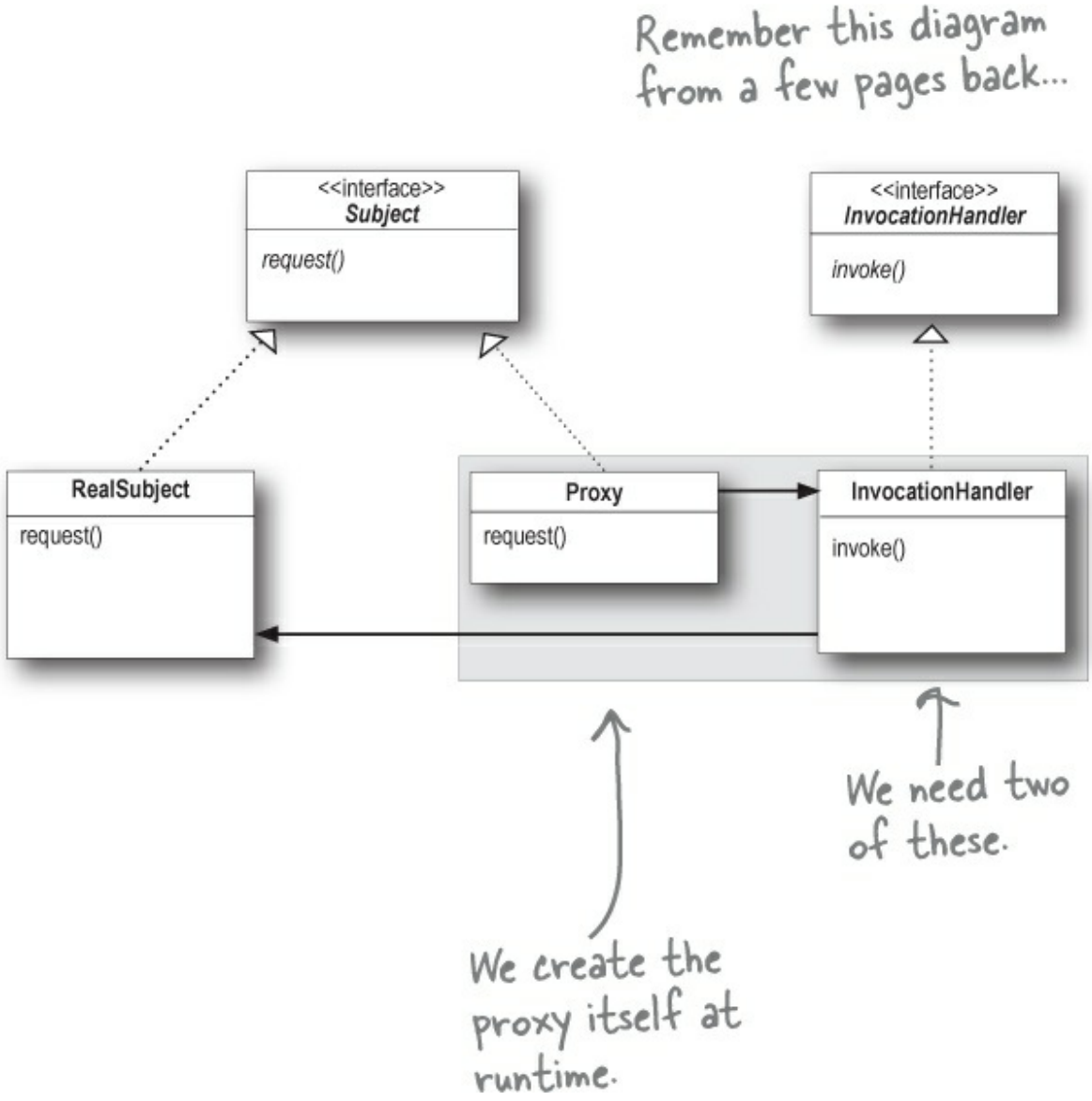
Big Picture: creating a Dynamic Proxy for the PersonBean

We have a couple of problems to fix: customers shouldn't be changing their own HotOrNot rating and customers shouldn't be able to change other customers' personal information. To fix these problems we're going to create two proxies: one for accessing your own PersonBean object and one for accessing another customer's PersonBean object. That way, the proxies can control what requests can be made in each circumstance.

To create these proxies we're going to use the Java API's dynamic proxy that you saw a few pages back. Java will create two proxies for us; all we need to do is supply the handlers that know what to do when a method is invoked on the proxy.



Step one:

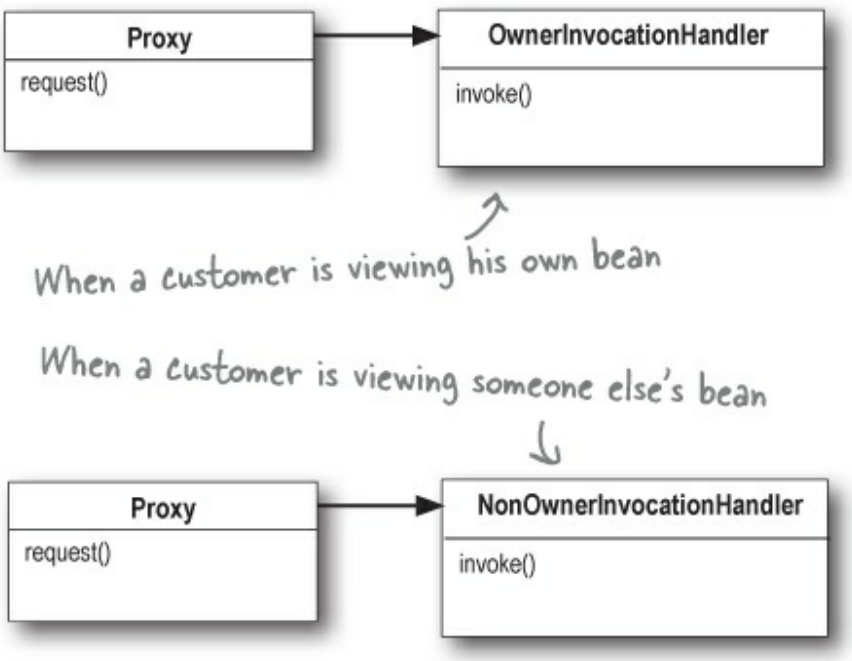
Create two **InvocationHandlers.**
InvocationHandlers implement the behavior of the proxy. As you'll see, Java will take care of creating the actual proxy class and object; we just need to supply a handler that knows what to do when a method is called on it.

Step two:

Write the code that creates the dynamic proxies.
We need to write a little bit of code to generate the proxy class and instantiate it. We'll step through this code in just a bit.

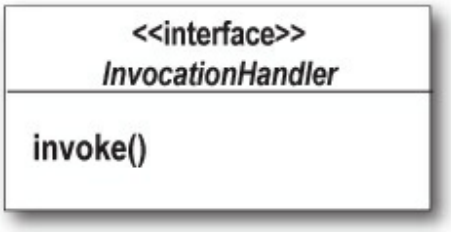
Step three:

Wrap any PersonBean object with the appropriate proxy.
When we need to use a PersonBean object, either it's the object of the customer himself (in that case, will call him the "owner"), or it's another user of the service that the customer is checking out (in that case we'll call him "non-owner").
In either case, we create the appropriate proxy for the PersonBean.

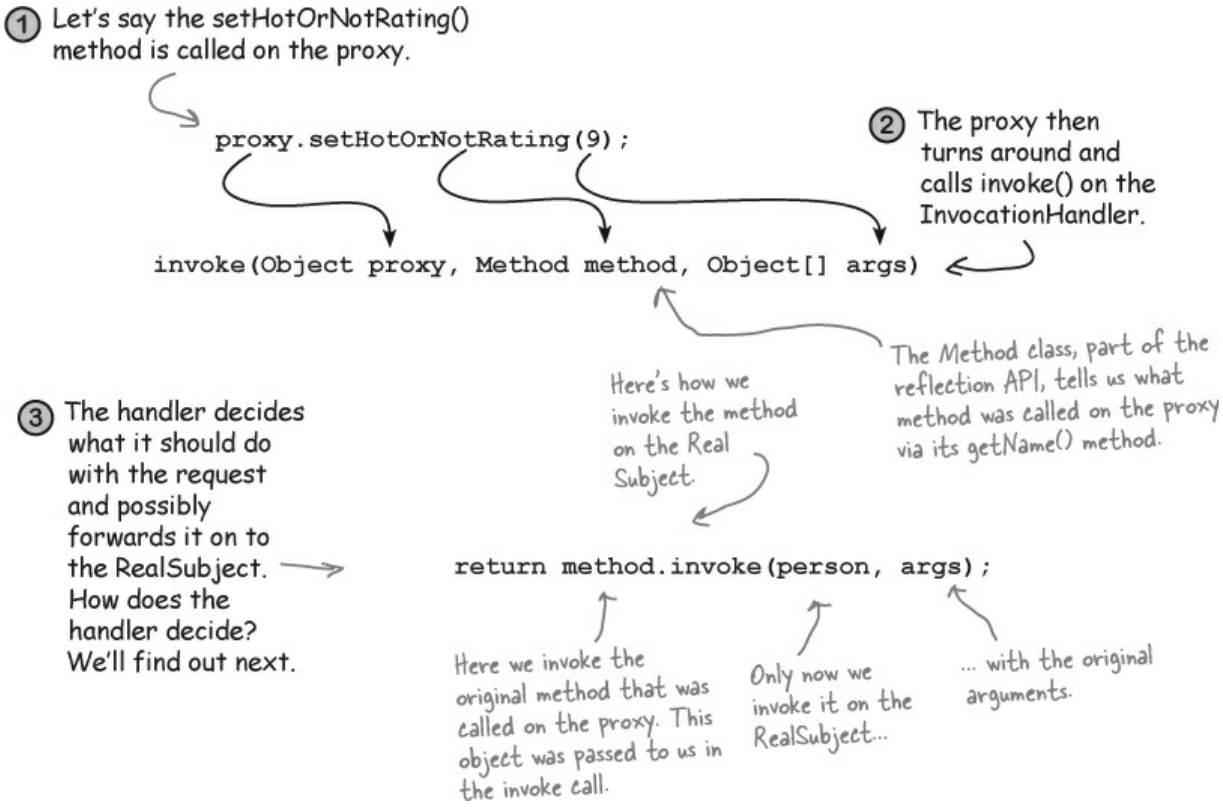


Step one: creating Invocation Handlers

We know we need to write two invocation handlers, one for the owner and one for the non-owner. But what are invocation handlers? Here's the way to think about them: when a method call is made on the proxy, the proxy forwards that call to your invocation handler, but not by calling the invocation handler's corresponding method. So, what does it call? Have a look at the InvocationHandler interface:



There's only one method, invoke(), and no matter what methods get called on the proxy, the invoke() method is what gets called on the handler. Let's see how this works:



Creating Invocation Handlers continued...

When invoke() is called by the proxy, how do you know what to do with the call? Typically, you'll examine the method that was called on the proxy and make decisions based on the method's name and possibly its arguments. Let's implement the OwnerInvocationHandler to see how this works:


```
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;
    }
}
```

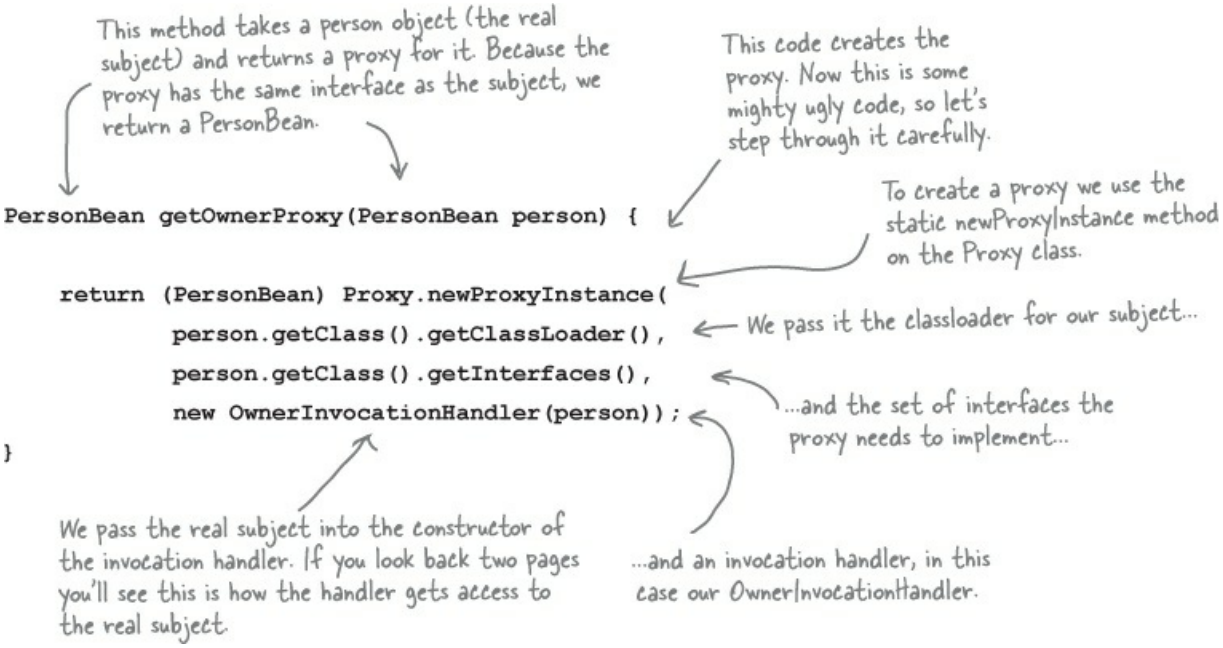
EXERCISE

The NonOwnerInvocationHandler works just like the OwnerInvocationHandler except that it *allows* calls to setHotOrNotRating() and it *disallows* calls to any other set method. Go ahead and write this handler yourself:

Step two: creating the Proxy class and instantiating the Proxy object

Now, all we have left is to dynamically create the Proxy class and instantiate the proxy object. Let's start by writing a method that takes a PersonBean and

knows how to create an owner proxy for it. That is, we're going to create the kind of proxy that forwards its method calls to the OwnerInvocationHandler. Here's the code:



SHARPEN YOUR PENCIL

While it is a little complicated, there isn't much to creating a dynamic proxy. Why don't you write getNonOwnerProxy(), which returns a proxy for the NonOwnerInvocationHandler:

Take it further: can you write one method getProxy() that takes a handler and a person and returns a proxy that uses that handler?

Testing the matchmaking service

Let's give the matchmaking service a test run and see how it controls access to the setter methods based on the proxy that is used.

```
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...

THERE ARE NO DUMB QUESTIONS

Q: **Q: So what exactly is the "dynamic" aspect of dynamic proxies? Is it that I'm instantiating the proxy and setting it to a handler at runtime?**

A: **A:** No, the proxy is dynamic because its class is created at runtime. Think about it: before your code runs there is no proxy class; it is created on demand from the set of interfaces you pass it.

Q: **Q: My InvocationHandler seems like a very strange proxy, it doesn't implement any of the methods of the class it's proxying.**

A: **A:** That is because the InvocationHandler isn't a proxy — it is a class that the proxy dispatches to for handling method calls. The proxy itself is created dynamically at runtime by the static Proxy.newProxyInstance() method.

Q: **Q: Is there any way to tell if a class is a Proxy class?**

A: **A:** Yes. The Proxy class has a static method called isProxyClass(). Calling this method with a class will return true if the class is a dynamic proxy class. Other than that, the proxy class will act like any other class that implements a particular set of interfaces.

Q: **Q: Are there any restrictions on the types of interfaces I can pass into newProxyInstance()?**

A: **A:** Yes, there are a few. First, it is worth pointing out that we always pass newProxyInstance() an array of interfaces — only interfaces are allowed, no classes. The major restrictions are that all non-public interfaces need to be from the same package. You also can't have interfaces with clashing method names (that is, two interfaces with a method with the same signature). There are a few other minor nuances as well, so at some point you should take a look at the fine print on dynamic proxies in the javadoc.

WHO DOES WHAT?

Match each pattern with its description:

Pattern	Description
Decorator	Wraps another object and provides a different interface to it.
Facade	Wraps another object and provides additional behavior for it.
Proxy	Wraps another object to control access to it.
Adapter	Wraps a bunch of objects to simplify their interface.

The Proxy Zoo

Welcome to the Objectville Zoo!

You now know about the remote, virtual and protection proxies, but out in the wild you're going to see lots of mutations of this pattern. Over here in the Proxy corner of the zoo we've got a nice collection of wild proxy patterns that we've captured for your study.

Our job isn't done; we are sure you're going to see more variations of this pattern in the real world, so give us a hand in cataloging more proxies. Let's take a look at the existing collection:

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.

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.

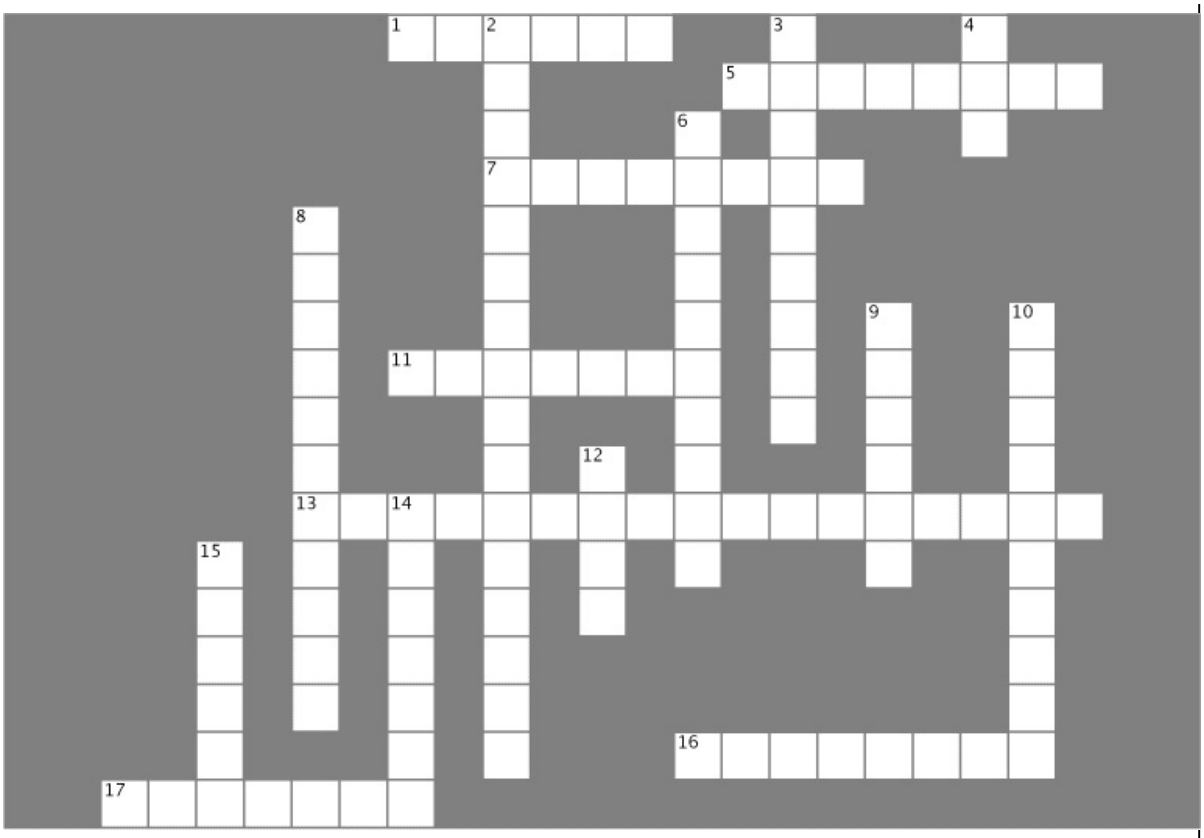
Habitat: seen in the vicinity of the Java's CopyOnWriteArrayList.

NOTE

Field Notes: please add your observations of other proxies in the wild here:

DESIGN PATTERNS CROSSWORD

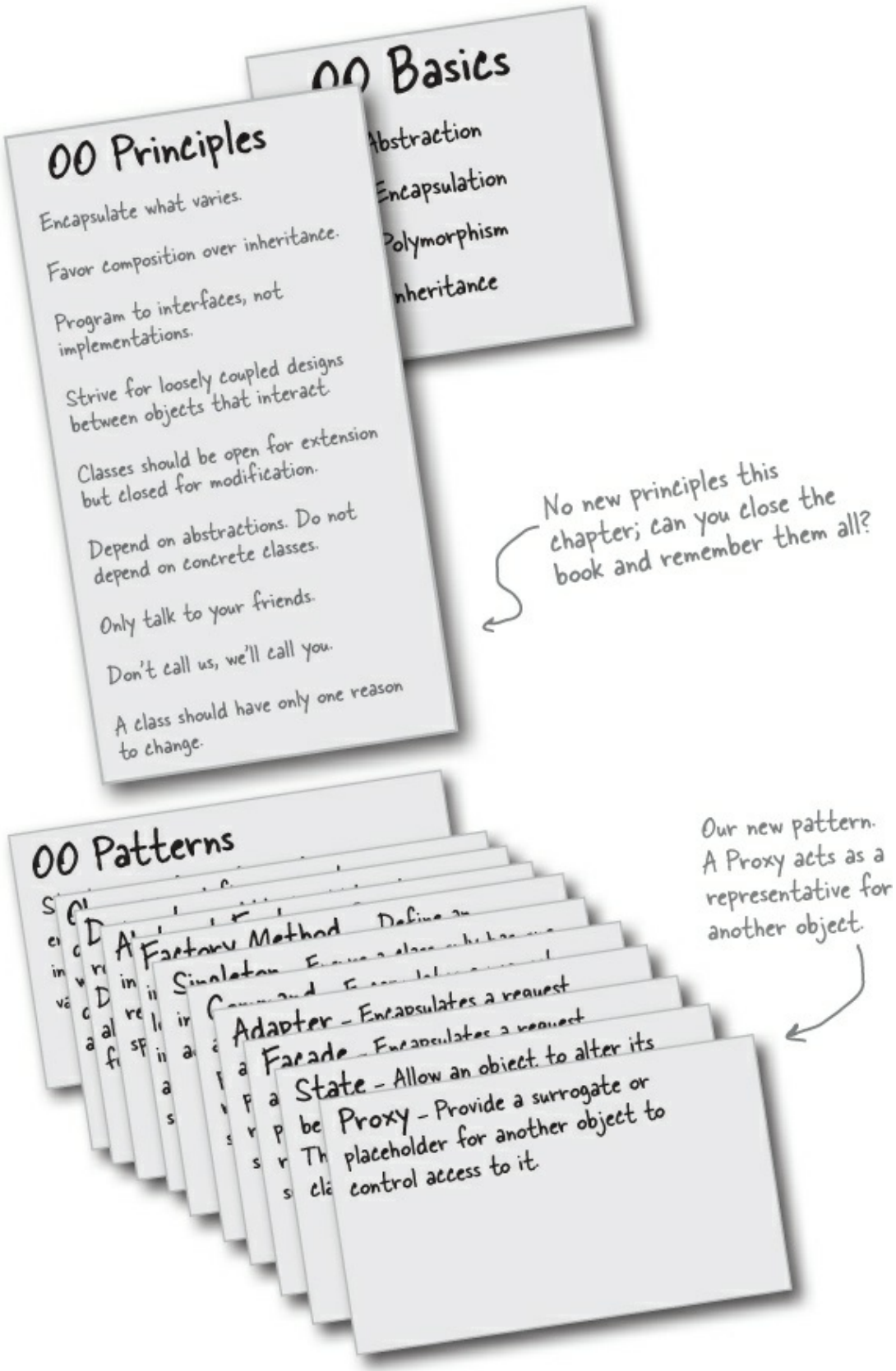
It's been a LONG chapter. Why not unwind by doing a crossword puzzle before it ends?



Across	Down
1. Our first mistake: the gumball machine reporting was not _____. 5. Commonly used proxy for web services (two words). 7. Objectville matchmaking gimmick (three words). 11. A _____ proxy class is created at runtime. 13. Java's dynamic proxy forwards all requests to this (two words). 16. In RMI, the object that takes the network requests on the service side. 17. The CD viewer used this kind of proxy.	2. Remote _____ was used to implement the gumball machine monitor (two words). 3. Similar to proxy, but with a different purpose. 4. Place to learn about the many proxy variants. 6. Proxy that protects method calls from unauthorized callers. 8. This utility acts as a lookup service for RMI. 9. Why Elroy couldn't get dates. 10. Software developer agent was being this kind of proxy. 12. In RMI, the proxy is called this. 14. Proxy that stands in for expensive objects. 15. We took one of these to learn RMI.

Tools for your Design Toolbox

Your design toolbox is almost full; you’re prepared for almost any design problem that comes your way.


OO Basics
Abstraction
Encapsulation
Polymorphism
Inheritance
OO Principles
Encapsulate what varies.
Favor composition over inheritance.
Program to interfaces, not implementations.
Strive for loosely coupled designs between objects that interact.
Classes should be open for extension but closed for modification.
Depend on abstractions. Do not depend on concrete classes.
Only talk to your friends.
Don't call us, we'll call you.
A class should have only one reason to change.
No new principles this chapter; can you close the book and remember them all?
OO Patterns
Factory Method
Adapter – Encapsulates a request
State – Allow an object to alter its
Proxy – Provide a surrogate or placeholder for another object to control access to it.
Our new pattern. A Proxy acts as a representative for another object.

BULLET 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.

EXERCISE SOLUTION

The NonOwnerInvocationHandler works just like the OwnerInvocationHandler except that it *allows* calls to setHotOrNotRating() and it *disallows* calls to any other set method. Here's our solution:

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

public class NonOwnerInvocationHandler implements InvocationHandler {
    PersonBean person;

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

DESIGN PUZZLE SOLUTION

The ImageProxy class appears to have two states that are controlled by conditional statements. Can you think of another pattern that might clean up this code? How would you redesign ImageProxy?

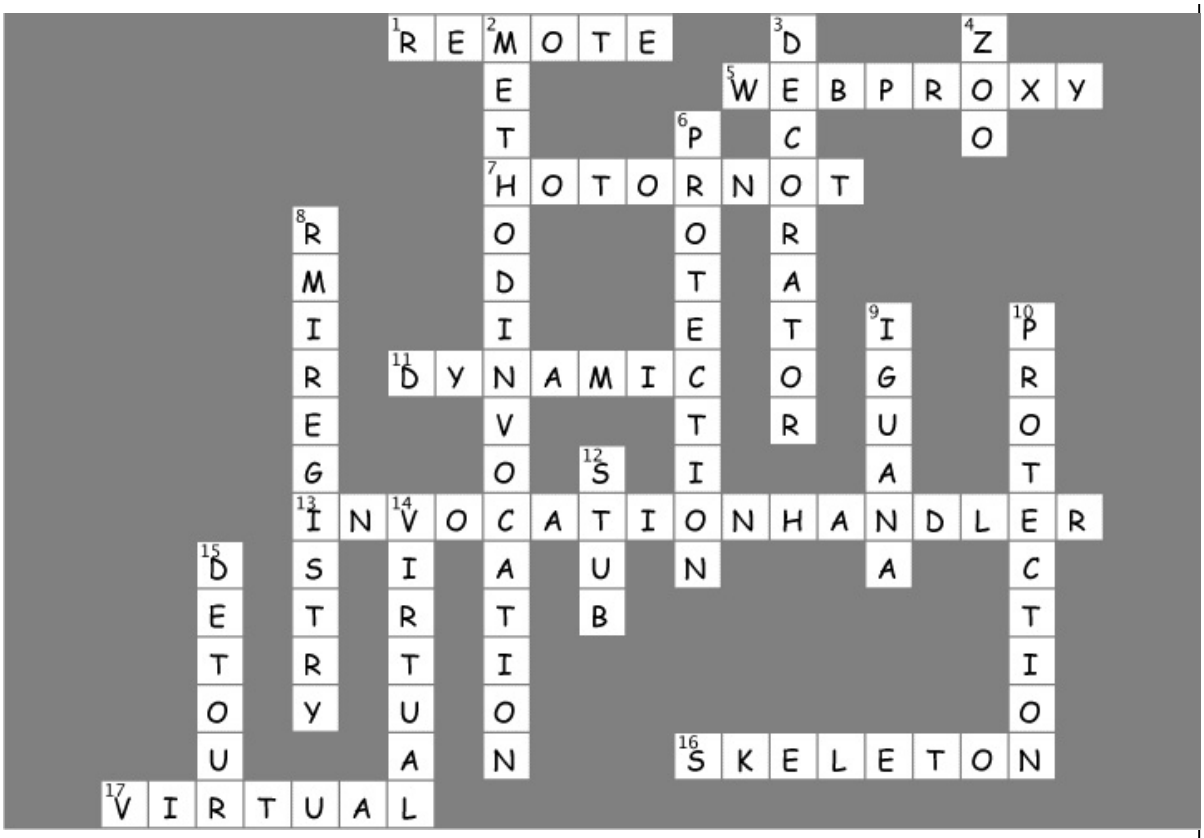
Use State Pattern: implement two states, ImageLoaded and ImageNotLoaded. Then put the code from the if statements into their respective states. Start in the ImageNotLoaded state and then transition to the ImageLoaded state once the ImageIcon had been retrieved.

SHARPEN YOUR PENCIL SOLUTION

While it is a little complicated, there isn't much to creating a dynamic proxy. Why don't you write getNonOwnerProxy(), which returns a proxy for the NonOwnerInvocationHandler. Here's our solution:

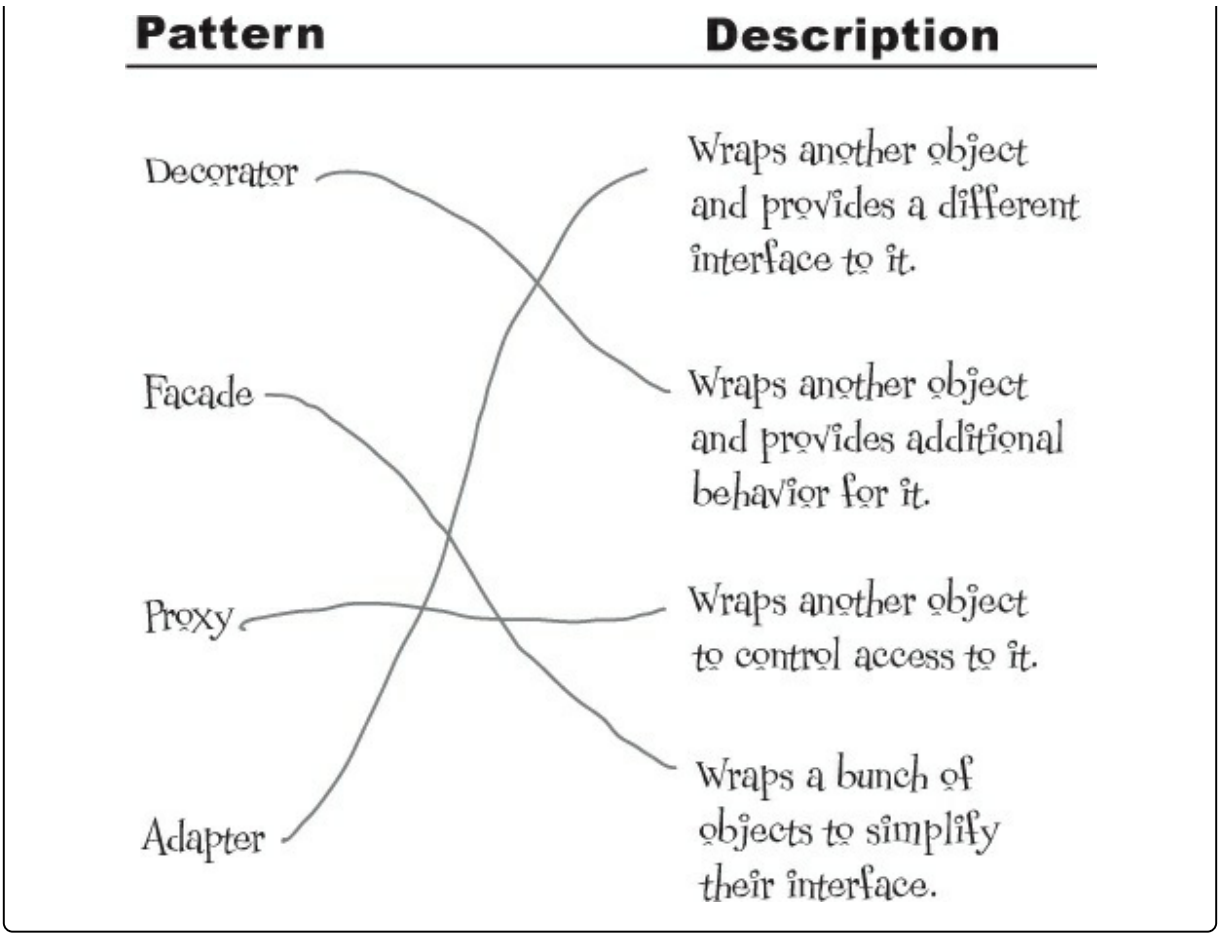
```
PersonBean getNonOwnerProxy(PersonBean person) {

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

DESIGN PATTERNS CROSSWORD SOLUTION



WHO DOES WHAT? SOLUTION

Match each pattern with its description:



The code for the CD Cover Viewer

READY BAKE CODE

```
package headfirst.designpatterns.proxy.virtualproxy;

import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;
public class ImageProxyTestDrive {
    ImageComponent imageComponent;
    JFrame frame = new JFrame("CD Cover Viewer");
    JMenuBar menuBar;
    JMenu menu;
    Hashtable<String, String> cds = new Hashtable<String, String>();

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

    public ImageProxyTestDrive() throws Exception{
        cds.put("Buddha
```

```
Bar","http://images.amazon.com/images/P/B00009XBYK.01.LZZZZZZZ.
jpg");

cds.put("Ima","http://images.amazon.com/images/P/B000005IRM.01.LZZZZZZZ.jpg");

cds.put("Karma","http://images.amazon.com/images/P/B000005DCB.01.LZZZZZZZ.gif");
        cds.put("MCMXC
A.D.","http://images.amazon.com/images/P/B000002URV.01.LZZZZZZZ.
jpg");
        cds.put("Northern
Exposure","http://images.amazon.com/images/P/B000003SFN.01.
LZZZZZZZ.jpg");
        cds.put("Selected Ambient Works, Vol.
2","http://images.amazon.com/images/P/
B000002MNZ.01.LZZZZZZZ.jpg");

        URL initialURL = new URL((String)cds.get("Selected Ambient Works, Vol.
2"));
        menuBar = new JMenuBar();
        menu = new JMenu("Favorite CDs");
        menuBar.add(menu);
        frame.setJMenuBar(menuBar);
        for(Enumeration e = cds.keys(); e.hasMoreElements();) {
            String name = (String)e.nextElement();
            JMenuItem menuItem = new JMenuItem(name);
            menu.add(menuItem);
            menuItem.addActionListener(event -> {
                imageComponent.setIcon(new
ImageProxy(getCDUrl(event.getActionCommand())));
                frame.repaint();
            });
        }

        // set up frame and menus

        Icon icon = new ImageProxy(initialURL);
        imageComponent = new ImageComponent(icon);
        frame.getContentPane().add(imageComponent);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(800,600);
        frame.setVisible(true);

    }
    URL getCDUrl(String name) {
        try {
            return new URL((String)cds.get(name));
        } catch (MalformedURLException e) {
            e.printStackTrace();
            return null;
        }
    }
}
package headfirst.designpatterns.proxy.virtualproxy;

import java.net.*;
import java.awt.*;
import javax.swing.*;

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();
            }
        }
    }
}
package headfirst.designpatterns.proxy.virtualproxy;

import java.awt.*;
import javax.swing.*;

class ImageComponent extends JComponent {
    private Icon icon;

    public ImageComponent(Icon icon) {
        this.icon = icon;
    }

    public void setIcon(Icon icon) {
        this.icon = icon;
```

```
        }

        public void paintComponent(Graphics g) {
            super.paintComponent(g);
            int w = icon.getIconWidth();
            int h = icon.getIconHeight();
            int x = (800 - w)/2;
            int y = (600 - h)/2;
            icon.paintIcon(this, g, x, y);
        }
    }
```

Chapter 12. Compound Patterns: Patterns of Patterns

Who would have ever guessed that Patterns could work together? You've already witnessed the acrimonious Fireside Chats (and you haven't even seen the Pattern Death Match pages that the editor forced us to remove from the book[2]), so who would have thought patterns can actually get along well together? Well, believe it or not, some of the most powerful OO designs use several patterns together. Get ready to take your pattern skills to the next level; it's time for compound patterns.

Working together

One of the best ways to use patterns is to get them out of the house so they can interact with other patterns. The more you use patterns the more you're going to see them showing up together in your designs. We have a special

name for a set of patterns that work together in a design that can be applied over many problems: a *compound pattern*. That's right, we are now talking about patterns made of patterns!

You'll find a lot of compound patterns in use in the real world. Now that you've got patterns in your brain, you'll see that they are really just patterns working together, and that makes them easier to understand.

We're going to start this chapter by revisiting our friendly ducks in the SimUDuck duck simulator. It's only fitting that the ducks should be here when we combine patterns; after all, they've been with us throughout the entire book and they've been good sports about taking part in lots of patterns. The ducks are going to help you understand how patterns can work together in the same solution. But just because we've combined some patterns doesn't mean we have a solution that qualifies as a compound pattern. For that, it has to be a general-purpose solution that can be applied to many problems. So, in the second half of the chapter we'll visit a *real* compound pattern: that's right, Mr. Model-View-Controller himself. If you haven't heard of him, you will, and you'll find this compound pattern is one of the most powerful patterns in your design toolbox.

> **Patterns are often used together and combined within the same design solution. A compound pattern combines two or more patterns into a solution that solves a recurring or general problem.**

Duck reunion

As you've already heard, we're going to get to work with the ducks again. This time the ducks are going to show you how patterns can coexist and even cooperate within the same solution.

We’re going to rebuild our duck simulator from scratch and give it some interesting capabilities by using a bunch of patterns. Okay, let’s get started...

① **First, we’ll create a Quackable interface.**
Like we said, we’re starting from scratch. This time around, the Ducks are going to implement a Quackable interface. That way we’ll know what things in the simulator can quack() — like Mallard Ducks, Redhead Ducks, Duck Calls, and we might even see the Rubber Duck sneak back in.

```
public interface Quackable {
    public void quack();
}
```

Quackables only need to do one thing well: Quack!

② **Now, some Ducks that implement Quackable**
What good is an interface without some classes to implement it? Time to create some concrete ducks (but not the “lawn art” kind, if you know what we mean).

Your standard Mallard duck.

```
public class MallardDuck implements Quackable {
    public void quack() {
        System.out.println("Quack");
    }
}

public class RedheadDuck implements Quackable {
    public void quack() {
        System.out.println("Quack");
    }
}
```

We’ve got to have some variation of species if we want this to be an interesting simulator.

This wouldn’t be much fun if we didn’t add other kinds of Ducks too.
Remember last time? We had duck calls (those things hunters use — they are definitely quackable) and rubber ducks.

```
public class DuckCall implements Quackable {
    public void quack() {
        System.out.println("Kwak");
    }
}
public class RubberDuck implements Quackable {
    public void quack() {
        System.out.println("Squeak");
    }
}
```

A DuckCall that quacks but doesn't sound quite like the real thing.

A RubberDuck that makes a squeak when it quacks.

③ Okay, we've got our ducks; now all we need is a simulator.

Let's cook up a simulator that creates a few ducks and makes sure their quackers are working...

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        simulator.simulate();
    }

    void simulate() {
        Quackable mallardDuck = new MallardDuck();
        Quackable redheadDuck = new RedheadDuck();
        Quackable duckCall = new DuckCall();
        Quackable rubberDuck = new RubberDuck();

        System.out.println("\nDuck Simulator");

        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

Here's our main method to get everything going.

We create a simulator and then call its simulate() method.

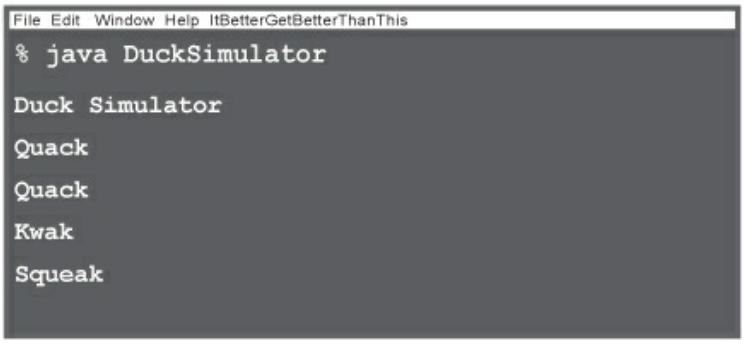
We need some ducks, so here we create one of each Quackable...

... then we simulate each one.

Here we overload the simulate method to simulate just one duck.

Here we let polymorphism do its magic: no matter what kind of Quackable gets passed in, the simulate() method asks it to quack.

Not too exciting yet, but we haven't added patterns!



NOTE

They all implement the same Quackable interface, but their implementations allow them to quack in their own way.

It looks like everything is working; so far, so good.

④ When ducks are around, geese can't be far.

Where there is one waterfowl, there are probably two. Here's a Goose class that has been hanging around the simulator.

```
public class Goose {
    public void honk() {
        System.out.println("Honk");
    }
}
```

A Goose is a honker, not a quacker.

BRAIN POWER

Let's say we wanted to be able to use a Goose anywhere we'd want to use a Duck. After all, geese make noise; geese fly; geese swim. Why can't we have Geese in the simulator?

What pattern would allow Geese to easily intermingle with Ducks?

⑤ We need a goose adapter.

Our simulator expects to see Quackable interfaces. Since geese aren't quackers (they're honkers), we can use an adapter to adapt a goose to a duck.

```
public class GooseAdapter implements Quackable {
    Goose goose;

    public GooseAdapter(Goose goose) {
        this.goose = goose;
    }

    public void quack() {
        goose.honk();
    }
}
```

Remember, an Adapter implements the target interface, which in this case is Quackable.

The constructor takes the goose we are going to adapt.

When quack is called, the call is delegated to the goose's honk() method.

⑥ Now geese should be able to play in the simulator, too.

All we need to do is create a Goose, wrap it in an adapter that implements Quackable, and we should be good to go.

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        simulator.simulate();
    }

    void simulate() {
        Quackable mallardDuck = new MallardDuck();
        Quackable redheadDuck = new RedheadDuck();
        Quackable duckCall = new DuckCall();
        Quackable rubberDuck = new RubberDuck();
        Quackable gooseDuck = new GooseAdapter(new Goose());

        System.out.println("\nDuck Simulator: With Goose Adapter");

        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
        simulate(gooseDuck);
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

We make a Goose that acts like a Duck by wrapping the Goose in the GooseAdapter.

Once the Goose is wrapped, we can treat it just like other duck Quackables.

⑦ Now let's give this a quick run....

This time when we run the simulator, the list of objects passed to the simulate() method includes a Goose wrapped in a duck adapter. The result? We should see some honking!

There's the goose! Now the Goose can quack with the rest of the Ducks.

```
File Edit Window Help GoldenEggs
% java DuckSimulator
Duck Simulator: With Goose Adapter
Quack
Quack
Kwak
Squeak
Honk
```

QUACKOLOGY

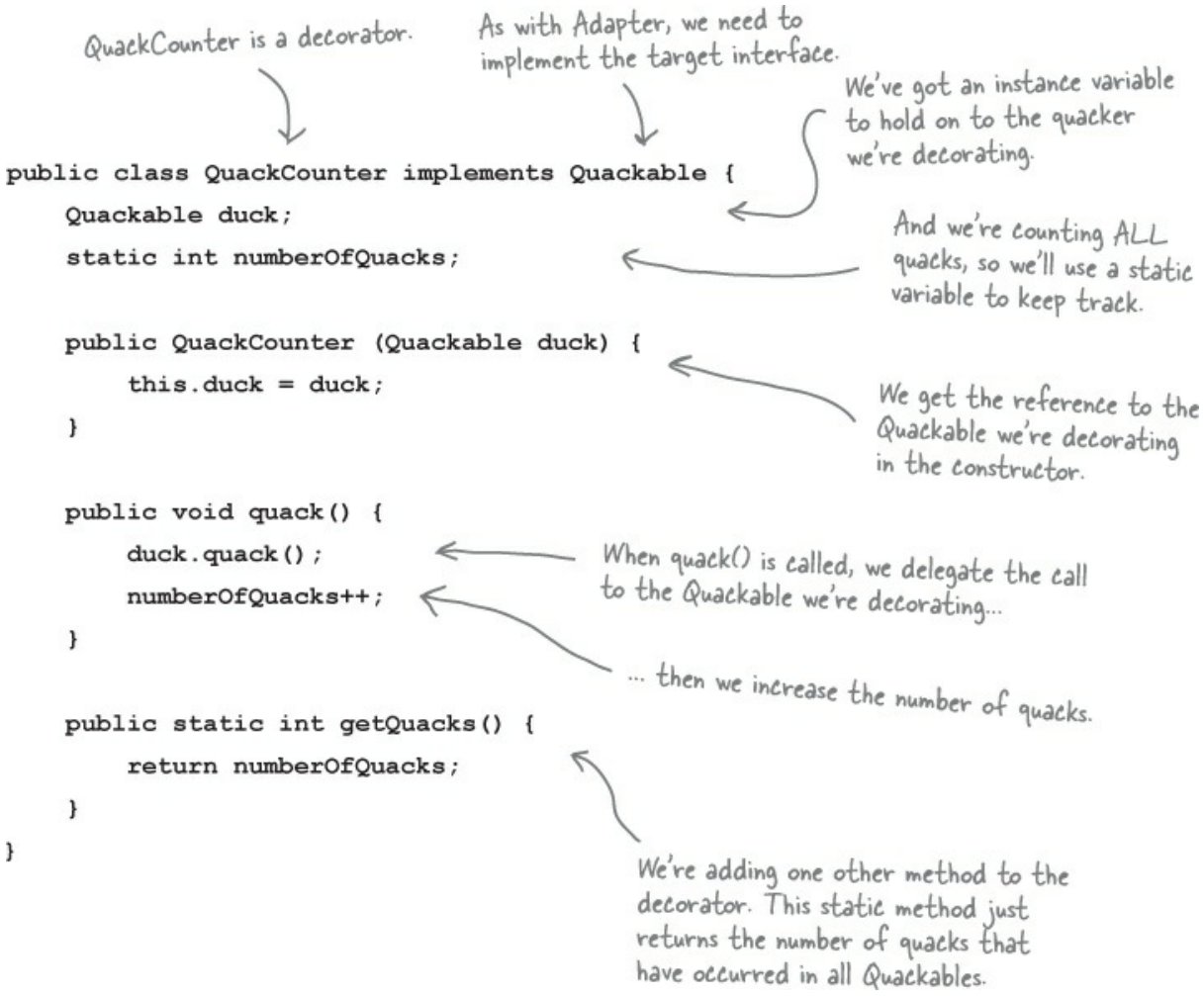
Quackologists are fascinated by all aspects of Quackable behavior. One thing Quackologists have always wanted to study is the total number of quacks made by a flock of ducks.

How can we add the ability to count duck quacks without having to change the duck classes?

Can you think of a pattern that would help?

⑧ **We're going to make those Quackologists happy and give them some quack counts.**

How? Let's create a decorator that gives the ducks some new behavior (the behavior of counting) by wrapping them with a decorator object. We won't have to change the Duck code at all.




```
public class QuackCounter implements Quackable {
    Quackable duck;
    static int numberOfQuacks;

    public QuackCounter (Quackable duck) {
        this.duck = duck;
    }

    public void quack() {
        duck.quack();
        numberOfQuacks++;
    }

    public static int getQuacks() {
        return numberOfQuacks;
    }
}
```

⑨ **We need to update the simulator to create decorated ducks.**
Now, we must wrap each Quackable object we instantiate in a QuackCounter decorator. If we don't, we'll have ducks running around making uncounted quacks.

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        simulator.simulate();
    }

    void simulate() {
        Quackable mallardDuck = new QuackCounter(new MallardDuck());
        Quackable redheadDuck = new QuackCounter(new RedheadDuck());
        Quackable duckCall = new QuackCounter(new DuckCall());
        Quackable rubberDuck = new QuackCounter(new RubberDuck());
        Quackable gooseDuck = new GooseAdapter(new Goose());

        System.out.println("\nDuck Simulator: With Decorator");

        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
        simulate(gooseDuck);

        System.out.println("The ducks quacked " +
                           QuackCounter.getQuacks() + " times");
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

Each time we create a Quackable, we wrap it with a new decorator.

The park ranger told us he didn't want to count geese honks, so we don't decorate it.

Here's where we gather the quacking behavior for the Quackologists.

Nothing changes here; the decorated objects are still Quackables.

Here's the output!

```
File Edit Window Help DecoratedEggs
% java DuckSimulator
Duck Simulator: With Decorator
Quack
Quack
Kwak
Squeak
Honk
4 quacks were counted
%
```

Remember, we're not counting geese.

You have to decorate objects to get decorated behavior.
He's right, that's the problem with wrapping objects: you have to make sure they get wrapped or they don't get the decorated behavior.
Why don't we take the creation of ducks and localize it in one place; in other words, let's take the duck creation and decorating and encapsulate it. What pattern does that sound like?
⑩ **We need a factory to produce ducks!**
Okay, we need some quality control to make sure our ducks get wrapped. We're going to build an entire factory just to produce them. The factory should produce a family of products that consists of different types of ducks, so we're going to use the Abstract Factory Pattern.
Let's start with the definition of the AbstractDuckFactory:

We're defining an abstract factory that subclasses will implement to create different families.

```
public abstract class AbstractDuckFactory {

    public abstract Quackable createMallardDuck();
    public abstract Quackable createRedheadDuck();
    public abstract Quackable createDuckCall();
    public abstract Quackable createRubberDuck();
}
```

Each method creates one kind of duck.

Let's start by creating a factory that creates ducks without decorators, just to get the hang of the factory:

DuckFactory extends the abstract factory.

```
public class DuckFactory extends AbstractDuckFactory {

    public Quackable createMallardDuck() {
        return new MallardDuck();
    }

    public Quackable createRedheadDuck() {
        return new RedheadDuck();
    }

    public Quackable createDuckCall() {
        return new DuckCall();
    }

    public Quackable createRubberDuck() {
        return new RubberDuck();
    }
}
```

Each method creates a product: a particular kind of Quackable. The actual product is unknown to the simulator – it just knows it's getting a Quackable.

Now let's create the factory we really want, the CountingDuckFactory:

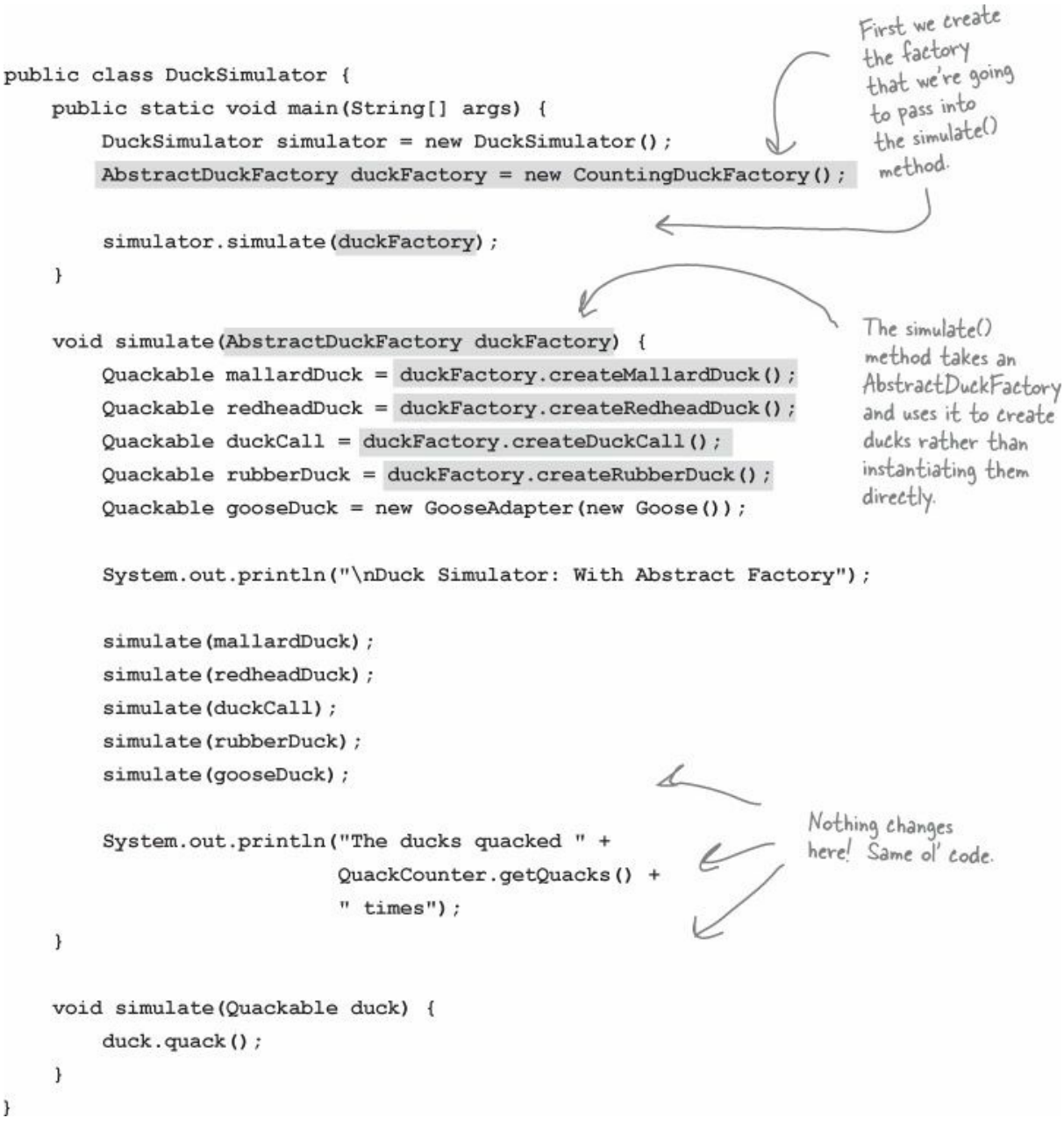
CountingDuckFactory also extends the abstract factory.

```
public class CountingDuckFactory extends AbstractDuckFactory {

    public Quackable createMallardDuck() {
        return new QuackCounter(new MallardDuck());
    }

    public Quackable createRedheadDuck() {
        return new QuackCounter(new RedheadDuck());
    }

    public Quackable createDuckCall() {
        return new QuackCounter(new DuckCall());
    }

    public Quackable createRubberDuck() {
        return new QuackCounter(new RubberDuck());
    }
}
```

Each method wraps the Quackable with the quack counting decorator. The simulator will never know the difference; it just gets back a Quackable. But now our rangers can be sure that all quacks are being counted.

⑪ **Let's set up the simulator to use the factory.**

Remember how Abstract Factory works? We create a polymorphic method that takes a factory and uses it to create objects. By passing in different factories, we get to use different product families in the method. We're going to alter the simulate() method so that it takes a factory and uses it to create ducks.




```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        AbstractDuckFactory duckFactory = new CountingDuckFactory();

        simulator.simulate(duckFactory);
    }

    void simulate(AbstractDuckFactory duckFactory) {
        Quackable mallardDuck = duckFactory.createMallardDuck();
        Quackable redheadDuck = duckFactory.createRedheadDuck();
        Quackable duckCall = duckFactory.createDuckCall();
        Quackable rubberDuck = duckFactory.createRubberDuck();
        Quackable gooseDuck = new GooseAdapter(new Goose());

        System.out.println("\nDuck Simulator: With Abstract Factory");

        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
        simulate(gooseDuck);

        System.out.println("The ducks quacked " +
                           QuackCounter.getQuacks() +
                           " times");
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

NOTE

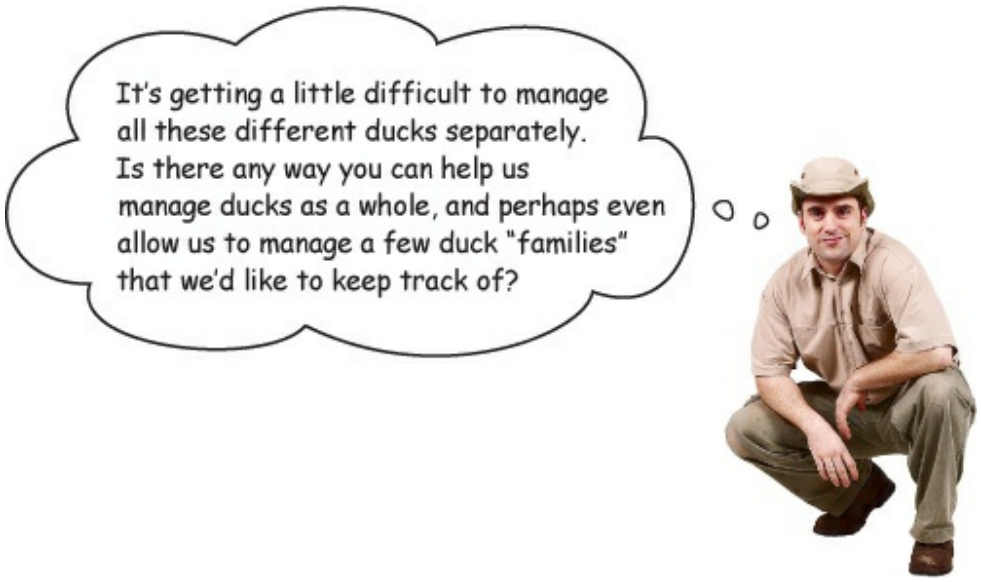
Here's the output using the factory...

Same as last time, but this time we're ensuring that the ducks are all decorated because we are using the CountingDuckFactory.

```
File Edit Window Help EggFactory
% java DuckSimulator
Duck Simulator: With Abstract Factory
Quack
Quack
Kwak
Squeak
Honk
4 quacks were counted
%
```

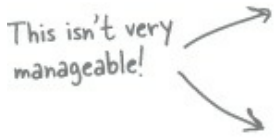
SHARPEN YOUR PENCIL

We're still directly instantiating Geese by relying on concrete classes. Can you write an Abstract Factory for Geese? How should it handle creating "goose ducks"?



Ah, he wants to manage a flock of ducks.

Here's another good question from Ranger Brewer: Why are we managing ducks individually?




```
Quackable mallardDuck = duckFactory.createMallardDuck();
Quackable redheadDuck = duckFactory.createRedheadDuck();
Quackable duckCall = duckFactory.createDuckCall();
Quackable rubberDuck = duckFactory.createRubberDuck();
Quackable gooseDuck = new GooseAdapter(new Goose());

simulate(mallardDuck);
simulate(redheadDuck);
simulate(duckCall);
simulate(rubberDuck);
simulate(gooseDuck);
```

What we need is a way to talk about collections of ducks and even sub-collections of ducks (to deal with the family request from Ranger Brewer). It would also be nice if we could apply operations across the whole set of ducks.

What pattern can help us?

⑫ **Let's create a flock of ducks (well, actually a flock of Quackables).** Remember the Composite Pattern that allows us to treat a collection of objects in the same way as individual objects? What better composite than a flock of Quackables!
Let's step through how this is going to work:

Remember, the composite needs to implement the same interface as the leaf elements. Our leaf elements are Quackables.

We're using an ArrayList inside each Flock to hold the Quackables that belong to the Flock.

```
public class Flock implements Quackable {
    ArrayList<Quackable> quackers = new ArrayList<Quackable>();

    public void add(Quackable quacker) {
        quackers.add(quacker);
    }

    public void quack() {
        Iterator<Quackable> iterator = quackers.iterator();
        while (iterator.hasNext()) {
            Quackable quacker = iterator.next();
            quacker.quack();
        }
    }
}
```

The add() method adds a Quackable to the Flock.

Now for the quack() method – after all, the Flock is a Quackable too. The quack() method in Flock needs to work over the entire Flock. Here we iterate through the ArrayList and call quack() on each element.

CODE UP CLOSE

Did you notice that we tried to sneak a Design Pattern by you without mentioning it?

```
public void quack() {
    Iterator<Quackable> iterator = quackers.iterator();
    while (iterator.hasNext()) {
        Quackable quacker = iterator.next();
        quacker.quack();
    }
}
```

There it is! The Iterator Pattern at work!

⑬ Now we need to alter the simulator.

Our composite is ready; we just need some code to round up the ducks into the composite structure.

```
public class DuckSimulator {
    // main method here

    void simulate(AbstractDuckFactory duckFactory) {
        Quackable redheadDuck = duckFactory.createRedheadDuck();
        Quackable duckCall = duckFactory.createDuckCall();
        Quackable rubberDuck = duckFactory.createRubberDuck();
        Quackable gooseDuck = new GooseAdapter(new Goose());

        System.out.println("\nDuck Simulator: With Composite - Flocks");

        Flock flockOfDucks = new Flock();

        flockOfDucks.add(redheadDuck);
        flockOfDucks.add(duckCall);
        flockOfDucks.add(rubberDuck);
        flockOfDucks.add(gooseDuck);

        Flock flockOfMallards = new Flock();

        Quackable mallardOne = duckFactory.createMallardDuck();
        Quackable mallardTwo = duckFactory.createMallardDuck();
        Quackable mallardThree = duckFactory.createMallardDuck();
        Quackable mallardFour = duckFactory.createMallardDuck();

        flockOfMallards.add(mallardOne);
        flockOfMallards.add(mallardTwo);
        flockOfMallards.add(mallardThree);
        flockOfMallards.add(mallardFour);

        flockOfDucks.add(flockOfMallards);

        System.out.println("\nDuck Simulator: Whole Flock Simulation");
        simulate(flockOfDucks);

        System.out.println("\nDuck Simulator: Mallard Flock Simulation");
        simulate(flockOfMallards);

        System.out.println("\nThe ducks quacked " +
                           QuackCounter.getQuacks() +
                           " times");
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

Create all the Quackables, just like before.

First we create a Flock, and load it up with Quackables.

Then we create a new Flock of mallards.

Here we're creating a little family of mallards...

...and adding them to the Flock of mallards.

Then we add the Flock of mallards to the main flock.

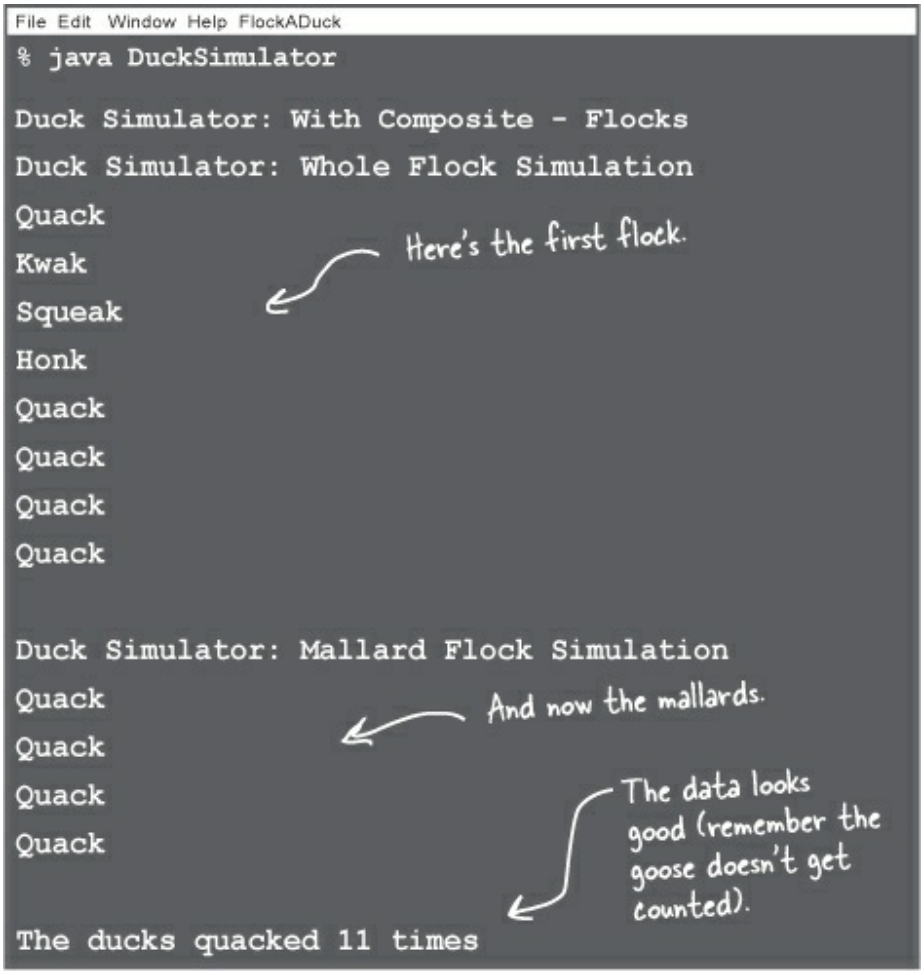
Let's test out the entire Flock!

Then let's just test out the mallard's Flock.

Finally, let's give the Quackologist the data.

Nothing needs to change here; a Flock is a Quackable!

Let’s give it a spin...



SAFETY VERSUS TRANSPARENCY

You might remember that in the Composite Pattern chapter the composites (the Menus) and the leaf nodes (the MenuItems) had the *same* exact set of methods, including the add() method. Because they had the same set of methods, we could call methods on MenuItems that didn't really make sense (like trying to add something to a MenuItem by calling add()). The benefit of this was that the distinction between leaves and composites was *transparent*: the client didn't have to know whether it was dealing with a leaf or a composite; it just called the same methods on both.

Here, we've decided to keep the composite's child maintenance methods separate from the leaf nodes: that is, only Flocks have the add() method. We know it doesn't make sense to try to add something to a Duck, and in this implementation, you can't. You can only add() to a Flock. So this design is *safer* — you can't call methods that don't make sense on components — but it's less transparent. Now the client has to know that a Quackable is a Flock in order to add Quackables to it.

As always, there are trade-offs when you do OO design and you need to consider them

as you create your own composites.

Can you say "observer"?

It sounds like the Quackologist would like to observe individual duck behavior. That leads us right to a pattern made for observing the behavior of objects: the Observer Pattern.

⑭ **First we need an Observable interface.**
Remember that an Observable is the object being observed. An Observable needs methods for registering and notifying observers. We could also have a method for removing observers, but we'll keep the implementation simple here and leave that out.

QuackObservable is the interface that Quackables should implement if they want to be observed.

```
public interface QuackObservable {
    public void registerObserver(Observer observer);
    public void notifyObservers();
}
```

It has a method for registering Observers. Any object implementing the Observer interface can listen to quacks. We'll define the Observer interface in a sec.

It also has a method for notifying the observers.

Now we need to make sure all Quackables implement this interface...

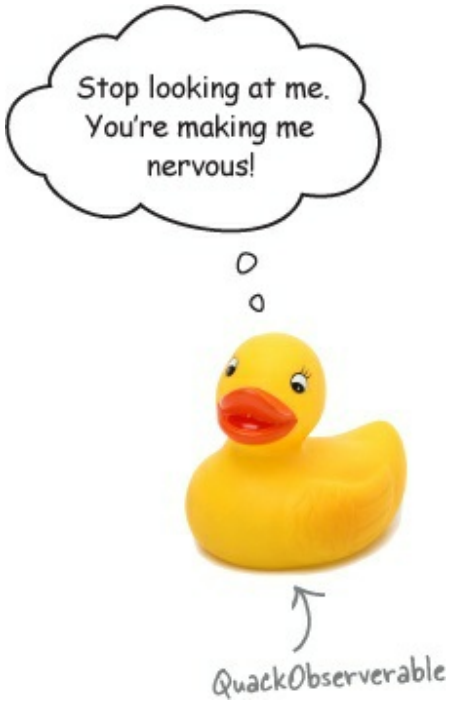
```
public interface Quackable extends QuackObservable {
    public void quack();
}
```

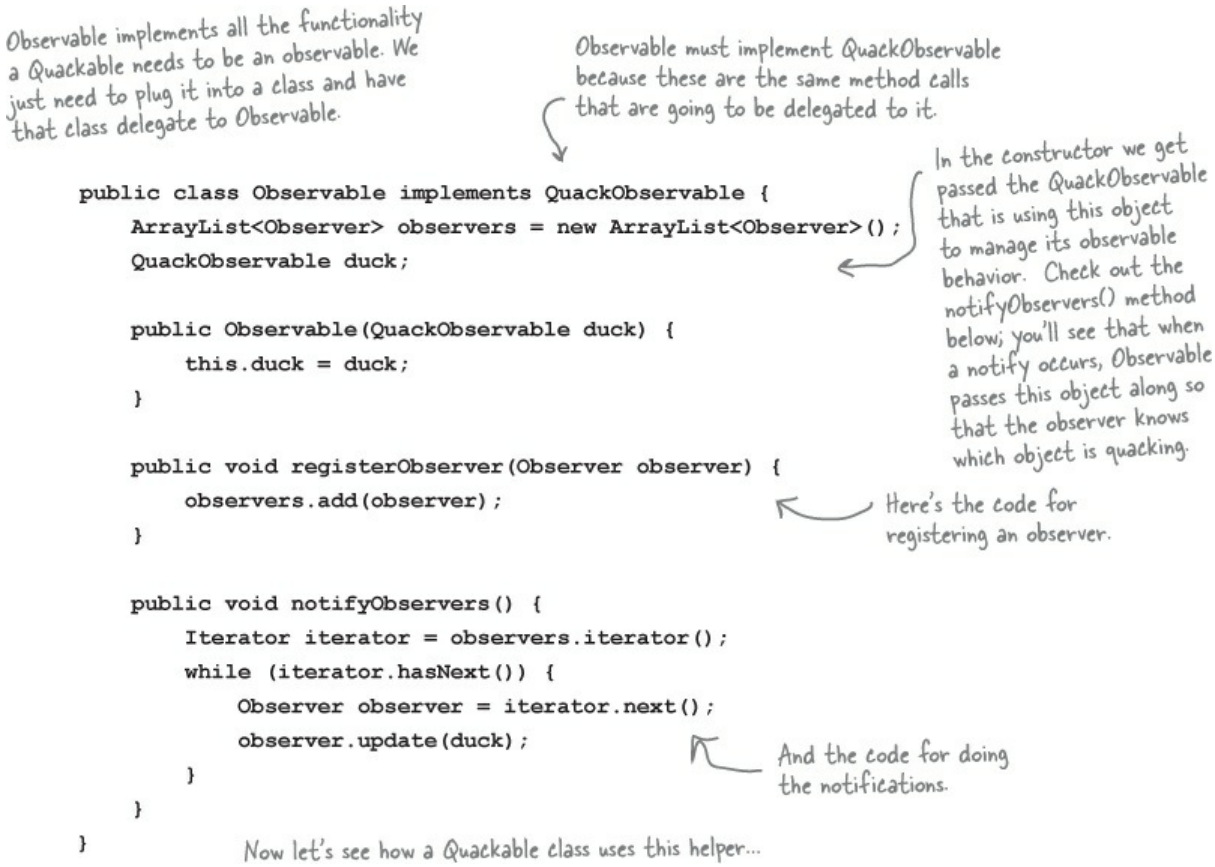
So, we extend the Quackable interface with QuackObserver.

⑮ Now, we need to make sure all the concrete classes that implement Quackable can handle being a QuackObservable.

We could approach this by implementing registration and notification in each and every class (like we did in Chapter 2). But we're going to do it a little differently this time: we're going to encapsulate the registration and notification code in another class, call it Observable, and compose it with a QuackObservable. That way, we only write the real code once and the QuackObservable just needs enough code to delegate to the helper class Observable.

Let's begin with the Observable helper class.






```
public class Observable implements QuackObservable {
    ArrayList<Observer> observers = new ArrayList<Observer>();
    QuackObservable duck;

    public Observable(QuackObservable duck) {
        this.duck = duck;
    }

    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    public void notifyObservers() {
        Iterator iterator = observers.iterator();
        while (iterator.hasNext()) {
            Observer observer = iterator.next();
            observer.update(duck);
        }
    }
}
```

⑯ Integrate the helper Observable with the Quackable classes.
This shouldn't be too bad. All we need to do is make sure the Quackable classes are composed with an Observable and that they know how to delegate to it. After that, they're ready to be Observables. Here's the implementation of MallardDuck; the other ducks are the same.

```
public class MallardDuck implements Quackable {
    Observable observable;

    public MallardDuck() {
        observable = new Observable(this);
    }

    public void quack() {
        System.out.println("Quack");
        notifyObservers();
    }

    public void registerObserver(Observer observer) {
        observable.registerObserver(observer);
    }

    public void notifyObservers() {
        observable.notifyObservers();
    }
}
```

Each Quackable has an Observable instance variable.

In the constructor, we create an Observable and pass it a reference to the MallardDuck object.

When we quack, we need to let the observers know about it.

Here are our two QuackObservable methods. Notice that we just delegate to the helper.

SHARPEN YOUR PENCIL

We haven't changed the implementation of one Quackable, the QuackCounter decorator. We need to make it an Observable too. Why don't you write that one:

⑰ We're almost there! We just need to work on the Observer side of the pattern.

We've implemented everything we need for the Observables; now we need some Observers. We'll start with the Observer interface:

The Observer interface just has one method, update(), which is passed the QuackObservable that is quacking.

```
public interface Observer {
    public void update(QuackObservable duck);
}
```

Now we need an Observer: where are those Quackologists?!

We need to implement the Observable interface or else we won't be able to register with a QuackObservable.

```
public class Quackologist implements Observer {

    public void update(QuackObservable duck) {
        System.out.println("Quackologist: " + duck + " just quacked.");
    }
}
```

The Quackologist is simple; it just has one method, update(), which prints out the Quackable that just quacked.

SHARPEN YOUR PENCIL

What if a Quackologist wants to observe an entire flock? What does that mean anyway? Think about it like this: if we observe a composite, then we're observing everything in the composite. So, when you register with a flock, the flock composite makes sure you get registered with all its children (sorry, all its little quackers), which may include other flocks.

Go ahead and write the Flock observer code before we go any further.

⑱ We're ready to observe. Let's update the simulator and give it a try:

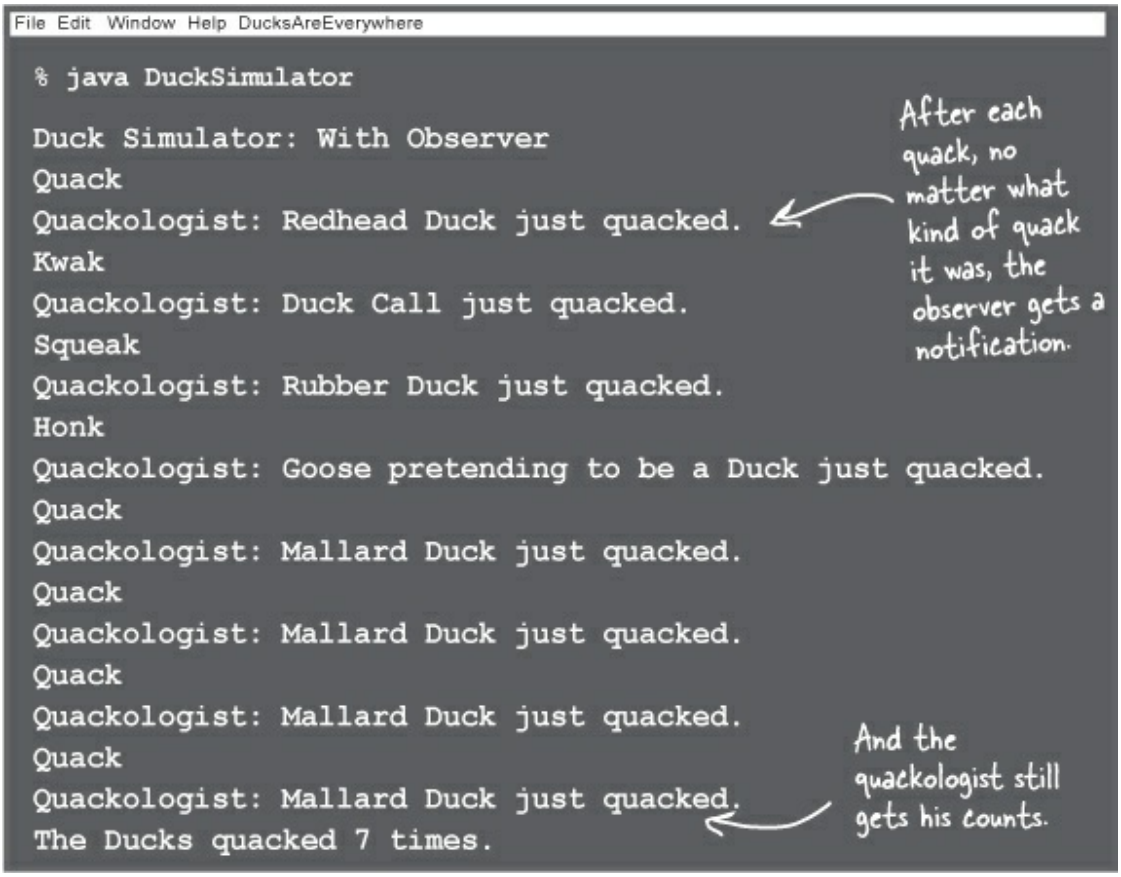
```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        AbstractDuckFactory duckFactory = new CountingDuckFactory();

        simulator.simulate(duckFactory);
    }

    void simulate(AbstractDuckFactory duckFactory) {

        // create duck factories and ducks here

        // create flocks here

        System.out.println("\nDuck Simulator: With Observer");

        Quackologist quackologist = new Quackologist();
        flockOfDucks.registerObserver(quackologist);

        simulate(flockOfDucks);

        System.out.println("\nThe ducks quacked " +
                           QuackCounter.getQuacks() +
                           " times");
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

All we do here is create a Quackologist and set him as an observer of the flock.

This time we'll we just simulate the entire flock.

Let's give it a try and see how it works!

This is the big finale. Five, no, six patterns have come together to create this amazing Duck Simulator. Without further ado, we present the DuckSimulator!



THERE ARE NO DUMB QUESTIONS

Q: **Q: So this was a compound pattern?**

A: **A:** No, this was just a set of patterns working together. A compound pattern is a set of a few patterns that are combined to solve a general problem. We're just about to take a look at the Model-View-Controller compound pattern; it's a collection of a few patterns that has been used over and over in many design solutions.

Q: **Q: So the real beauty of Design Patterns is that I can take a problem, and start applying patterns to it until I have a solution. Right?**

A: **A:** Wrong. We went through this exercise with Ducks to show you how patterns *can* work together. You'd never actually want to approach a design like we just did. In fact, there may be solutions to parts of the Duck Simulator for which some of these patterns were big time overkill. Sometimes just using good OO design principles can solve a problem well enough on its own.

We're going to talk more about this in the next chapter, but you only want to apply patterns when and where they make sense. You never want to start out with the intention of using patterns just for the sake of it. You should consider the design of the Duck Simulator to be forced and artificial. But hey, it was fun and gave us a good idea of how several patterns can fit into a solution.

What did we do?

We started with a bunch of Quackables...

A goose came along and wanted to act like a Quackable too. So we used

the *Adapter Pattern* to adapt the goose to a Quackable. Now, you can call quack() on a goose wrapped in the adapter and it will honk!

Then, the Quackologists decided they wanted to count quacks. So we used the *Decorator Pattern* to add a QuackCounter decorator that keeps track of the number of times quack() is called, and then delegates the quack to the Quackable it's wrapping.

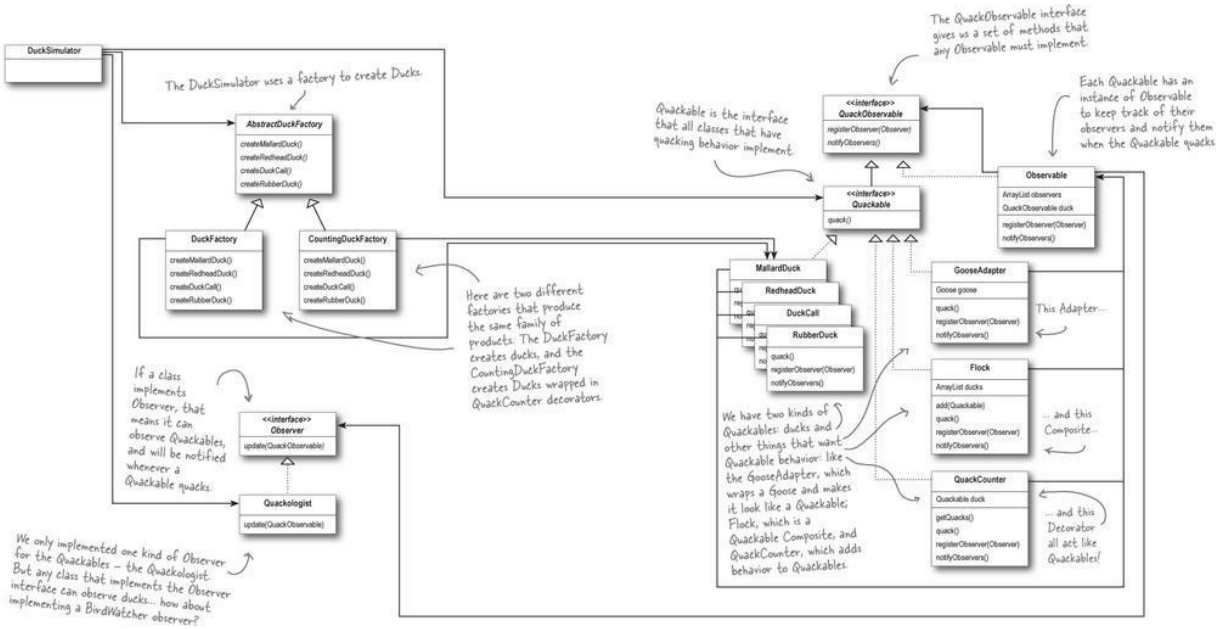
But the Quackologists were worried they'd forget to add the QuackCounter decorator. So we used the *Abstract Factory Pattern* to create ducks for them. Now, whenever they want a duck, they ask the factory for one, and it hands back a decorated duck. (And don't forget, they can also use another duck factory if they want an un-decorated duck!)

We had management problems keeping track of all those ducks and geese and quackables. So we used the *Composite Pattern* to group Quackables into Flocks. The pattern also allows the Quackologist to create sub-Flocks to manage duck families. We used the *Iterator Pattern* in our implementation by using java.util's iterator in ArrayList.

The Quackologists also wanted to be notified when any Quackable quacked. So we used the *Observer Pattern* to let the Quackologists register as Quackable Observers. Now they're notified every time any Quackable quacks. We used iterator again in this implementation. The Quackologists can even use the Observer Pattern with their composites.

A duck's eye view: the class diagram

We've packed a lot of patterns into one small duck simulator! Here's the big picture of what we did:



The King of Compound Patterns

If Elvis were a compound pattern, his name would be Model-View-Controller, and he'd be singing a little song like this...

Model, View, Controller Lyrics and music by James Dempsey.	Model a bottle of fine Chardonnay Model all the glottal stops people say Model the coddling of boiling eggs You can model the waddle in Hexley's legs
MVC's a paradigm for factoring your code into functional segments, so your brain does not explode. To achieve reusability, you gotta keep those boundaries clean Model on the one side, View on the other, the Controller's in between.	Model View, you can model all the models that pose for GQ Model View Controller
	

	NOTE So does Java! View objects tend to be controls used to display and edit Cocoa’s got a lot of those, well written to its credit. Take an NSTextView, hand it any old Unicode string The user can interact with it, it can hold most anything But the view don’t know about the Model That string could be a phone number or the works of Aristotle Keep the coupling loose and so achieve a massive level of reuse So does Java!
Model View, it’s got three layers like Oreos do Model View Controller Model View, Model View, Model View Controller	Model View, all rendered very nicely in Aqua blue Model View Controller
Model objects represent your application’s raison d’être Custom objects that contain data, logic, and et cetera You create custom classes, in your app’s problem domain you can choose to reuse them with all the views but the model objects stay the same.	You’re probably wondering now You’re probably wondering how Data flows between Model and View The Controller has to mediate Between each layer’s changing state To synchronize the data of the two It pulls and pushes every changed value
You can model a throttle and a manifold Model the toddle of a two year old	Model View, mad props to the smalltalk crew!

	Model View Controller
Model View, it’s pronounced Oh Oh not Ooo Ooo Model View Controller	Model View How we gonna deep six all that glue Model View Controller
There’s a little left to this story A few more miles upon this road Nobody seems to get much glory From writing the controller code	Controllers know the Model and View very intimately They often use hardcoding which can be foreboding for reusability But now you can connect each model key that you select to any view property
Well the model’s mission critical And gorgeous is the view I might be lazy, but sometimes it’s just crazy How much code I write is just glue And it wouldn’t be so tragic But the code ain’t doing magic It’s just moving values through	And once you start binding I think you’ll be finding less code in your source tree Yeah I know I was elated by the stuff they’ve automated and the things you get for free
And I don’t mean to be vicious But it gets repetitious Doing all the things controllers do	And I think it bears repeating all the code you won’t be needing when you hook it up in ~~XIB~~ Using Swing
And I wish I had a dime For every single time I sent a TextField StringValue.	Model View, even handles multiple selections too Model View Controller Model View, bet I ship my application before you Model View Controller

EAR POWER

Don’t just read! After all, this is a Head First book... grab your iPod, hit this URL:

http://www.youtube.com/watch?v=YYvOGPMLVDo

Sit back and give it a listen.

No. Design Patterns are your key to the MVC.

We were just trying to whet your appetite. Tell you what, after you finish reading this chapter, go back and listen to the song again — you'll have even more fun.

It sounds like you've had a bad run-in with MVC before? Most of us have. You've probably had other developers tell you it's changed their lives and could possibly create world peace. It's a powerful compound pattern, for sure, and while we can't claim it will create world peace, it will save you hours of writing code once you know it.

But first you have to learn it, right? Well, there's going to be a big difference this time around because *now you know patterns!*

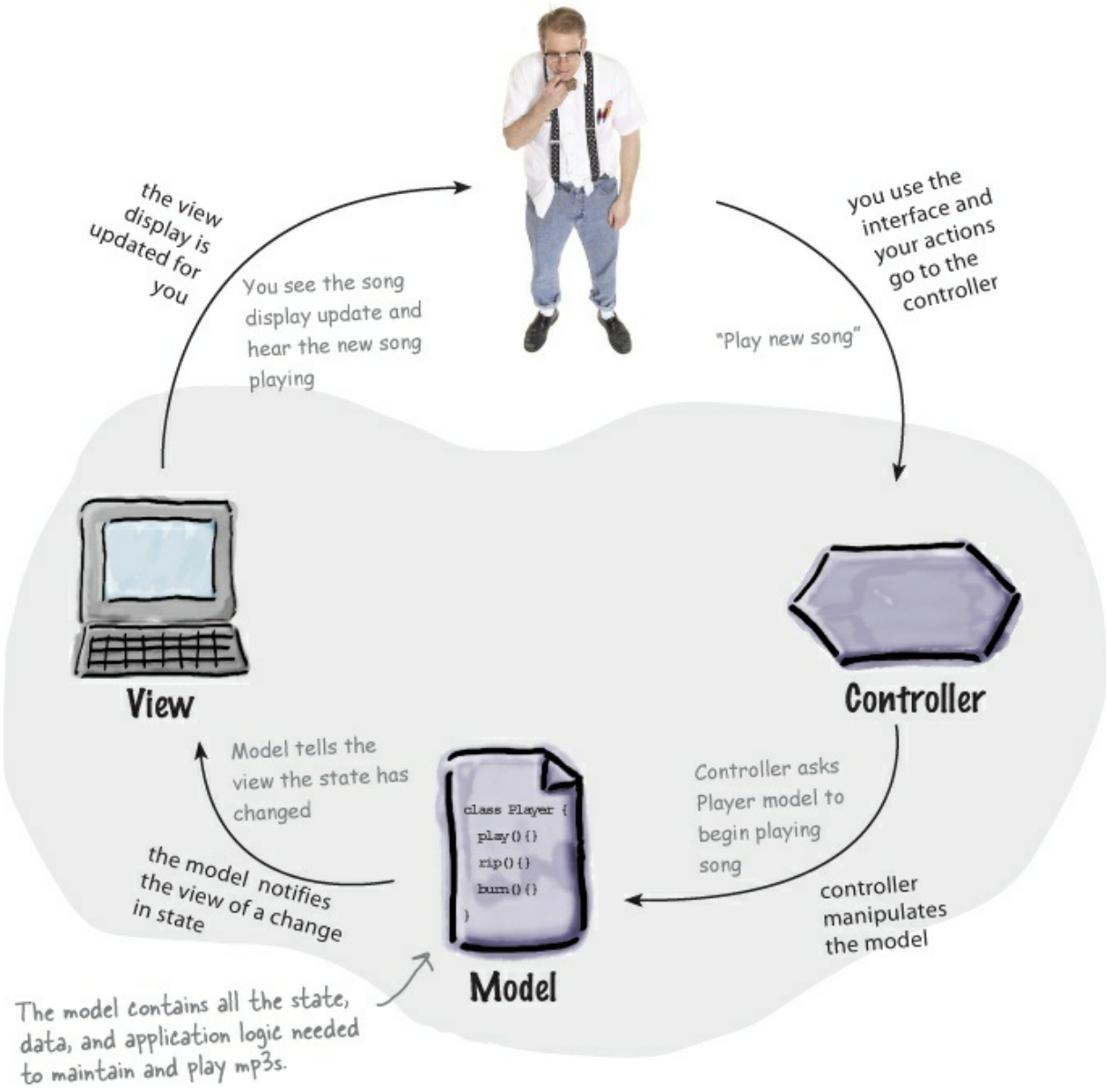
That’s right, patterns are the key to MVC. Learning MVC from the top down is difficult; not many developers succeed. Here’s the secret to learning MVC: *it’s just a few patterns put together.* When you approach learning MVC by looking at the patterns, all of a sudden it starts to make sense.

Let’s get started. This time around you’re going to nail MVC!

Meet the Model-View-Controller

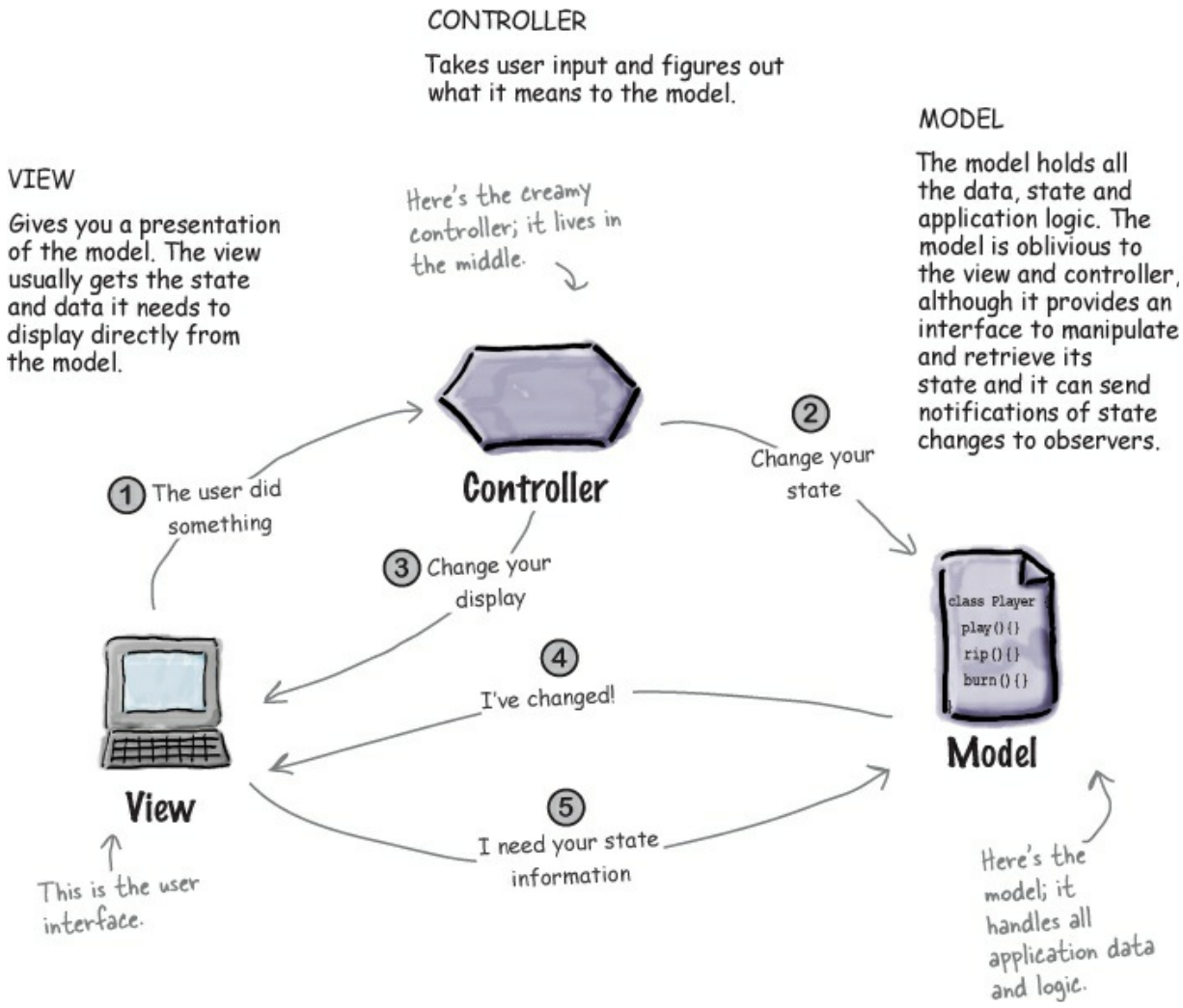
Imagine you’re using your favorite MP3 player, like iTunes. You can use its interface to add new songs, manage playlists and rename tracks. The player takes care of maintaining a little database of all your songs along with their associated names and data. It also takes care of playing the songs and, as it does, the user interface is constantly updated with the current song title, the running time, and so on.

Well, underneath it all sits the Model-View-Controller...



A closer look...

The MP3 player description gives us a high-level view of MVC, but it really doesn't help you understand the nitty gritty of how the compound pattern works, how you'd build one yourself, or why it's such a good thing. Let's start by stepping through the relationships among the model, view and controller, and then we'll take second look from the perspective of Design Patterns.



① **You're the user — you interact with the view.**
The view is your window to the model. When you do something to the view (like click the Play button) then the view tells the controller what you did. It's the controller's job to handle that.
② **The controller asks the model to change its state.**
The controller takes your actions and interprets them. If you click on a button, it's the controller's job to figure out what that means and how the model should be manipulated based on that action.
③ **The controller may also ask the view to change.**
When the controller receives an action from the view, it may need to tell the view to change as a result. For example, the controller could enable or disable certain buttons or menu items in the interface.
④ **The model notifies the view when its state has changed.**

When something changes in the model, based either on some action you took (like clicking a button) or some other internal change (like the next song in the playlist has started), the model notifies the view that its state has changed.

⑤ The view asks the model for state.

The view gets the state it displays directly from the model. For instance, when the model notifies the view that a new song has started playing, the view requests the song name from the model and displays it. The view might also ask the model for state as the result of the controller requesting some change in the view.

THERE ARE NO DUMB QUESTIONS

Q: **Q: Does the controller ever become an observer of the model?**

A: **A:** Sure. In some designs the controller registers with the model and is notified of changes. This can be the case when something in the model directly affects the user interface controls. For instance, certain states in the model may dictate that some interface items be enabled or disabled. If so, it is really controller's job to ask the view to update its display accordingly.

Q: **Q: All the controller does is take user input from the view and send it to the model, correct? Why have it at all if that is all it does? Why not just have the code in the view itself? In most cases isn't the controller just calling a method on the model?**

A: **A:** The controller does more than just "send it to the model"; it is responsible for interpreting the input and manipulating the model based on that input. But your real question is probably "why can't I just do that in the view code?"

You could; however, you don't want to for two reasons. First, you'll complicate your view code because it now has two responsibilities: managing the user interface and dealing with the logic of how to control the model. Second, you're tightly coupling your view to the model. If you want to reuse the view with another model, forget it. The controller separates the logic of control from the view and decouples the view from the model. By keeping the view and controller loosely coupled, you are building a more flexible and extensible design, one that can more easily accommodate change down the road.

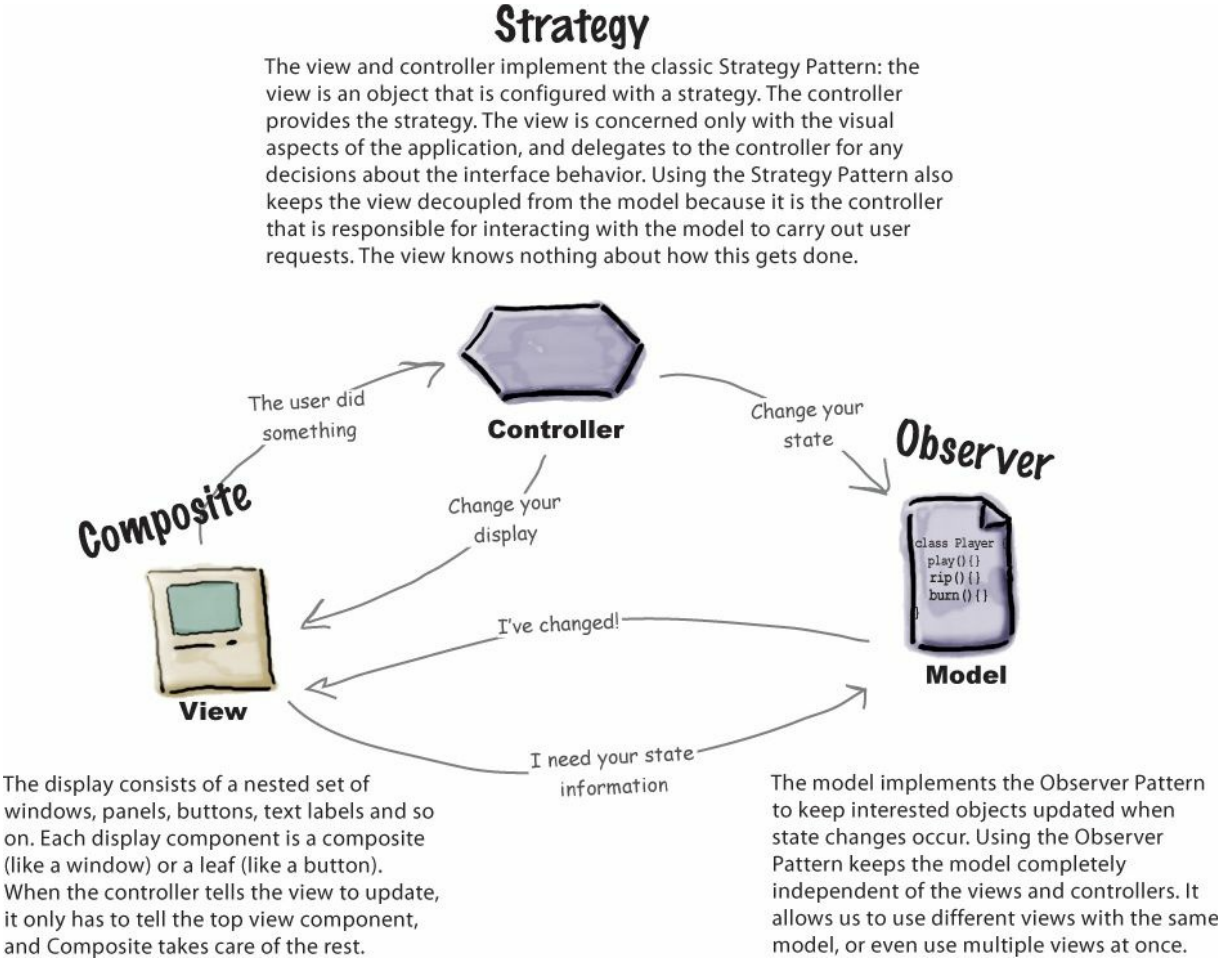
Looking at MVC through patterns-colored glasses

We've already told you the best path to learning the MVC is to see it for what it is: a set of patterns working together in the same design.

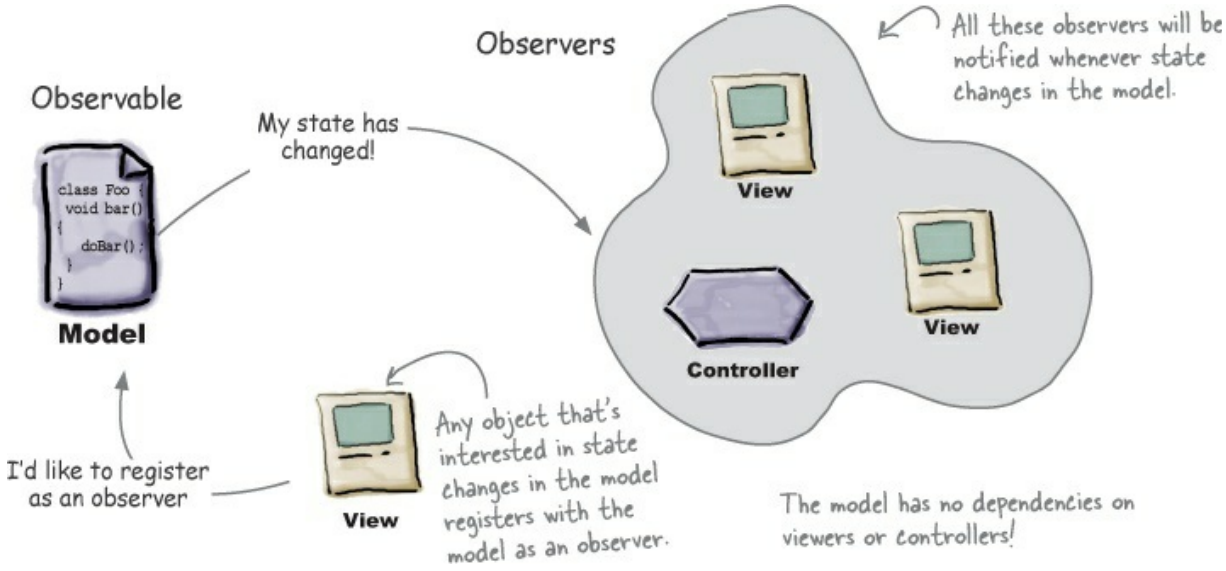
Let's start with the model. As you might have guessed, the model uses

Observer to keep the views and controllers updated on the latest state changes. The view and the controller, on the other hand, implement the Strategy Pattern. The controller is the behavior of the view, and it can be easily exchanged with another controller if you want different behavior. The view itself also uses a pattern internally to manage the windows, buttons and other components of the display: the Composite Pattern.

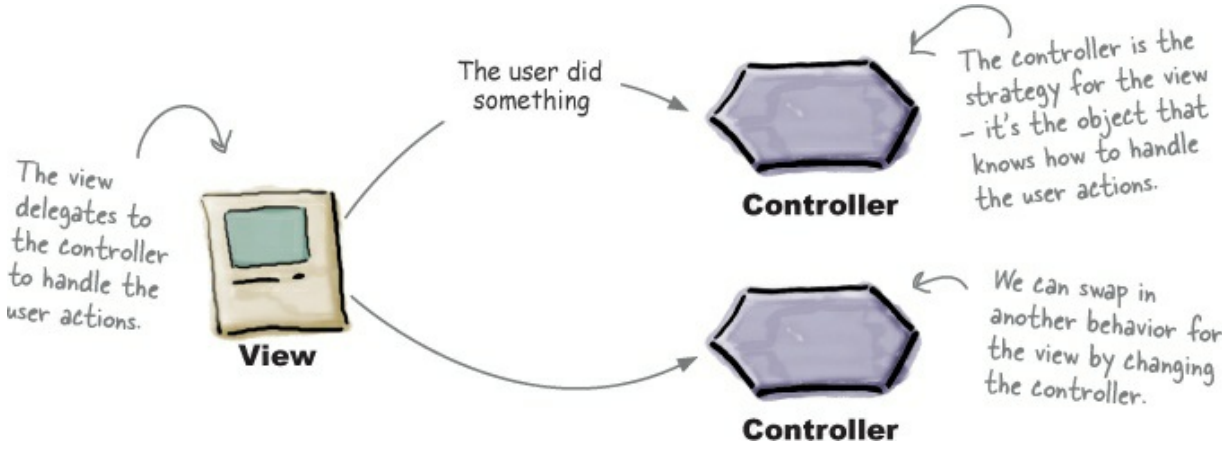
Let’s take a closer look:



Observer



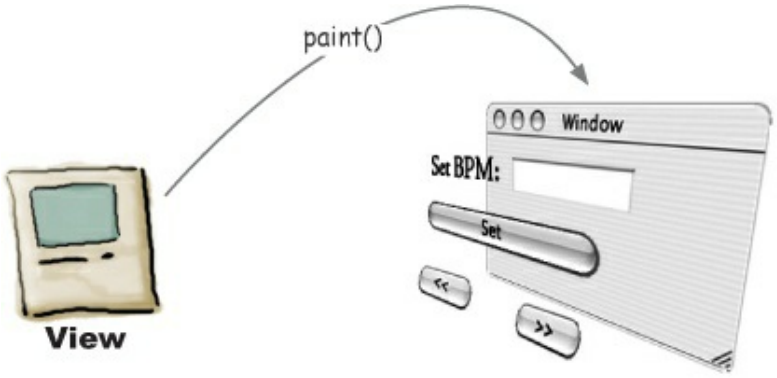
Strategy



NOTE

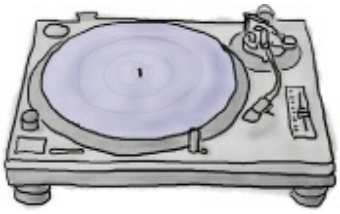
The view only worries about presentation. The controller worries about translating user input to actions on the model.

Composite



The view is a composite of GUI components (labels, buttons, text entry, etc.). The top-level component contains other components, which contain other components, and so on until you get to the leaf nodes.

Using MVC to control the beat...

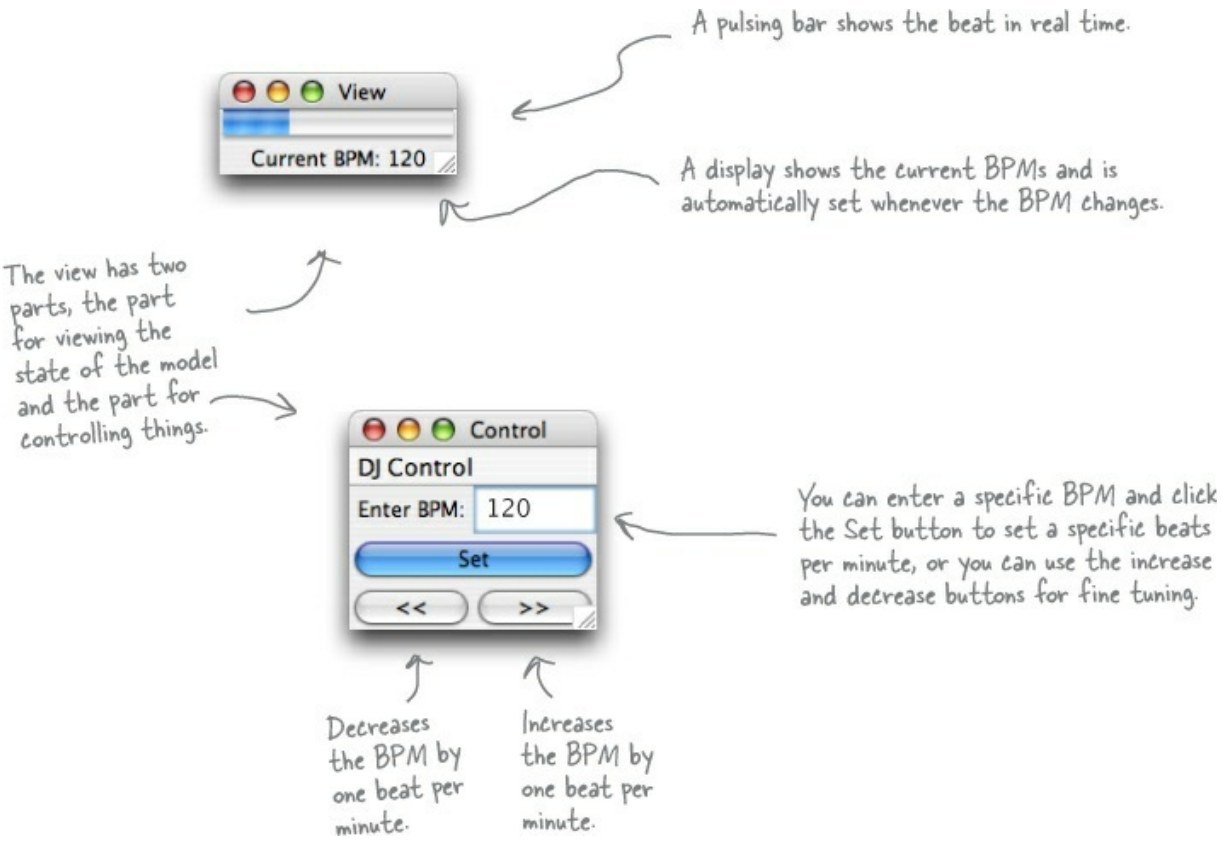


It's your time to be the DJ. When you're a DJ it's all about the beat. You might start your mix with a slowed, downtempo groove at 95 beats per minute (BPM) and then bring the crowd up to a frenzied 140 BPM of trance techno. You'll finish off your set with a mellow 80 BPM ambient mix.

How are you going to do that? You have to control the beat and you're going to build the tool to get you there.

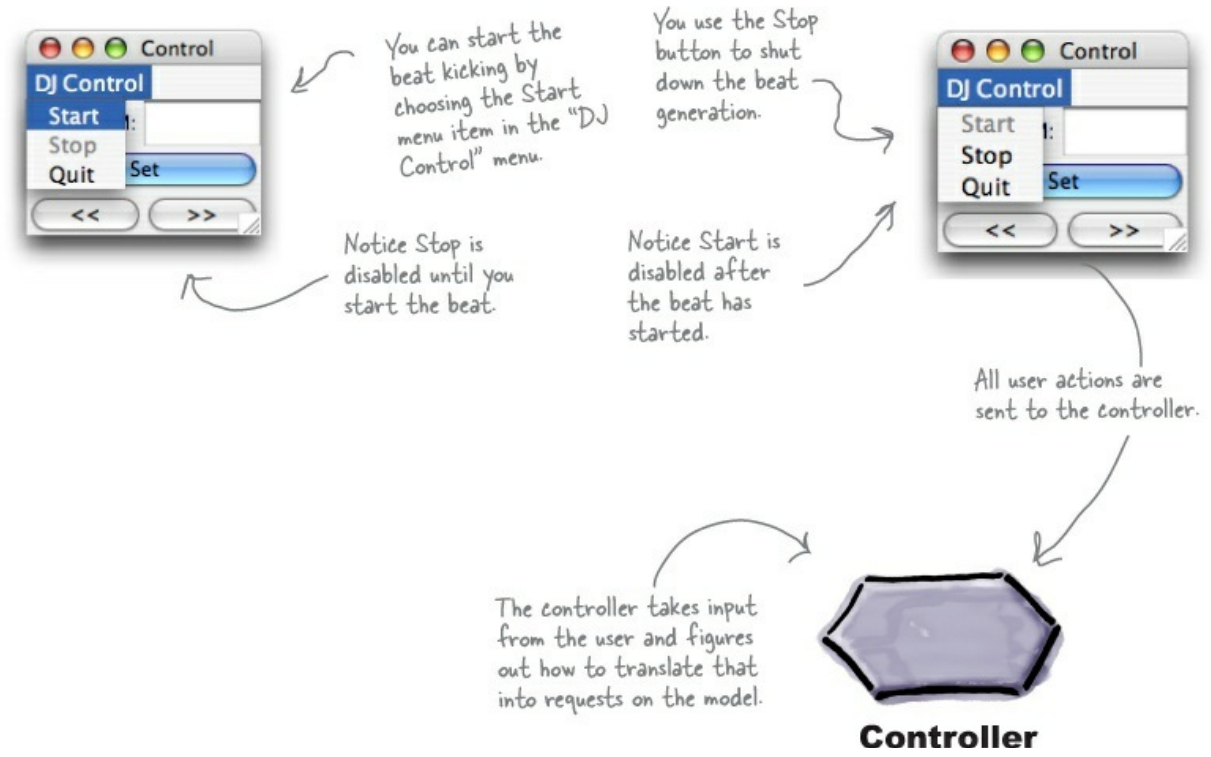
Meet the Java DJ View

Let's start with the **view** of the tool. The view allows you to create a driving drum beat and tune its beats per minute...



NOTE

Here are a few more ways to control the DJ View...

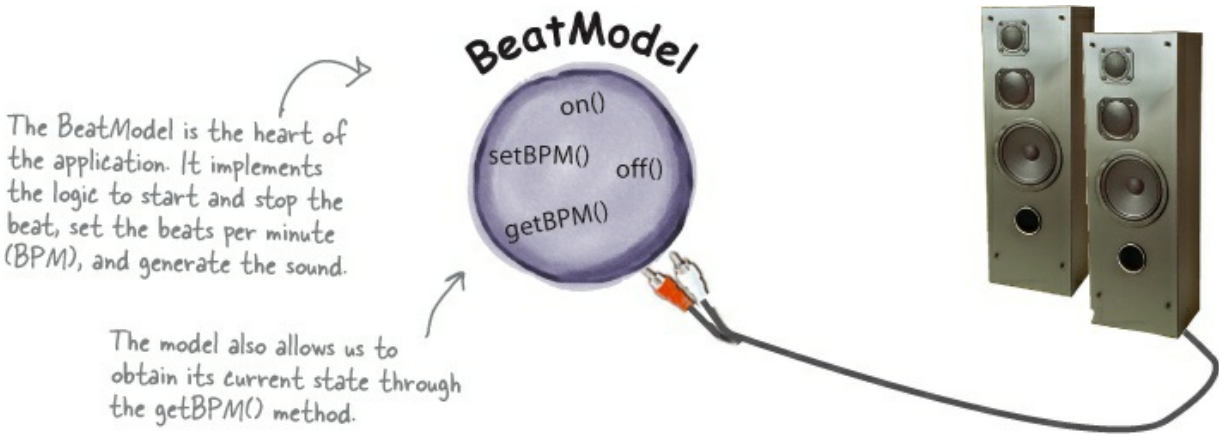


The controller is in the middle...

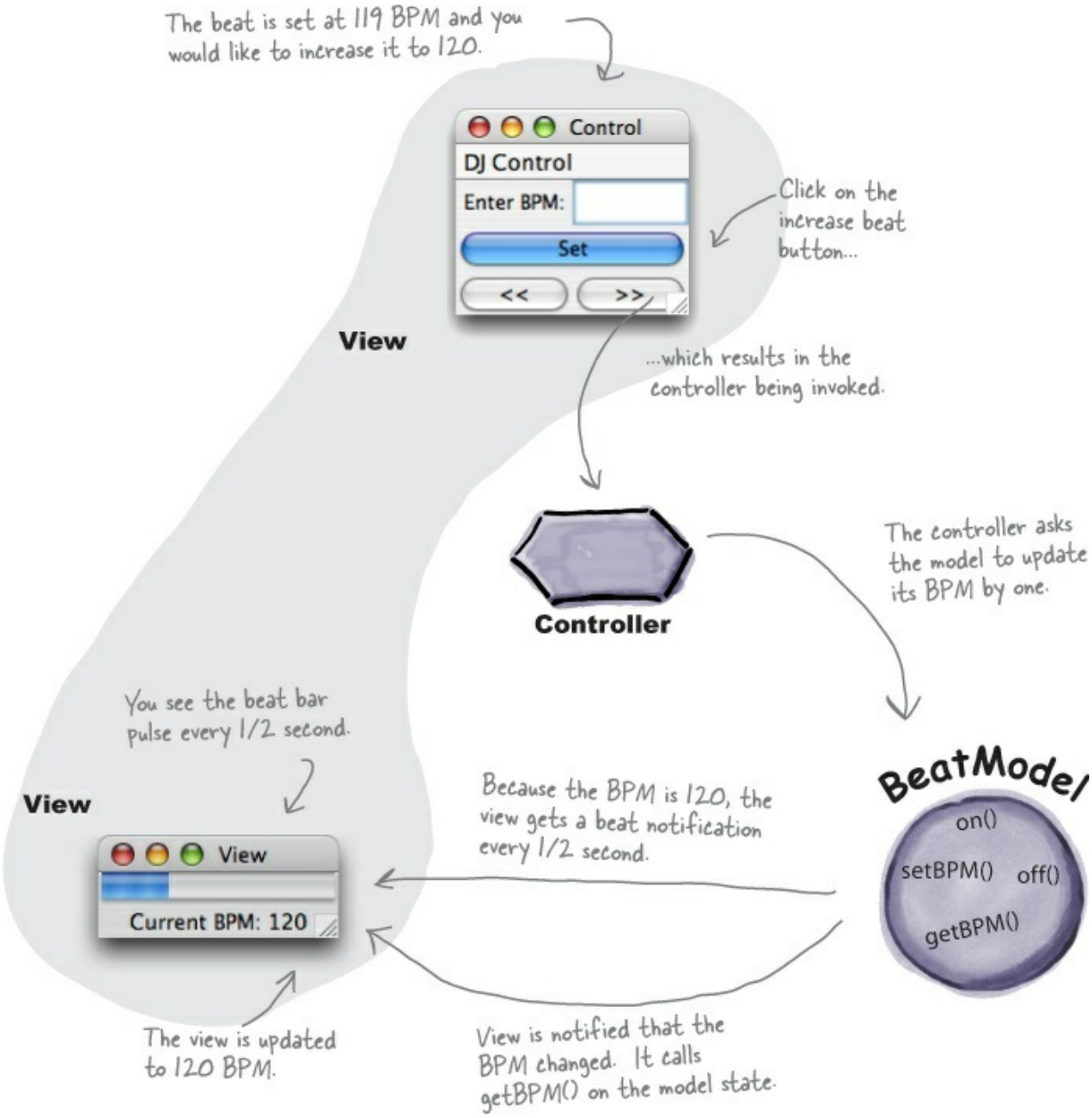
The **controller** sits between the view and model. It takes your input, like selecting “Start” from the DJ Control menu, and turns it into an action on the model to start the beat generation.

Let’s not forget about the model underneath it all...

You can’t see the **model**, but you can hear it. The model sits underneath everything else, managing the beat and driving the speakers with MIDI.



Putting the pieces together

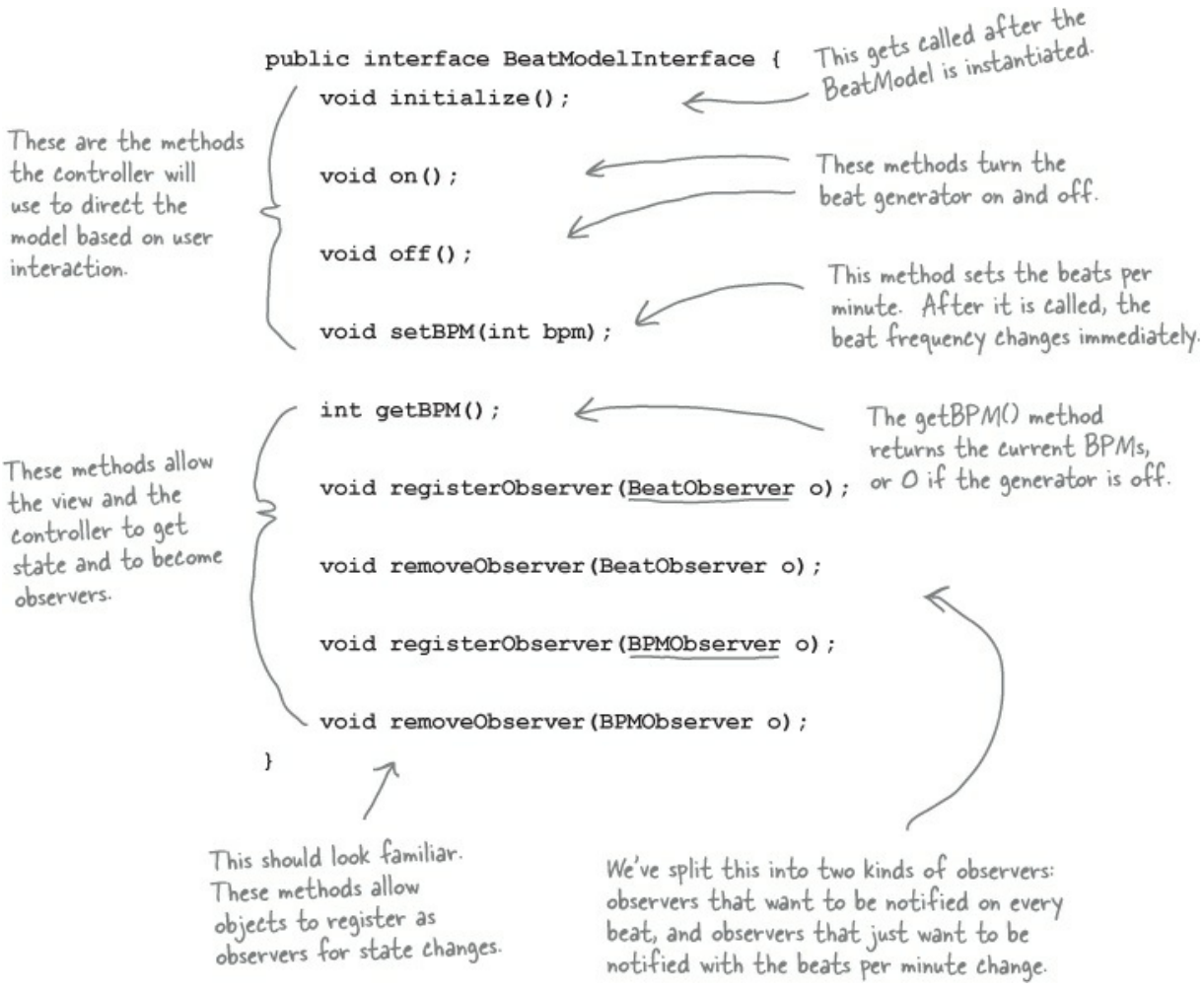


Building the pieces

Okay, you know the model is responsible for maintaining all the data, state and any application logic. So what's the BeatModel got in it? Its main job is managing the beat, so it has state that maintains the current beats per minute and lots of code that generates MIDI events to create the beat that we hear. It also exposes an interface that lets the controller manipulate the beat and lets

the view and controller obtain the model's state. Also, don't forget that the model uses the Observer Pattern, so we also need some methods to let objects register as observers and send out notifications.

Let's check out the BeatModelInterface before looking at the implementation



Now let's have a look at the concrete BeatModel class


```
public class BeatModel implements BeatModelInterface, MetaEventListener {
    Sequencer sequencer;
    ArrayList<BeatObserver> beatObservers = new ArrayList<BeatObserver>();
    ArrayList<BPMObserver> bpmObservers = new ArrayList<BPMObserver>();
    int bpm = 90;
    // other instance variables here

    public void initialize() {
        setUpMidi();
        buildTrackAndStart();
    }

    public void on() {
        sequencer.start();
        setBPM(90);
    }

    public void off() {
        setBPM(0);
        sequencer.stop();
    }

    public void setBPM(int bpm) {
        this.bpm = bpm;
        sequencer.setTempoInBPM(getBPM());
        notifyBPMObservers();
    }

    public int getBPM() {
        return bpm;
    }

    void beatEvent() {
        notifyBeatObservers();
    }

    // Code to register and notify observers
    // Lots of MIDI code to handle the beat
}
```

READY BAKE CODE

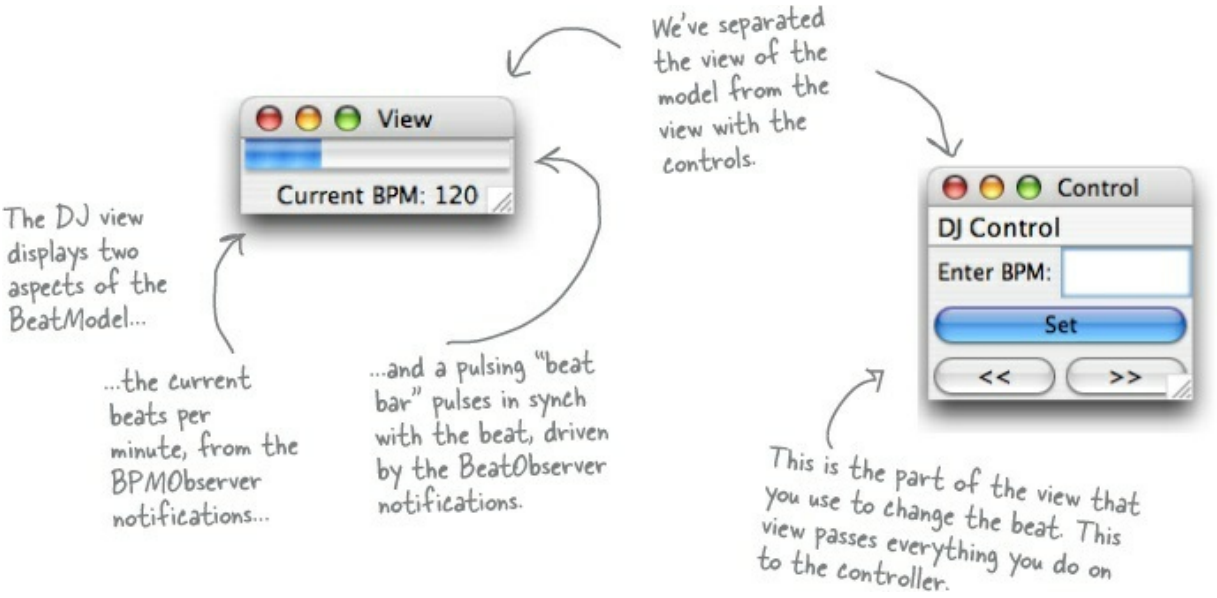
This model uses Java's MIDI support to generate beats. You can check out the complete implementation of all the DJ classes in the Java source files available on the wickedlysmart.com site, or look at the code at the end of the chapter.

The View

Now the fun starts; we get to hook up a view and visualize the BeatModel!

The first thing to notice about the view is that we've implemented it so that it is displayed in two separate windows. One window contains the current BPM

and the pulse; the other contains the interface controls. Why? We wanted to emphasize the difference between the interface that contains the view of the model and the rest of the interface that contains the set of user controls. Let's take a closer look at the two parts of the view:



BRAIN POWER

Our BeatModel makes no assumptions about the view. The model is implemented using the Observer Pattern, so it just notifies any view registered as an observer when its state changes. The view uses the model's API to get access to the state. We've implemented one type of view; can you think of other views that could make use of the notifications and state in the BeatModel?

A lightshow that is based on the real-time beat.

A textual view that displays a music genre based on the BPM (ambient, downbeat, techno, etc.).

Implementing the View

The two parts of the view — the view of the model, and the view with the user interface controls — are displayed in two windows, but live together in

one Java class. We'll first show you just the code that creates the view of the model, which displays the current BPM and the beat bar. Then we'll come back on the next page and show you just the code that creates the user interface controls, which displays the BPM text entry field, and the buttons.

WATCH IT!

The code on these two pages is just an outline!

What we've done here is split ONE class into TWO, showing you one part of the view on this page, and the other part on the next page. All this code is really in ONE class — DJView.java. It's all listed at the end of the chapter.

DJView is an observer for both real-time beats and BPM changes.

```
public class DJView implements ActionListener,  BeatObserver, BPMObserver {
    BeatModelInterface model;
    ControllerInterface controller;
    JFrame viewFrame;
    JPanel viewPanel;
    BeatBar beatBar;
    JLabel bpmOutputLabel;

    public DJView(ControllerInterface controller, BeatModelInterface model) {
        this.controller = controller;
        this.model = model;
        model.registerObserver((BeatObserver)this);
        model.registerObserver((BPMObserver)this);
    }

    public void createView() {
        // Create all Swing components here
    }

    public void updateBPM() {
        int bpm = model.getBPM();
        if (bpm == 0) {
            bpmOutputLabel.setText("offline");
        } else {
            bpmOutputLabel.setText("Current BPM: " + model.getBPM());
        }
    }

    public void updateBeat() {
        beatBar.setValue(100);
    }
}
```

The view holds a reference to both the model and the controller. The controller is only used by the control interface, which we'll go over in a sec...

Here, we create a few components for the display.

The constructor gets a reference to the controller and the model, and we store references to those in the instance variables.

We also register as a BeatObserver and a BPMObserver of the model.

The updateBPM() method is called when a state change occurs in the model. When that happens we update the display with the current BPM. We can get this value by requesting it directly from the model.

Likewise, the updateBeat() method is called when the model starts a new beat. When that happens, we need to pulse our "beat bar." We do this by setting it to its maximum value (100) and letting it handle the animation of the pulse.

Implementing the View, continued...

Now, we'll look at the code for the user interface controls part of the view. This view lets you control the model by telling the controller what to do, which in turn, tells the model what to do. Remember, this code is in the same class file as the other view code.

```
public class DJView implements ActionListener,  BeatObserver, BPMObserver {
    BeatModelInterface model;
    ControllerInterface controller;
    JLabel bpmLabel;
    JTextField bpmTextField;
    JButton setBPMButton;
    JButton increaseBPMButton;
    JButton decreaseBPMButton;
    JMenuBar menuBar;
    JMenu menu;
    JMenuItem startMenuItem;
    JMenuItem stopMenuItem;

    public void createControls() {
        // Create all Swing components here
    }

    public void enableStopMenuItem() {
        stopMenuItem.setEnabled(true);
    }

    public void disableStopMenuItem() {
        stopMenuItem.setEnabled(false);
    }

    public void enableStartMenuItem() {
        startMenuItem.setEnabled(true);
    }

    public void disableStartMenuItem() {
        startMenuItem.setEnabled(false);
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getSource() == setBPMButton) {
            int bpm = Integer.parseInt(bpmTextField.getText());
            controller.setBPM(bpm);
        } else if (event.getSource() == increaseBPMButton) {
            controller.increaseBPM();
        } else if (event.getSource() == decreaseBPMButton) {
            controller.decreaseBPM();
        }
    }
}
```

This method creates all the controls and places them in the interface. It also takes care of the menu. When the stop or start items are chosen, the corresponding methods are called on the controller.

All these methods allow the start and stop items in the menu to be enabled and disabled. We'll see that the controller uses these to change the interface.

This method is called when a button is clicked.

If the Set button is clicked then it is passed on to the controller along with the new bpm.

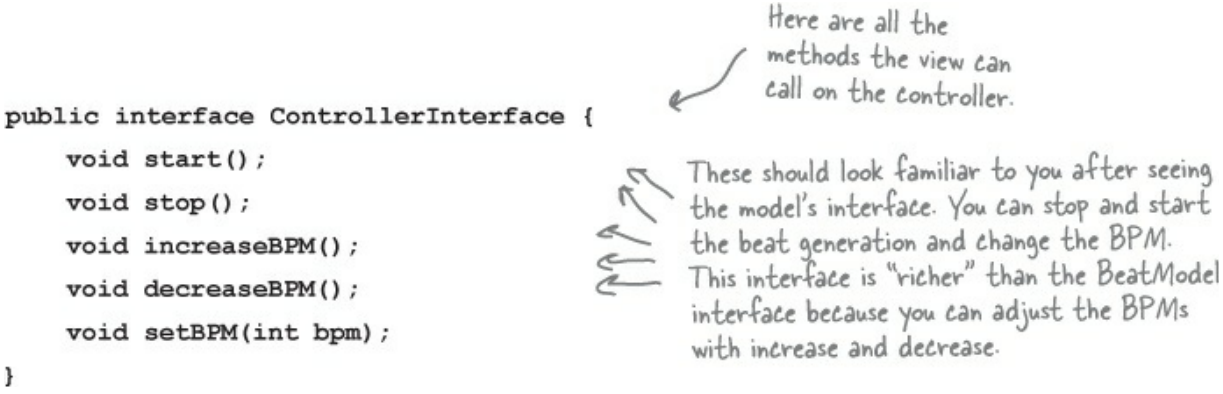
Likewise, if the increase or decrease buttons are clicked, this information is passed on to the controller.

Now for the Controller

It's time to write the missing piece: the controller. Remember the controller is the strategy that we plug into the view to give it some smarts.

Because we are implementing the Strategy Pattern, we need to start with an

interface for any Strategy that might be plugged into the DJ View. We're going to call it ControllerInterface.




```
public interface ControllerInterface {
    void start();
    void stop();
    void increaseBPM();
    void decreaseBPM();
    void setBPM(int bpm);
}
```

DESIGN PUZZLE

You've seen that the view and controller together make use of the Strategy Pattern. Can you draw a class diagram of the two that represents this pattern?

And here's the implementation of the controller

Putting it all together...

We've got everything we need: a model, a view, and a controller. Now it's time to put them all together into a MVC! We're going to see and hear how well they work together.

All we need is a little code to get things started; it won't take much:

```
public class DJTestDrive {

    public static void main (String[] args) {
        BeatModelInterface model = new BeatModel();
        ControllerInterface controller = new BeatController(model);
    }
}
```

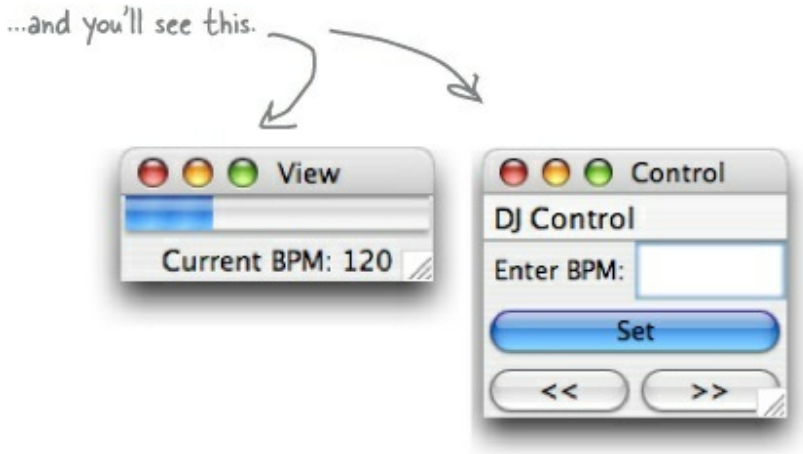
First create a model...

...then create a controller and pass it the model. Remember, the controller creates the view, so we don't have to do that.

And now for a test run...

```
File Edit Window Help LetTheBassKick
% java DJTestDrive
%
```

Run this...



Things to do

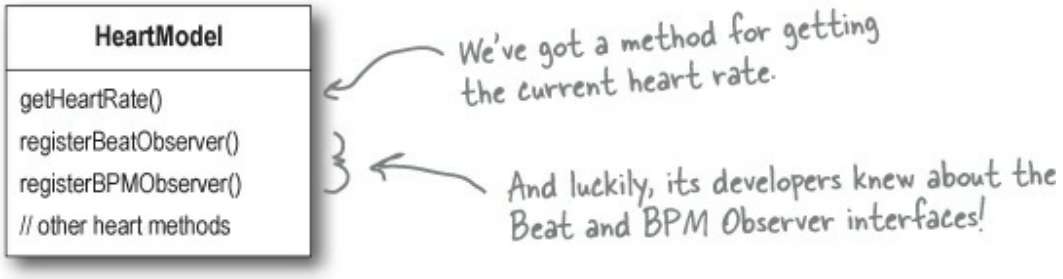
① Start the beat generation with the Start menu item; notice the controller disables the item afterwards.
② Use the text entry along with the increase and decrease buttons to change the BPM. Notice how the view display reflects the changes despite the fact that it has no logical link to the controls.
③ Notice how the beat bar always keeps up with the beat since it's an observer of the model.
④ Put on your favorite song and see if you can beat match the beat by using the increase and decrease controls.
⑤ Stop the generator. Notice how the controller disables the Stop menu item and enables the Start menu item.

Exploring Strategy

Let's take the Strategy Pattern just a little further to get a better feel for how it is used in MVC. We're going to see another friendly pattern pop up too — a pattern you'll often see hanging around the MVC trio: the Adapter Pattern.

Think for a second about what the DJ View does: it displays a beat rate and a

pulse. Does that sound like something else? How about a heartbeat? It just so happens that we have a heart monitor class; here's the class diagram:



BRAIN POWER

It certainly would be nice to reuse our current view with the HeartModel, but we need a controller that works with this model. Also, the interface of the HeartModel doesn't match what the view expects because it has a getHeartRate() method rather than a getBPM(). How would you design a set of classes to allow the view to be reused with the new model? Jot down your class design ideas below.

Adapting the Model

For starters, we're going to need to adapt the HeartModel to a BeatModel. If we don't, the view won't be able to work with the model, because the view only knows how to getBPM(), and the equivalent heart model method is getHeartRate(). How are we going to do this? We're going to use the Adapter Pattern, of course! It turns out that this is a common technique when working with the MVC: use an adapter to adapt a model to work with existing controllers and views.

Here's the code to adapt a HeartModel to a BeatModel:


```
public class HeartAdapter implements BeatModelInterface {
    HeartModelInterface heart;

    public HeartAdapter(HeartModelInterface heart) {
        this.heart = heart;
    }

    public void initialize() {}

    public void on() {}

    public void off() {}

    public int getBPM() {
        return heart.getHeartRate();
    }

    public void setBPM(int bpm) {}

    public void registerObserver(BeatObserver o) {
        heart.registerObserver(o);
    }

    public void removeObserver(BeatObserver o) {
        heart.removeObserver(o);
    }

    public void registerObserver(BPMObserver o) {
        heart.registerObserver(o);
    }

    public void removeObserver(BPMObserver o) {
        heart.removeObserver(o);
    }
}
```

Now we're ready for a HeartController

With our HeartAdapter in hand we should be ready to create a controller and get the view running with the HeartModel. Talk about reuse!

```
public class HeartController implements ControllerInterface {
    HeartModelInterface model;
    DJView view;

    public HeartController(HeartModelInterface model) {
        this.model = model;
        view = new DJView(this, new HeartAdapter(model));
        view.createView();
        view.createControls();
        view.disableStopMenuItem();
        view.disableStartMenuItem();
    }

    public void start() {}

    public void stop() {}

    public void increaseBPM() {}

    public void decreaseBPM() {}

    public void setBPM(int bpm) {}
}
```

The HeartController implements the ControllerInterface, just like the BeatController did.

Like before, the controller creates the view and gets everything glued together.

There is one change: we are passed a HeartModel, not a BeatModel...

...and we need to wrap that model with an adapter before we hand it to the view.

Finally, the HeartController disables the menu items because they aren't needed.

There's not a lot to do here; after all, we can't really control hearts like we can beat machines.

And that's it! Now it's time for some test code...

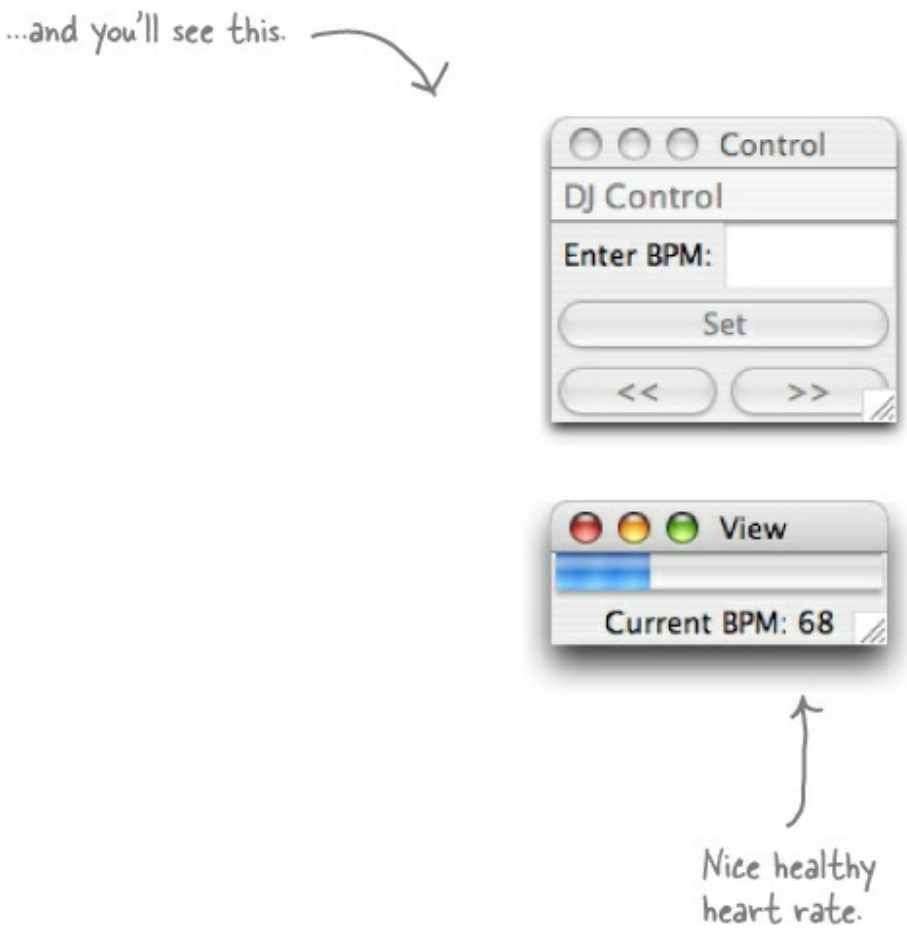
```
public class HeartTestDrive {

    public static void main (String[] args) {
        HeartModel heartModel = new HeartModel();
        ControllerInterface model = new HeartController(heartModel);
    }
}
```

All we need to do is create the controller and pass it a heart monitor.

And now for a test run...

```
File Edit Window Help CheckMyPulse
% java HeartTestDrive
%
```

Run this...



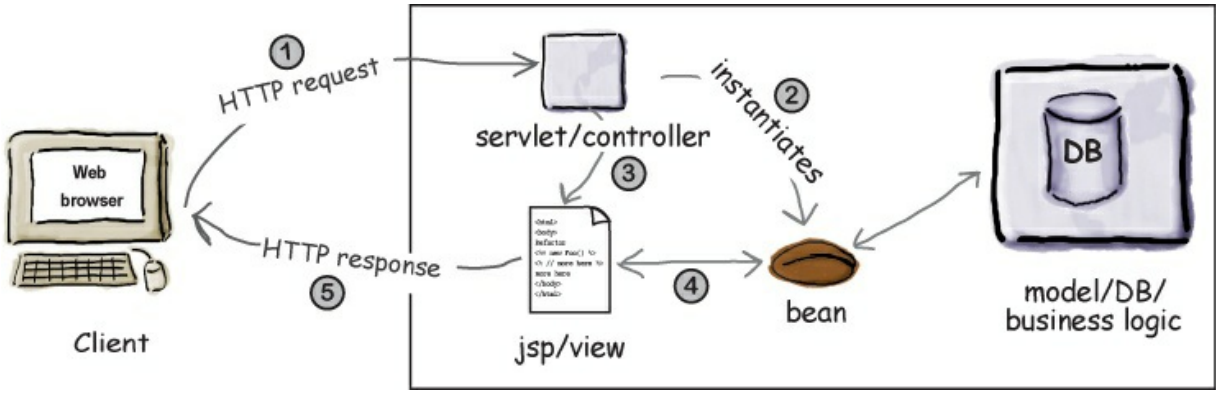
Things to do

① Notice that the display works great with a heart! The beat bar looks just like a pulse. Because the HeartModel also supports BPM and Beat Observers we can get beat updates just like with the DJ beats.
② As the heartbeat has natural variation, notice the display is updated with the new beats per minute.
③ Each time we get a BPM update the adapter is doing its job of translating getBPM() calls to getHeartRate() calls.
④ The Start and Stop menu items are not enabled because the controller disabled them.
⑤ The other buttons still work but have no effect because the controller implements no ops for them. The view could be changed to support the disabling of these items.

MVC and the Web

It wasn't long after the Web was spun that developers started adapting the MVC to fit the browser/server model. The prevailing adaptation is known simply as "Model 2" and uses a combination of servlet and JSP technology to achieve the same separation of model, view and controller that we see in conventional GUIs.

Let's check out how Model 2 works:



①	**You make an HTTP request, which is received by a servlet.** Using your web browser you make an HTTP request. This typically involves sending along some form data, like your username and password. A servlet receives this form data and parses it.
②	**The servlet acts as the controller.** The servlet plays the role of the controller and processes your request, most likely making requests on the model (usually a database). The result of processing the request is usually bundled up in the form of a JavaBean.
③	**The controller forwards control to the view.** The View is represented by a JSP. The JSP's only job is to generate the page representing the view of model (❹ which it obtains via the JavaBean) along with any controls needed for further actions.
④	**The view returns a page to the browser via HTTP.** A page is returned to the browser, where it is displayed as the view. The user submits further requests, which are processed in the same fashion.

Model 2 is more than just a clean design.

The benefits of the separation of the view, model and controller are pretty clear to you now. But you need to know the "rest of the story" with Model 2 — that it saved many web shops from sinking into chaos.

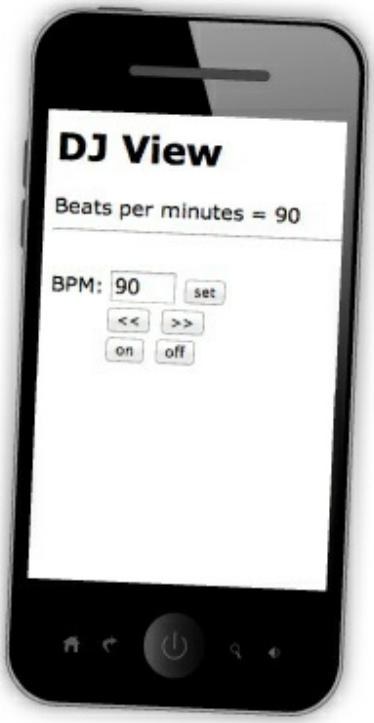
How? Well, Model 2 not only provides a separation of components in terms of design, it also provides a separation in *production responsibilities*. Let's face it, in the old days, anyone with access to your JSPs could get in and write any Java code they wanted, right? And that included a lot of people who didn't know a jar file from a jar of peanut butter. The reality is that most web producers *know about content and HTML, not software*.

Luckily Model 2 came to the rescue. With Model 2 we can leave the developer jobs to the men & women who know their servlets and let the web producers loose on simple Model 2-style JSPs where all the producers have access to is HTML and simple JavaBeans.

Model 2: DJ'ing from a cell phone

You didn't think we'd try to skip out without moving that great BeatModel over to the Web, did you? Just think, you can control your entire DJ session

through a web page on your cellular phone. So now you can get out of that DJ booth and get down in the crowd. What are you waiting for? Let's write that code!



The plan

① **Fix up the model.**
Well, actually, we don't have to fix the model; it's fine just like it is!
② **Create a servlet controller**
We need a simple servlet that can receive our HTTP requests and perform a few operations on the model. All it needs to do is stop, start and change the beats per minute.
③ **Create a HTML view.**
We'll create a simple view with a JSP. It's going to receive a JavaBean from the controller that will tell it everything it needs to display. The JSP will then generate an HTML interface.

GEEK BITS

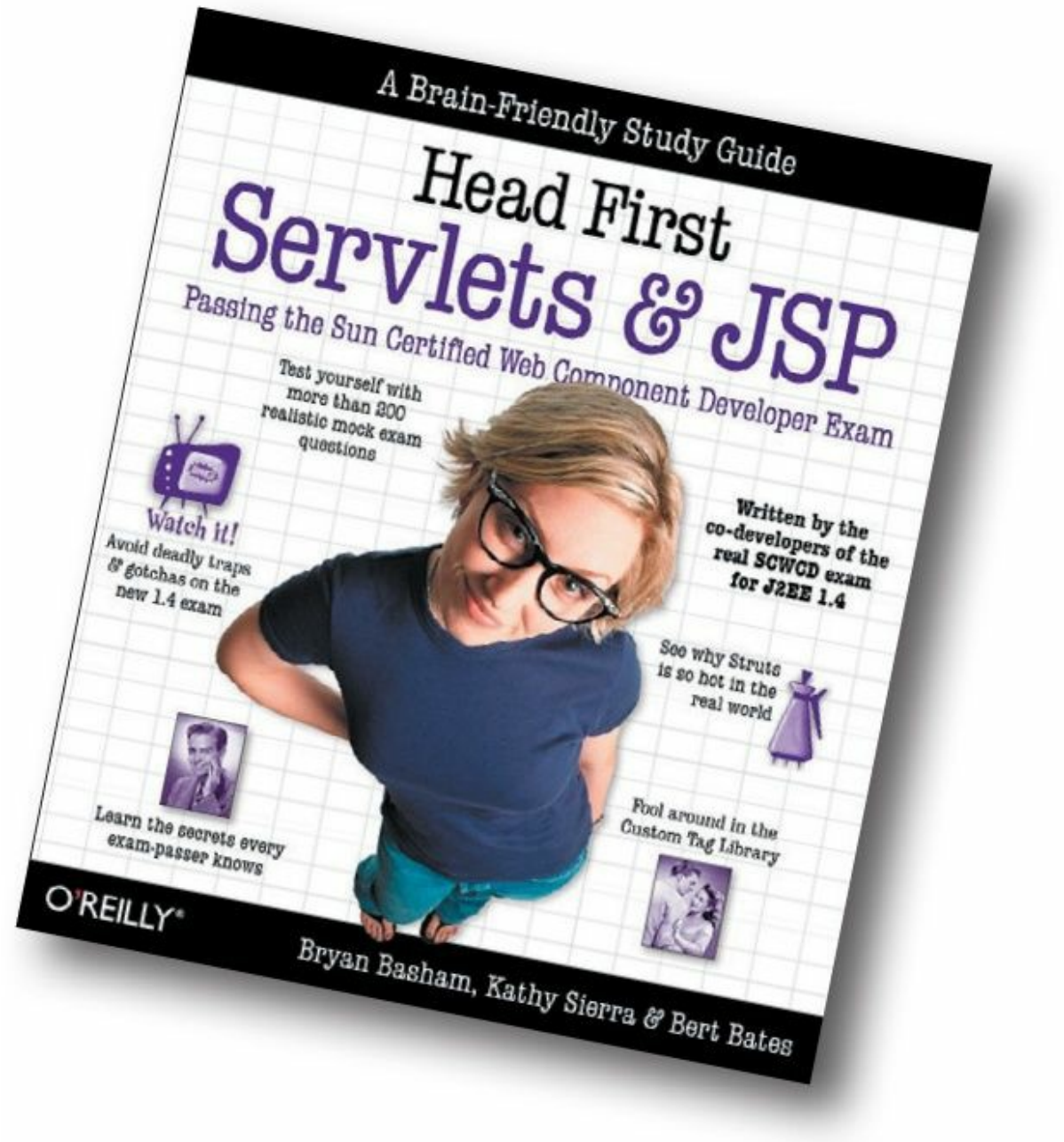
Setting up your servlet environment

Showing you how to set up your servlet environment is a little bit off topic for a book on

Design Patterns, at least if you don't want the book to weigh more than you do!

Fire up your web browser and head straight to http://jakarta.apache.org/tomcat/ for the Apache Jakarta Project's Tomcat Servlet Container. You'll find everything you need there to get you up and running.

You'll also want to check out *Head First Servlets & JSP* by Bryan Basham, Kathy Sierra and Bert Bates.



Step one: the model

Remember that in MVC, the model doesn't know anything about the views or controllers. In other words, it is totally decoupled. All it knows is that it may have observers it needs to notify. That's the beauty of the Observer Pattern. It also provides an interface the views and controllers can use to get and set its state.

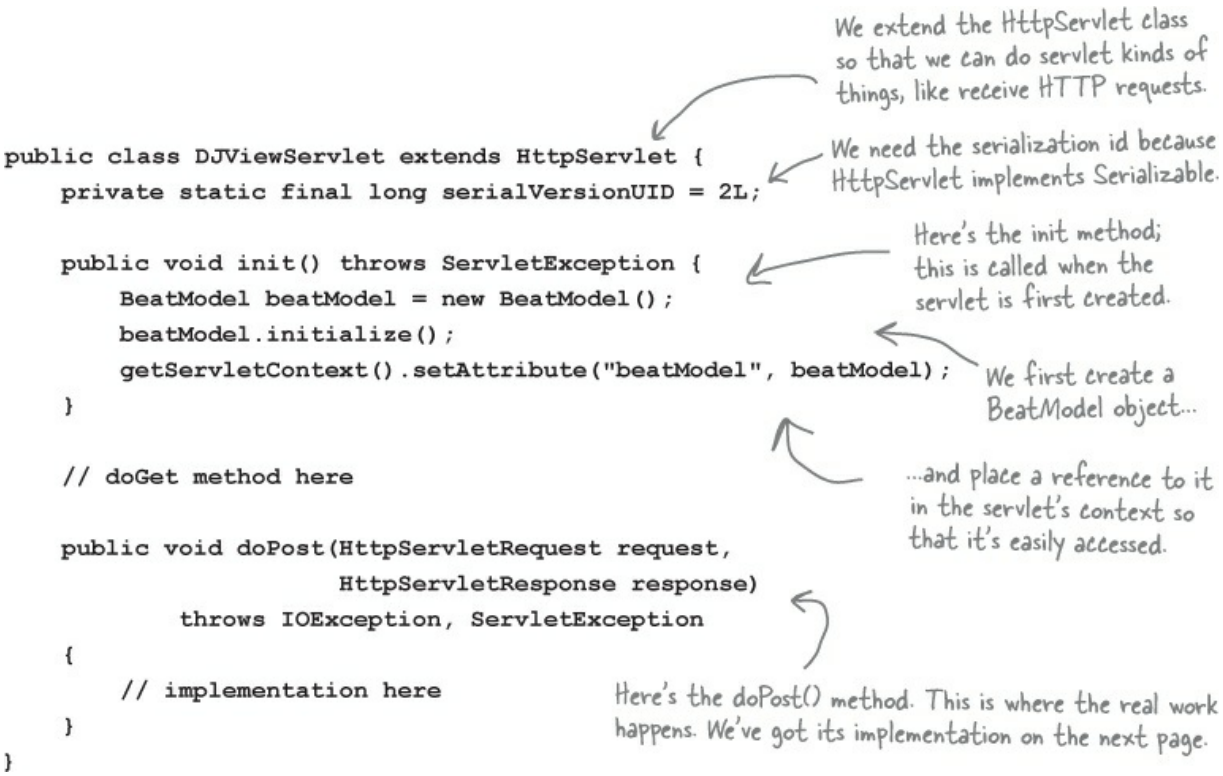
Now all we need to do is adapt it to work in the web environment, but, given that it doesn't depend on any outside classes, there is really no work to be done. We can use our BeatModel off the shelf without changes. So, let's be productive and move on to step two!

Step two: the controller servlet

Remember, the servlet is going to act as our controller; it will receive web browser input in a HTTP request and translate it into actions that can be applied to the model.

Then, given the way the Web works, we need to return a view to the browser. To do this we'll pass control to the view, which takes the form of a JSP. We'll get to that in step three.

Here's the outline of the servlet; on the next page, we'll look at the full implementation.




```
public class DJViewServlet extends HttpServlet {
    private static final long serialVersionUID = 2L;

    public void init() throws ServletException {
        BeatModel beatModel = new BeatModel();
        beatModel.initialize();
        getServletContext().setAttribute("beatModel", beatModel);
    }

    // doGet method here

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
            throws IOException, ServletException
    {
        // implementation here
    }
}
```

Here's the implementation of the doGet() method from the page before:

```
public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
        throws IOException, ServletException
{
    BeatModel beatModel =
        (BeatModel)getServletContext().getAttribute("beatModel");

    String bpm = request.getParameter("bpm");
    if (bpm == null) {
        bpm = beatModel.getBPM() + "";
    }

    String set = request.getParameter("set");
    if (set != null) {
        int bpmNumber = 90;
        bpmNumber = Integer.parseInt(bpm);
        beatModel.setBPM(bpmNumber);
    }

    String decrease = request.getParameter("decrease");
    if (decrease != null) {
        beatModel.setBPM(beatModel.getBPM() - 1);
    }
    String increase = request.getParameter("increase");
    if (increase != null) {
        beatModel.setBPM(beatModel.getBPM() + 1);
    }
    String on = request.getParameter("on");
    if (on != null) {
        beatModel.on();
    }
    String off = request.getParameter("off");
    if (off != null) {
        beatModel.off();
    }

    request.setAttribute("beatModel", beatModel);

    RequestDispatcher dispatcher =
        request.getRequestDispatcher("/djview.jsp");
    dispatcher.forward(request, response);
}
```

First we grab the model from the servlet context. We can't manipulate the model without a reference to it.

Next we grab all the HTTP commands/parameters...

If we get a set command, then we get the value of the set, and tell the model.

To increase or decrease, we get the current BPMs from the model, and adjust up or down by one.

If we get an on or off command, we tell the model to turn off or on.

Following the Model 2 definition, we pass the JSP a bean with the model state in it. In this case, we pass it the actual model, since it happens to be a bean.

Finally, our job as a controller is done. All we need to do is ask the view to take over and create an HTML view.

Now we need a view...

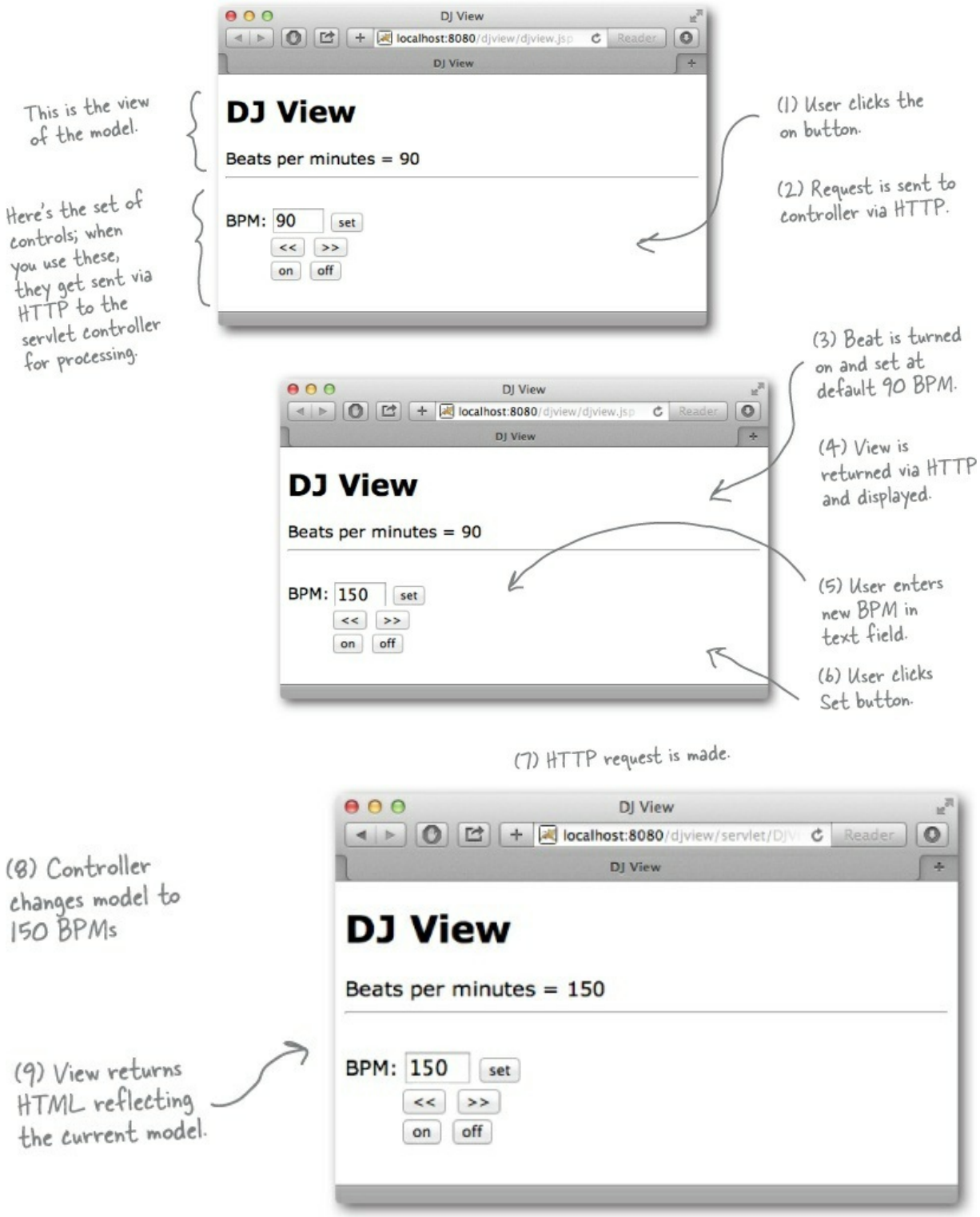
All we need is a view and we've got our browser-based beat generator ready to go! In Model 2, the view is just a JSP. All the JSP knows about is the bean it receives from the controller. In our case, that bean is just the model and the JSP is only going to use its BPM property to extract the current beats per minute. With that data in hand, it creates the view and also the user interface controls.

NOTE

NOTICE that just like MVC, in Model 2 the view doesn't alter the model (that's the controller's job); all it does is use its state!

Putting Model 2 to the test...

It's time to start your web browser, hit the DJView Servlet and give the system a spin...


This is the view of the model.
Here's the set of controls; when you use these, they get sent via HTTP to the servlet controller for processing.
DJ View
localhost:8080/djview/djview.jsp
Reader
DJ View
DJ View
Beats per minutes = 90
BPM: 90 set
<< >>
on off
(1) User clicks the on button.
(2) Request is sent to controller via HTTP.
(3) Beat is turned on and set at default 90 BPM.
(4) View is returned via HTTP and displayed.
DJ View
localhost:8080/djview/djview.jsp
Reader
DJ View
DJ View
Beats per minutes = 90
BPM: 150 set
<< >>
on off
(5) User enters new BPM in text field.
(6) User clicks Set button.
(7) HTTP request is made.
(8) Controller changes model to 150 BPMs
(9) View returns HTML reflecting the current model.
DJ View
localhost:8080/djview/servlet/DJV
Reader
DJ View
DJ View
Beats per minutes = 150
BPM: 150 set
<< >>
on off

Things to do

① First, hit the web page; you'll see the beats per minute at 0. Go ahead and click the "on" button.
② Now you should see the beats per minute at the default setting: 90 BPM. You should also hear a beat on the machine the server is running on.
③ Enter a specific beat, say, 120, and click the "set" button. The page should refresh with a beats per minute of 120 (and you should hear the beat increase).
④ Now play with the increase/decrease buttons to adjust the beat up and down.
⑤ Think about how each step of the system works. The HTML interface makes a request to the servlet (the controller); the servlet parses the user input and then makes requests to the model. The servlet then passes control to the JSP (the view), which creates the HTML view that is returned and displayed.

Design Patterns and Model 2

After implementing the DJ control for the Web using Model 2, you might be wondering where the patterns went. We have a view created in HTML from a JSP, but the view is no longer a listener of the model. We have a controller that's a servlet that receives HTTP requests, but are we still using the Strategy Pattern? And what about Composite? We have a view that is made from HTML and displayed in a web browser. Is that still the Composite Pattern?

Model 2 is an adaptation of MVC to the Web

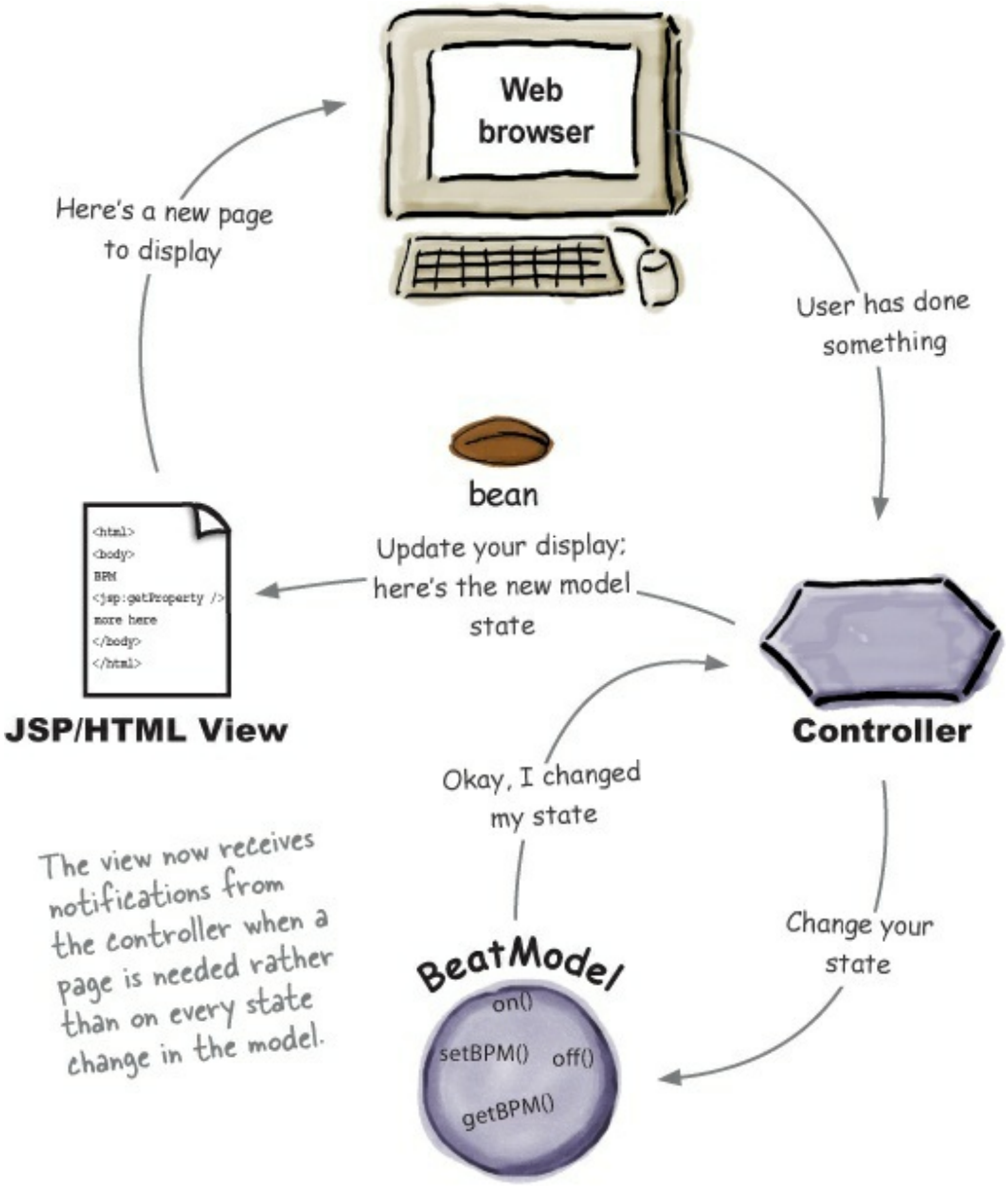
Even though Model 2 doesn't look exactly like "textbook" MVC, all the parts are still there; they've just been adapted to reflect the idiosyncrasies of the web browser model. Let's take another look...

Observer

The view is no longer an observer of the model in the classic sense; that is, it doesn't register with the model to receive state change notifications.

However, the view does receive the equivalent of notifications indirectly from the controller when the model has been changed. The controller even passes the view a bean that allows the view to retrieve the model's state.

If you think about the browser model, the view only needs an update of state information when an HTTP response is returned to the browser; notifications at any other time would be pointless. Only when a page is being created and returned does it make sense to create the view and incorporate the model's state.



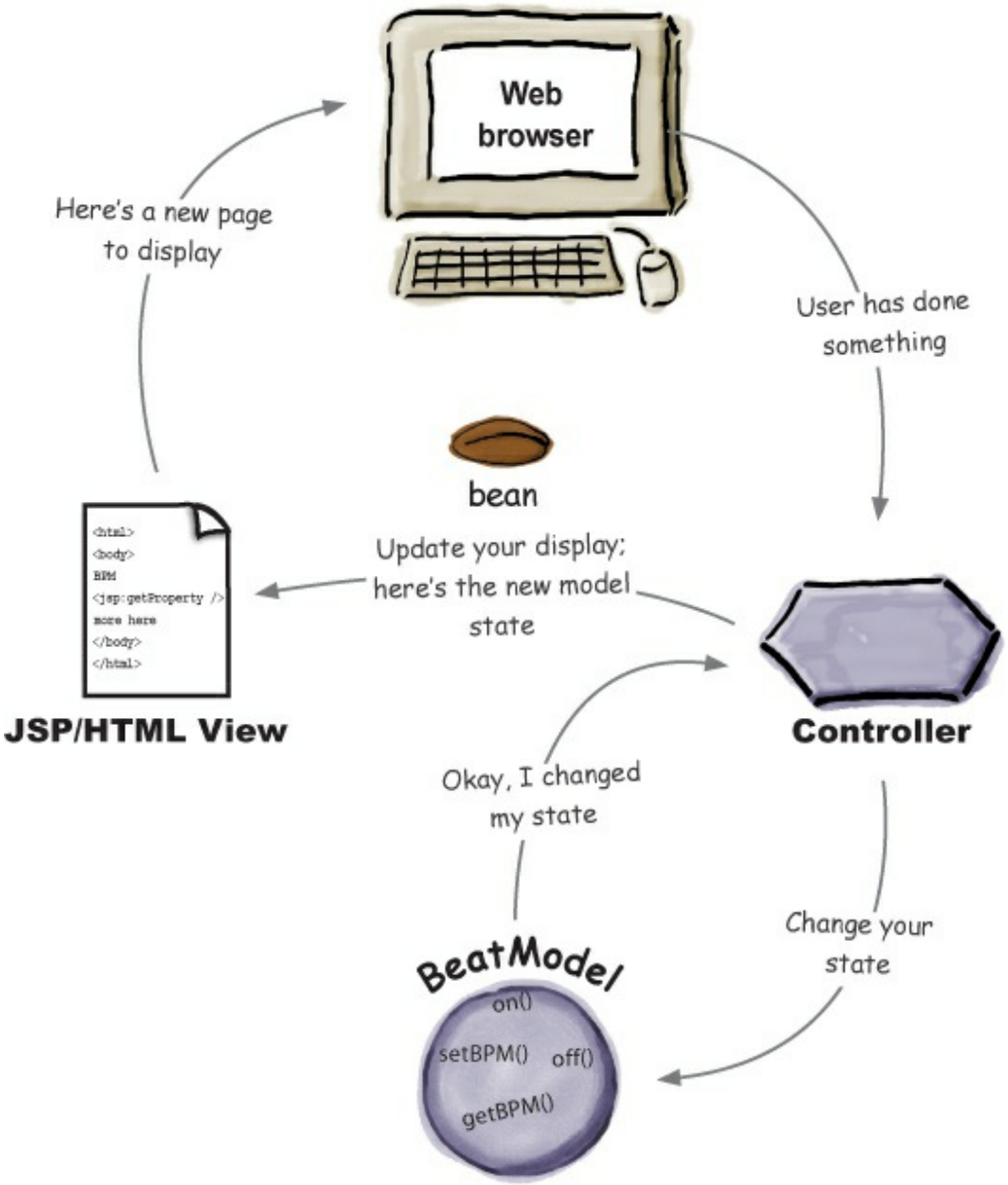
Strategy

In Model 2, the Strategy object is still the controller servlet; however, it's not directly composed with the view in the classic manner. That said, it is an

object that implements behavior for the view, and we can swap it out for another controller if we want different behavior.

Composite

Like our Swing GUI, the view is ultimately made up of a nested set of graphical components. In this case, they are rendered by a web browser from an HTML description; however, underneath there is an object system that most likely forms a composite.



NOTE

The controller still provides the view behavior, even if it isn't composed with the view using object composition.

THERE ARE NO DUMB QUESTIONS

Q: **Q: It seems like you are really hand-waving the fact that the Composite Pattern is really in MVC. Is it really there?**

A: **A:** Yes, Virginia, there really is a Composite Pattern in MVC. But, actually, this is a very good question. Today GUI packages, like Swing, have become so sophisticated that we hardly notice the internal structure and the use of Composite in the building and update of the display. It's even harder to see when we have web browsers that can take markup language and convert it into a user interface.
Back when MVC was first discovered, creating GUIs required a lot more manual intervention and the pattern was more obviously part of the MVC.

Q: **Q: Does the controller ever implement any application logic?**

A: **A:** No, the controller implements behavior for the view. It is the smarts that translates the actions from the view to actions on the model. The model takes those actions and implements the application logic to decide what to do in response to those actions. The controller might have to do a little work to determine what method calls to make on the model, but that's not considered the "application logic." The application logic is the code that manages and manipulates your data and it lives in your model.

Q: **Q: I've always found the word "model" hard to wrap my head around. I now get that it's the guts of the application, but why was such a vague, hard-to-understand word used to describe this aspect of the MVC?**

A: **A:** When MVC was named they needed a word that began with a "M" or otherwise they couldn't have called it MVC.
But seriously, we agree with you. Everyone scratches their head and wonders what a model is. But then everyone comes to the realization that they can't think of a better word either.

Q: **Q: You've talked a lot about the state of the model. Does this mean it has the State Pattern in it?**

A: **A:** No, we mean the general idea of state. But certainly some models do use the State Pattern to manage their internal states.

Q: **Q: I've seen descriptions of the MVC where the controller is described as a "mediator" between the view and the model. Is the controller implementing the Mediator Pattern?**

A: **A:** We haven't covered the Mediator Pattern (although you'll find a summary of the pattern in the appendix), so we won't go into too much detail here, but the intent of the mediator is to encapsulate how objects interact and promote loose coupling by keeping two objects from referring to each other explicitly. So, to some degree, the controller can be seen as a mediator, since the view never sets state directly on the model, but rather always goes through the controller. Remember, however, that the view does have a reference to the model to access its state. If the controller were truly a mediator, the view would have to go through the controller to get the state of the model as well.

Q: **Q: Does the view always have to ask the model for its state? Couldn't we use the push model and send the model's state with the update notification?**

A: **A:** Yes, the model could certainly send its state with the notification, and in fact, if you look again at the JSP/HTML view, that's exactly what we're doing. We're sending the entire model in a bean, which the view uses to access the state it needs using the bean properties. We could do something similar with the BeatModel by sending just the state that the view is interested in. If you remember the Observer Pattern chapter, however, you'll also remember that there's a couple of disadvantages to this. If you don't, go back and have a second look.

Q: **Q: If I have more than one view, do I always need more than one controller?**

A: **A:** Typically, you need one controller per view at runtime; however, the same controller class can easily manage many views.

Q: **Q: The view is not supposed to manipulate the model; however, I noticed in your implementation that the**

	view has full access to the methods that change the model's state. Is this dangerous?
A:	**A:** You are correct; we gave the view full access to the model's set of methods. We did this to keep things simple, but there may be circumstances where you want to give the view access to only part of your model's API. There's a great design pattern that allows you to adapt an interface to only provide a subset. Can you think of it?

Tools for your Design Toolbox

You could impress anyone with your design toolbox. Wow, look at all those principles, patterns and now, compound patterns!


OO Principles
Encapsulate what varies.
Favor composition over inheritance.
Program to interfaces, not implementations.
Strive for loosely coupled designs between objects that interact.
Classes should be open for extension but closed for modification.
Depend on abstractions. Do not depend on concrete classes.
Only talk to your friends.
Don't call us, we'll call you.
A class should have only one reason to change.
OO Basics
Abstraction
Encapsulation
Polymorphism
Inheritance
OO Patterns
Proxy – Provide a surrogate or placeholder for another object to control access to it.
Compound Patterns
A Compound Pattern combines two or more patterns into a solution that solves a recurring or general problem.
We have a new category! MVC and Model 2 are compound patterns.

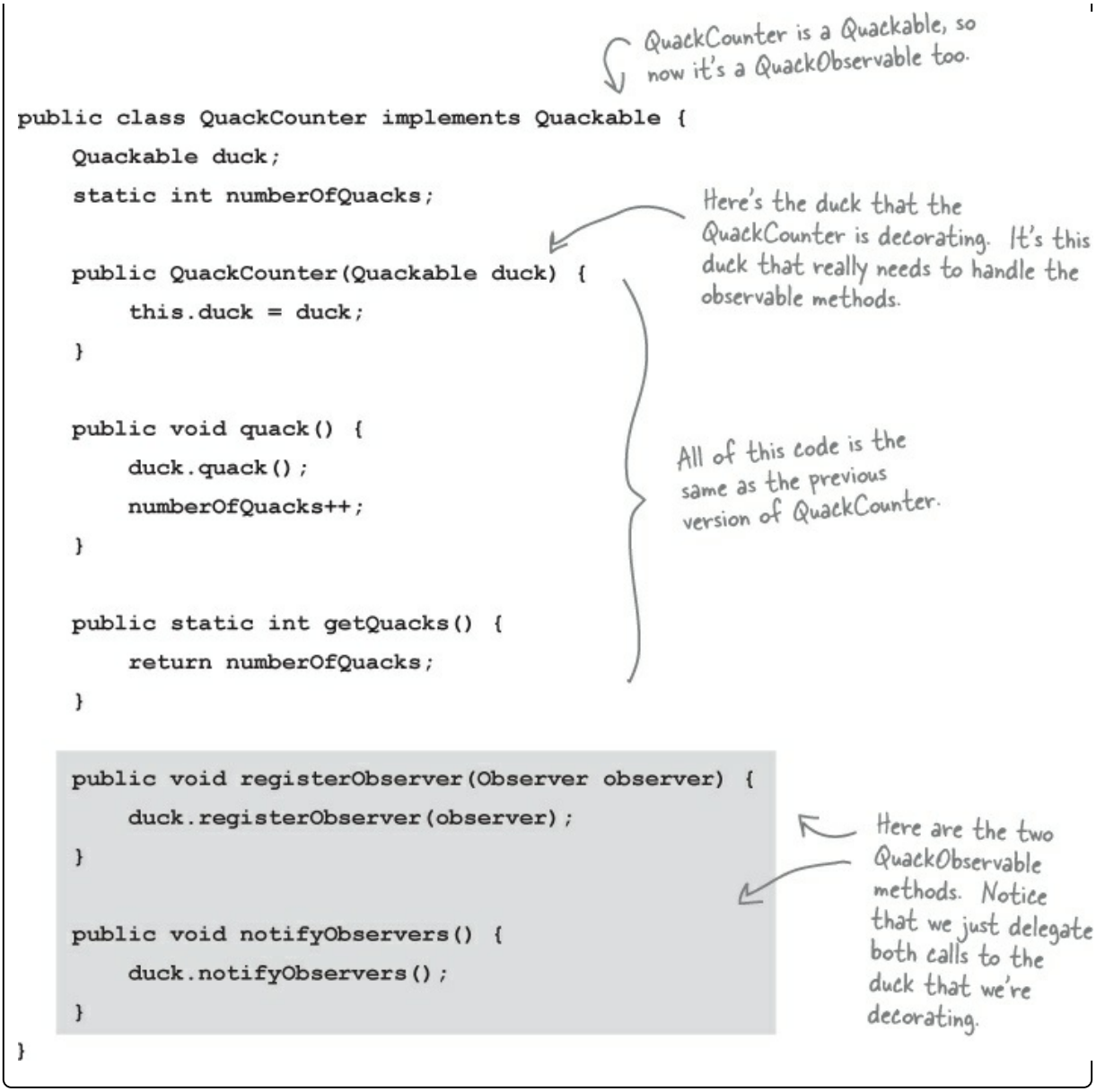
BULLET POINTS

- The Model View Controller Pattern (MVC) is a compound pattern consisting of the Observer, Strategy and Composite patterns.
- The model makes use of the Observer Pattern so that it can keep observers updated yet stay decoupled from them.
- The controller is the strategy for the view. The view can use different implementations of the controller to get different behavior.
- The view uses the Composite Pattern to implement the user interface, which usually consists of nested components like panels, frames and buttons.
- These patterns work together to decouple the three players in the MVC model, which keeps designs clear and flexible.
- The Adapter Pattern can be used to adapt a new model to an existing view and controller.
- Model 2 is an adaptation of MVC for web applications.
- In Model 2, the controller is implemented as a servlet and JSP & HTML implement the view.

Exercise Solutions

SHARPEN YOUR PENCIL SOLUTION

The QuackCounter is a Quackable too. When we change Quackable to extend QuackObservable, we have to change every class that implements Quackable, including QuackCounter:



SHARPEN YOUR PENCIL SOLUTION

What if our Quackologist wants to observe an entire flock? What does that mean anyway? Think about it like this: if we observe a composite, then we're observing everything in the composite. So, when you register with a flock, the flock composite makes sure you get registered with all its children, which may include other flocks.


```
public class Flock implements Quackable {
    ArrayList<Quackable> quackers = new ArrayList<Quackable>();

    public void add(Quackable duck) {
        ducks.add(duck);
    }

    public void quack() {
        Iterator<Quackable> iterator = quackers.iterator();
        while (iterator.hasNext()) {
            Quackable duck = iterator.next();
            duck.quack();
        }
    }

    public void registerObserver(Observer observer) {
        Iterator<Quackable> iterator = ducks.iterator();
        while (iterator.hasNext()) {
            Quackable duck = iterator.next();
            duck.registerObserver(observer);
        }
    }

    public void notifyObservers() { }
}
```

SHARPEN YOUR PENCIL SOLUTION

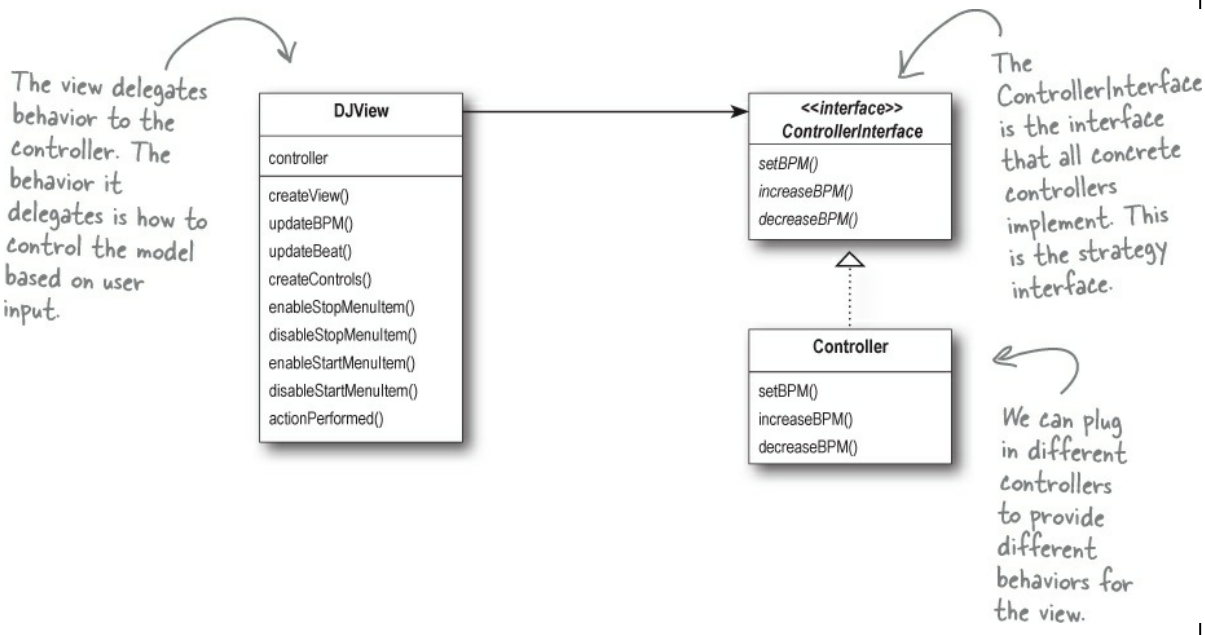
We're still directly instantiating Geese by relying on concrete classes. Can you write an Abstract Factory for Geese? How should it handle creating "goose ducks"?

You could add a createGooseDuck() method to the existing Duck Factories. Or, you could create a completely separate Factory for creating families of Geese.

DESIGN PUZZLE SOLUTION

You've seen that the view and controller together make use of the Strategy Pattern. Can

you draw a class diagram of the two that represents this pattern?



READY BAKE CODE

Here's the complete implementation of the DJView. It shows all the MIDI code to generate the sound, and all the Swing components to create the view. You can also download this code at http://www.wickedlysmart.com. Have fun!

```
package headfirst.designpatterns.combined.djview;

public class DJTestDrive {

    public static void main (String[] args) {
        BeatModelInterface model = new BeatModel();
        ControllerInterface controller = new BeatController(model);
    }
}
```

The Beat Model

```
package headfirst.designpatterns.combined.djview;

public interface BeatModelInterface {
    void initialize();

    void on();

    void off();

    void setBPM(int bpm);

    int getBPM();

    void registerObserver(BeatObserver o);
```

```
    void removeObserver(BeatObserver o);

    void registerObserver(BPMObserver o);

    void removeObserver(BPMObserver o);
}
package headfirst.designpatterns.combined.djview;

import javax.sound.midi.*;

import java.util.*;

public class BeatModel implements BeatModelInterface, MetaEventListener {
       Sequencer sequencer;
       ArrayList<BeatObserver> beatObservers = new ArrayList<BeatObserver>();
       ArrayList<BPMObserver> bpmObservers = new ArrayList<BPMObserver>();
       int bpm = 90;
       Sequence sequence;
       Track track;

       public void initialize() {
              setUpMidi();
              buildTrackAndStart();
       }

       public void on() {
              System.out.println("Starting the sequencer");
              sequencer.start();
              setBPM(90);
       }

       public void off() {
              setBPM(0);
              sequencer.stop();
       }

       public void setBPM(int bpm) {
              this.bpm = bpm;
              sequencer.setTempoInBPM(getBPM());
              notifyBPMObservers();
       }

       public int getBPM() {
              return bpm;
       }

       void beatEvent() {
              notifyBeatObservers();
       }

       public void registerObserver(BeatObserver o) {
              beatObservers.add(o);
       }

      public void notifyBeatObservers() {
             for(int i = 0; i < beatObservers.size(); i++) {
                    BeatObserver observer = (BeatObserver)beatObservers.get(i);
                    observer.updateBeat();
             }
      }

      public void registerObserver(BPMObserver o) {
```

```
        bpmObservers.add(o);
    }

    public void notifyBPMObservers() {
        for(int i = 0; i < bpmObservers.size(); i++) {
            BPMObserver observer = (BPMObserver)bpmObservers.get(i);
            observer.updateBPM();
        }
    }

    public void removeObserver(BeatObserver o) {
        int i = beatObservers.indexOf(o);
        if (i >= 0) {
            beatObservers.remove(i);
        }
    }

    public void removeObserver(BPMObserver o) {
        int i = bpmObservers.indexOf(o);
        if (i >= 0) {
            bpmObservers.remove(i);
        }
    }

    public void meta(MetaMessage message) {
        if (message.getType() == 47) {
            beatEvent();
            sequencer.start();
            setBPM(getBPM());
        }
    }

    public void setUpMidi() {
        try {
              sequencer = MidiSystem.getSequencer();
              sequencer.open();
              sequencer.addMetaEventListener(this);
              sequence = new Sequence(Sequence.PPQ,4);
              track = sequence.createTrack();
              sequencer.setTempoInBPM(getBPM());
              sequencer.setLoopCount(Sequencer.LOOP_CONTINUOUSLY);
        } catch(Exception e) {
              e.printStackTrace();
        }
    }

    public void buildTrackAndStart() {
        int[] trackList = {35, 0, 46, 0};

        sequence.deleteTrack(null);
        track = sequence.createTrack();

        makeTracks(trackList);
        track.add(makeEvent(192,9,1,0,4));
        try {
              sequencer.setSequence(sequence);
        } catch(Exception e) {
              e.printStackTrace();
        }
    }
```

```
        public void makeTracks(int[] list) {

               for (int i = 0; i < list.length; i++) {
                      int key = list[i];

                      if (key != 0) {
                             track.add(makeEvent(144,9,key, 100, i));
                             track.add(makeEvent(128,9,key, 100, i+1));
                      }
               }
        }

        public MidiEvent makeEvent(int comd, int chan, int one, int two, int tick)
  {
              MidiEvent event = null;
              try {
                     ShortMessage a = new ShortMessage();
                     a.setMessage(comd, chan, one, two);
                     event = new MidiEvent(a, tick);

              } catch(Exception e) {
                     e.printStackTrace();
              }
              return event;
        }
  }
```

The View

```
  package headfirst.designpatterns.combined.djview;

  public interface BeatObserver {
      void updateBeat();
  }

  package headfirst.designpatterns.combined.djview;

  public interface BPMObserver {
      void updateBPM();
  }

  package headfirst.designpatterns.combined.djview;

  import java.awt.*;
  import java.awt.event.*;
  import javax.swing.*;

  public class DJView implements ActionListener, BeatObserver, BPMObserver {
      BeatModelInterface model;
      ControllerInterface controller;
      JFrame viewFrame;
      JPanel viewPanel;
      BeatBar beatBar;
      JLabel bpmOutputLabel;
      JFrame controlFrame;
      JPanel controlPanel;
      JLabel bpmLabel;
      JTextField bpmTextField;
      JButton setBPMButton;
      JButton increaseBPMButton;
      JButton decreaseBPMButton;
```

```
    JMenuBar menuBar;
    JMenu menu;
    JMenuItem startMenuItem;
    JMenuItem stopMenuItem;

    public DJView(ControllerInterface controller, BeatModelInterface model) {
        this.controller = controller;
        this.model = model;
        model.registerObserver((BeatObserver)this);
        model.registerObserver((BPMObserver)this);
    }

public void createView() {
    // Create all Swing components here
    viewPanel = new JPanel(new GridLayout(1, 2));
    viewFrame = new JFrame("View");
    viewFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    viewFrame.setSize(new Dimension(100, 80));
    bpmOutputLabel = new JLabel("offline", SwingConstants.CENTER);
    beatBar = new BeatBar();
    beatBar.setValue(0);
    JPanel bpmPanel = new JPanel(new GridLayout(2, 1));
    bpmPanel.add(beatBar);
    bpmPanel.add(bpmOutputLabel);
    viewPanel.add(bpmPanel);
    viewFrame.getContentPane().add(viewPanel, BorderLayout.CENTER);
    viewFrame.pack();
    viewFrame.setVisible(true);
}

public void createControls() {
    // Create all Swing components here
    JFrame.setDefaultLookAndFeelDecorated(true);
    controlFrame = new JFrame("Control");
    controlFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    controlFrame.setSize(new Dimension(100, 80));

    controlPanel = new JPanel(new GridLayout(1, 2));

    menuBar = new JMenuBar();
    menu = new JMenu("DJ Control");
    startMenuItem = new JMenuItem("Start");
    menu.add(startMenuItem);
    startMenuItem.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            controller.start();
        }
    });
    stopMenuItem = new JMenuItem("Stop");
    menu.add(stopMenuItem);
    stopMenuItem.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            controller.stop();
        }
    });
    JMenuItem exit = new JMenuItem("Quit");
    exit.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            System.exit(0);
        }
    });
```

```
        menu.add(exit);
        menuBar.add(menu);
        controlFrame.setJMenuBar(menuBar);

        bpmTextField = new JTextField(2);
        bpmLabel = new JLabel("Enter BPM:", SwingConstants.RIGHT);
        setBPMButton = new JButton("Set");
        setBPMButton.setSize(new Dimension(10,40));
        increaseBPMButton = new JButton(">>");
        decreaseBPMButton = new JButton("<<");
        setBPMButton.addActionListener(this);
        increaseBPMButton.addActionListener(this);
        decreaseBPMButton.addActionListener(this);

        JPanel buttonPanel = new JPanel(new GridLayout(1, 2));

        buttonPanel.add(decreaseBPMButton);
        buttonPanel.add(increaseBPMButton);

        JPanel enterPanel = new JPanel(new GridLayout(1, 2));
        enterPanel.add(bpmLabel);
        enterPanel.add(bpmTextField);
        JPanel insideControlPanel = new JPanel(new GridLayout(3, 1));
        insideControlPanel.add(enterPanel);
        insideControlPanel.add(setBPMButton);
        insideControlPanel.add(buttonPanel);
        controlPanel.add(insideControlPanel);

        bpmLabel.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));
        bpmOutputLabel.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));

        controlFrame.getRootPane().setDefaultButton(setBPMButton);
        controlFrame.getContentPane().add(controlPanel, BorderLayout.CENTER);

        controlFrame.pack();
        controlFrame.setVisible(true);
    }

    public void enableStopMenuItem() {
        stopMenuItem.setEnabled(true);
    }

    public void disableStopMenuItem() {
        stopMenuItem.setEnabled(false);
    }

    public void enableStartMenuItem() {
        startMenuItem.setEnabled(true);
    }

    public void disableStartMenuItem() {
        startMenuItem.setEnabled(false);
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getSource() == setBPMButton) {
            int bpm = Integer.parseInt(bpmTextField.getText());
            controller.setBPM(bpm);
        } else if (event.getSource() == increaseBPMButton) {
            controller.increaseBPM();
        } else if (event.getSource() == decreaseBPMButton) {
```

```
            controller.decreaseBPM();
        }
    }

    public void updateBPM() {
        int bpm = model.getBPM();
        if (bpm == 0) {
            bpmOutputLabel.setText("offline");
        } else {
            bpmOutputLabel.setText("Current BPM: " + model.getBPM());
        }
    }

    public void updateBeat() {
        beatBar.setValue(100);
    }
}
```

The Controller

```
package headfirst.designpatterns.combined.djview;

public interface ControllerInterface {
    void start();
    void stop();
    void increaseBPM();
    void decreaseBPM();
    void setBPM(int bpm);
}
```

```
package headfirst.designpatterns.combined.djview;

public class BeatController implements ControllerInterface {
    BeatModelInterface model;
    DJView view;

    public BeatController(BeatModelInterface model) {
        this.model = model;
        view = new DJView(this, model);
        view.createView();
        view.createControls();
        view.disableStopMenuItem();
        view.enableStartMenuItem();
        model.initialize();
    }

    public void start() {
        model.on();
        view.disableStartMenuItem();
        view.enableStopMenuItem();
    }

    public void stop() {
        model.off();
        view.disableStopMenuItem();
        view.enableStartMenuItem();
    }

    public void increaseBPM() {
        int bpm = model.getBPM();
        model.setBPM(bpm + 1);
    }
```

```
    public void decreaseBPM() {
        int bpm = model.getBPM();
        model.setBPM(bpm - 1);
    }

    public void setBPM(int bpm) {
        model.setBPM(bpm);
    }
}
```

The Heart Model

```
package headfirst.designpatterns.combined.djview;

public class HeartTestDrive {

    public static void main (String[] args) {
        HeartModel heartModel = new HeartModel();
        ControllerInterface model = new HeartController(heartModel);
    }
}

package headfirst.designpatterns.combined.djview;

public interface HeartModelInterface {
    int getHeartRate();
    void registerObserver(BeatObserver o);
    void removeObserver(BeatObserver o);
    void registerObserver(BPMObserver o);
    void removeObserver(BPMObserver o);
}

package headfirst.designpatterns.combined.djview;

import java.util.*;

public class HeartModel implements HeartModelInterface, Runnable {
    ArrayList<BeatObserver> beatObservers = new ArrayList<BeatObserver>();
    ArrayList<BPMObserver> bpmObservers = new ArrayList<BPMObserver>();
    int time = 1000;
    int bpm = 90;
    Random random = new Random(System.currentTimeMillis());
    Thread thread;

    public HeartModel() {
        thread = new Thread(this);
        thread.start();
    }

    public void run() {
        int lastrate = -1;

        for(;;) {
            int change = random.nextInt(10);
            if (random.nextInt(2) == 0) {
                change = 0 - change;
            }
            int rate = 60000/(time + change);
            if (rate < 120 && rate > 50) {
                time += change;
```

```
                notifyBeatObservers();
                if (rate != lastrate) {
                    lastrate = rate;
                    notifyBPMObservers();
                }
            }
            try {
                Thread.sleep(time);
            } catch (Exception e) {}
        }
    }

    public int getHeartRate() {
        return 60000/time;
    }

    public void registerObserver(BeatObserver o) {
        beatObservers.add(o);
    }

    public void removeObserver(BeatObserver o) {
        int i = beatObservers.indexOf(o);
            if (i >= 0) {
            beatObservers.remove(i);
        }
    }

    public void notifyBeatObservers() {
        for(int i = 0; i < beatObservers.size(); i++) {
            BeatObserver observer = (BeatObserver)beatObservers.get(i);
            observer.updateBeat();
        }
    }

    public void registerObserver(BPMObserver o) {
        bpmObservers.add(o);
    }

    public void removeObserver(BPMObserver o) {
        int i = bpmObservers.indexOf(o);
        if (i >= 0) {
            bpmObservers.remove(i);
        }
    }

    public void notifyBPMObservers() {
        for(int i = 0; i < bpmObservers.size(); i++) {
            BPMObserver observer = (BPMObserver)bpmObservers.get(i);
            observer.updateBPM();
        }
    }
}
```

The Heart Adapter

```
package headfirst.designpatterns.combined.djview;

public class HeartAdapter implements BeatModelInterface {
    HeartModelInterface heart;

    public HeartAdapter(HeartModelInterface heart) {
```

```
        this.heart = heart;
    }

    public void initialize() {}

    public void on() {}

    public void off() {}

    public int getBPM() {
        return heart.getHeartRate();
    }

    public void setBPM(int bpm) {}

    public void registerObserver(BeatObserver o) {
        heart.registerObserver(o);
    }

    public void removeObserver(BeatObserver o) {
        heart.removeObserver(o);
    }

    public void registerObserver(BPMObserver o) {
        heart.registerObserver(o);
    }

    public void removeObserver(BPMObserver o) {
        heart.removeObserver(o);
    }
}
```

The Controller

```
package headfirst.designpatterns.combined.djview;

public class HeartController implements ControllerInterface {
    HeartModelInterface model;
    DJView view;

    public HeartController(HeartModelInterface model) {
        this.model = model;
        view = new DJView(this, new HeartAdapter(model));
        view.createView();
        view.createControls();
        view.disableStopMenuItem();
        view.disableStartMenuItem();
    }

    public void start() {}

    public void stop() {}

    public void increaseBPM() {}

    public void decreaseBPM() {}

    public void setBPM(int bpm) {}
}
```

[2] **send us email for a copy.**

Chapter 13. Better Living with Patterns: Patterns in the Real World

Ahhhh, now you're ready for a bright new world filled with Design Patterns. But, before you go opening all those new doors of opportunity, we need to cover a few details that you'll encounter out in the real world — that's right, things get a little more complex than they are here in Objectville. Come along, we've got a nice guide to help you through the transition on the next page...

THE OBJECTVILLE GUIDE TO BETTER LIVING WITH DESIGN PATTERNS

Please accept our handy guide with tips & tricks for living with patterns in the real world. In this guide you will:

☞	Learn the all too common misconceptions about the definition of a “Design Pattern.”
☞	Discover those nifty Design Patterns catalogs and why you just have to get one.
☞	Avoid the embarrassment of using a Design Pattern at the wrong time.
☞	Learn how to keep patterns in classifications where they belong.
☞	See that discovering patterns isn’t just for the gurus; read our quick How To and become a patterns writer too.
☞	Be there when the true identity of the mysterious Gang of Four is revealed.
☞	Keep up with the neighbors — the coffee table books any patterns user must own.
☞	Learn to train your mind like a Zen master.
☞	Win friends and influence developers by improving your patterns vocabulary.

Design Pattern defined

We bet you’ve got a pretty good idea of what a pattern is after reading this book. But we’ve never really given a definition for a Design Pattern. Well, you might be a bit surprised by the definition that is in common use:

NOTE

A Pattern is a solution to a problem in a context.

That’s not the most revealing definition is it? But don’t worry, we’re going to step through each of these parts: context, problem and solution:

The **context** is the situation in which the pattern applies. This should be a recurring situation.

NOTE

Example: You have a collection of objects.

The **problem** refers to the goal you are trying to achieve in this context, but it also refers to any constraints that occur in the context.

NOTE

You need to step through the objects without exposing the collection's implementation.

The **solution** is what you're after: a general design that anyone can apply which resolves the goal and set of constraints.

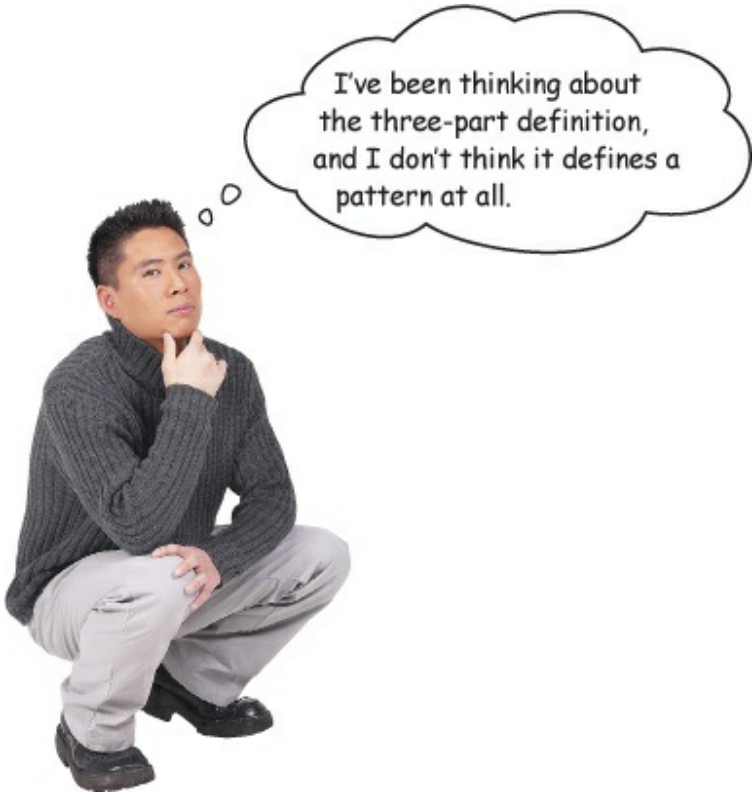
NOTE

Encapsulate the iteration into a separate class.

This is one of those definitions that takes a while to sink in, but take it one step at a time. Here's a little mnemonic you can repeat to yourself to remember it:

> "If you find yourself in a context with a problem that has a goal that is affected by a set of constraints, then you can apply a design that resolves the goal and constraints and leads to a solution."

Now, this seems like a lot of work just to figure out what a Design Pattern is. After all, you already know that a Design Pattern gives you a solution to a common recurring design problem. What is all this formality getting you? Well, you're going to see that by having a formal way of describing patterns we can create a catalog of patterns, which has all kinds of benefits.



You might be right; let's think about this a bit... We need a *problem*, a *solution* and a *context*:

> **Problem**: How do I get to work on time?
> **Context**: I've locked my keys in the car.
> **Solution**: Break the window, get in the car, start the engine and drive to work.

We have all the components of the definition: we have a problem, which includes the goal of getting to work, and the constraints of time, distance and probably some other factors. We also have a context in which the keys to the car are inaccessible. And we have a solution that gets us to the keys and resolves both the time and distance constraints. We must have a pattern now! Right?

BRAIN POWER

We followed the Design Pattern definition and defined a problem, a context, and a solution (which works!). Is this a pattern? If not, how did it fail? Could we fail the same way when defining an OO Design Pattern?

Looking more closely at the Design Pattern definition

Our example does seem to match the Design Pattern definition, but it isn't a true pattern. Why? For starters, we know that a pattern needs to apply to a recurring problem. While an absent-minded person might lock his keys in the car often, breaking the car window doesn't qualify as a solution that can be applied over and over (or at least isn't likely to if we balance the goal with another constraint: cost).

It also fails in a couple of other ways: first, it isn't easy to take this description, hand it to someone and have him apply it to his own unique problem. Second, we've violated an important but simple aspect of a pattern: we haven't even given it a name! Without a name, the pattern doesn't become part of a vocabulary that can be shared with other developers.

Luckily, patterns are not described and documented as a simple problem, context and solution; we have much better ways of describing patterns and collecting them together into *patterns catalogs*.

THERE ARE NO DUMB QUESTIONS

Q: **Q: Am I going to see pattern descriptions that are stated as a problem, a context and a solution?**

A: **A:** Pattern descriptions, which you'll typically find in pattern catalogs, are usually a bit more revealing than that. We're going to look at patterns catalogs in detail in just a minute; they describe a lot more about a pattern's intent and motivation and where it might apply, along with the solution design and the consequences (good and bad) of using it.

Q: **Q: Is it okay to slightly alter a pattern's structure to fit my design? Or am I going to have to go by the strict definition?**

A: **A:** Of course you can alter it. Like design principles, patterns are not meant to be laws or rules; they are guidelines that you can alter to fit your needs. As you've seen, a lot of real-world examples don't fit the classic pattern designs.
However, when you adapt patterns, it never hurts to document how your pattern differs from the classic design — that way, other developers can quickly recognize the patterns you're using and any differences between your pattern and the classic pattern.

Q: **Q: Where can I get a patterns catalog?**

A: **A:** The first and most definitive patterns catalog is *Design Patterns: Elements of Reusable Object-Oriented*

Software, by Gamma, Helm, Johnson & Vlissides (Addison Wesley). This catalog lays out 23 fundamental patterns. We'll talk a little more about this book in a few pages.
Many other patterns catalogs are starting to be published in various domain areas such as enterprise software, concurrent systems and business systems.

GEEK BITS

May the force be with you

The Design Pattern definition tells us that the *problem* consists of a *goal* and a *set of constraints*. Patterns gurus have a term for these: they call them forces. Why? Well, we're sure they have their own reasons, but if you remember the movie, the force "shapes and controls the Universe." Likewise, the forces in the pattern definition shape and control the solution. Only when a solution balances both sides of the force (the light side: your goal, and the dark side: the constraints) do we have a useful pattern.

This "force" terminology can be quite confusing when you first see it in pattern discussions, but just remember that there are two sides of the force (goals and constraints) and that they need to be balanced or resolved to create a pattern solution. Don't let the lingo get in your way and may the force be with you!

Frank: Fill us in, Jim. I've just been learning patterns by reading a few articles here and there.

Jim: Sure, each patterns catalog takes a set of patterns and describes each in detail along with its relationship to the other patterns.

Joe: Are you saying there is more than one patterns catalog?

Jim: Of course; there are catalogs for fundamental Design Patterns and there are also catalogs on domain-specific patterns, like EJB patterns.

Frank: Which catalog are you looking at?

Jim: This is the classic GoF catalog; it contains 23 fundamental Design Patterns.

Frank: GoF?

Jim: Right, that stands for the Gang of Four. The Gang of Four are the guys that put together the first patterns catalog.

Joe: What's in the catalog?

Jim: There is a set of related patterns. For each pattern there is a description that follows a template and spells out a lot of details of the pattern. For instance, each pattern has a *name*.

Frank: Wow, that's earth-shattering — a name! Imagine that.

Jim: Hold on, Frank; actually, the name is really important. When we have a name for a pattern, it gives us a way to talk about the pattern; you know, that whole shared vocabulary thing.

Frank: Okay, okay. I was just kidding. Go on, what else is there?

Jim: Well, like I was saying, every pattern follows a template. For each pattern we have a name and a few sections that tell us more about the pattern. For instance, there is an Intent section that describes what the pattern is, kind of like a definition. Then there are Motivation and Applicability sections that describe when and where the pattern might be used.

Joe: What about the design itself?

Jim: There are several sections that describe the class design along with all the classes that make it up and what their roles are. There is also a section that describes how to implement the pattern and often sample code to show you how.

Frank: It sounds like they've thought of everything.

Jim: There's more. There are also examples of where the pattern has been used in real systems, as well as what I think is one of the most useful sections: how the pattern relates to other patterns.

Frank: Oh, you mean they tell you things like how state and strategy differ?

Jim: Exactly!

Joe: So Jim, how are you actually using the catalog? When you have a problem, do you go fishing in the catalog for a solution?

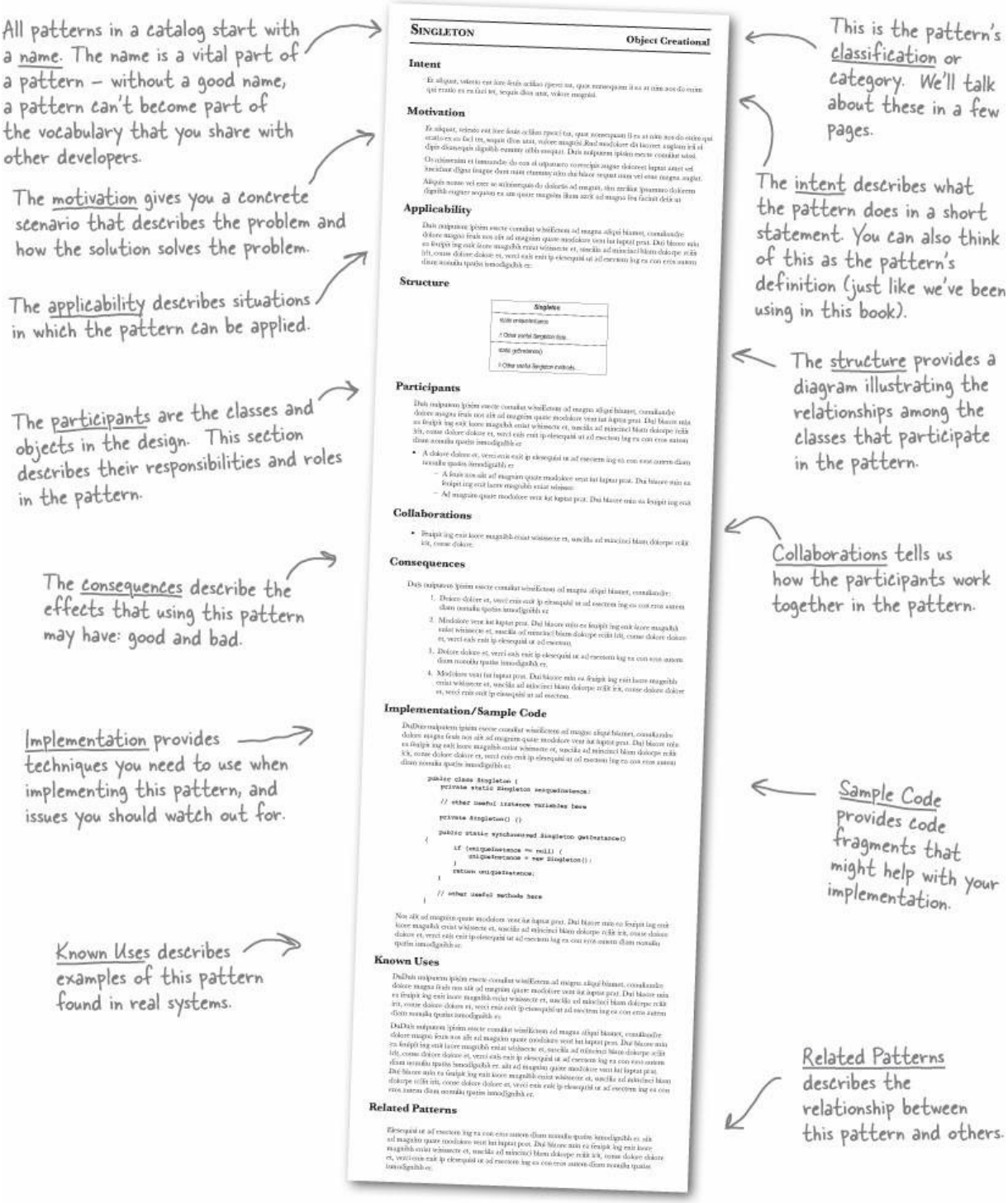
Jim: I try to get familiar with all the patterns and their relationships first. Then, when I need a pattern, I have some idea of what it is. I go back and look at the Motivation and Applicability sections to make sure I've got it right. There is also another really important section: Consequences. I review that to make sure there won't be some unintended effect on my design.

Frank: That makes sense. So once you know the pattern is right, how do you approach working it into your design and implementing it?

Jim: That's where the class diagram comes in. I first read over the Structure section to review the diagram and then over the Participants section to make sure I understand each class's role. From there, I work it into my design, making any alterations I need to make it fit. Then I review the Implementation and Sample code sections to make sure I know about any good implementation techniques or gotchas I might encounter.

Joe: I can see how a catalog is really going to accelerate my use of patterns!

Frank: Totally. Jim, can you walk us through a pattern description?



THERE ARE NO DUMB QUESTIONS

Q: Q: Is it possible to create your own Design Patterns? Or is that something you have to be a "patterns guru" to do?

A:	**A:** First, remember that patterns are discovered, not created. So, anyone can discover a Design Pattern and then author its description; however, it's not easy and doesn't happen quickly, nor often. Being a "patterns writer" takes commitment. You should first think about why you'd want to — the majority of people don't author patterns; they just use them. However, you might work in a specialized domain for which you think new patterns would be helpful, or you might have come across a solution to what you think is a recurring problem, or you may just want to get involved in the patterns community and contribute to the growing body of work.
Q:	**Q: I'm game; how do I get started?**
A:	**A:** As with any discipline, the more you know the better. Studying existing patterns, what they do, and how they relate to other patterns is crucial. Not only does it make you familiar with how patterns are crafted, it also prevents you from reinventing the wheel. From there you'll want to start writing your patterns on paper, so you can communicate them to other developers; we're going to talk more about how to communicate your patterns in a bit. If you're really interested, you'll want to read the section that follows these Q&As.
Q:	**Q: How do I know when I really have a pattern?**
A:	**A:** That's a very good question: you don't have a pattern until others have used it and found it to work. In general, you don't have a pattern until it passes the "Rule of Three." This rule states that a pattern can be called a pattern only if it has been applied in a real-world solution at least three times.

So you wanna be a design patterns star?
Well, listen now to what I tell.
Get yourself a patterns catalog,
Then take some time and learn it well.
And when you've got your description right,
And three developers agree without a fight,
Then you'll know it's a pattern alright.

NOTE

To the tune of "So you wanna be a Rock'n Roll Star."

So you wanna be a Design Patterns writer

Do your homework. You need to be well versed in the existing patterns before you can create a new one. Most patterns that appear to be new, are, in fact, just variants of existing patterns. By studying patterns, you become better at recognizing them, and you learn to relate them to other patterns.

Take time to reflect, evaluate. Your experience — the problems you've encountered, and the solutions you've used — are where ideas for patterns are born. So take some time to reflect on your experiences and comb them for novel designs that recur. Remember that most designs are variations on existing patterns and not new patterns. And when you do find what looks like a new pattern, its applicability may be too narrow to qualify as a real pattern.

Get your ideas down on paper in a way others can understand. Locating

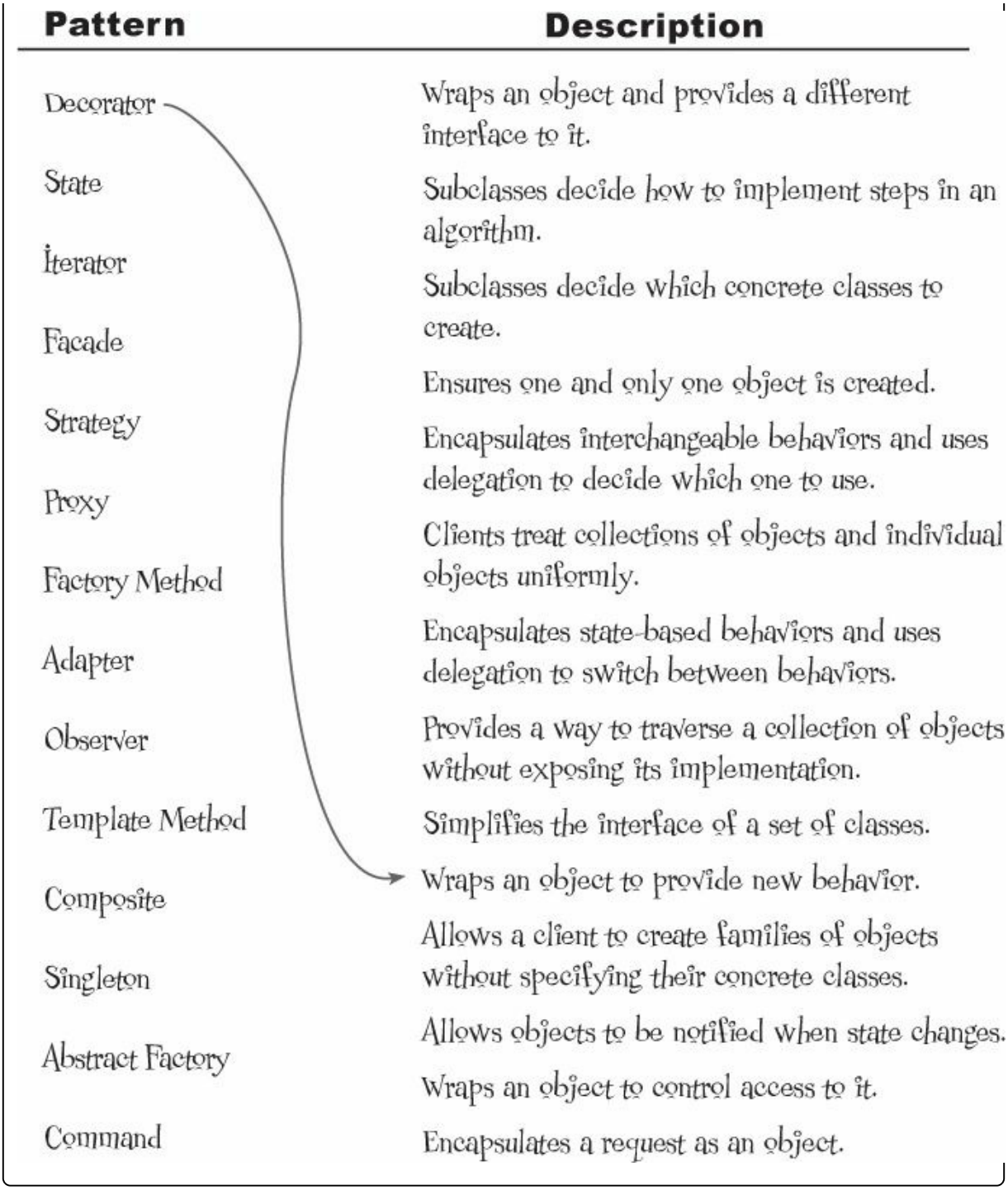
new patterns isn’t of much use if others can’t make use of your find; you need to document your pattern candidates so that others can read, understand, and apply them to their own solution and then supply you with feedback. Luckily, you don’t need to invent your own method of documenting your patterns. As you’ve already seen with the GoF template, a lot of thought has already gone into how to describe patterns and their characteristics.

Have others try your patterns; then refine and refine some more. Don’t expect to get your pattern right the first time. Think of your pattern as a work in progress that will improve over time. Have other developers review your candidate pattern, try it out, and give you feedback. Incorporate that feedback into your description and try again. Your description will never be perfect, but at some point it should be solid enough that other developers can read and understand it.

Don’t forget the Rule of Three. Remember, unless your pattern has been successfully applied in three real-world solutions, it can’t qualify as a pattern. That’s another good reason to get your pattern into the hands of others so they can try it, give feedback, and allow you to converge on a working pattern.

WHO DOES WHAT?

Match each pattern with its description:



Organizing Design Patterns

As the number of discovered Design Patterns grows, it makes sense to partition them into classifications so that we can organize them, narrow our searches to a subset of all Design Patterns, and make comparisons within a

group of patterns.

In most catalogs, you'll find patterns grouped into one of a few classification schemes. The most well-known scheme was used by the first patterns catalog and partitions patterns into three distinct categories based on their purposes: Creational, Behavioral, and Structural.

SHARPEN YOUR PENCIL

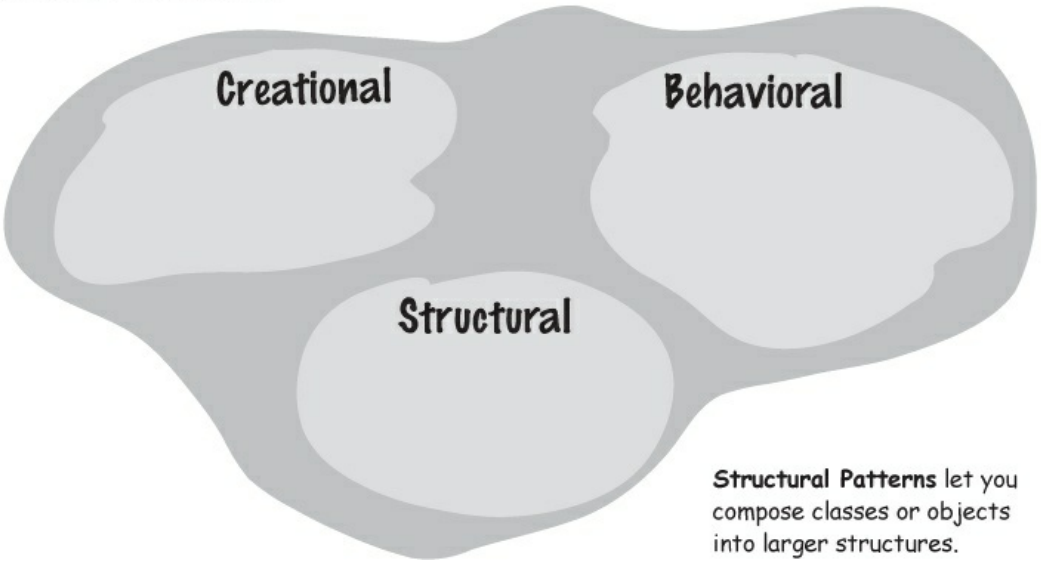
Read each category description and see if you can corral these patterns into their correct categories. This is a toughy! But give it your best shot and then check out the answers on the next page.

Abstract Factory
Composite
Observer
Strategy
Decorator
State
Adapter
Singleton
Factory Method
Command
Proxy
Template Method
Iterator
Facade

Each of these patterns belongs in one of those categories

Creational Patterns involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate.

Any pattern that is a **Behavioral Pattern** is concerned with how classes and objects interact and distribute responsibility.



Structural Patterns let you compose classes or objects into larger structures.

Pattern Categories

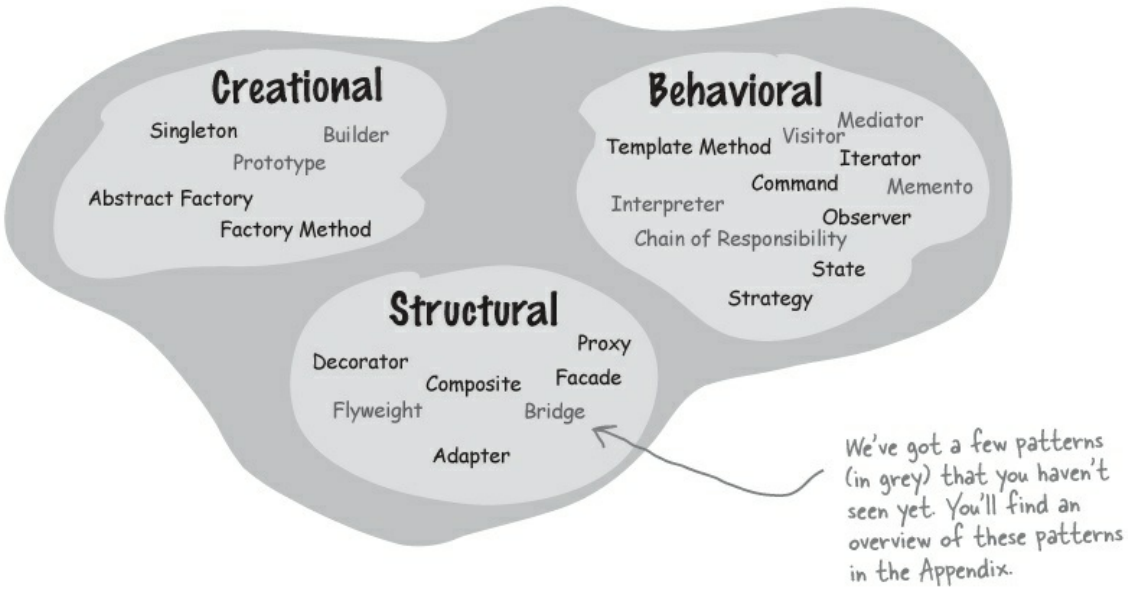
Sharpen your pencil Solution

Here's the grouping of patterns into categories. You probably found the

exercise difficult, because many of the patterns seem like they could fit into more than one category. Don’t worry, everyone has trouble figuring out the right categories for the patterns.

Creational Patterns involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate.

Any pattern that is a **Behavioral Pattern** is concerned with how classes and objects interact and distribute responsibility.

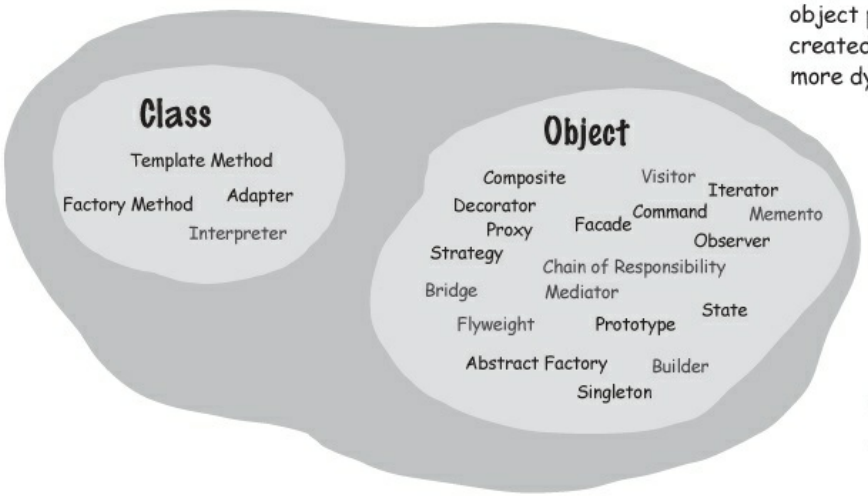


Structural Patterns let you compose classes or objects into larger structures.

Patterns are often classified by a second attribute: whether or not the pattern deals with classes or objects:

Class Patterns describe how relationships between classes are defined via inheritance. Relationships in class patterns are established at compile time.

Object Patterns describe relationships between objects and are primarily defined by composition. Relationships in object patterns are typically created at runtime and are more dynamic and flexible.



Notice there are a lot more object patterns than class patterns!

THERE ARE NO DUMB QUESTIONS

Q: Q: Are these the only classification schemes?

A: A: No, other schemes have been proposed. Some other schemes start with the three categories and then add subcategories, like "Decoupling Patterns." You'll want to be familiar with the most common schemes for organizing patterns, but also feel free to create your own, if it helps you to understand the patterns better.

Q: Q: Does organizing patterns into categories really help you remember them?

A: A: It certainly gives you a framework for the sake of comparison. But many people are confused by the creational, structural and behavioral categories; often a pattern seems to fit into more than one category. The most important thing is to know the patterns and the relationships among them. When categories help, use them!

Q: Q: Why is the Decorator Pattern in the structural category? I would have thought of that as a behavioral pattern; after all, it adds behavior!

A: A: Yes, lots of developers say that! Here's the thinking behind the Gang of Four classification: structural patterns describe how classes and objects are composed to create new structures or new functionality. The Decorator Pattern allows you to compose objects by wrapping one object with another to provide new functionality. So the focus is on how you compose the objects dynamically to gain functionality, rather than on the communication and interconnection between objects, which is the purpose of behavioral patterns. But remember, the intent of these patterns is different, and that's often the key to understanding which category a pattern belongs to.

MASTER AND STUDENT...

Master: *Grasshopper, you look troubled.*

Student: *Yes, I've just learned about pattern classification and I'm confused.*

Master: *Grasshopper, continue...*

Student: *After learning much about patterns, I've just been told that each pattern fits into one of three classifications: structural, behavioral, or creational. Why do we need*

these classifications?

Master: *Grasshopper, whenever we have a large collection of anything, we naturally find categories to fit those things into. It helps us to think of the items at a more abstract level.*

Student: *Master; can you give me an example?*

Master: *Of course. Take automobiles; there are many different models of automobiles and we naturally put them into categories like economy cars, sports cars, SUVs, trucks, and luxury car categories.*

Master: *Grasshopper, you look shocked; does this not make sense?*

Student: *Master, it makes a lot of sense, but I am shocked you know so much about cars!*

Master: *Grasshopper, I can't relate* ***everything*** *to lotus flowers or rice bowls. Now, may I continue?*

Student: *Yes, yes, I'm sorry, please continue.*

Master: *Once you have classifications or categories you can easily talk about the different groupings: "If you're doing the mountain drive from Silicon Valley to Santa Cruz, a sports car with good handling is the best option." Or, "With the worsening oil situation, you really want to buy a economy car; they're more fuel-efficient."*

Student: *So by having categories we can talk about a set of patterns as a group. We might know we need a creational pattern, without knowing exactly which one, but we can still talk about creational patterns.*

Master*: Yes, and it also gives us a way to compare a member to the rest of the category. For example, "the Mini really is the most stylish compact car around," or to narrow our search, "I need a fuel-efficient car."*

Student: *I see. So I might say that the Adapter Pattern is the best structural pattern for changing an object's interface.*

Master: *Yes. We also can use categories for one more purpose: to launch into new territory. For instance, "we really want to deliver a sports car with Ferrari performance at Miata prices."*

Student: *That sounds like a death trap.*

Master: *I'm sorry, I did not hear you Grasshopper.*

Student: *Uh, I said "I see that."*

Student: *So categories give us a way to think about the way groups of patterns relate and how patterns within a group relate to one another. They also give us a way to extrapolate to new patterns. But why are there three categories and not four, or five?*

Master: *Ah, like stars in the night sky, there are as many categories as you want to see. Three is a convenient number and a number that many people have decided makes for a*

nice grouping of patterns. But others have suggested four, five or more.

Thinking in Patterns

Contexts, constraints, forces, catalogs, classifications... boy, this is starting to sound mighty academic. Okay, all that stuff is important and knowledge is power. But, let's face it, if you understand the academic stuff and don't have the *experience* and practice using patterns, then it's not going to make much difference in your life.

Here's a quick guide to help you start to *think in patterns.* What do we mean by that? We mean being able to look at a design and see where patterns naturally fit and where they don't.

Keep it simple (KISS)

First of all, when you design, solve things in the simplest way possible. Your

goal should be simplicity, not “how can I apply a pattern to this problem?” Don’t feel like you aren’t a sophisticated developer if you don’t use a pattern to solve a problem. Other developers will appreciate and admire the simplicity of your design. That said, sometimes the best way to keep your design simple and flexible is to use a pattern.

Design Patterns aren’t a magic bullet; in fact, they’re not even a bullet!

Patterns, as you know, are general solutions to recurring problems. Patterns also have the benefit of being well tested by lots of developers. So, when you see a need for one, you can sleep well knowing many developers have been there before and solved the problem using similar techniques.

However, patterns aren’t a magic bullet. You can’t plug one in, compile and then take an early lunch. To use patterns, you also need to think through the consequences for the rest of your design.

You know you need a pattern when...

Ah... the most important question: when do you use a pattern? As you approach your design, introduce a pattern when you’re sure it addresses a problem in your design. If a simpler solution might work, give that consideration before you commit to using a pattern.

Knowing when a pattern applies is where your experience and knowledge come in. Once you’re sure a simple solution will not meet your needs, you should consider the problem along with the set of constraints under which the solution will need to operate — these will help you match your problem to a pattern. If you’ve got a good knowledge of patterns, you may know of a pattern that is a good match. Otherwise, survey patterns that look like they might solve the problem. The intent and applicability sections of the patterns catalogs are particularly useful for this. Once you’ve found a pattern that appears to be a good match, make sure it has a set of consequences you can live with and study its effect on the rest of your design. If everything looks good, go for it!

There is one situation in which you’ll want to use a pattern even if a simpler solution would work: when you expect aspects of your system to vary. As we’ve seen, identifying areas of change in your design is usually a good sign

that a pattern is needed. Just make sure you are adding patterns to deal with *practical change* that is likely to happen, not *hypothetical change* that may happen.

Design time isn't the only time you want to consider introducing patterns; you'll also want to do so at refactoring time.

Refactoring time is Patterns time!

Refactoring is the process of making changes to your code to improve the way it is organized. The goal is to improve its structure, not change its behavior. This is a great time to reexamine your design to see if it might be better structured with patterns. For instance, code that is full of conditional statements might signal the need for the State Pattern. Or, it may be time to clean up concrete dependencies with a Factory. Entire books have been written on the topic of refactoring with patterns, and as your skills grow, you'll want to study this area more.

Take out what you don't really need. Don't be afraid to remove a Design Pattern from your design.

No one ever talks about when to remove a pattern. You'd think it was blasphemy! Nah, we're all adults here; we can take it.

So when do you remove a pattern? When your system has become complex and the flexibility you planned for isn't needed. In other words, when a simpler solution without the pattern would be better.

If you don't need it now, don't do it now.

Design Patterns are powerful, and it's easy to see all kinds of ways they can be used in your current designs. Developers naturally love to create beautiful architectures that are ready to take on change from any direction.

Resist the temptation. If you have a practical need to support change in a design today, go ahead and employ a pattern to handle that change. However, if the reason is only hypothetical, don't add the pattern; it is only going to add complexity to your system, and you might never need it!

MASTER AND STUDENT...

Master: *Grasshopper, your initial training is almost complete. What are your plans?*

Student: *I'm going to Disneyland! And, then I'm going to start creating lots of code with patterns!*

Master: *Whoa, hold on. Never use your big guns unless you have to.*

Student: *What do you mean, Master? Now that I've learned design patterns shouldn't I be using them in all my designs to achieve maximum power, flexibility and manageability?*

Master: *No; patterns are a tool, and a tool that should only be used when needed. You've also spent a lot of time learning design principles. Always start from your*

principles and create the simplest code you can that does the job. However, if you see the need for a pattern emerge, then use it.

Student: *So I shouldn't build my designs from patterns?*

Master: *That should not be your goal when beginning a design. Let patterns emerge naturally as your design progresses.*

Student: *If patterns are so great, why should I be so careful about using them?*

Master: *Patterns can introduce complexity, and we never want complexity where it is not needed. But patterns are powerful when used where they are needed. As you already know, patterns are proven design experience that can be used to avoid common mistakes. They're also a shared vocabulary for communicating our design to others.*

Student: *Well, when do we know it's okay to introduce design patterns?*

Master: *Introduce a pattern when you are sure it's necessary to solve a problem in your design, or when you are quite sure that it is needed to deal with a future change in the requirements of your application.*

Student: *I guess my learning is going to continue even though I already understand a lot of patterns.*

Master: *Yes, grasshopper; learning to manage the complexity and change in software is a life-long pursuit. But now that you know a good set of patterns, the time has come to apply them where needed in your design and to continue learning more patterns.*

Student: *Wait a minute, you mean I don't know them ALL?*

Master: *Grasshopper, you've learned the fundamental patterns; you're going to find there are many more, including patterns that just apply to particular domains such as concurrent systems and enterprise systems. But now that you know the basics, you're in good shape to learn them.*

Your Mind on Patterns

The Beginner uses patterns everywhere. This is good: the beginner gets lots of experience with and practice using patterns. The beginner also thinks, "The more patterns I use, the better the design." The beginner will learn this is not so, that all designs should be as simple as possible. Complexity and patterns should only be used where they are needed for practical extensibility.

“I need a pattern for Hello World.”

As learning progresses, the Intermediate mind starts to see where patterns are needed and where they aren’t. The intermediate mind still tries to fit too many square patterns into round holes, but also begins to see that patterns can be adapted to fit situations where the canonical pattern doesn’t fit.

“Maybe I need a Singleton here.”

The Zen mind is able to see patterns where they fit naturally. The Zen mind is not obsessed with using patterns; rather it looks for simple solutions that best solve the problem. The Zen mind thinks in terms of the object principles and their trade-offs. When a need for a pattern naturally arises, the Zen mind applies it knowing well that it may require adaptation. The Zen mind also sees relationships to similar patterns and understands the subtleties of differences in the intent of related patterns. *The Zen mind is also a Beginner mind* — it doesn’t let all that pattern knowledge overly influence design decisions.

“This is a natural place for Decorator.”

NOTE

WARNING: Overuse of design patterns can lead to code that is downright over-engineered. Always go with the simplest solution that does the job and introduce patterns where the need emerges.

Of course we want you to use Design Patterns!

But we want you to be a good OO designer even more.

When a design solution calls for a pattern, you get the benefits of using a solution that has been time-tested by lots of developers. You’re also using a solution that is well documented and that other developers are going to recognize (you know, that whole shared vocabulary thing).

However, when you use Design Patterns, there can also be a downside.

Design Patterns often introduce additional classes and objects, and so they can increase the complexity of your designs. Design Patterns can also add more layers to your design, which adds not only complexity, but also inefficiency.

Also, using a Design Pattern can sometimes be outright overkill. Many times you can fall back on your design principles and find a much simpler solution to solve the same problem. If that happens, don't fight it. Use the simpler solution.

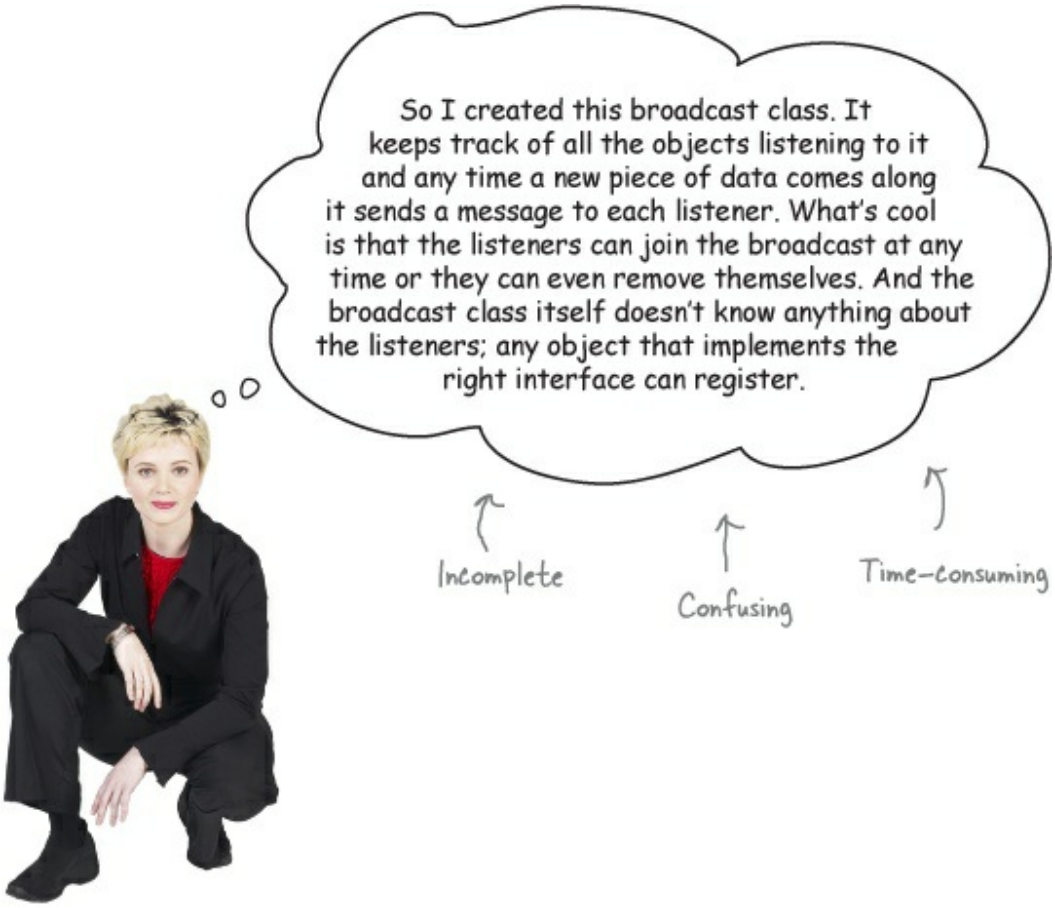
Don't let us discourage you, though. When a Design Pattern is the right tool for the job, the advantages are many.

Don't forget the power of the shared vocabulary

We've spent so much time in this book discussing OO nuts and bolts that it's easy to forget the human side of Design Patterns — they don't just help load your brain with solutions, they also give you a shared vocabulary with other developers. Don't underestimate the power of a shared vocabulary, it's one of the *biggest benefits* of Design Patterns.

Just think, something has changed since the last time we talked about shared vocabularies; you've now started to build up quite a vocabulary of your own! Not to mention, you have also learned a full set of OO design principles from which you can easily understand the motivation and workings of any new patterns you encounter.

Now that you've got the Design Pattern basics down, it's time for you to go out and spread the word to others. Why? Because when your fellow developers know patterns and use a shared vocabulary as well, it leads to better designs, better communication, and, best of all, it'll save you a lot of time that you can spend on cooler things.

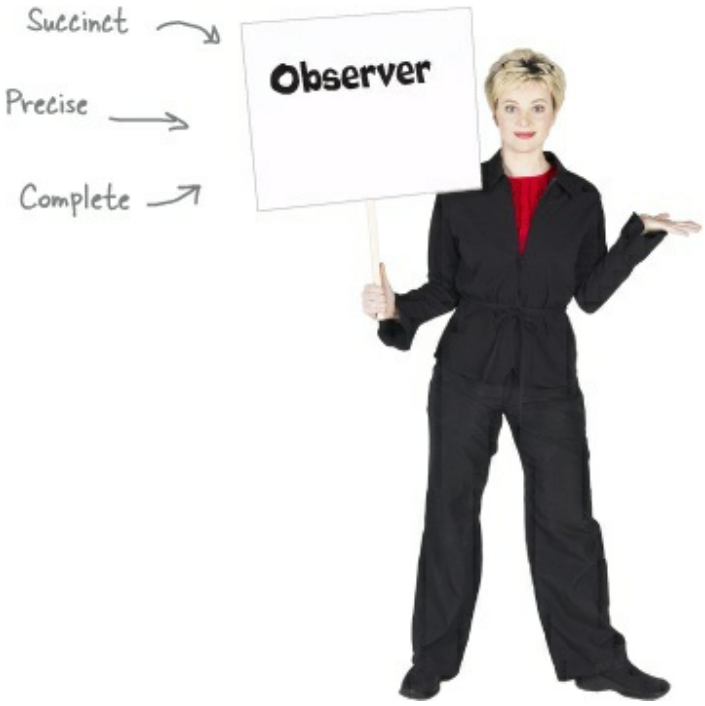


Top five ways to share your vocabulary

1. **In design meetings:** When you meet with your team to discuss a software design, use design patterns to help stay "in the design" longer. Discussing designs from the perspective of Design Patterns and OO principles keeps your team from getting bogged down in implementation details and prevent many misunderstandings.
2. **With other developers:** Use patterns in your discussions with other developers. This helps other developers learn about new patterns and builds a community. The best part about sharing what you've learned is that great feeling when someone else "gets it"!
3. **In architecture documentation:** When you write architectural documentation, using patterns will reduce the amount of documentation you need to write and gives the reader a clearer picture of the design.
4. **In code comments and naming conventions:** When you're writing code, clearly identify the patterns you're using in comments. Also, choose class and method names that reveal any patterns underneath.

Other developers who have to read your code will thank you for allowing them to quickly understand your implementation.

5. **To groups of interested developers:** Share your knowledge. Many developers have heard about patterns but don't have a good understanding of what they are. Volunteer to give a brown-bag lunch on patterns or a talk at your local user group.



Cruisin' Objectville with the Gang of Four

You won’t find the Jets or Sharks hanging around Objectville, but you will find the Gang of Four. As you’ve probably noticed, you can’t get far in the World of Patterns without running into them. So, who is this mysterious gang?

Put simply, “the GoF,” which includes Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, is the group of guys who put together the first patterns catalog and in the process, started an entire movement in the software field!

How did they get that name? No one knows for sure; it’s just a name that stuck. But think about it: if you’re going to have a “gang element” running around Objectville, could you think of a nicer bunch of guys? In fact, they’ve even agreed to pay us a visit...

*John Vlissides passed away in 2005. A great loss to the Design Patterns community.

Your journey has just begun...

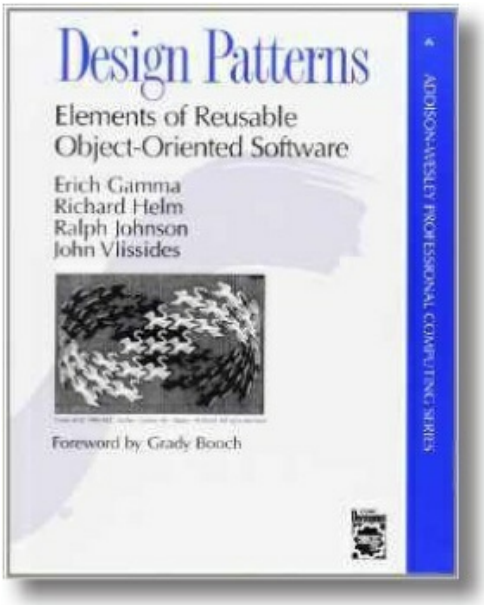
Now that you're on top of Design Patterns and ready to dig deeper, we've got three definitive texts that you need to add to your bookshelf...

The definitive Design Patterns text

This is the book that kicked off the entire field of Design Patterns when it was released in 1995. You'll find all the fundamental patterns here. In fact, this book is the basis for the set of patterns we used in *Head First Design Patterns*.

You won't find this book to be the last word on Design Patterns — the field has grown substantially since its publication — but it is the first and most definitive.

Picking up a copy of *Design Patterns* is a great way to start exploring patterns after Head First.



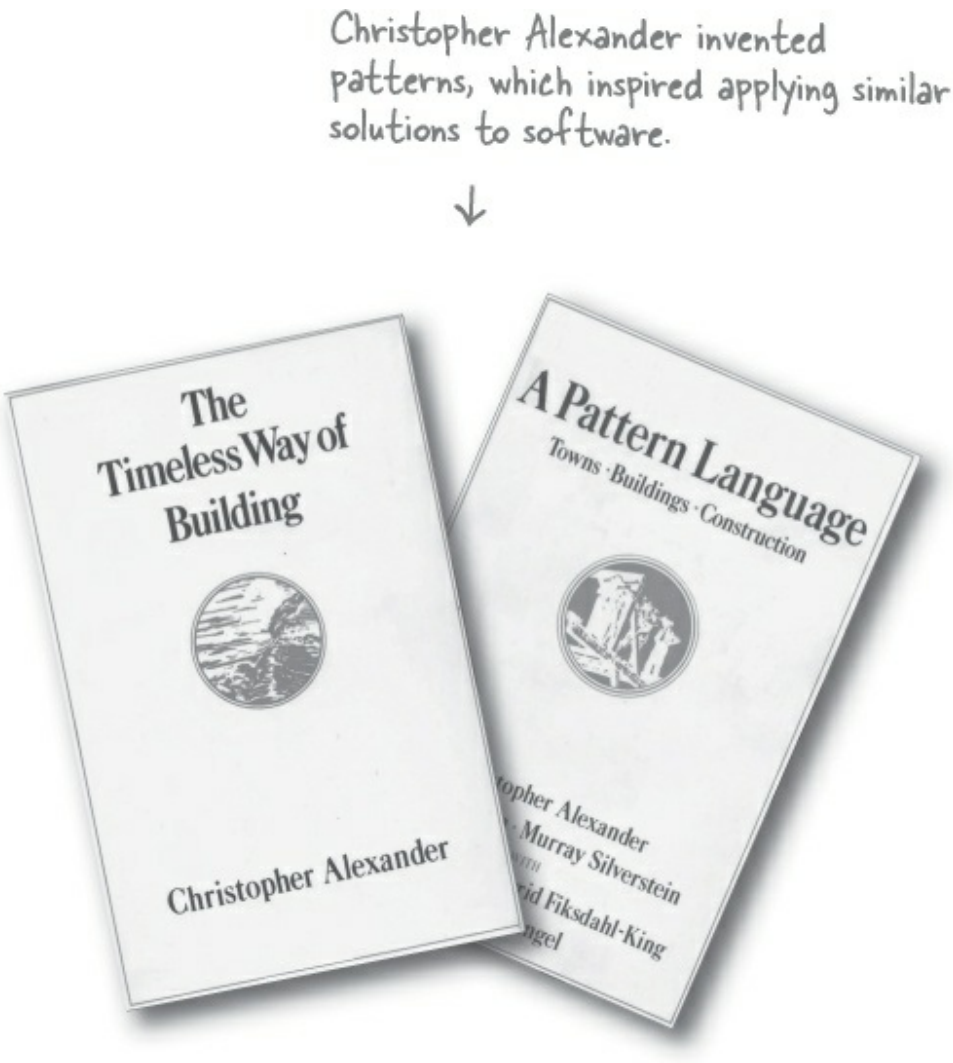
The authors of Design Patterns are affectionately known as the "Gang of Four," or GoF for short.

The definitive Patterns texts

Patterns didn't start with the GoF; they started with Christopher Alexander, a professor of architecture at Berkeley — that's right, Alexander is an *architect*, not a computer scientist. Alexander invented patterns for building living architectures (like houses, towns and cities).

The next time you're in the mood for some deep, engaging reading, pick up *The Timeless Way of Building* and *A Pattern Language*. You'll see the true beginnings of Design Patterns and recognize the direct analogies between creating "living architecture" and flexible, extensible software.

So grab a cup of Starbuzz Coffee, sit back, and enjoy...



Other Design Patterns resources

You’re going to find there is a vibrant, friendly community of patterns users and writers out there and they’re glad to have you join them. Here are a few resources to get you started...

Websites

The Portland Patterns Repository, run by Ward Cunningham, is a wiki devoted to all things related to patterns. Anyone can participate. You’ll find threads of discussion on every topic you can think of related to patterns and OO systems.

```
http://c2.com/cgi/wiki?WelcomeVisitors
```

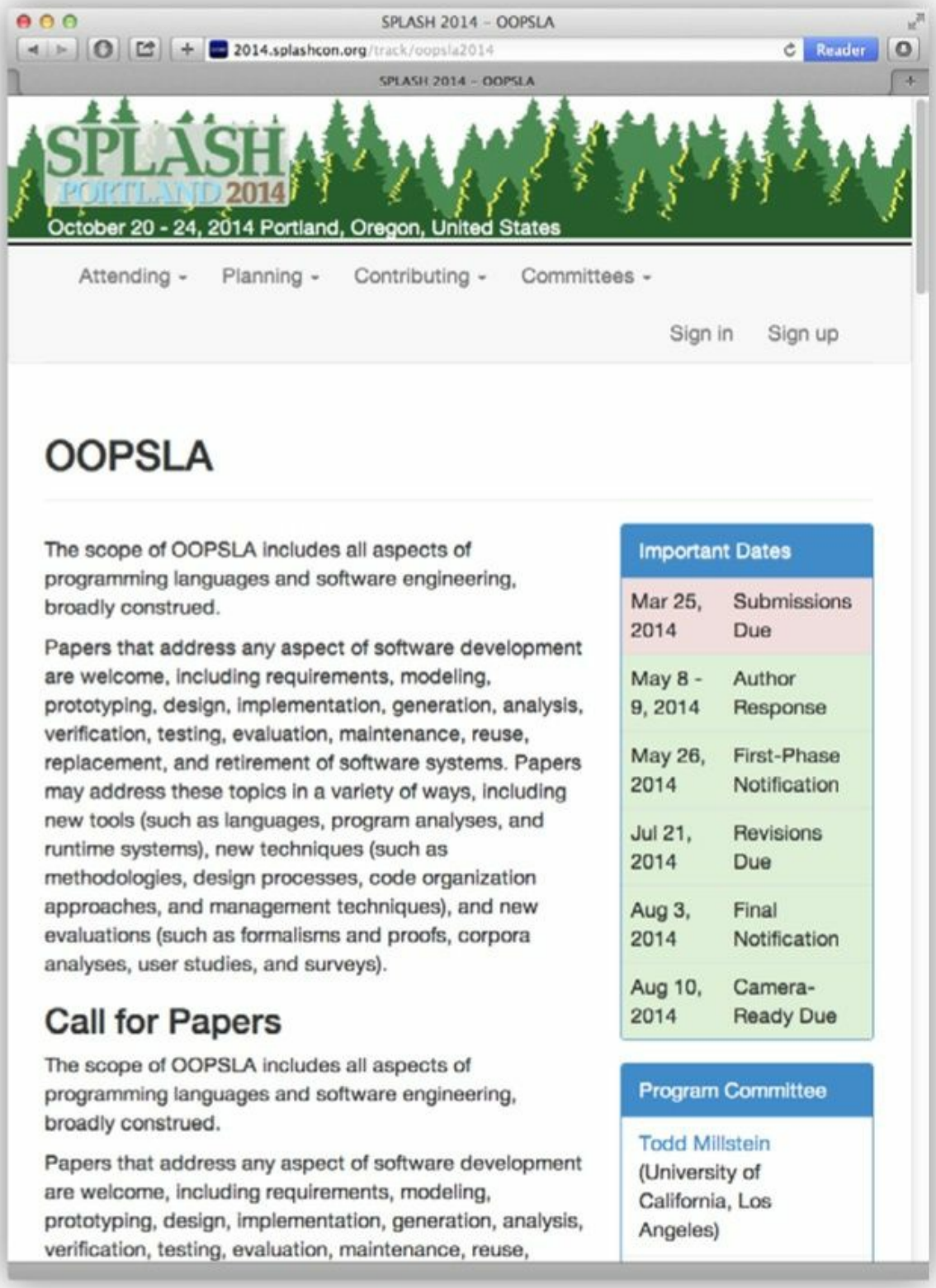
The **Hillside Group** fosters common programming and design practices and

provides a central resource for patterns work. The site includes information on many patterns-related resources such as articles, books, mailing lists and tools.

```
http://hillside.net/
```

Conferences and Workshops

And if you'd like to get some face-to-face time with the patterns community, be sure to check out the many patterns-related conferences and workshops. The Hillside site maintains a complete list. At the least you'll want to check out Pattern Languages of Programs (PLoP), and the ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA).


SPLASH 2014 - OOPSLA
2014.splashcon.org/track/oopsla2014
Reader
SPLASH 2014 - OOPSLA
SPLASH
PORTLAND 2014
October 20 - 24, 2014 Portland, Oregon, United States
Attending
Planning
Contributing
Committees
Sign in
Sign up
OOPSLA
The scope of OOPSLA includes all aspects of programming languages and software engineering, broadly construed.
Papers that address any aspect of software development are welcome, including requirements, modeling, prototyping, design, implementation, generation, analysis, verification, testing, evaluation, maintenance, reuse, replacement, and retirement of software systems. Papers may address these topics in a variety of ways, including new tools (such as languages, program analyses, and runtime systems), new techniques (such as methodologies, design processes, code organization approaches, and management techniques), and new evaluations (such as formalisms and proofs, corpora analyses, user studies, and surveys).
Call for Papers
The scope of OOPSLA includes all aspects of programming languages and software engineering, broadly construed.
Papers that address any aspect of software development are welcome, including requirements, modeling, prototyping, design, implementation, generation, analysis, verification, testing, evaluation, maintenance, reuse,
Important Dates
Mar 25, 2014 Submissions Due
May 8 - 9, 2014 Author Response
May 26, 2014 First-Phase Notification
Jul 21, 2014 Revisions Due
Aug 3, 2014 Final Notification
Aug 10, 2014 Camera-Ready Due
Program Committee
Todd Millstein (University of California, Los Angeles)

The Patterns Zoo

As you've just seen, patterns didn't start with software; they started with the architecture of buildings and towns. In fact, the patterns concept can be applied in many different domains. Take a walk around the Patterns Zoo to see a few...

Architectural Patterns are used to create the living, vibrant architecture of buildings, towns, and cities. This is where patterns got their start.

> **NOTE**
>
> Habitat: found in buildings you like to live in, look at and visit.

> **NOTE**
>
> Habitat: seen hanging around 3-tier architectures, client-server systems and the web.

Application Patterns are patterns for creating system-level architecture. Many multi-tier architectures fall into this category.

> **NOTE**
>
> Field note: MVC has been known to pass for an application pattern.

Domain-Specific Patterns are patterns that concern problems in specific domains, like concurrent systems or real-time systems.

NOTE

Help find a habitat_____

_____J2EE_____

_____

_____

_____

Business Process Patterns describe the interaction between businesses, customers and data, and can be applied to problems such as how to effectively make and communicate decisions.

NOTE

Help find a habitat_____

Development team_____

Customer support team_____

_____

_____

_____

Organizational Patterns describe the structures and practices of human organizations. Most efforts to date have focused on organizations that produce and/or support software.

User Interface Design Patterns address the problems of how to design interactive software programs.

> **NOTE**
>
> Habitat: seen in the vicinity of video game designers, GUI builders, and producers.

> **NOTE**
>
> Field notes: please add your observations of pattern domains here:
>
> _____
>
> _____
>
> _____
>
> _____

Annihilating evil with Anti-Patterns

The Universe just wouldn’t be complete if we had patterns and no anti-patterns, now would it?

If a Design Pattern gives you a general solution to a recurring problem in a

particular context, then what does an anti-pattern give you?

> **NOTE**
>
> An **Anti-Pattern** tells you how to go from a problem to a BAD solution.

You're probably asking yourself, "Why on earth would anyone waste their time documenting bad solutions?"

Think about it like this: if there is a recurring bad solution to a common problem, then by documenting it we can prevent other developers from making the same mistake. After all, avoiding bad solutions can be just as valuable as finding good ones!

Let's look at the elements of an anti-pattern:

An anti-pattern tells you why a bad solution is attractive. Let's face it, no one would choose a bad solution if there wasn't something about it that seemed attractive up front. One of the biggest jobs of the anti-pattern is to alert you to the seductive aspect of the solution.

An anti-pattern tells you why that solution in the long term is bad. In order to understand why it's an anti-pattern, you've got to understand how it's going to have a negative effect down the road. The anti-pattern describes where you'll get into trouble using the solution.

An anti-pattern suggests other patterns that are applicable which may provide good solutions. To be truly helpful, an anti-pattern needs to point you in the right direction; it should suggest other possibilities that may lead to good solutions.

Let's have a look at an anti-pattern.

> **An anti-pattern always looks like a good solution, but then turns out to be a bad solution when it is applied.**
> **By documenting anti-patterns we help others to recognize bad solutions before they implement them.**
> **Like patterns, there are many types of anti-patterns including development, OO, organizational, and domain-specific anti-patterns.**

> **NOTE**
>
> Here's an example of a software development anti-pattern.

ANTI-PATTERN

Name: Golden Hammer

> **NOTE**
>
> Just like a Design Pattern, an anti-pattern has a name so we can create a shared vocabulary.

Problem: You need to choose technologies for your development and you believe that exactly one technology must dominate the architecture.

Context: You need to develop some new system or piece of software that doesn't fit well with the technology that the development team is familiar with.

> **NOTE**
>
> The problem and context, just like a Design Pattern description.

Forces:

> **NOTE**
>
> Tells you why the solution is attractive.

- The development team is committed to the technology they know.
- The development team is not familiar with other technologies.
- Unfamiliar technologies are seen as risky.
- It is easy to plan and estimate for development using the familiar technology.

Supposed Solution: Use the familiar technology anyway. The technology is applied obsessively to many problems, including places where it is clearly inappropriate.

> **NOTE**
>
> The bad, yet attractive, solution.

Refactored Solution: Expanding the knowledge of developers through education, training, and book study groups that expose developers to new solutions.

> **NOTE**
>
> How to get to a good solution.

Examples:

NOTE

Example of where this anti-pattern has been observed.

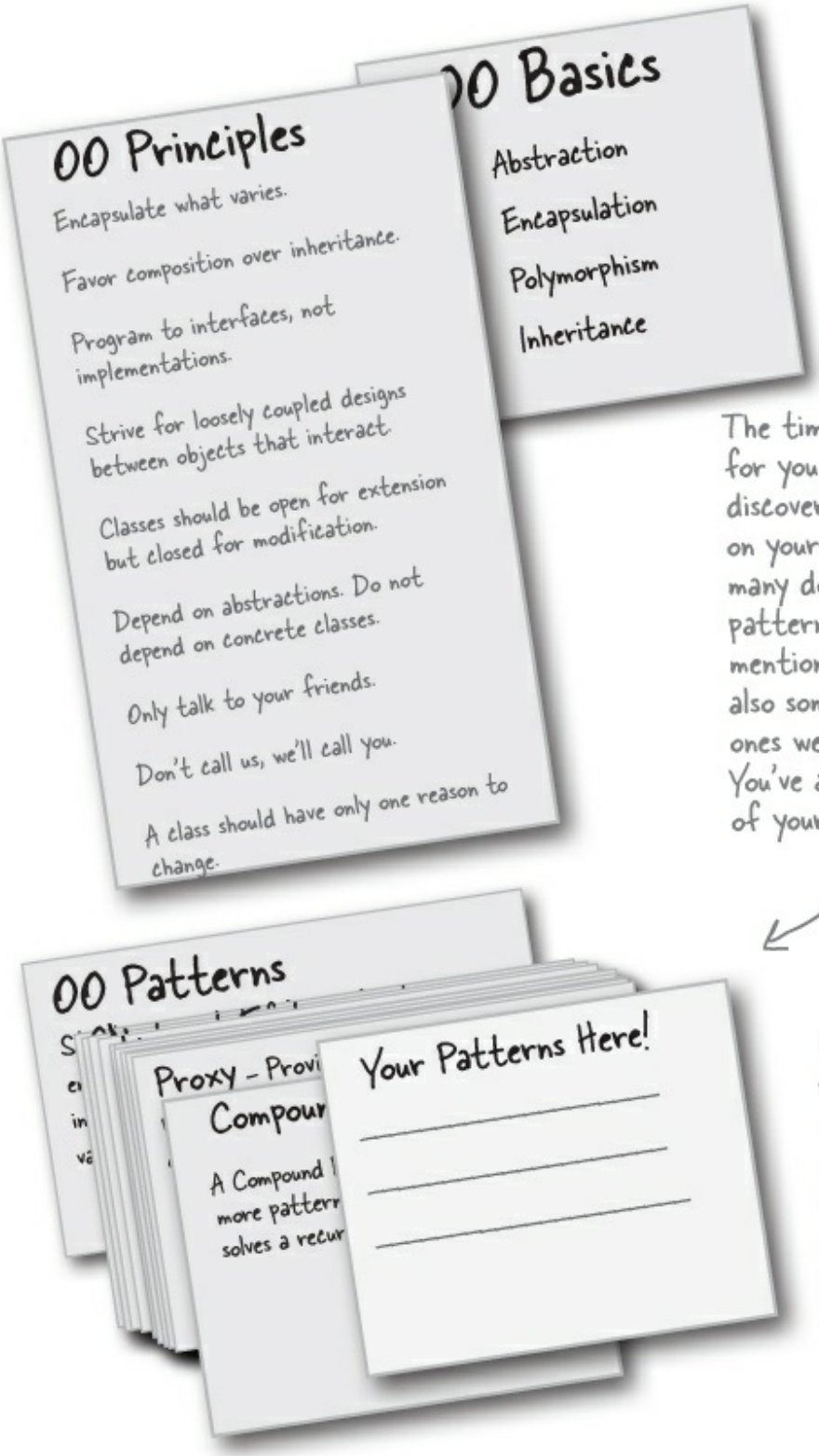
Web companies keep using and maintaining their internal homegrown caching systems when open source alternatives are in use.

NOTE

Adapted from the Portland Pattern Repository's WIKI at http://c2.com/ where you'll find many anti patterns and discussions.

Tools for your Design Toolbox

You've reached that point where you've outgrown us. Now's the time to go out in the world and explore patterns on your own...



The time has come for you to go out and discover more patterns on your own. There are many domain-specific patterns we haven't even mentioned and there are also some foundational ones we didn't cover. You've also got patterns of your own to create.

Check out the Appendix; we'll give you a heads up on some more foundational patterns you'll probably want to have a look at.

BULLET POINTS

- Let Design Patterns emerge in your designs; don’t force them in just for the sake of using a pattern.
- Design Patterns aren’t set in stone; adapt and tweak them to meet your needs.
- Always use the simplest solution that meets your needs, even if it doesn’t include a pattern.
- Study Design Patterns catalogs to familiarize yourself with patterns and the relationships among them.
- Pattern classifications (or categories) provide groupings for patterns. When they help, use them.
- You need to be committed to be a patterns writer: it takes time and patience, and you have to be willing to do lots of refinement.
- Remember, most patterns you encounter will be adaptations of existing patterns, not new patterns.
- Build your team’s shared vocabulary. This is one of the most powerful benefits of using patterns.
- Like any community, the patterns community has its own lingo. Don’t let that hold you back. Having read this book, you now know most of it.

Leaving Objectville...

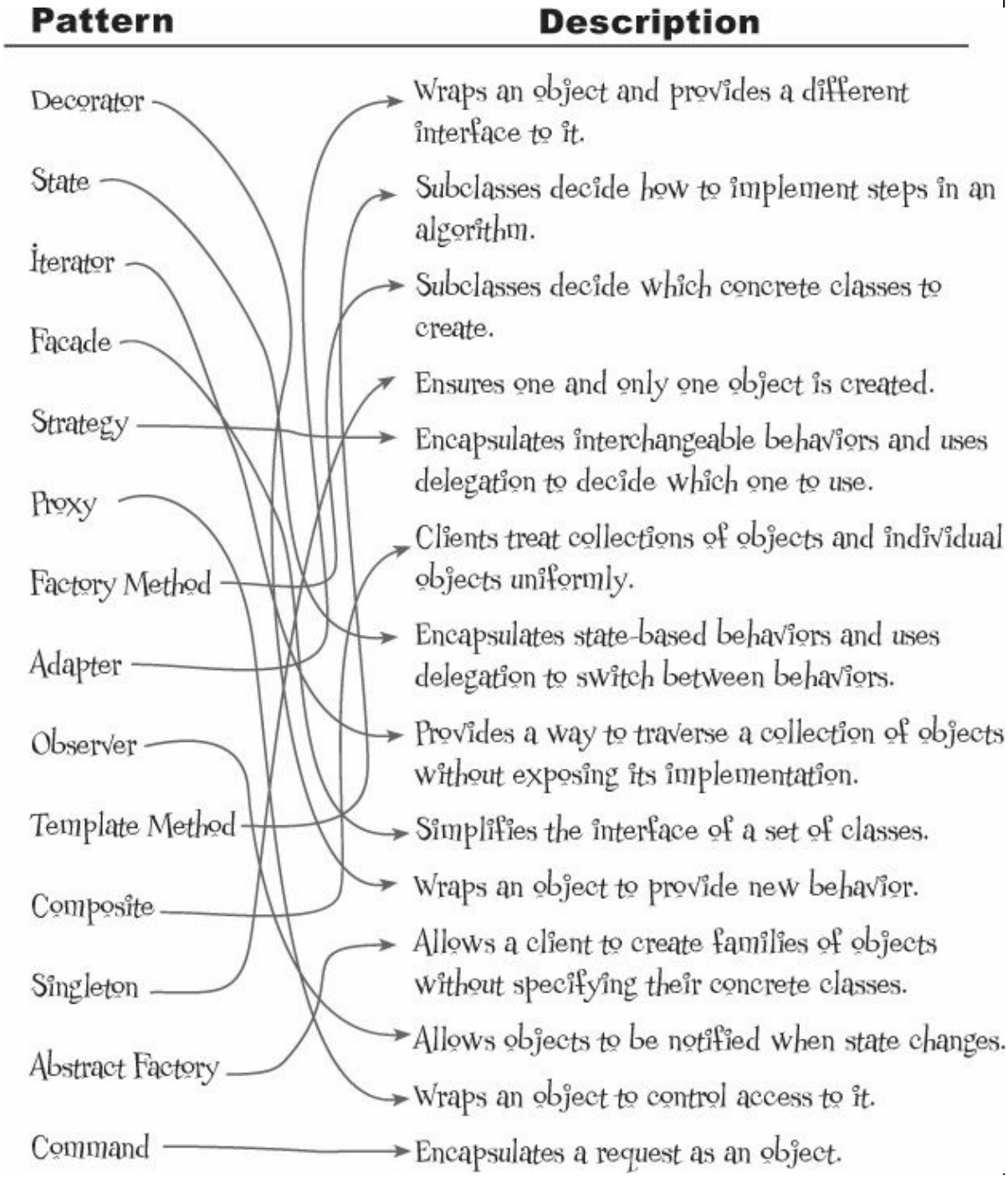
Boy, it’s been great having you in Objectville.

We’re going to miss you, for sure. But don’t worry — before you know it,

the next Head First book will be out and you can visit again. What's the next book, you ask? Hmmm, good question! Why don't you help us decide? Send email to booksuggestions@wickedlysmart.com.

WHO DOES WHAT? SOLUTION

Match each pattern with its description:



Appendix A. Leftover Patterns

Not everyone can be the most popular. A lot has changed in the last 20 years. Since *Design Patterns: Elements of Reusable Object-Oriented Software* first came out, developers have applied these patterns thousands of times. The patterns we summarize in this appendix are full-fledged, card-carrying, official GoF patterns, but aren't used as often as the patterns we've explored so far. But these patterns are awesome in their own right, and if your situation calls for them, you should apply them with your head held high. Our goal in this appendix is to give you a high-level idea of what these patterns are all about.

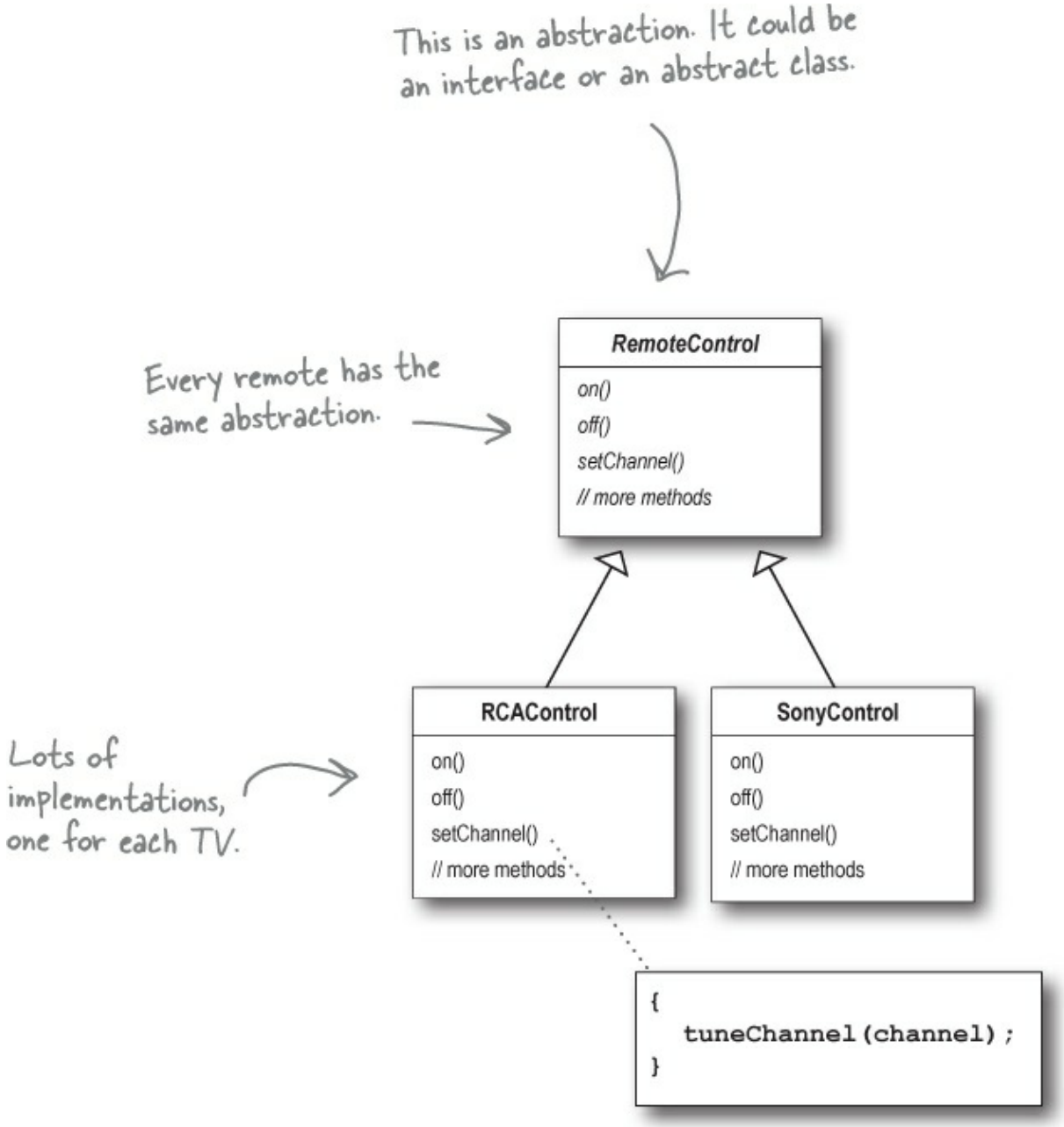
Bridge

Use the Bridge Pattern to vary not only your implementations, but also

your abstractions.

A scenario

Imagine you're going to revolutionize "extreme lounging." You're writing the code for a new ergonomic and user-friendly remote control for TVs. You already know that you've got to use good OO techniques because while the remote is based on the same *abstraction*, there will be lots of *implementations* — one for each model of TV.



Your dilemma

You know that the remote's user interface won't be right the first time. In fact, you expect that the product will be refined many times as usability data is collected on the remote control.

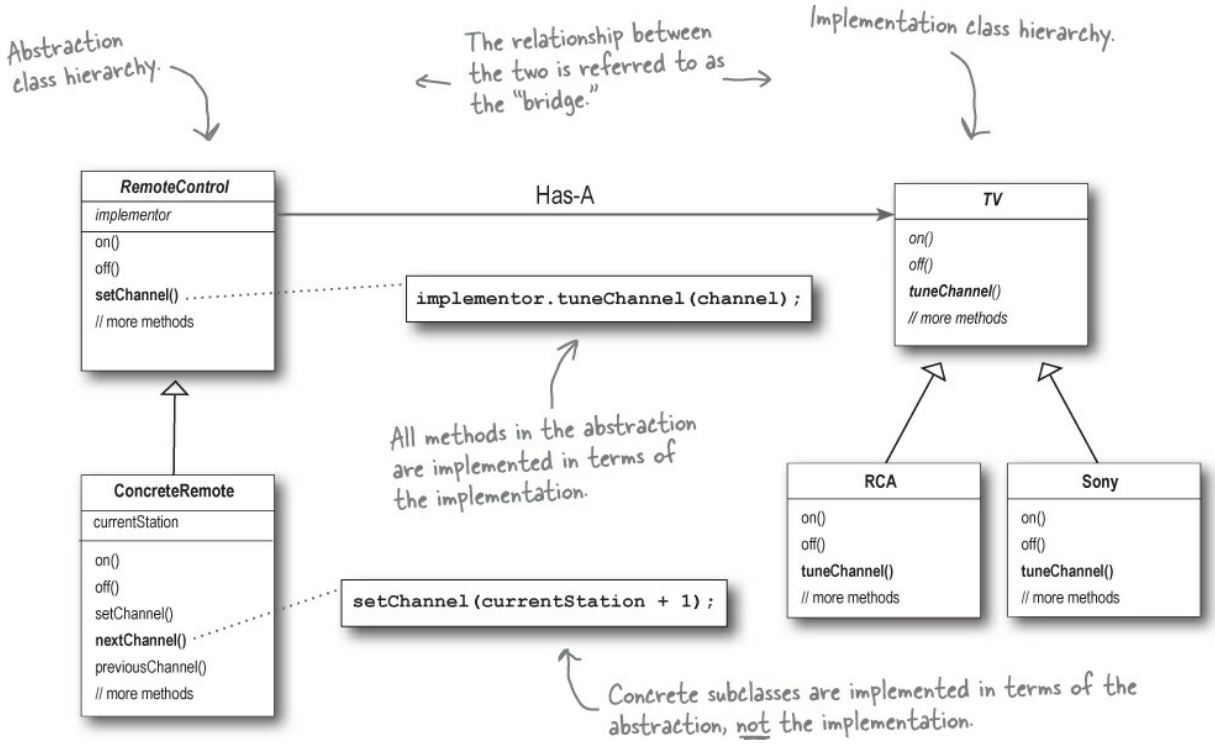
So your dilemma is that the remotes are going to change and the TVs are going to change. You've already *abstracted* the user interface so that you can vary the *implementation* over the many TVs your customers will own. But you are also going to need to *vary the abstraction* because it is going to change over time as the remote is improved based on the user feedback.

> **NOTE**
>
> Using this design we can vary only the TV implementation, not the user interface.

So how are you going to create an OO design that allows you to vary the implementation *and* the abstraction?

Why use the Bridge Pattern?

The Bridge Pattern allows you to vary the implementation *and* the abstraction by placing the two in separate class hierarchies.



Now you have two hierarchies, one for the remotes and a separate one for platform-specific TV implementations. The bridge allows you to vary either side of the two hierarchies independently.

BRIDGE BENEFITS

- Decouples an implementation so that it is not bound permanently to an interface.
- Abstraction and implementation can be extended independently.
- Changes to the concrete abstraction classes don’t affect the client.

BRIDGE USES AND DRAWBACKS

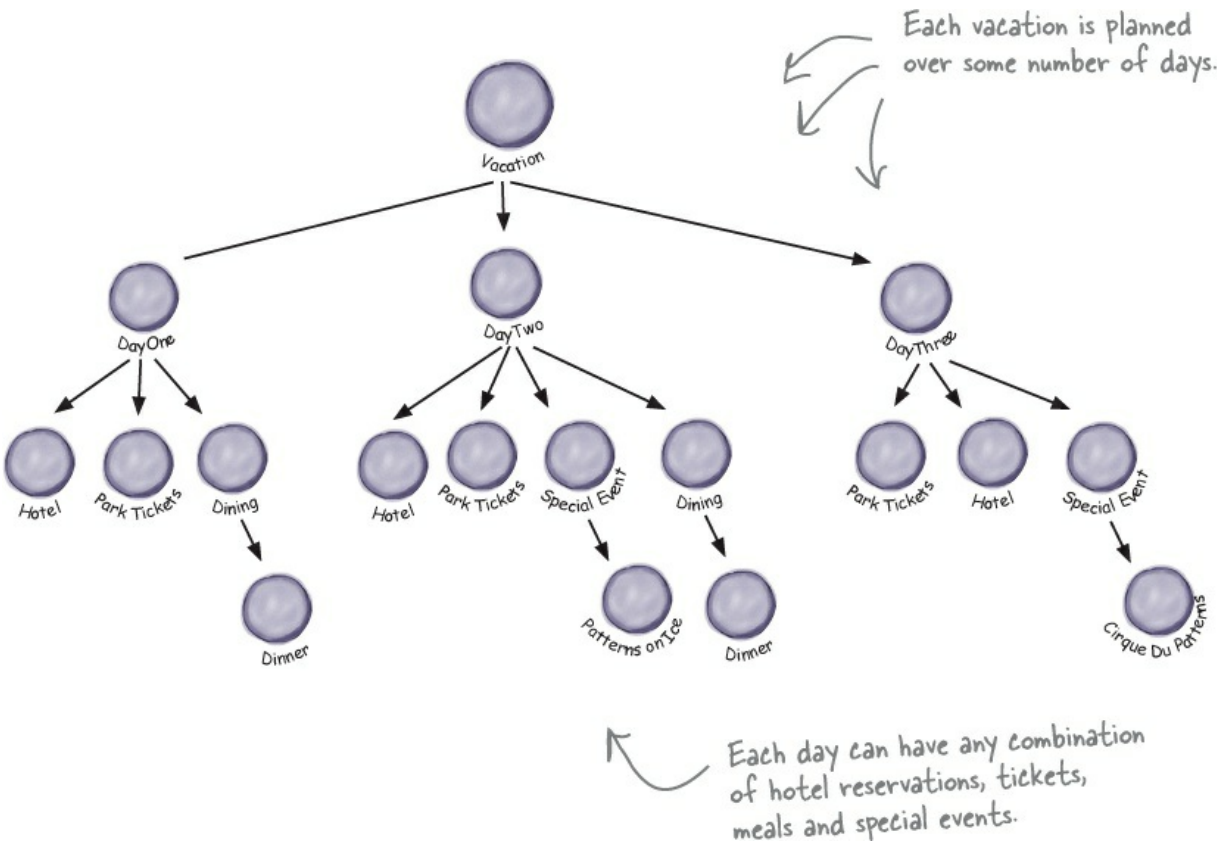
- Useful in graphics and windowing systems that need to run over multiple platforms.
- Useful any time you need to vary an interface and an implementation in different ways.
- Increases complexity.

Builder

Use the Builder Pattern to encapsulate the construction of a product and allow it to be constructed in steps.

A scenario

You’ve just been asked to build a vacation planner for Patternsland, a new theme park just outside of Objectville. Park guests can choose a hotel and various types of admission tickets, make restaurant reservations, and even book special events. To create a vacation planner, you need to be able to create structures like this:



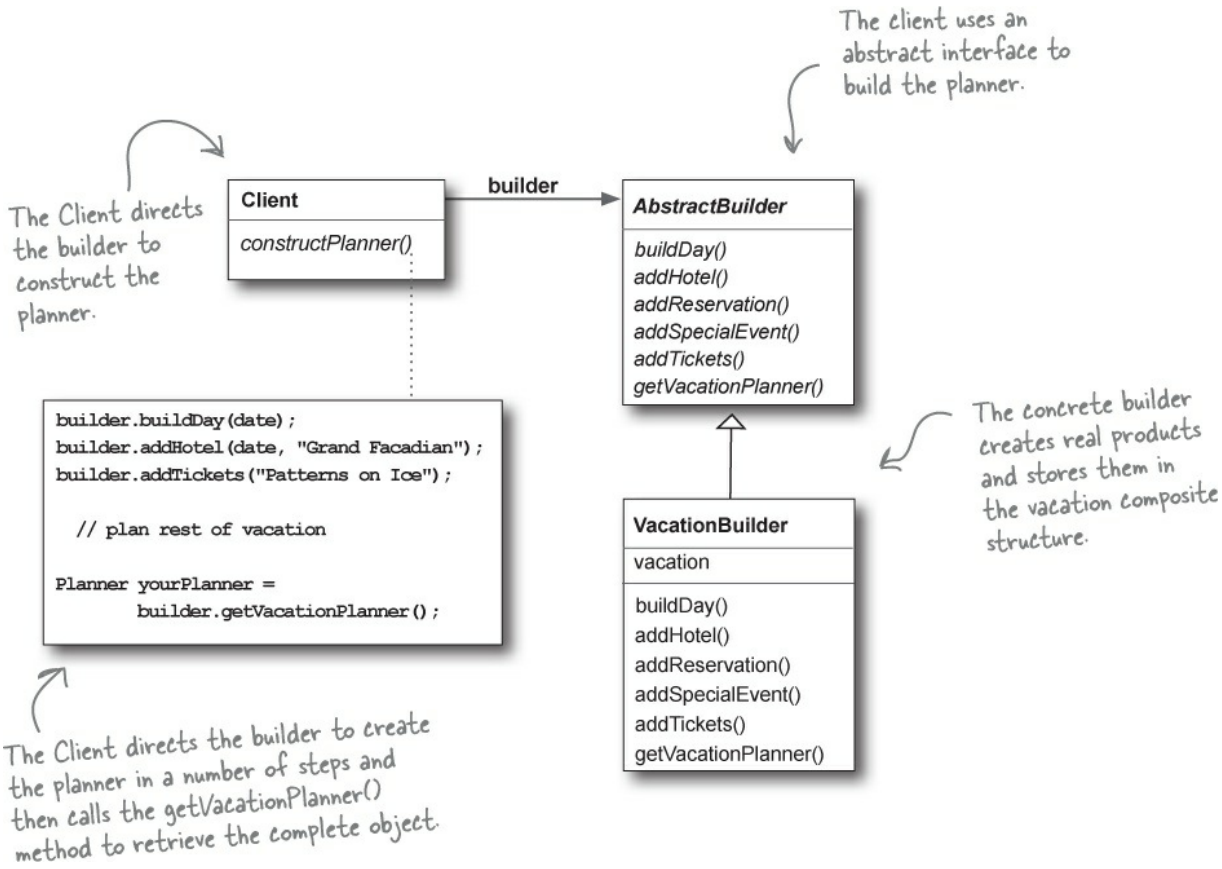
You need a flexible design

Each guest's planner can vary in the number of days and types of activities it includes. For instance, a local resident might not need a hotel, but wants to make dinner and special event reservations. Another guest might be flying into Objectville and needs a hotel, dinner reservations, and admission tickets.

So, you need a flexible data structure that can represent guest planners and all their variations; you also need to follow a sequence of potentially complex steps to create the planner. How can you provide a way to create the complex structure without mixing it with the steps for creating it?

Why use the Builder Pattern?

Remember Iterator? We encapsulated the iteration into a separate object and hid the internal representation of the collection from the client. It's the same idea here: we encapsulate the creation of the trip planner in an object (let's call it a builder), and have our client ask the builder to construct the trip planner structure for it.



BUILDER BENEFITS

- Encapsulates the way a complex object is constructed.
- Allows objects to be constructed in a multistep and varying process (as opposed to one-step factories).
- Hides the internal representation of the product from the client.
- Product implementations can be swapped in and out because the client only sees an abstract interface.

BUILDER USES AND DRAWBACKS

- Often used for building composite structures.
- Constructing objects requires more domain knowledge of the client than when using a Factory.

Chain of Responsibility

Use the Chain of Responsibility Pattern when you want to give more than one object a chance to handle a request.

A scenario

Mighty Gumball has been getting more email than they can handle since the release of the Java-powered Gumball Machine. From their own analysis they get four kinds of email: fan mail from customers that love the new 1-in-10 game, complaints from parents whose kids are addicted to the game, and requests to put machines in new locations. They also get a fair amount of spam.

All fan mail should go straight to the CEO, all complaints should go to the legal department and all requests for new machines should go to business development. Spam should be deleted.

Your task

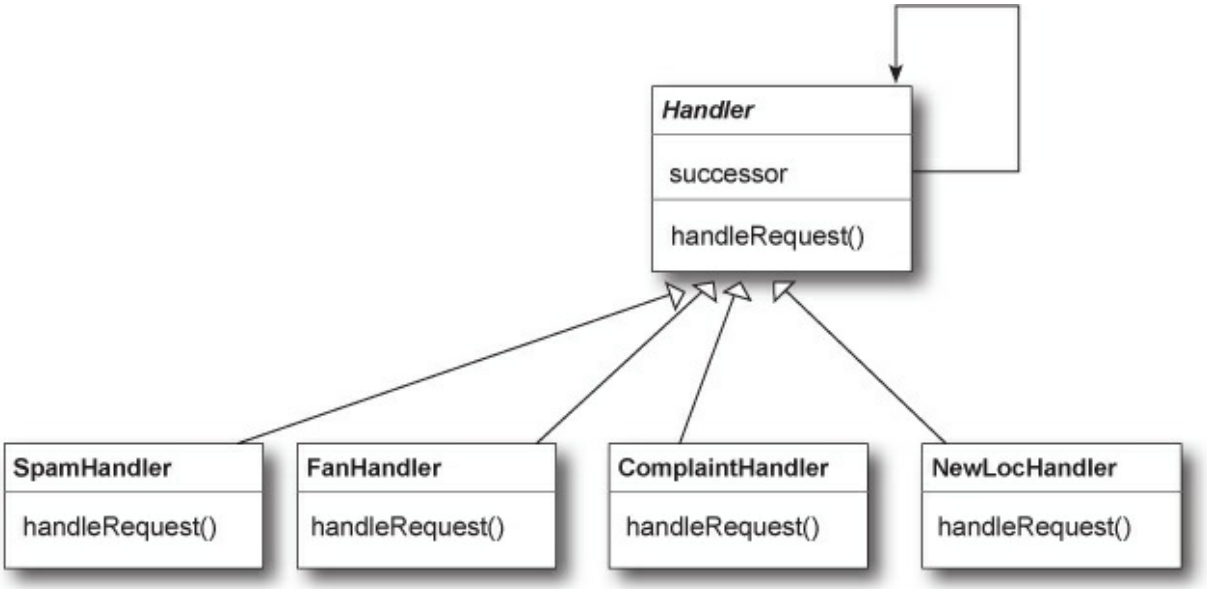
Mighty Gumball has already written some AI detectors that can tell if an email is spam, fan mail, a complaint, or a request, but they need you to create a design that can use the detectors to handle incoming email.

How to use the Chain of Responsibility Pattern

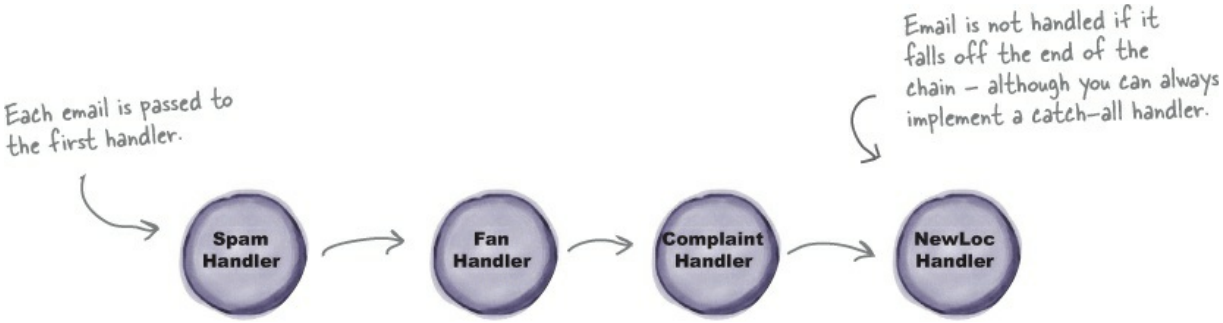
With the Chain of Responsibility Pattern, you create a chain of objects to examine requests. Each object in turn examines a request and either handles it, or passes it on to the next object in the chain.

NOTE

Each object in the chain acts as a handler and has a successor object. If it can handle the request, it does; otherwise, it forwards the request to its successor.



As email is received, it is passed to the first handler: the SpamHandler. If the SpamHandler can't handle the request, it is passed on to the FanHandler. And so on...



CHAIN OF RESPONSIBILITY BENEFITS

- Decouples the sender of the request and its receivers.
- Simplifies your object because it doesn't have to know the chain's structure and keep direct references to its members.
- Allows you to add or remove responsibilities dynamically by changing the members or order of the chain.

CHAIN OF RESPONSIBILITY USES AND DRAWBACKS

- Commonly used in windows systems to handle events like mouse clicks and keyboard events.
- Execution of the request isn't guaranteed; it may fall off the end of the chain if no

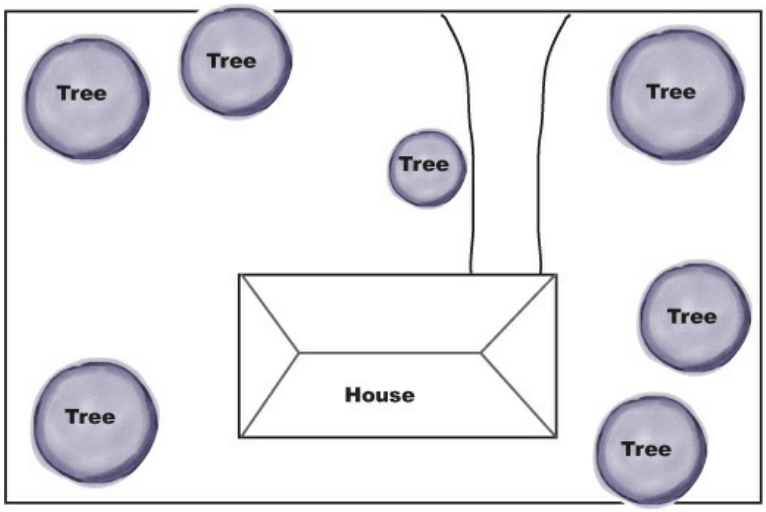
object handles it (this can be an advantage or a disadvantage).
- Can be hard to observe and debug at runtime.

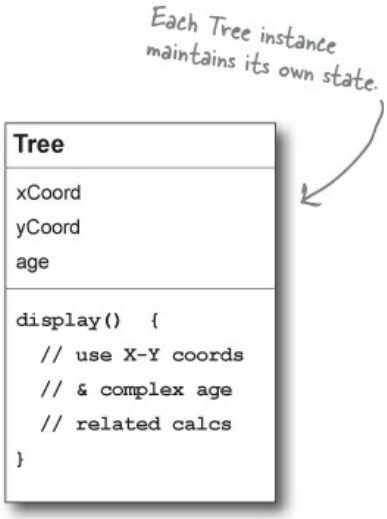
Flyweight

Use the Flyweight Pattern when one instance of a class can be used to provide many "virtual instances."

A scenario

You want to add trees as objects in your hot new landscape design application. In your application, trees don't really do very much; they have an X-Y location, and they can draw themselves dynamically, depending on how old they are. The thing is, a user might want to have lots and lots of trees in one of their home landscape designs. It might look something like this:





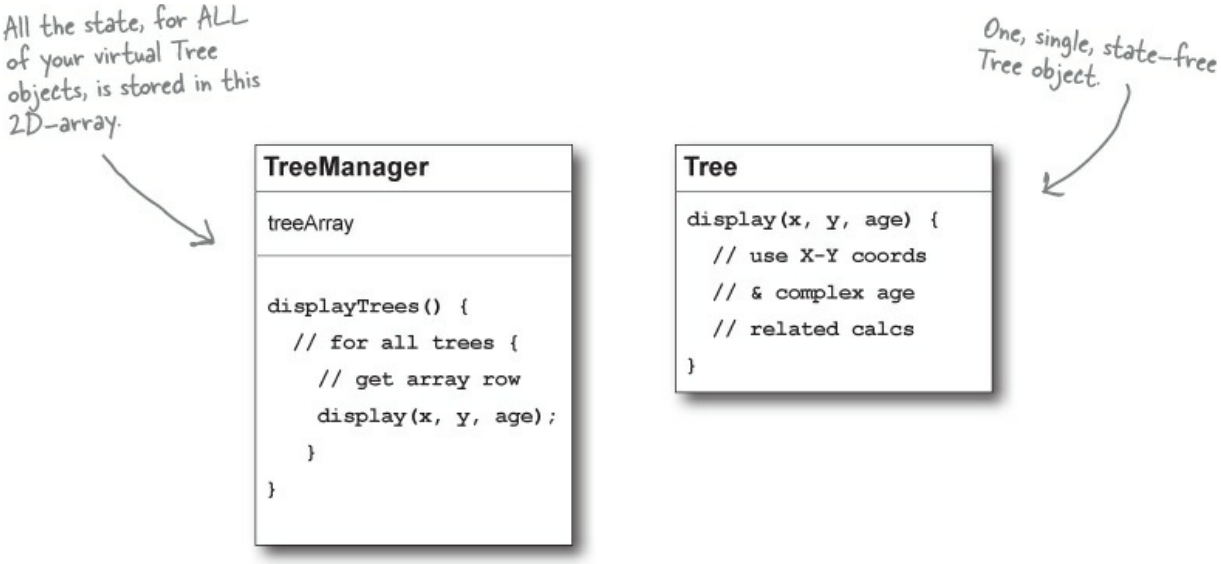
Your big client's dilemma

You've just landed your "reference account." That key client you've been pitching for months. They're going to buy 1,000 seats of your application, and they're using your software to do the landscape design for huge planned communities. After using your software for a week, your client is

complaining that when they create large groves of trees, the app starts getting sluggish...

Why use the Flyweight Pattern?

What if, instead of having thousands of Tree objects, you could redesign your system so that you've got only one instance of Tree, and a client object that maintains the state of ALL your trees? That's the Flyweight!



FLYWEIGHT BENEFITS

- Reduces the number of object instances at runtime, saving memory.
- Centralizes state for many "virtual" objects into a single location.

FLYWEIGHT USES AND DRAWBACKS

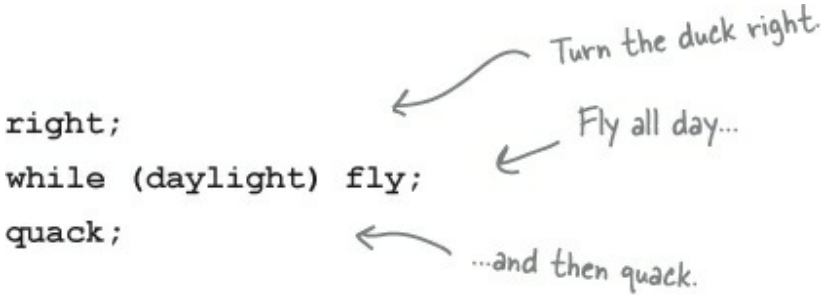
- The Flyweight is used when a class has many instances, and they can all be controlled identically.
- A drawback of the Flyweight pattern is that once you've implemented it, single, logical instances of the class will not be able to behave independently from the other instances.

Interpreter

Use the Interpreter Pattern to build an interpreter for a language.

A scenario

Remember the Duck Simulator? You have a hunch it would also make a great educational tool for children to learn programming. Using the simulator, each child gets to control one duck with a simple language. Here's an example of the language:

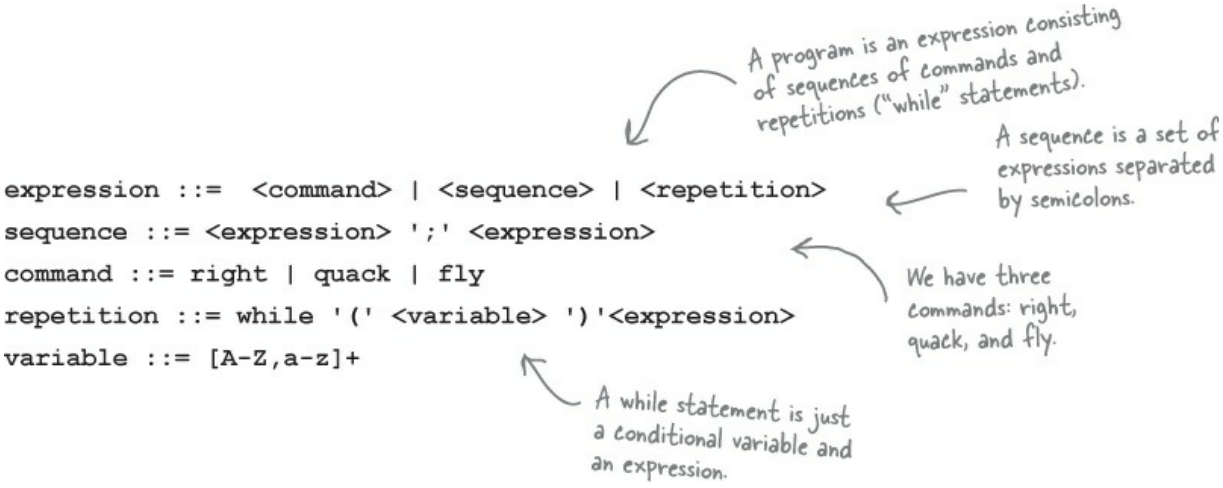


RELAX

The Interpreter Pattern requires some knowledge of formal grammars.

If you've never studied formal grammars, go ahead and read through the pattern; you'll still get the gist of it.

Now, remembering how to create grammars from one of your old introductory programming classes, you write out the grammar:

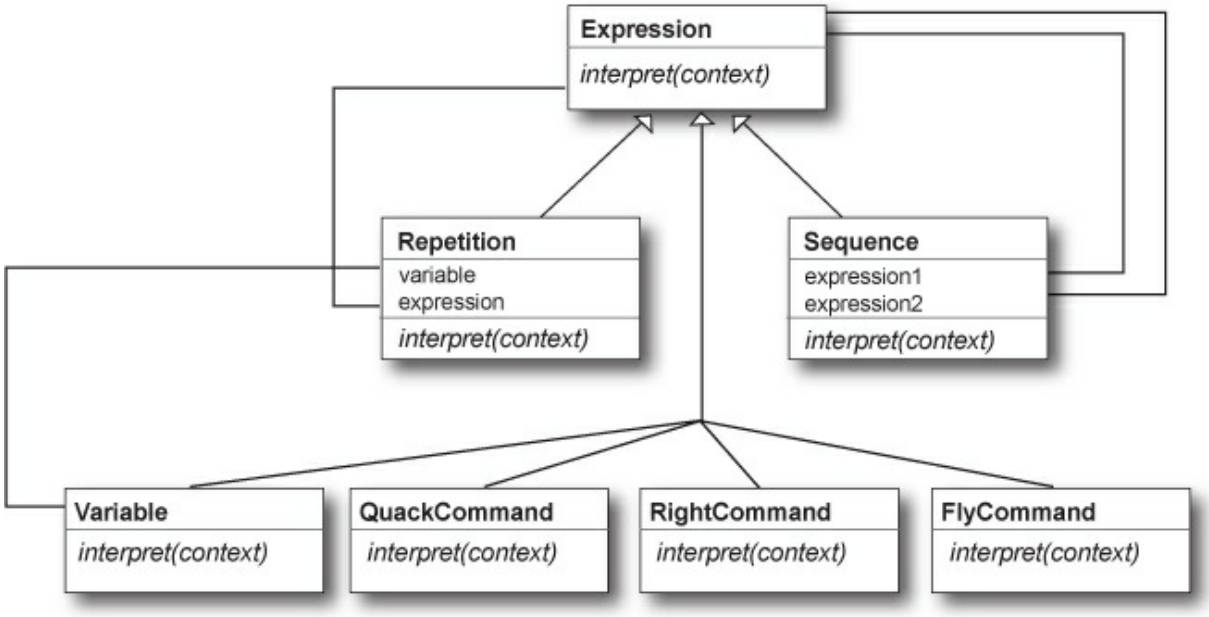


Now what?

You've got a grammar; now all you need is a way to represent and interpret sentences in the grammar so that the students can see the effects of their programming on the simulated ducks.

How to implement an interpreter

When you need to implement a simple language, the Interpreter Pattern defines a class-based representation for its grammar along with an interpreter to interpret its sentences. To represent the language, you use a class to represent each rule in the language. Here's the duck language translated into classes. Notice the direct mapping to the grammar.



To interpret the language, call the interpret() method on each expression type. This method is passed a context — which contains the input stream of the program we're parsing — and matches the input and evaluates it.

INTERPRETER BENEFITS

- Representing each grammar rule in a class makes the language easy to implement.
- Because the grammar is represented by classes, you can easily change or extend the language.
- By adding methods to the class structure, you can add new behaviors beyond interpretation, like pretty printing and more sophisticated program validation.

INTERPRETER USES AND DRAWBACKS

- Use interpreter when you need to implement a simple language.
- Appropriate when you have a simple grammar and simplicity is more important than efficiency.

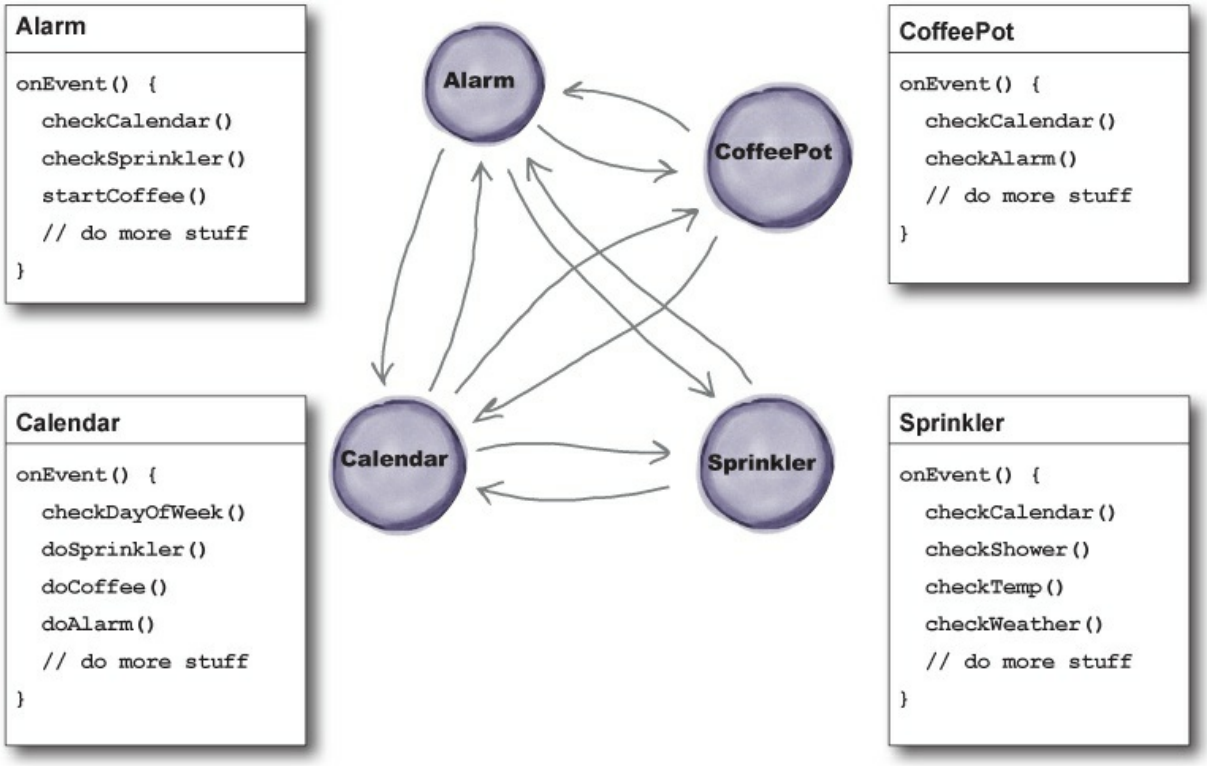
- Used for scripting and programming languages.
- This pattern can become cumbersome when the number of grammar rules is large. In these cases a parser/compiler generator may be more appropriate.

Mediator

Use the Mediator Pattern to centralize complex communications and control between related objects.

A scenario

Bob has a Java-enabled auto-house, thanks to the good folks at HouseOfTheFuture. All of his appliances are designed to make his life easier. When Bob stops hitting the snooze button, his alarm clock tells the coffee maker to start brewing. Even though life is good for Bob, he and other clients are always asking for lots of new features: No coffee on the weekends... Turn off the sprinkler 15 minutes before a shower is scheduled... Set the alarm early on trash days...



HouseOfTheFuture's dilemma

It's getting really hard to keep track of which rules reside in which objects,

and how the various objects should relate to each other.

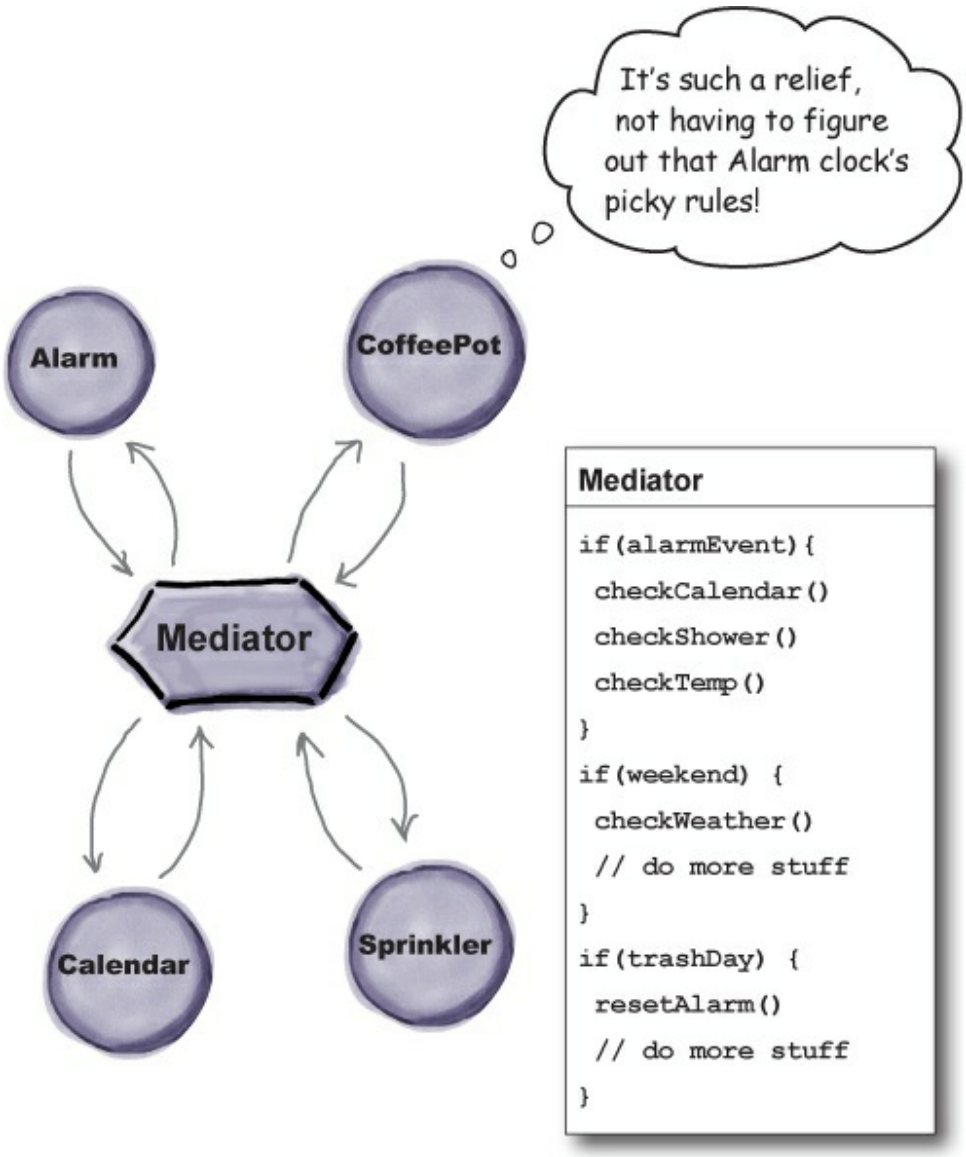
Mediator in action...

With a Mediator added to the system, all of the appliance objects can be greatly simplified:

- They tell the Mediator when their state changes.
- They respond to requests from the Mediator.

Before we added the Mediator, all of the appliance objects needed to know about each other... they were all tightly coupled. With the Mediator in place, the appliance objects are all completely decoupled from each other.

The Mediator contains all of the control logic for the entire system. When an existing appliance needs a new rule, or a new appliance is added to the system, you'll know that all of the necessary logic will be added to the Mediator.



MEDIATOR BENEFITS

- Increases the reusability of the objects supported by the Mediator by decoupling them from the system.
- Simplifies maintenance of the system by centralizing control logic.
- Simplifies and reduces the variety of messages sent between objects in the system.

MEDIATOR USES AND DRAWBACKS

- The Mediator is commonly used to coordinate related GUI components.
- A drawback of the Mediator Pattern is that without proper design, the Mediator

object itself can become overly complex.

Memento

Use the Memento Pattern when you need to be able to return an object to one of its previous states; for instance, if your user requests an "undo."

A scenario

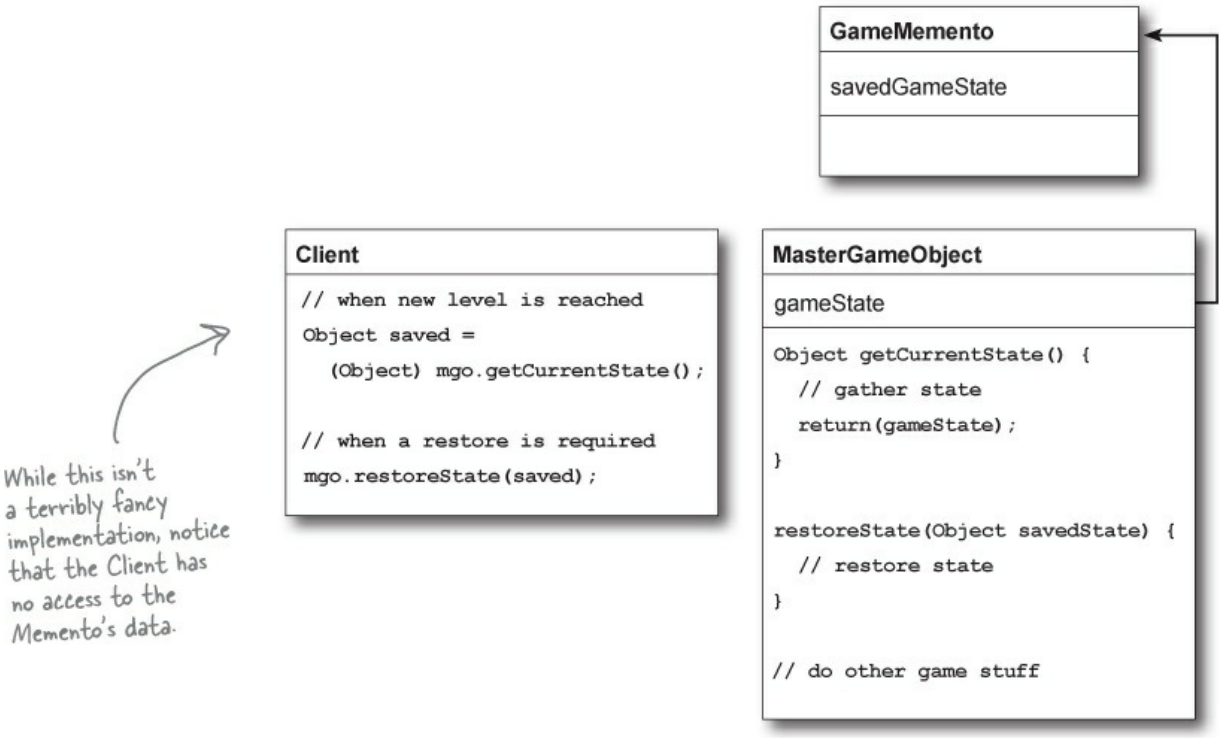
Your interactive role playing game is hugely successful, and has created a legion of addicts, all trying to get to the fabled "level 13." As users progress to more challenging game levels, the odds of encountering a game-ending situation increase. Fans who have spent days progressing to an advanced level are understandably miffed when their character gets snuffed, and they have to start all over. The cry goes out for a "save progress" command, so that players can store their game progress and at least recover most of their efforts when their character is unfairly extinguished. The "save progress" function needs to be designed to return a resurrected player to the last level she completed successfully.

The Memento at work

The Memento has two goals:

- Saving the important state of a system's key object.
- Maintaining the key object's encapsulation.

Keeping the single responsibility principle in mind, it's also a good idea to keep the state that you're saving separate from the key object. This separate object that holds the state is known as the Memento object.



MEMENTO BENEFITS

- Keeping the saved state external from the key object helps to maintain cohesion.
- Keeps the key object's data encapsulated.
- Provides easy-to-implement recovery capability.

MEMENTO USES AND DRAWBACKS

- The Memento is used to save state.
- A drawback to using Memento is that saving and restoring state can be time consuming.
- In Java systems, consider using Serialization to save a system's state.

Prototype

Use the Prototype Pattern when creating an instance of a given class is either expensive or complicated.

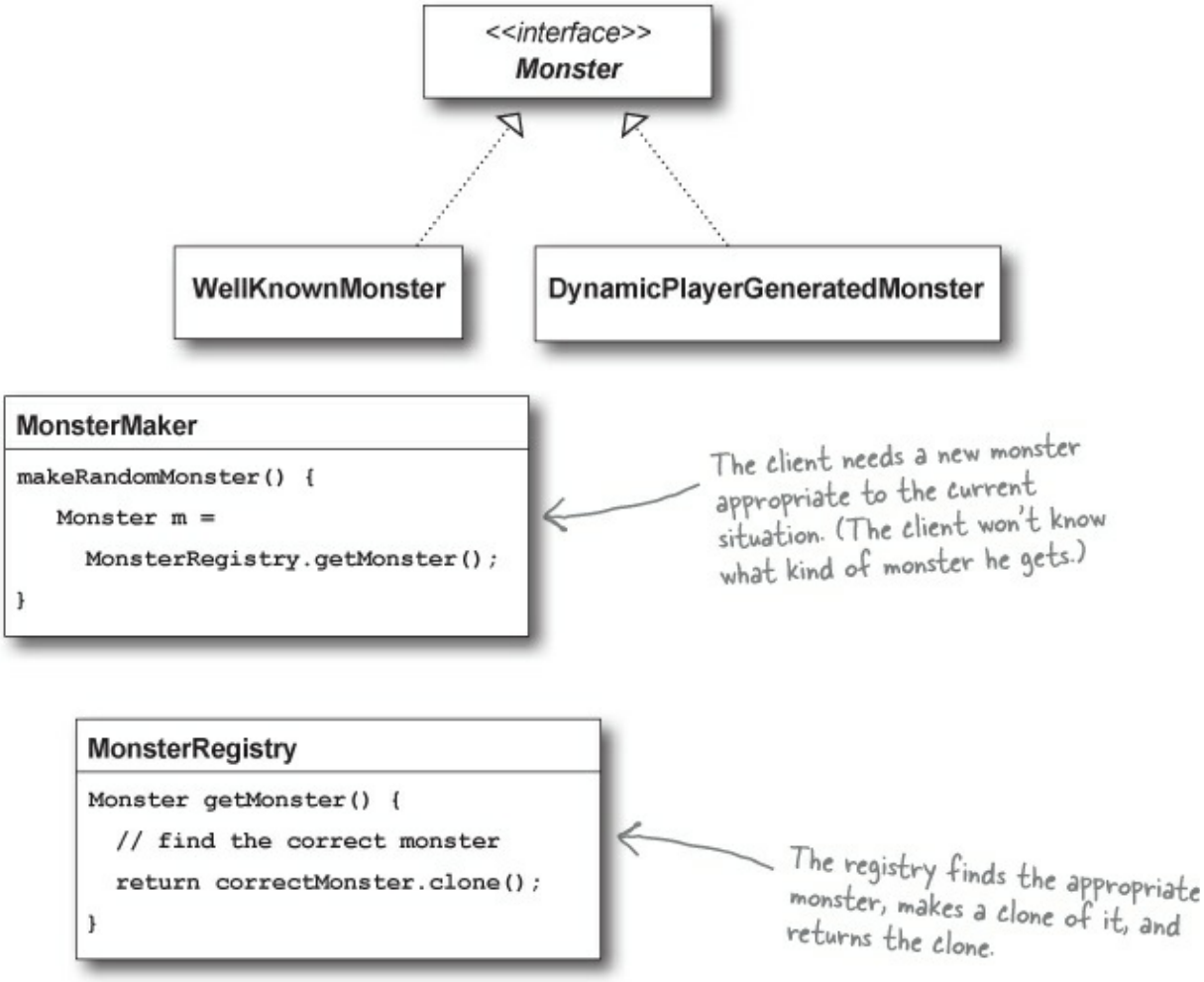
A scenario

Your interactive role playing game has an insatiable appetite for monsters. As

your heroes make their journey through a dynamically created landscape, they encounter an endless chain of foes that must be subdued. You'd like the monster's characteristics to evolve with the changing landscape. It doesn't make a lot of sense for bird-like monsters to follow your characters into underseas realms. Finally, you'd like to allow advanced players to create their own custom monsters.

Prototype to the rescue

The Prototype Pattern allows you to make new instances by copying existing instances. (In Java this typically means using the clone() method, or de-serialization when you need deep copies.) A key aspect of this pattern is that the client code can make new instances without knowing which specific class is being instantiated.



PROTOTYPE BENEFITS

- Hides the complexities of making new instances from the client.
- Provides the option for the client to generate objects whose type is not known.
- In some circumstances, copying an object can be more efficient than creating a new object.

PROTOTYPE USES AND DRAWBACKS

- Prototype should be considered when a system must create new objects of many types in a complex class hierarchy.
- A drawback to using the Prototype is that making a copy of an object can sometimes be complicated.

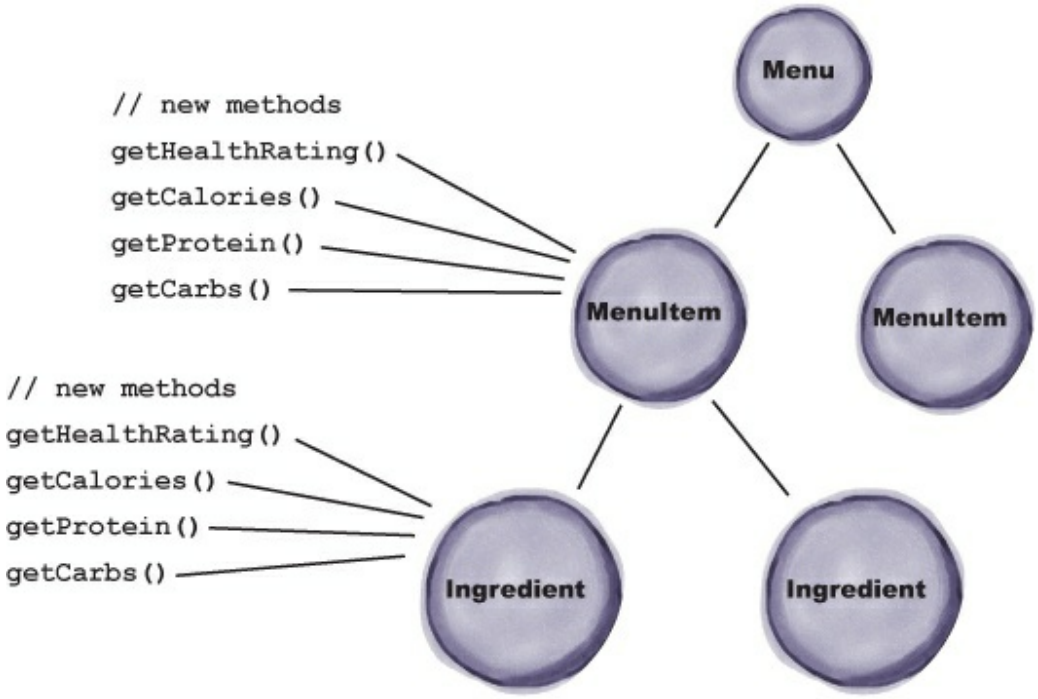
Visitor

Use the Visitor Pattern when you want to add capabilities to a composite of objects and encapsulation is not important.

A scenario

Customers who frequent the Objectville Diner and Objectville Pancake House have recently become more health conscious. They are asking for nutritional information before ordering their meals. Because both establishments are so willing to create special orders, some customers are even asking for nutritional information on a per ingredient basis.

Lou's proposed solution:

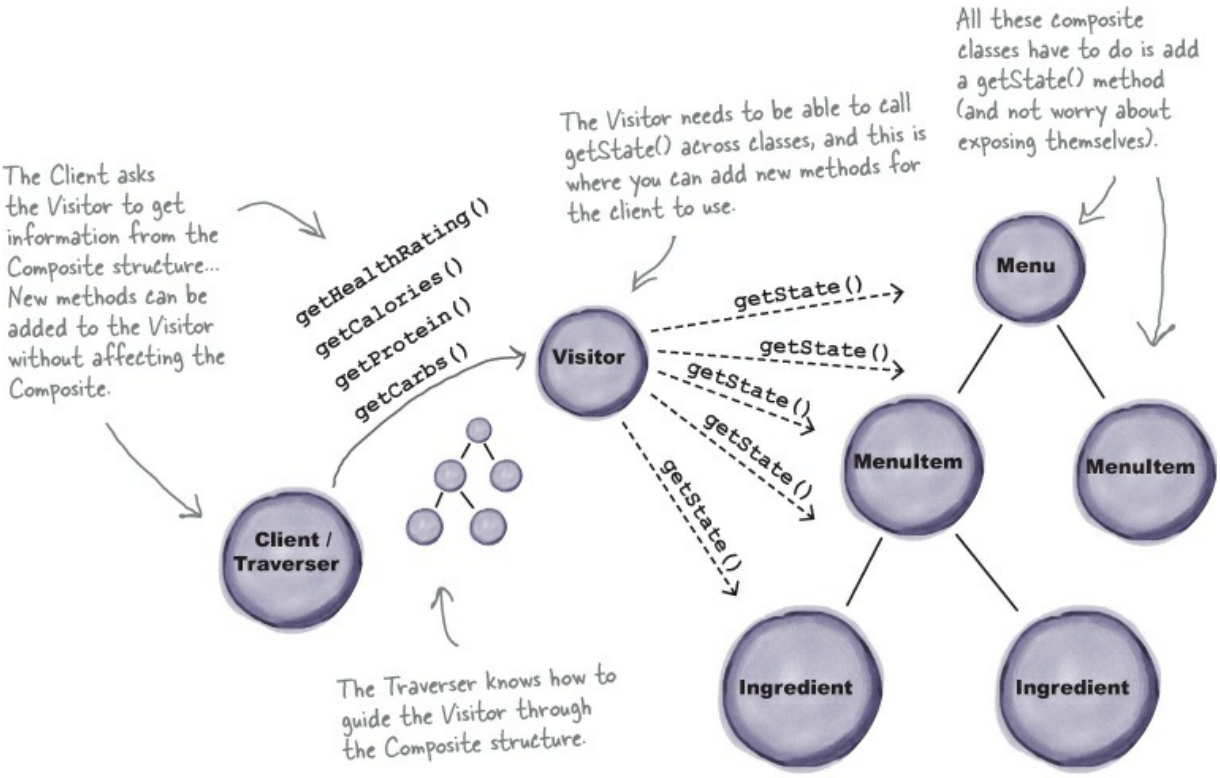


Mel's concerns...

"Boy, it seems like we're opening Pandora's box. Who knows what new method we're going to have to add next, and every time we add a new method we have to do it in two places. Plus, what if we want to enhance the base application with, say, a recipes class? Then we'll have to make these changes in three different places..."

The Visitor drops by

The Visitor works hand in hand with a Traverser. The Traverser knows how to navigate to all of the objects in a Composite. The Traverser guides the Visitor through the Composite so that the Visitor can collect state as it goes. Once state has been gathered, the Client can have the Visitor perform various operations on the state. When new functionality is required, only the Visitor must be enhanced.



VISITOR BENEFITS

- Allows you to add operations to a Composite structure without changing the structure itself.
- Adding new operations is relatively easy.
- The code for operations performed by the Visitor is centralized.

VISITOR DRAWBACKS

- The Composite classes' encapsulation is broken when the Visitor is used.
- Because the traversal function is involved, changes to the Composite structure are more difficult.

Appendix B.

Head First Institute

And now, a final word from the Head First Institute...

Our world class researchers are working day and night in a mad race to uncover the mysteries of Life, the Universe and Everything–before it's too late. Never before has a research team with such noble and daunting goals been assembled. Currently, we are focusing our collective energy and brain power on creating the ultimate learning machine. Once perfected, you and others will join us in our quest!

You're fortunate to be holding one of our first prototypes in your hands. But only through constant refinement can our goal be achieved. We ask you, a pioneer user of the technology, to send us periodic field reports of your progress, at fieldreports@wickedlysmart.com

Appendix C. Mighty Gumball

Without your help the next generation may never know the joys of the gumball machine. Today, inflexible, poorly designed code is putting our Java-powered machines at risk. Mighty Gumball won't let that happen. We're devoting ourselves to helping you improve your Java and OO design skills so that you can help us build the next generation of Mighty Gumball machines.

Come on, Java toasters are *sooo* ‘90s, visit us at *http://www.wickedlysmart.com*.

Index

A NOTE ON THE DIGITAL INDEX

A link in an index entry is displayed as the section title in which that entry appears. Because some sections have multiple index markers, it is not unusual for an entry to have several links to the same section. Clicking on any link will take you directly to the place in the text in which the marker appears.

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

About the Authors

Eric Freeman recently ended nearly a decade as a media company executive, having held the position of CTO of Disney Online & Disney.com at The Walt Disney Company. Eric is now devoting his time to WickedlySmart.com and lives with his wife and young daughter in Austin, TX. He holds a Ph.D. in Computer Science from Yale University.

Elisabeth Robson is co-founder of Wickedly Smart, an education company devoted to helping customers gain mastery in web technologies. She's co-author of four bestselling books, Head First Design Patterns, Head First HTML and CSS, Head First HTML5 Programming, and Head First JavaScript Programming.

Bert Bates is a 20-year software developer, a Java instructor, and a co-developer of Sun's upcoming EJB exam (Sun Certified Business Component Developer). His background features a long stint in artificial intelligence, with clients like the Weather Channel, A&E Network, Rockwell, and Timken.

Kathy Sierra has been interested in learning theory since her days as a game developer (Virgin, MGM, Amblin'). More recently, she's been a master trainer for Sun Microsystems, teaching Sun's Java instructors how to teach the latest technologies to customers, and a lead developer of several Sun certification exams. Along with her partner Bert Bates, Kathy created the Head First series. She's also the original founder of the Software Development/Jolt Productivity Award-winning javaranch.com, the largest (and friendliest) all-volunteer Java community.

Colophon

All interior layouts were designed by Eric Freeman, Elisabeth Robson, Kathy Sierra and Bert Bates. Kathy and Bert created the look & feel of the Head First series. The book was produced using Adobe InDesign CS (an unbelievably cool design tool that we can't get enough of) and Adobe Photoshop CS. The book was typeset using Uncle Stinky, Mister Frisky (you think we're kidding), Ann Satellite, Baskerville, Comic Sans, Myriad Pro, Skippy Sharp, Savoye LET, Jokerman LET, Courier New and Woodrow typefaces.

Interior design and production all happened exclusively on Apple Macintoshes — at Head First we're all about "Think Different" (even if it isn't grammatical). All Java code was created using James Gosling's favorite IDE, *vi*, Erich Gamma's Eclipse.

Long days of writing were powered by the caffeine fuel of Honest Tea and Tejava, the clean Santa Fe air, and the grooving sounds of Banco de Gaia, Cocteau Twins, Buddha Bar I-VI, Delerium, Enigma, Mike Oldfield, Olive, Orb, Orbital, LTJ Bukem, Massive Attack, Steve Roach, Sasha and Digweed, Thievery Corporation, Zero 7 and Neil Finn (in all his incarnations) along

with a heck of a lot of acid trance and more 80s music than you’d care to know about.

Head First: Design Patterns

Eric Freeman

Elisabeth Robson

Bert Bates

Kathy Sierra

Editor
Mike Hendrickson

Editor
Mike Loukides

Head First Design Patterns

by Eric Freeman, Elisabeth Robson, Kathy Sierra, and Bert Bates

Editors:	Mike Hendrickson, Mike Loukides
Cover Designer:	Ellie Volckhausen
Pattern Wranglers:	Eric Freeman, Elisabeth Freeman
Facade Decoration:	Elisabeth Robson
Strategy:	Kathy Sierra and Bert Bates
Observer:	Oliver

Printing History:

July 2014: Second release.

October 2004: First release.

In other words, if you use anything in *Head First Design Patterns* to, say, run a nuclear power plant, you’re on your own. We do, however, encourage you to use the DJ View app.

No ducks were harmed in the making of this book.

The original GoF agreed to have their photos in this book. Yes, they really are that good-looking.

[LSI]
06-30]

O’Reilly Media
1005 Gravenstein Highway North
Sebastopol, CA 95472

2017-09-13T12:54:18-07:00