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 [3]), 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");
    }
}
```

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

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

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...

Here's our main method to get everything going.

```
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();
    }
}
```

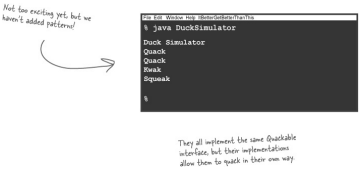
We create a simulator and then call its simulate() method.

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.



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!

QUACKOLOGY

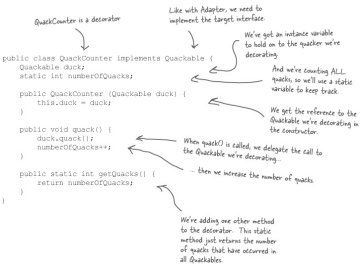
Quackologists are fascinated by all aspects of Quackable behavior. One thing Quackologists have always wanted to study is the total number of quacks made by a flock of ducks.

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.

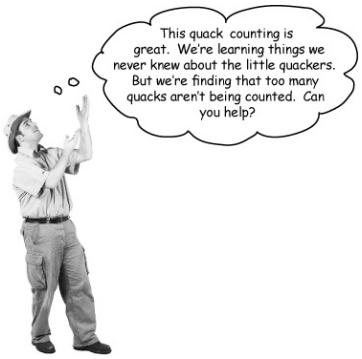



```
public class QuackCounter implements Quackable {
    Quackable duck;
    static int numberOfQuacks;

    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();
    }
}
```



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:

```
public abstract class AbstractDuckFactory {

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

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

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:

```
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();
    }
}
```

DuckFactory extends the abstract factory.

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:

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

CountingDuckFactory also extends the abstract factory.

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();
    }
}
```

First we create the factory that we're going to pass into the simulate() method

The simulate() method takes an AbstractDuckFactory and uses it to create ducks rather than instantiating them directly.

Nothing changes here! Same ol' code.

NOTE

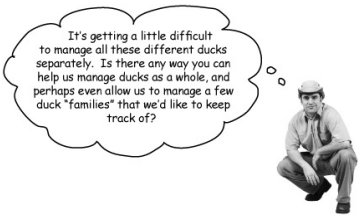
Here's the output using the factory...

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?

This isn't very manageable!

```
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.

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

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

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

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

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 iterator = quackers.iterator();
    while (iterator.hasNext()) {
        Quackable quacker = (Quackable)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 add 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...

```
File Edit Window Help FlockADuck
% java DuckSimulator
Duck Simulator: With Composite - Flocks
Duck Simulator: Whole Flock Simulation
Quack
Kwak
Squeak
Honk
Quack
Quack
Quack
Quack

Duck Simulator: Mallard Flock Simulation
Quack
Quack
Quack
Quack

The ducks quacked 11 times
```

Here's the first flock.

And now the mallards.

The data looks good (remember the goose doesn't get counted).

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.

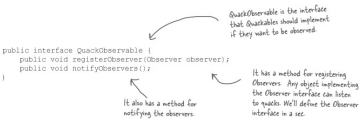
As always, there are trade-offs when you do OO design and you need to consider them as you create your own composites.

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.

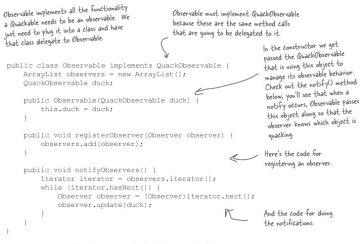


Now we need to make sure all Quackables implement this interface...

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

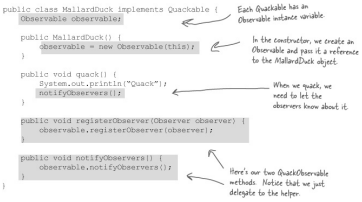
We could approach this by implementing registration and notification in each and every class (like we did in Chapter 2). But we're going to do it a little differently this time: we're going to encapsulate the registration and notification code in another class, call it Observable, and compose it with 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 start with the Observable helper class...



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.

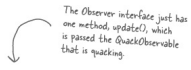


SHARPEN YOUR PENCIL

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:

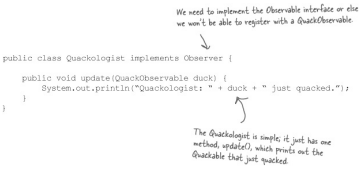
17. **We're almost there! We just need to work on the Observer side of the pattern.**

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




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

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



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!

```
File  Edit  Window  Help  DucksAreEverywhere

% java DuckSimulator
Duck Simulator: With Observer
Quack
Quackologist: Redhead Duck just quacked.
Kwak
Quackologist: Duck Call just quacked.
Squeak
Quackologist: Rubber Duck just quacked.
Honk
Quackologist: Goose pretending to be a Duck just quacked.
Quack
Quackologist: Mallard Duck just quacked.
Quack
Quackologist: Mallard Duck just quacked.
Quack
Quackologist: Mallard Duck just quacked.
Quack
Quackologist: Mallard Duck just quacked.
The Ducks quacked 7 times.

%
```

After each quack, no matter what kind of quack it was, the observer gets a notification.

And the quackologist still gets his counts

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 DuckSimulator 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

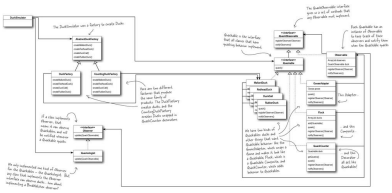
back a decorated duck. (And don't forget, they can also use another duck factory if they want an un-decorated duck!)

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 ~~Birds~~ 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:

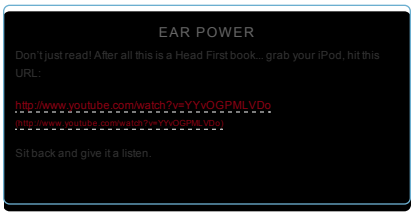


The King of Compound Patterns

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	You can model a throttle and a manifold
Lyrics and music by James Dempsey.	Model the toddle of a two year old Model a bottle of fine Chardonnay
MVC's a paradigm for factoring your code into functional segments, so your brain does not explode.	Model all the glottal stops people say Model the coddling of boiling eggs
To achieve reusability, you gotta keep those boundaries clean	You can model the waddle in Hexley's legs
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
Model View, it's got three layers like Oreos do Model View Controller	Keep the coupling loose and so achieve a massive level of reuse
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're probably wondering now
You create custom classes, in your app's problem domain	You're probably wondering how
you can choose to reuse them with all the views but the model objects stay the same.	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	I sent a TextField StringValue.
Model View, mad props to the smalltalk crew! Model View Controller	Model View How we gonna deep six all that glue Model View Controller

Model View, it's pronounced Oh Oh not Ooo Ooo	
Model View Controller	Controllers know the Model and View very intimately They often use hardcoding which can be foreboding for reusability
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	But now you can connect each model key that you select to any view property
	And once you start binding I think you'll be finding less code in your source tree
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	Yeah I know I was elated by the stuff they've automated and the things you get for free And I think it bears repeating all the code you won't be needing when you hook it up to JGB Using Swing
And I don't mean to be vicious But it gets repetitious Doing all the things controllers do	Model View, even handles multiple selections too Model View Controller
And I wish I had a dime For every single time	Model View, bet I ship my application before you Model View Controller



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!*

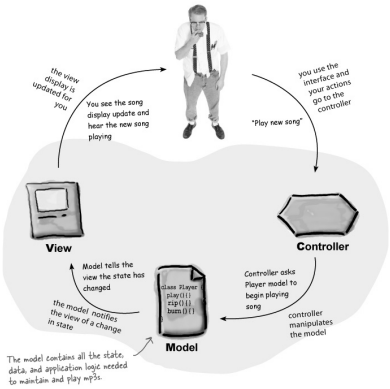
That's right, patterns are the key to MVC. Learning MVC from the top down is difficult; not many developers succeed. Here's the secret to learning MVC: *it's just a few patterns put together*. When you approach learning MVC by looking at the patterns, all of the sudden it starts to make sense.

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.

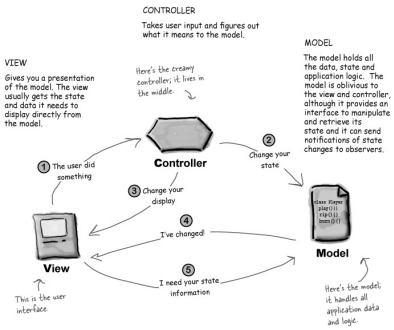
Well, underneath it all sits the Model-View-Controller...



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", the controller 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?"*

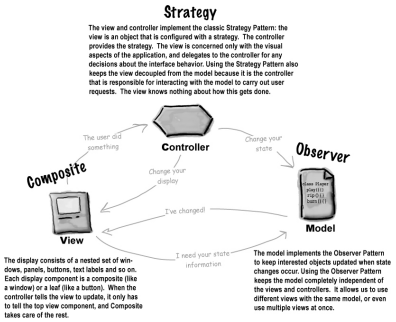
You could; however, you don't want to for two reasons: First, you'll complicate your view code because it now has two responsibilities: managing the user interface and dealing with logic of how to control the model. Second, you're tightly coupling your view to the model. If you want to reuse the view 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

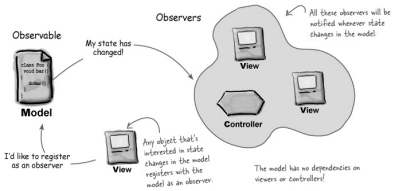
We've already told you the best path to learning the MVC is to see it for what it is: a set of patterns working together in the same design.

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.

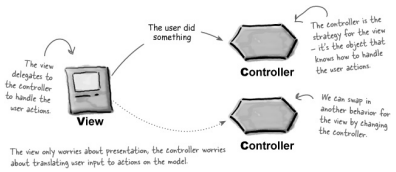
Let's take a closer look:



Observer



Strategy



Composite

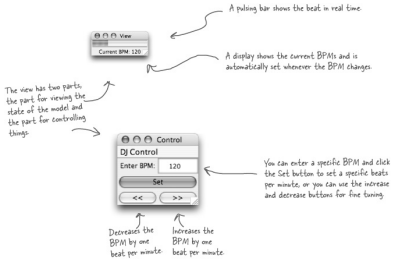
Using MVC to control the beat...

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.

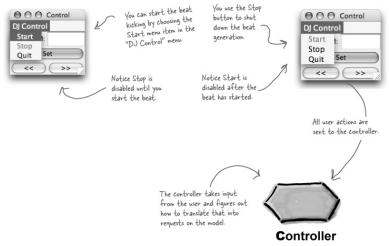
Meet the Java DJ View

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's a few more ways to control the DJ View...

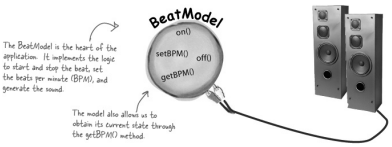


The controller is in the middle...

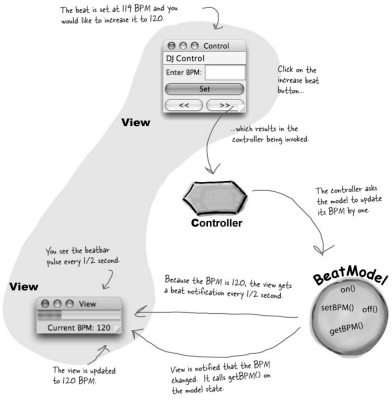
The **controller** sits between the view and model. It takes your input, like selecting "Start" from the DJ Control menu, and turns it into an action on the model to start the beat generation.

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.



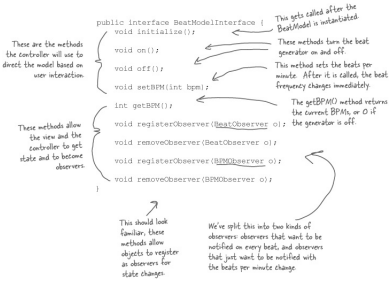
Putting the pieces together



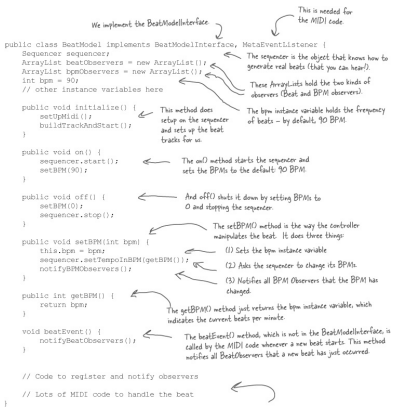
Building the pieces

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



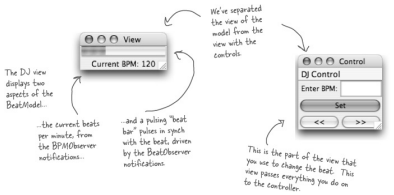
READY-BAKE CODE

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.).

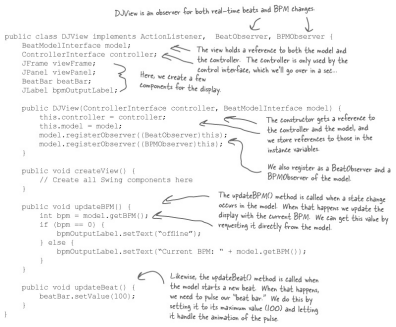
Implementing the View

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.

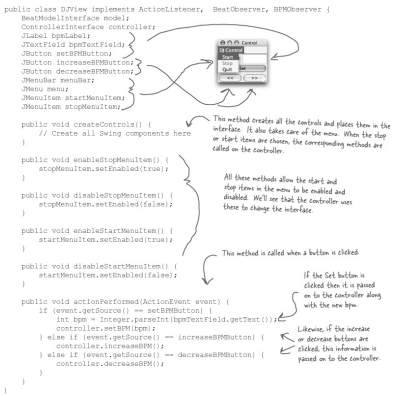
WATCH IT!

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 back of the chapter.



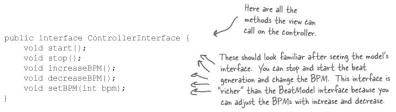
Now, we'll look at the code for the user interface controls part of the view. This view lets you control the model by telling the controller what to do, which in turn, tells the model what to do. Remember, this code is in the same class file as the other view code.



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.

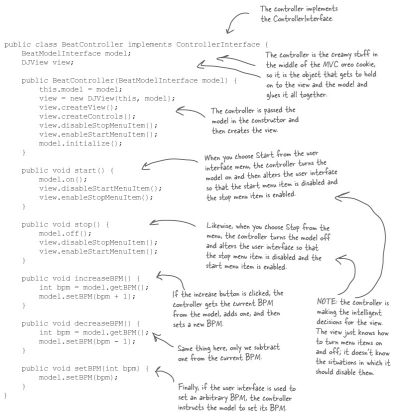
Because we are implementing the Strategy Pattern, we need to start with an interface for any Strategy that might be plugged into the DJ View. We're going to call it ControllerInterface.



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




```
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);
    }
}
```

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...

...and you'll see this.

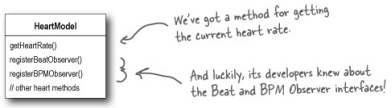
Things to do

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 we happen to have a heart monitor class; here's the class diagram:



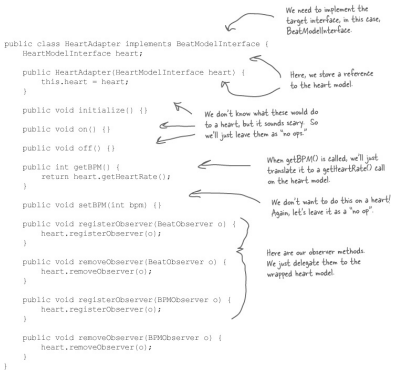
BRAIN POWER

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 HeatModel 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?

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.

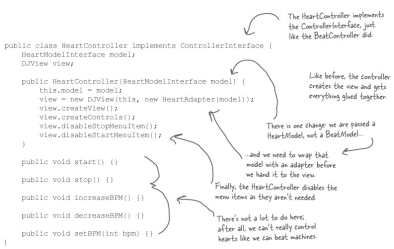
Here's the code to adapt a HeartModel to a BeatModel:




```
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) {}
}
```

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);
    }
}
```

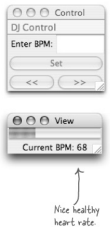
All we need to do is create the controller and pass it a heart monitor.

And now for a test run...

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

Run this...

...and you'll see this.



Nice healthy heart rate.

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.**

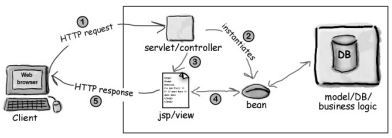
4. **The Start and Stop menu items are not enabled because the controller disabled them.**

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 (4) 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 guys & girls 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/* (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.

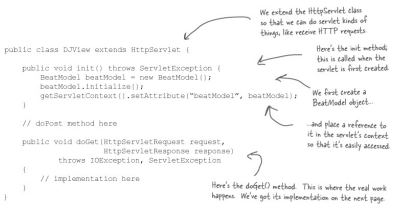
Now all we need to do is adapt it to work in the web environment, but, given that it doesn't depend on any outside classes, there is really no work to be done. We can use our BeatModel off the shelf without changes. So, let's be productive and move on to step two!

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.



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

```
public void doGet(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.start();
    }
    String off = request.getParameter("off");
    if (off != null) {
        beatModel.stop();
    }

    request.setAttribute("beatModel", beatModel);

    RequestDispatcher dispatcher =
        request.getRequestDispatcher("/jsp/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 start or stop.

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.

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.

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.

```
<jsp:useBean id="beatModel" scope="request" class="headfirst.combined.djview.BeatModel" />

<html>
<head>
<title>DJ View</title>
</head>
<body>

<h1>DJ View</h1>
Beats per minutes = <jsp:getProperty name="beatModel" property="BPM" />
<br />
<br>
<br />

<form method="post" action="/djview/servlet/DJView">
BPM: <input type=text name="bpm"
           value="<jsp:getProperty name="beatModel"
           property="BPM" />">
 

<input type="submit" name="set" value="set"><br />
<input type="submit" name="decrease" value="<<">
<input type="submit" name="increase" value=">>"><br />
<input type="submit" name="on" value="on">
<input type="submit" name="off" value="off"><br />
</form>

</body>
</html>
```

Here's our bean, which the servlet passed us.

Beginning of the HTML.

Here we use the model bean to extract the BPM property.

Now we generate the view, which prints out the current beats per minute.

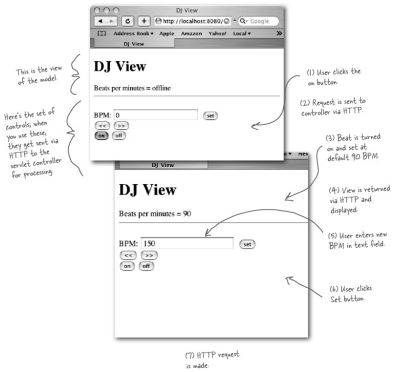
And here's the control part of the view. We have a text entry for entering a BPM along with increase/decrease and on/off buttons.

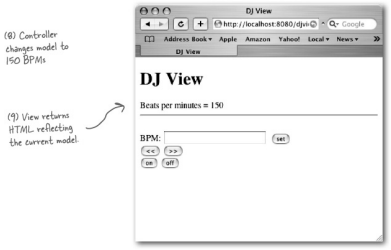
And here's the end of the HTML.

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...





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

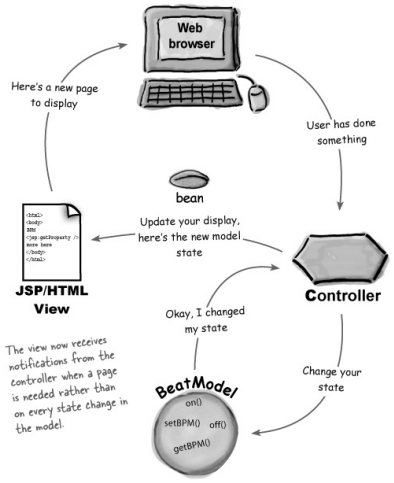
Even though Model 2 doesn't look exactly like "textbook" MVC, all the parts are still there; they've just been adapted to reflect the idiosyncrasies of the web browser model. Let's take another look...

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.

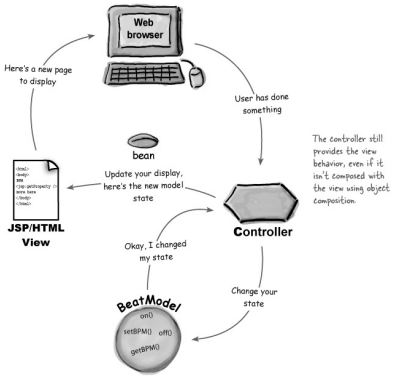


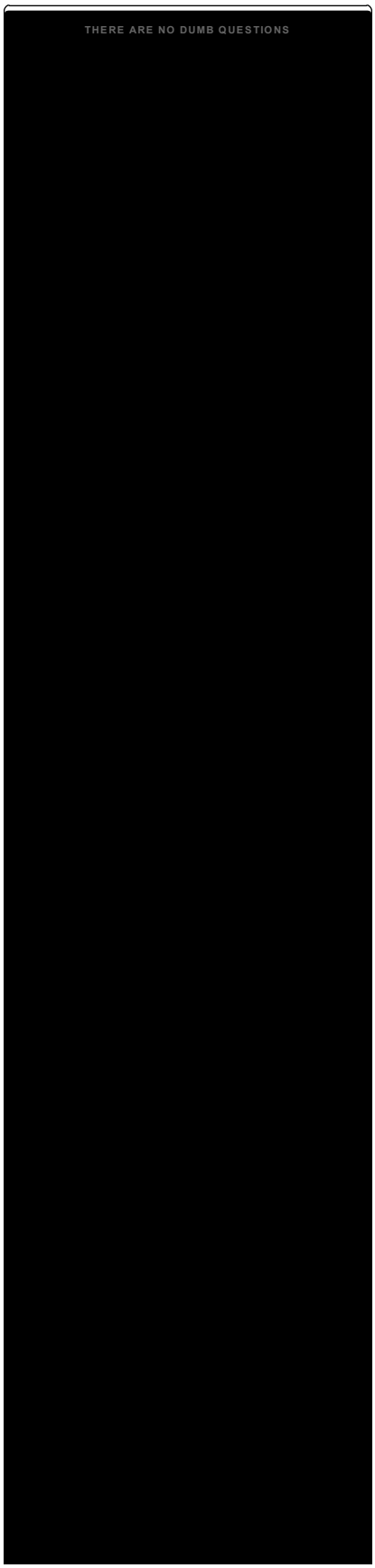
Strategy

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.




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!

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

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

```
                                    QuackCounter is a Quackable, so
                                    now it's a QuackObservable too.
public class QuackCounter implements Quackable {
    Quackable duck;
    static int numberOfQuacks;                  Here's the duck that the
                                                QuackCounter is decorating.
    public QuackCounter(Quackable duck) {       It's this duck that really
        this.duck = duck;                       needs to handle the
    }                                           observable methods.

    public void quack() {
        duck.quack();                           All of this code is the same
        numberOfQuacks++;                       as the previous version of
    }                                           QuackCounter.

    public static int getQuacks() {
        return numberOfQuacks;
    }

    public void registerObserver(Observer observer) {
        duck.registerObserver(observer);        Here are the two
    }                                           QuackObservable
                                                methods. Notice that
    public void notifyObservers() {             we just delegate both
        duck.notifyObservers();                 calls to the duck that
    }                                           we're decorating.
}
```

SHARPEN YOUR PENCIL

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.

```
                                    Flock is a Quackable, so now
                                    it's a QuackObservable too
public class Flock implements Quackable {
    ArrayList ducks = new ArrayList();          Here's the Quackables that
                                                are in the Flock.
    public void add(Quackable duck) {
        ducks.add(duck);
    }

    public void quack() {
        Iterator iterator = ducks.iterator();
        while (iterator.hasNext()) {
            Quackable duck = (Quackable)iterator.next();
            duck.quack();
        }
    }                                           When you register an Observer
                                                with the Flock, you actually get
    public void registerObserver(Observer observer) {   registered with everything that's
        Iterator iterator = ducks.iterator();           IN the flock, which is every
        while (iterator.hasNext()) {                    Quackable, whether it's a duck or
            Quackable duck = (Quackable)iterator.next();    another Flock.
            duck.registerObserver(observer);        We iterate through all the
        }                                           Quackables in the Flock and
    }                                               delegate the call to each
                                                    Quackable. If the Quackable
    public void notifyObservers() { }               is another Flock, it will do
                                                    the same.
}
                                    Each Quackable does its own notification,
                                    so Flock doesn't have to worry about it.
                                    This happens when Flock delegates quack()
                                    to each Quackable in the Flock.
```

SHARPEN YOUR PENCIL

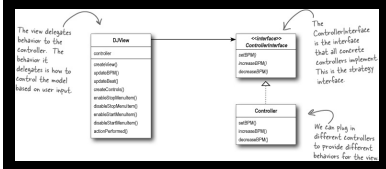
We're still directly instantiating Geese by relying on concrete classes. Can you write an Abstract Factory for Geese? How should it handle creating "goose ducks?"

NOTE

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 CLASS

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 shows 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 http://www.wickedlysmart.com. Have fun!

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

import javax.sound.midi.*;
import java.util.*;

public class BeatModel implements BeatModelInterface, MetaEventList
    Sequencer sequencer;
    ArrayList beatObservers = new ArrayList();
    ArrayList bpmObservers = new ArrayList();
    int bpm = 90;
    // other instance variables here
    Sequence sequence;
    Track track;

    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();
    }



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

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

public interface BeatObserver {
    void updateBeat();
}



package headfirst.combined.djview;

public interface BPMObserver {
    void updateBPM();
}


package headfirst.combined.djview;

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

public class DJView implements ActionListener, BeatObserver, BPMOb
    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, BeatModelInterfac
    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.CENTE
    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.CENT
    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();
            //bpmOutputLabel.setText("offline");
        }
    });
    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,

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

    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.combined.djview;

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

package headfirst.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.combined.djview;

public class HeartTestDrive {

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

package headfirst.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.combined.djview;

import java.util.*;

public class HeartModel implements HeartModelInterface, Runnable {
    ArrayList beatObservers = new ArrayList();
    ArrayList bpmObservers = new ArrayList();
```

```
    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
            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.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.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) {}
}
```

[3]

send us email for a copy.

PREV
11. The Proxy Pattern: Controlling Object Access

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

Terms of Service / Privacy Policy