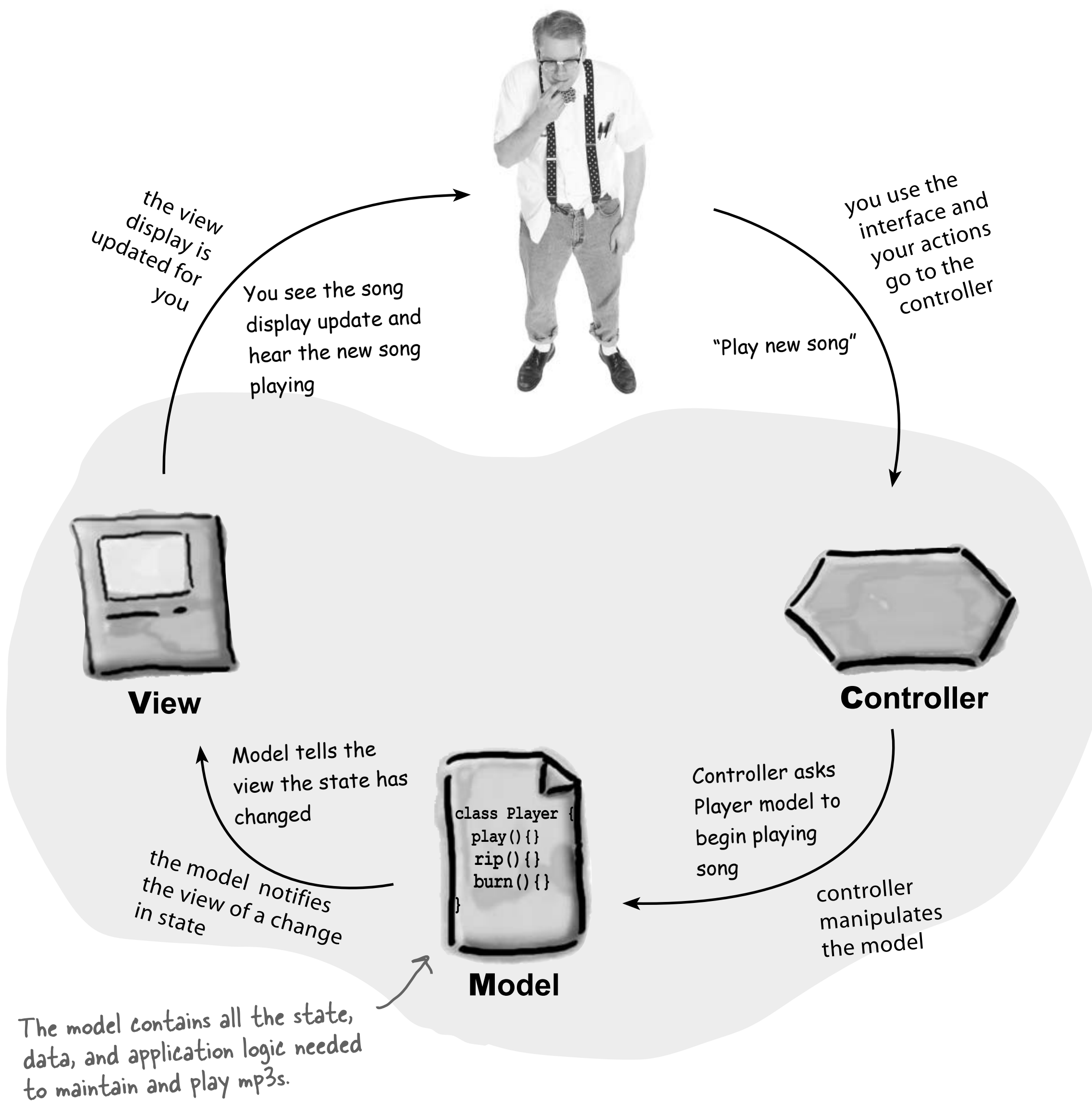
Meet the Model-View-Controller

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.

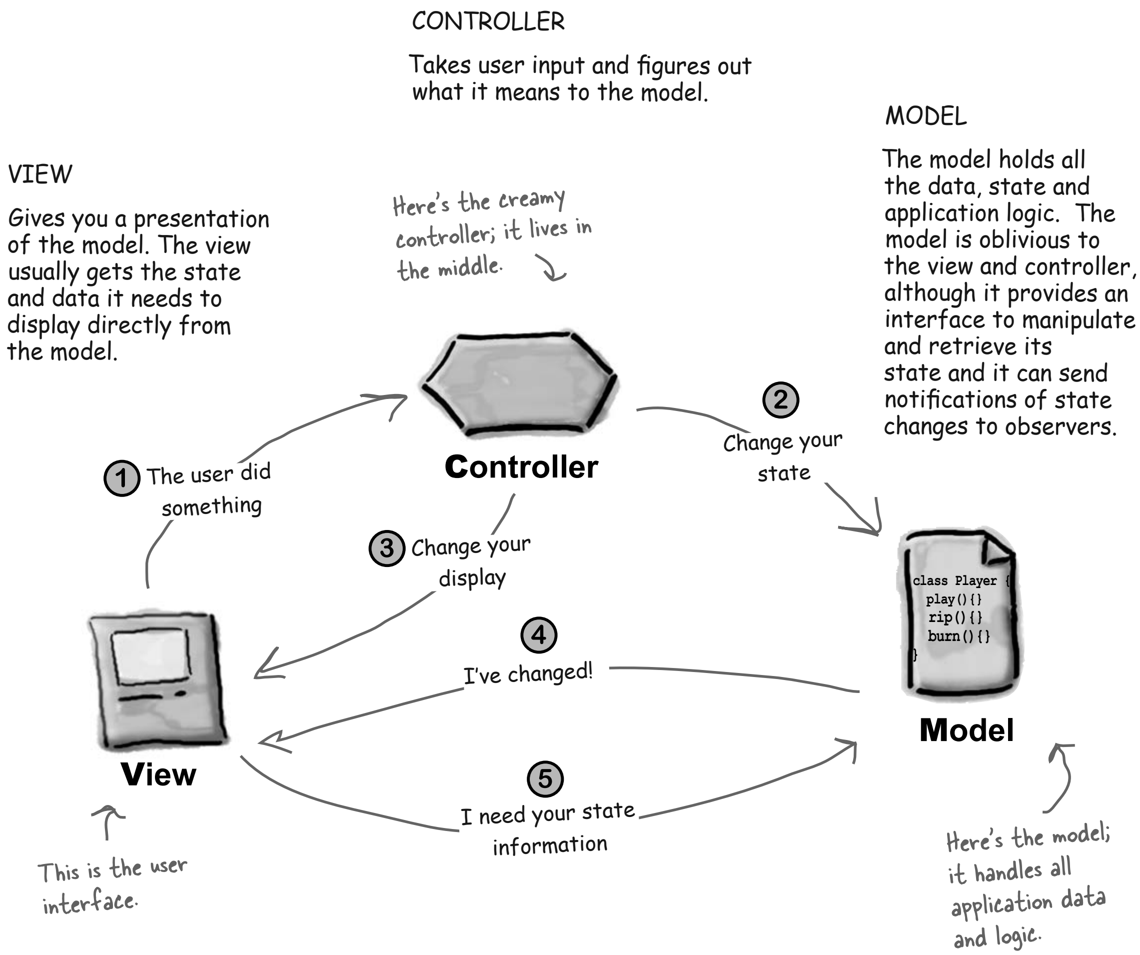
Well, underneath it all sits the Model-View-Controller...



Now let's zoom into the

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

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: 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: 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

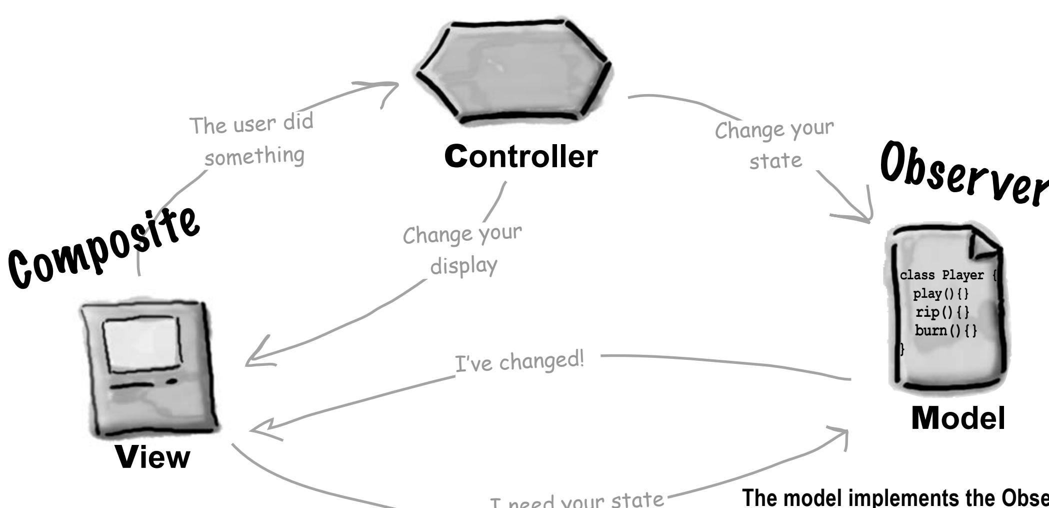
We've already told you the best path to learning the MVC is to see it for what it is: a set of patterns working together in the same design.

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:

Strategy

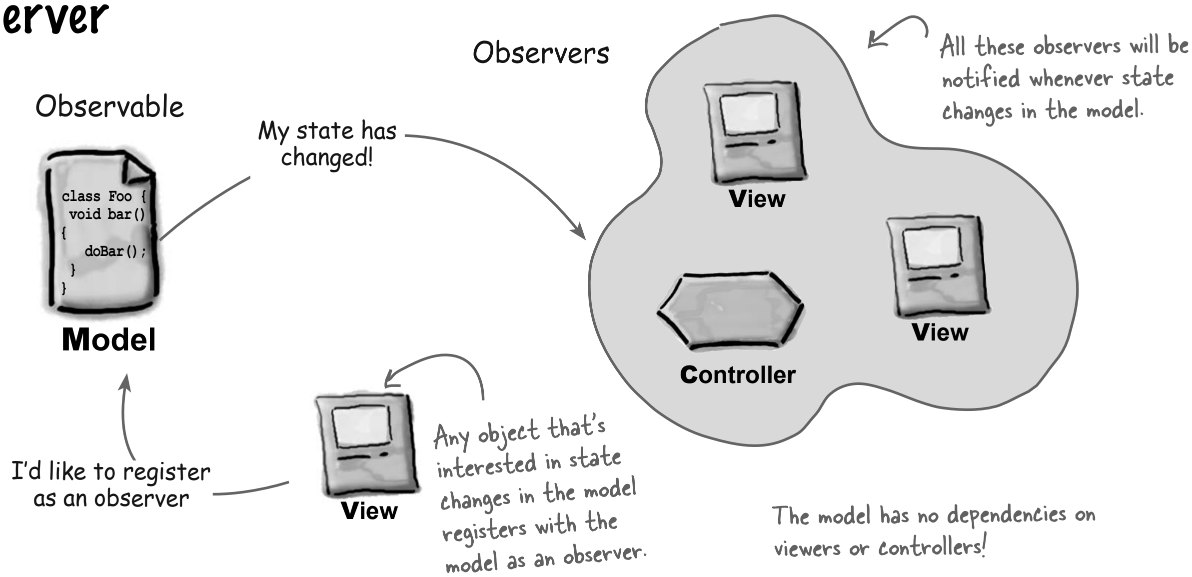
The view and controller implement the classic Strategy Pattern: the view is an object that is configured with a strategy. The controller provides the strategy. The view is concerned only with the visual aspects of the application, and delegates to the controller for any decisions about the interface behavior. Using the Strategy Pattern also keeps the view decoupled from the model because it is the controller that is responsible for interacting with the model to carry out user requests. The view knows nothing about how this gets done.



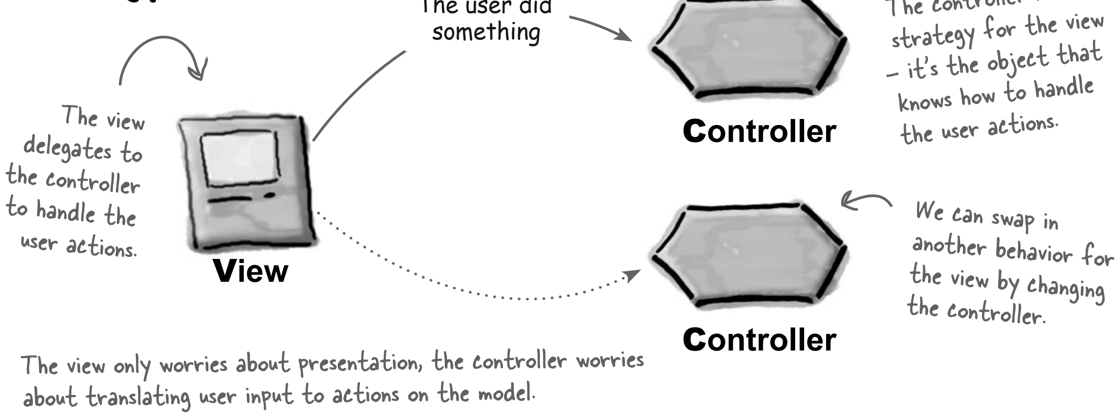
The display consists of a nested set of windows, panels, buttons, text labels and so on. Each display component is a composite (like a window) or a leaf (like a button). When the controller tells the view to update, it only has to tell the top view component, and Composite takes care of the rest.

The model implements the Observer Pattern to keep interested objects updated when state changes occur. Using the Observer Pattern keeps the model completely independent of the views and controllers. It allows us to use different views with the same model, or even use multiple views at once.

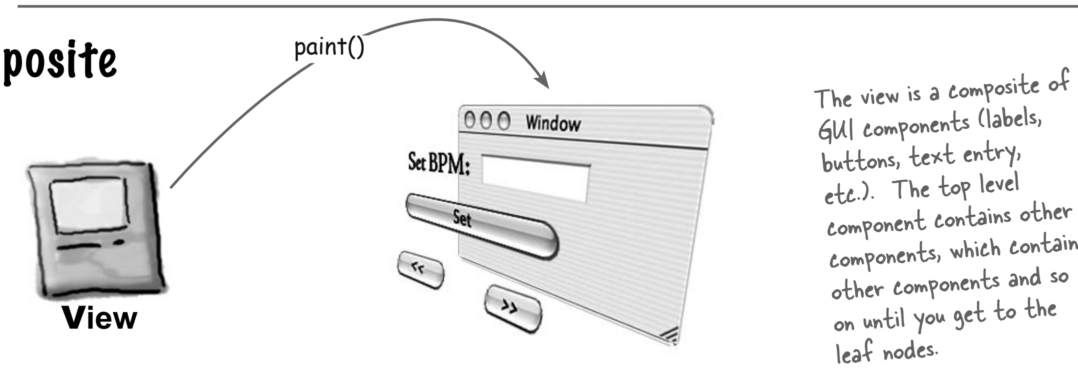
Observer



Strategy



Composite



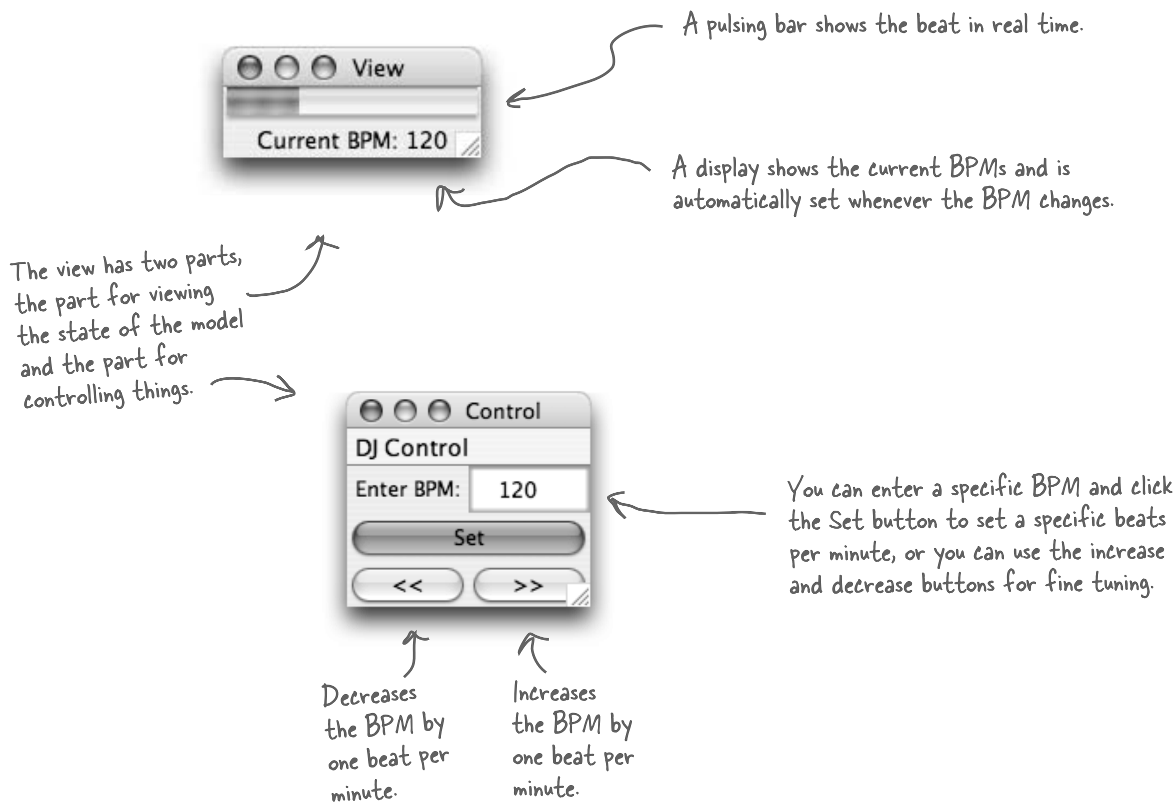
Using MVC to control the beat...

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.

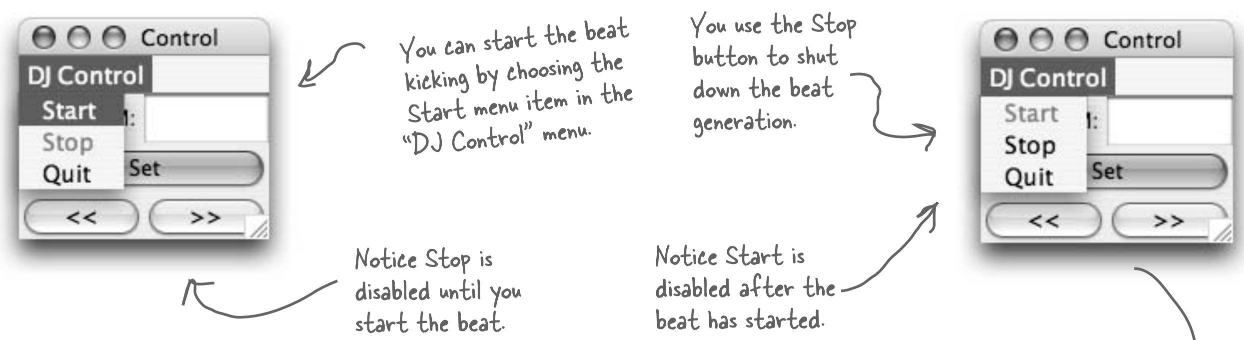
How are you going to do that? You have to control the beat and you're going to build the tool to get you there.

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...

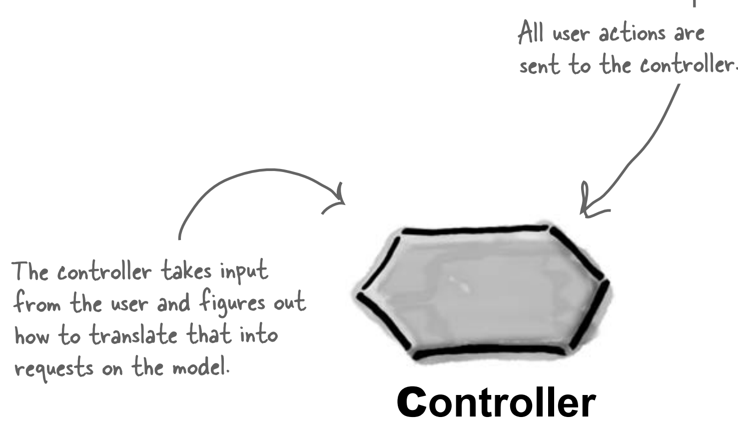


Here's a few more ways to control the DJ View...



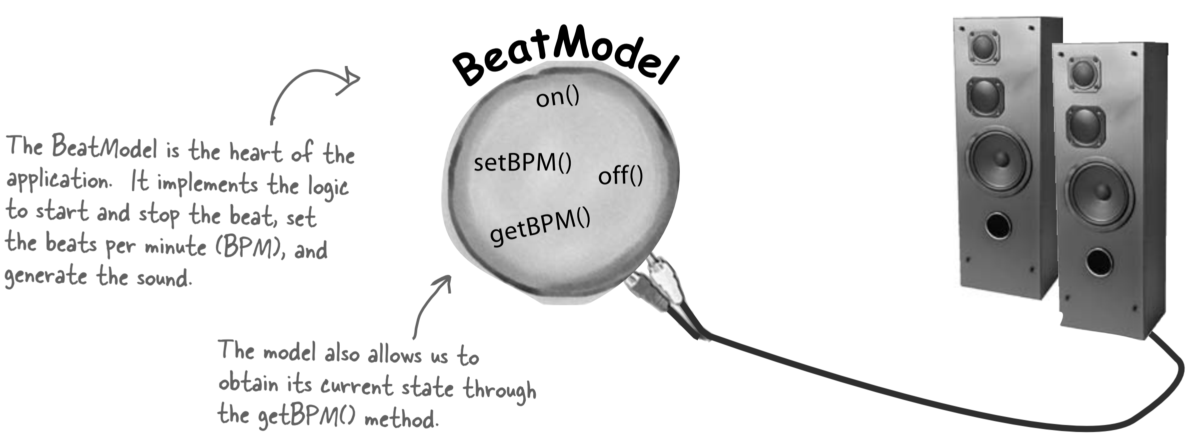
The controller is in the middle...

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.

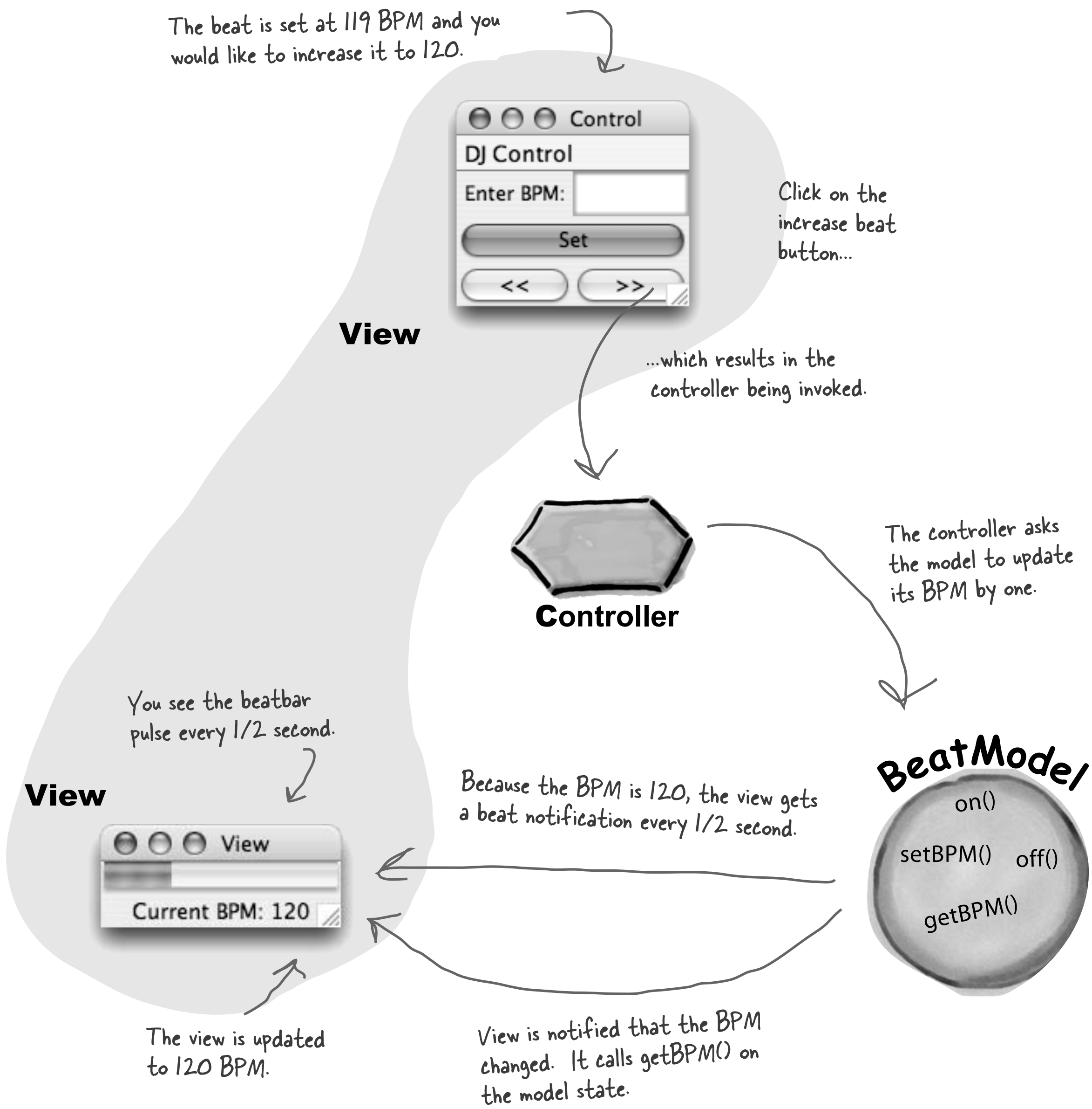


Let's not forget about the model underneath it all...

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:

```
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);
}
```

This gets called after the BeatModel is instantiated.

These methods turn the beat generator on and off.

This method sets the beats per minute. After it is called, the beat frequency changes immediately.

The getBPM() method returns the current BPMs, or 0 if the generator is off.

These are the methods the controller will use to direct the model based on user interaction.

These methods allow the view and the controller to get state and to become observers.

This should look familiar, these methods allow objects to register as observers for state changes.

We've split this into two kinds of observers: observers that want to be notified on every beat, and observers that just want to be notified with the beats per minute change.

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

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

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

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

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

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

    public int getBPM() {
        return bpm;
    }

    void beatEvent() {
        notifyBeatObservers();
    }

    // Code to register and notify observers

    // Lots of MIDI code to handle the beat
}
```

- We implement the BeatModelInterface.
- This is needed for the MIDI code.
- The sequencer is the object that knows how to generate real beats (that you can hear!).
- These ArrayLists hold the two kinds of observers (Beat and BPM observers).
- The bpm instance variable holds the frequency of beats – by default, 90 BPM.
- This method does setup on the sequencer and sets up the beat tracks for us.
- The on() method starts the sequencer and sets the BPMs to the default: 90 BPM.
- And off() shuts it down by setting BPMs to 0 and stopping the sequencer.
- The setBPM() method is the way the controller manipulates the beat. It does three things:
  - (1) Sets the bpm instance variable
  - (2) Asks the sequencer to change its BPMs.
  - (3) Notifies all BPM Observers that the BPM has changed.
- The getBPM() method just returns the bpm instance variable, which indicates the current beats per minute.
- The beatEvent() method, which is not in the BeatModelInterface, is called by the MIDI code whenever a new beat starts. This method notifies all BeatObservers that a new beat has just occurred.

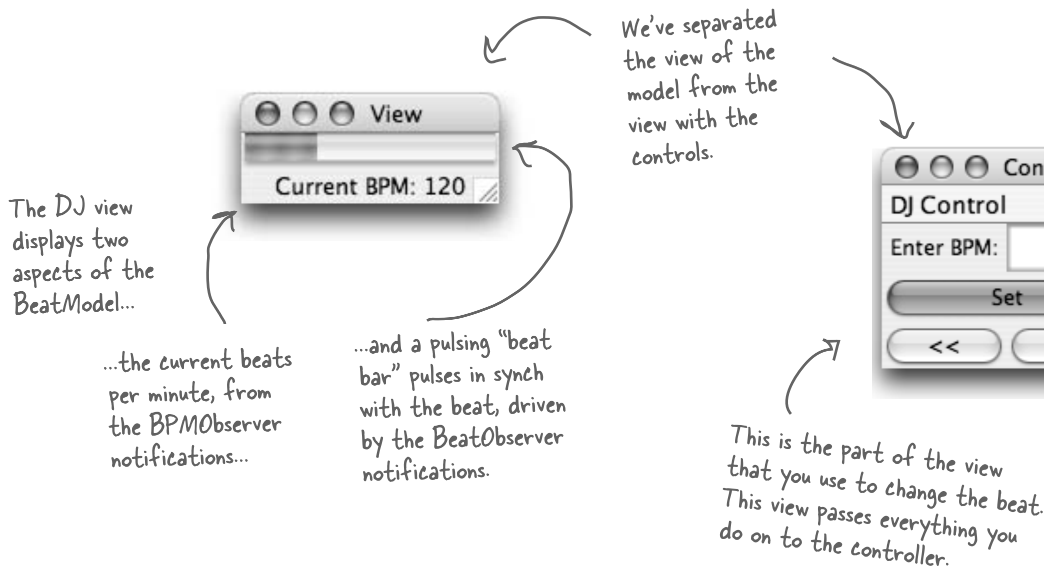
Ready-bake Code

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 headfirstlabs.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

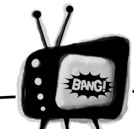
Our BeatModel makes no assumptions about the view. The model is implemented using the Observer Pattern, so it just notifies any view registered as an observer when its state changes. The view uses the model's API to get access to the state. We've implemented one type of view, can you think of other views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

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

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

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

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

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

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

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

Here, we create a few components for the display.

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

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

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

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

Implementing the View, continued...

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

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

    public void createControls() {
        // Create all Swing components here
    }
    public void enableStopMenuItem() {
        stopMenuItem.setEnabled(true);
    }

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

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

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

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

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

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

This method is called when a button is clicked.

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

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

Now for the Controller

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

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

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

Here are all the methods the view can call on the controller.

These should look familiar after seeing the model's interface. You can stop and start the beat generation and change the BPM. This interface is "richer" than the BeatModel interface because you can adjust the BPMs with increase and decrease.

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:

The controller implements the ControllerInterface.

```
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 controller is the creamy stuff in the middle of the MVC oreo cookie, so it is the object that gets to hold on to the view and the model and glues it all together.

The controller is passed the model in the constructor and then creates the view.

When you choose Start from the user interface menu, the controller turns the model on and then alters the user interface so that the start menu item is disabled and the stop menu item is enabled.

Likewise, when you choose Stop from the menu, the controller turns the model off and alters the user interface so that the stop menu item is disabled and the start menu item is enabled.

If the increase button is clicked, the controller gets the current BPM from the model, adds one, and then sets a new BPM.

Same thing here, only we subtract one from the current BPM.

Finally, if the user interface is used to set an arbitrary BPM, the controller instructs the model to set its BPM.

NOTE: the controller is making the intelligent decisions for the view. The view just knows how to turn menu items on and off; it doesn't know the situations in which it should disable them.

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.**

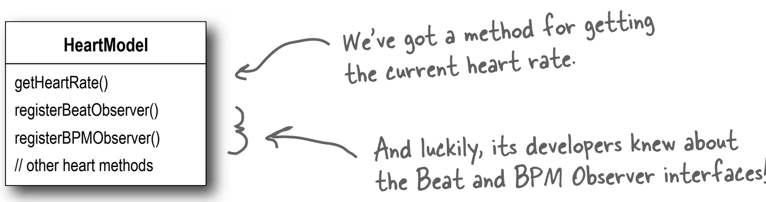
4. **Put on your favorite song and see if you can beat match the beat by using the increase and decrease controls.**

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:

We need to implement the target interface, in this case, BeatModelInterface.

```
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);
    }
}
```

Here, we store a reference to the heart model.

We don't know what these would do to a heart, but it sounds scary. So we'll just leave them as "no ops."

When getBPM() is called, we'll just translate it to a getHeartRate() call on the heart model.

We don't want to do this on a heart! Again, let's leave it as a "no op".

Here are our observer methods. We just delegate them to the wrapped heart model.

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!

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

```
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) {}
}
```

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

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

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

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

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

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

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

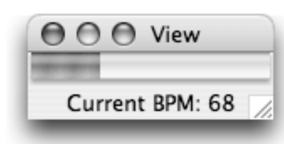
All we need to do is create the controller and pass it a heart monitor.

And now for a test run...

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

Run this...

...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.**