subject from "bad" clients.

Habitat: often seen in the location of corporate firewall systems.

Help find a habitat

Smart Reference Proxy provides additional actions whenever a subject is referenced, such as counting the number of references to an object.

Caching Proxy provides temporary storage for results of operations that are expensive. It can also allow multiple clients to share the results to reduce computation or network latency.

Habitat: often seen in web server proxies as well as content management and publishing systems.

The Proxy Zoo

Synchronization Proxy provides safe access to a subject from multiple threads.

Seen hanging around JavaSpaces, where it controls synchronized access to an underlying set of objects in a distributed environment.

Help find a habitat

Complexity Hiding Proxy hides the complexity of and controls access to a complex set of classes. This is sometimes called the Facade Proxy for obvious reasons. The Complexity Hiding Proxy differs from the Facade Pattern in that the proxy controls access, while the Facade Pattern just provides an alternative interface.

Copy-On-Write Proxy controls the copying of an object by deferring the copying of an object until it is required by a client. This is a variant of the Virtual Proxy.

Habitat: seen in the vicinity of the Java's CopyOnWriteArrayList.

Recap

The Proxy Pattern provides a **representative** for another object in order to **control the client's access** to it.

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.

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.