Compound Patterns?

- What is a compound pattern?
- Patterns of patterns
- Let us meet the Model-View-Controller ….

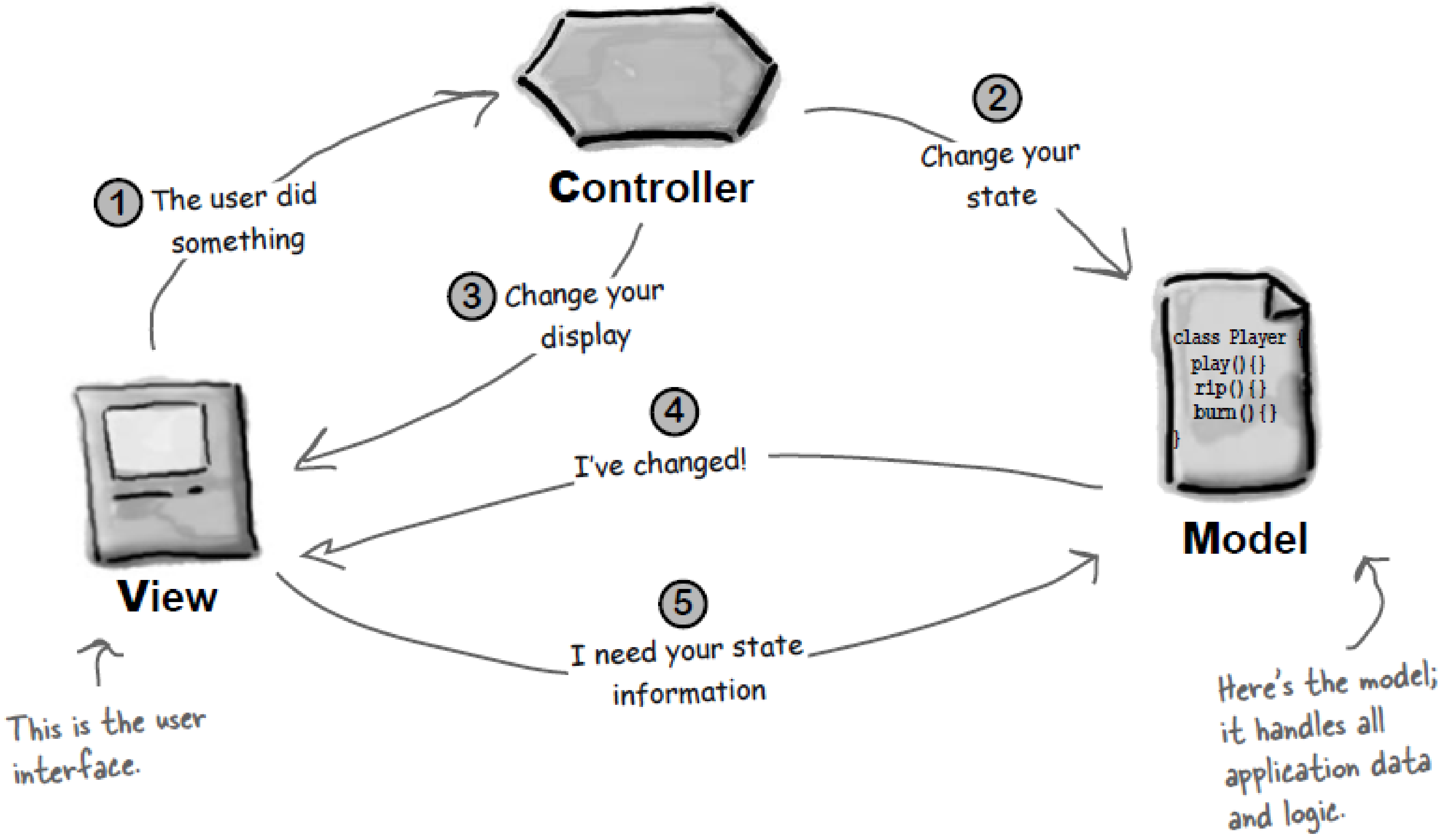
Meet the Model-View-Controller

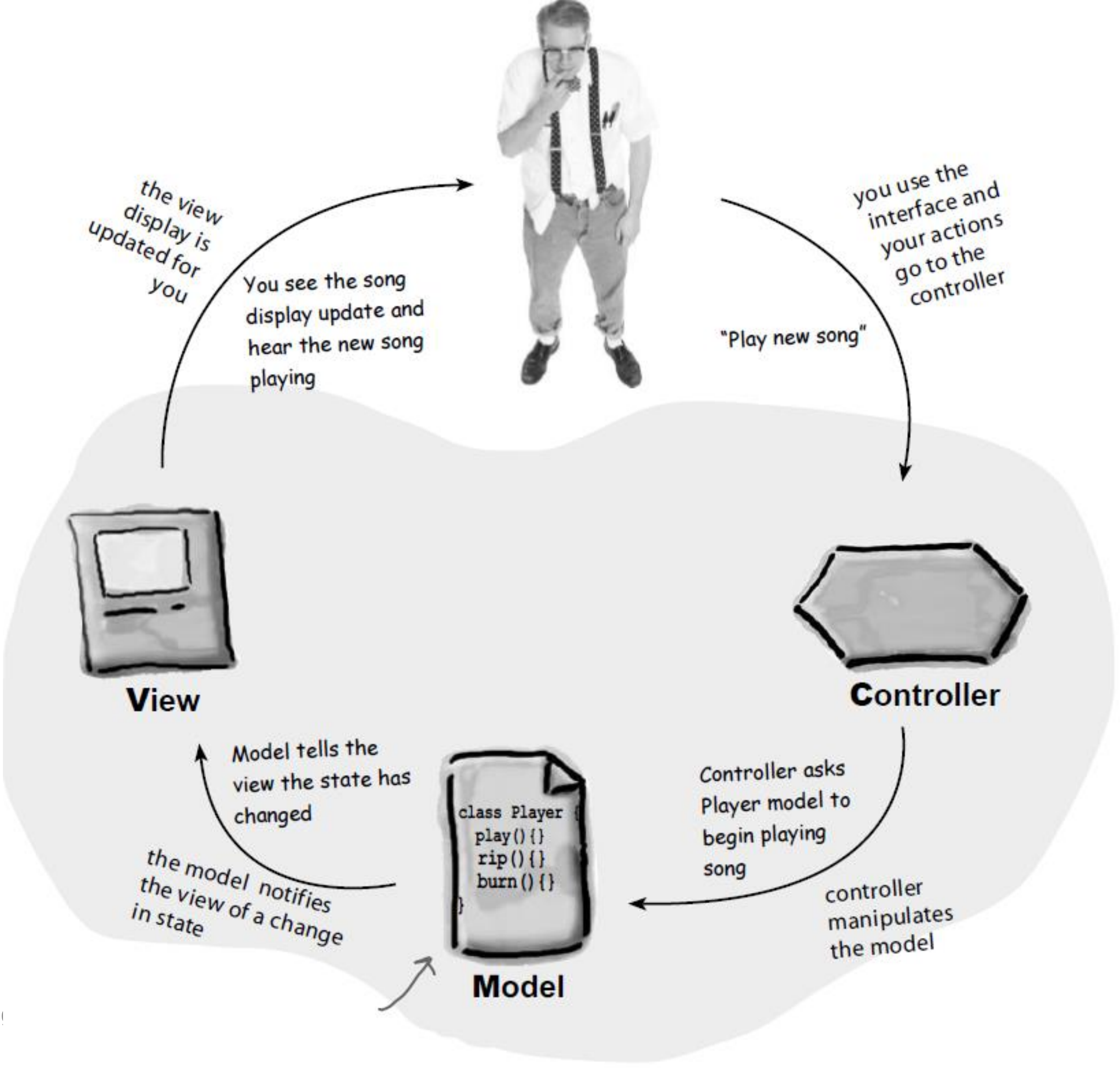
- Imagine you're using your favorite MP3 player, like iTunes.
- You can use its interface to add new songs, manage playlists and rename tracks.
- The player takes care of maintaining a little database of all your songs along with their associated names and data.
- It also takes care of playing the songs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

Meet the Model-View-Controller

- VIEW: Gives you a presentation of the model. The view usually gets the state and data it needs to display directly from the model.
- CONTROLLER: Takes user input and figures out what it means to the model.
- MODEL: The model holds all the data, state and application logic.

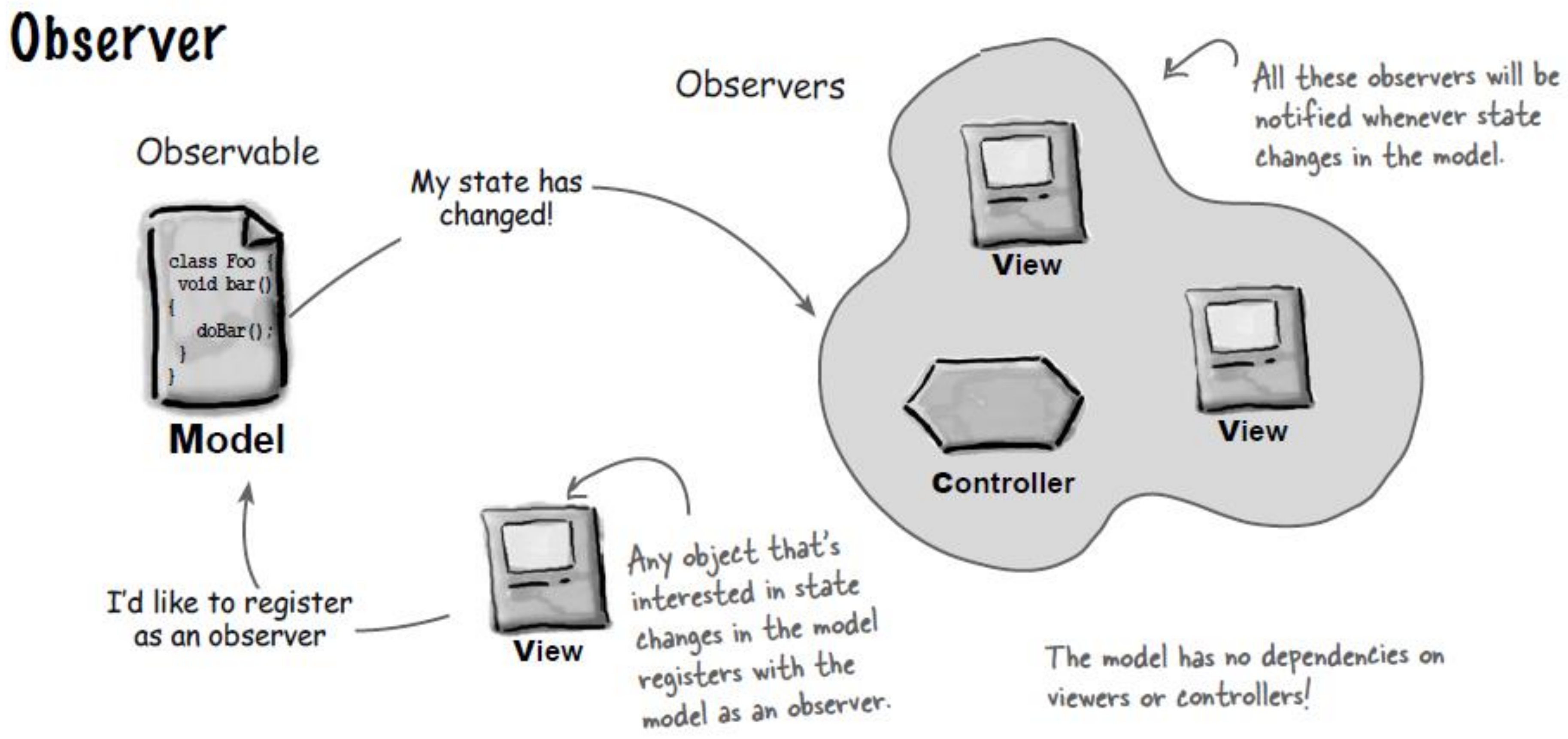
Meet the Model-View-Controller



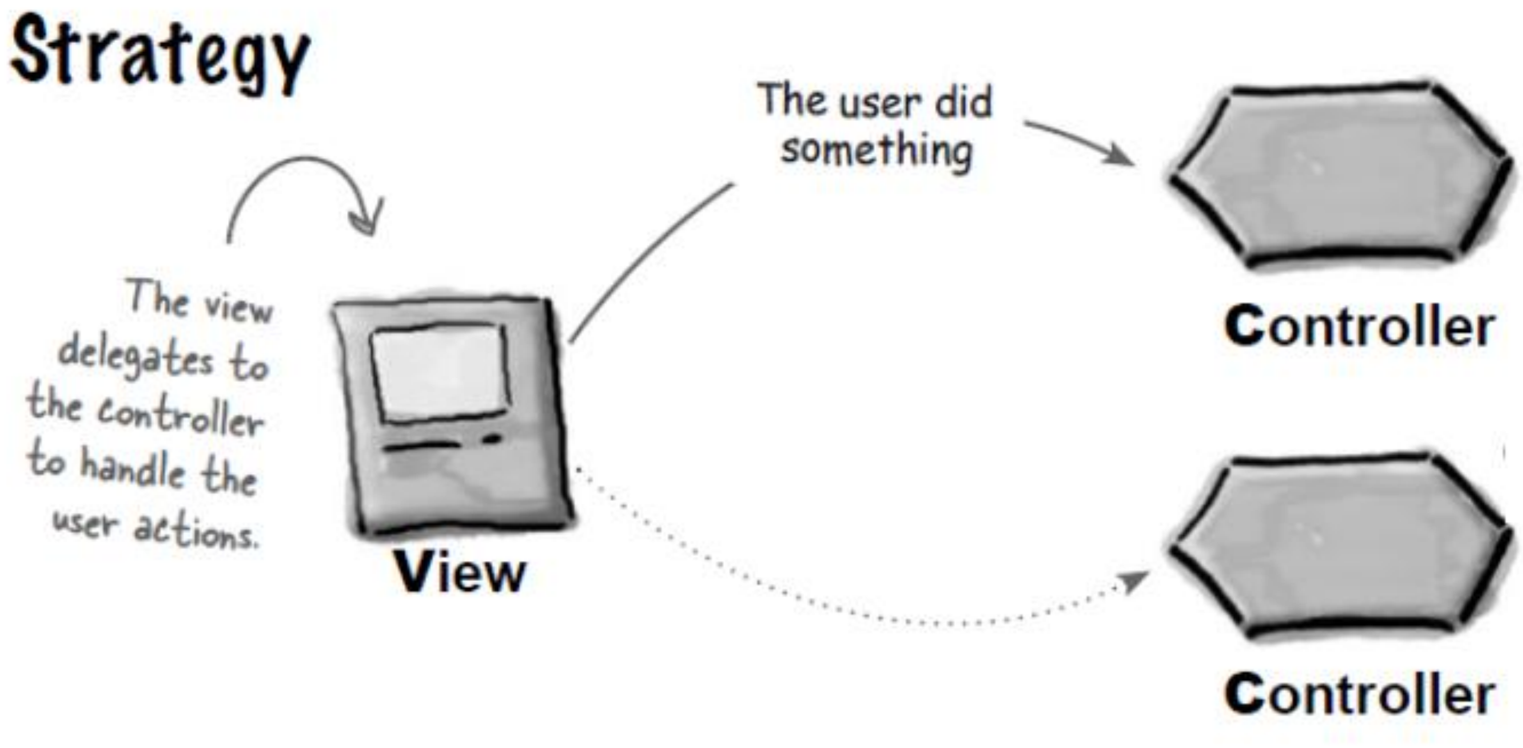


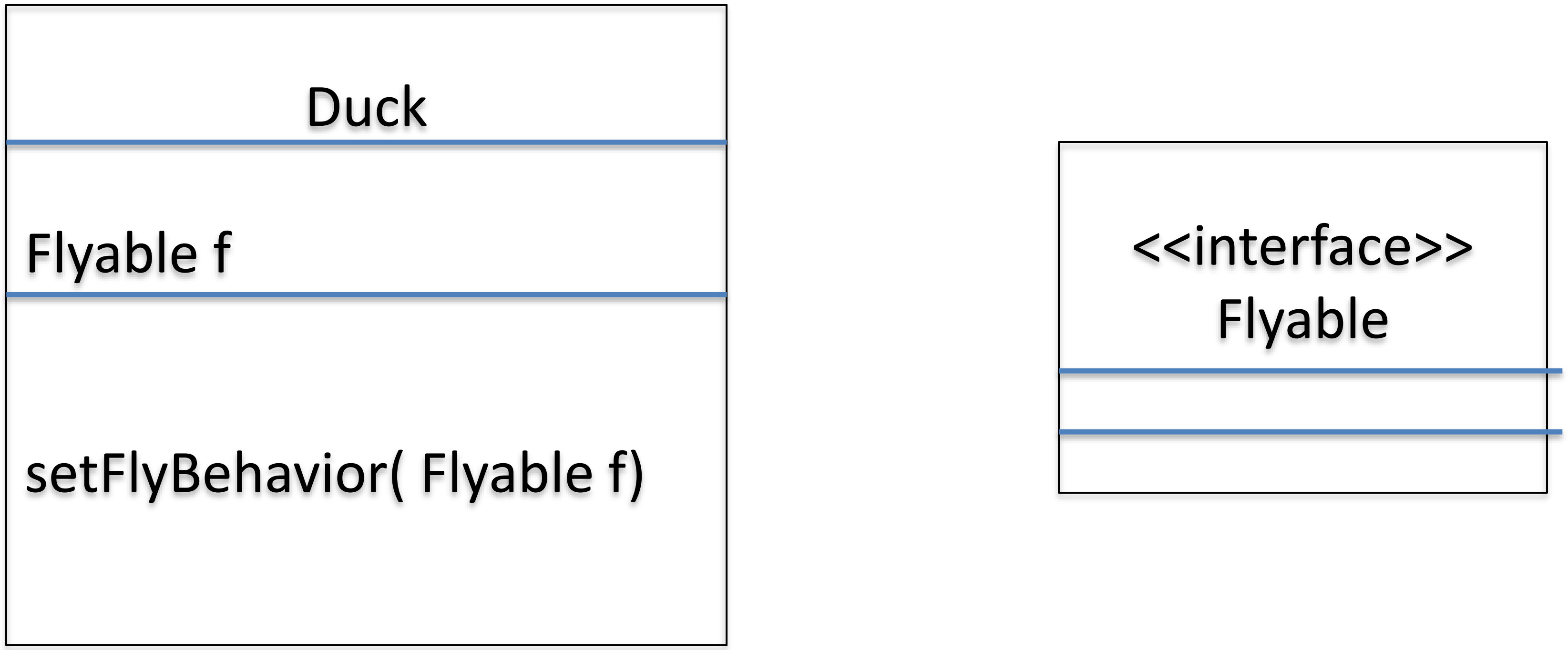
the view display is updated for you
You see the song display update and hear the new song playing
you use the interface and your actions go to the controller
"Play new song"
View
Controller
Model tells the view the state has changed
the model notifies the view of a change in state
class Player {
play() {}
rip() {}
burn() {}
}
Model
Controller asks Player model to begin playing song
controller manipulates the model

Looking at MVC through patterns-colored glasses

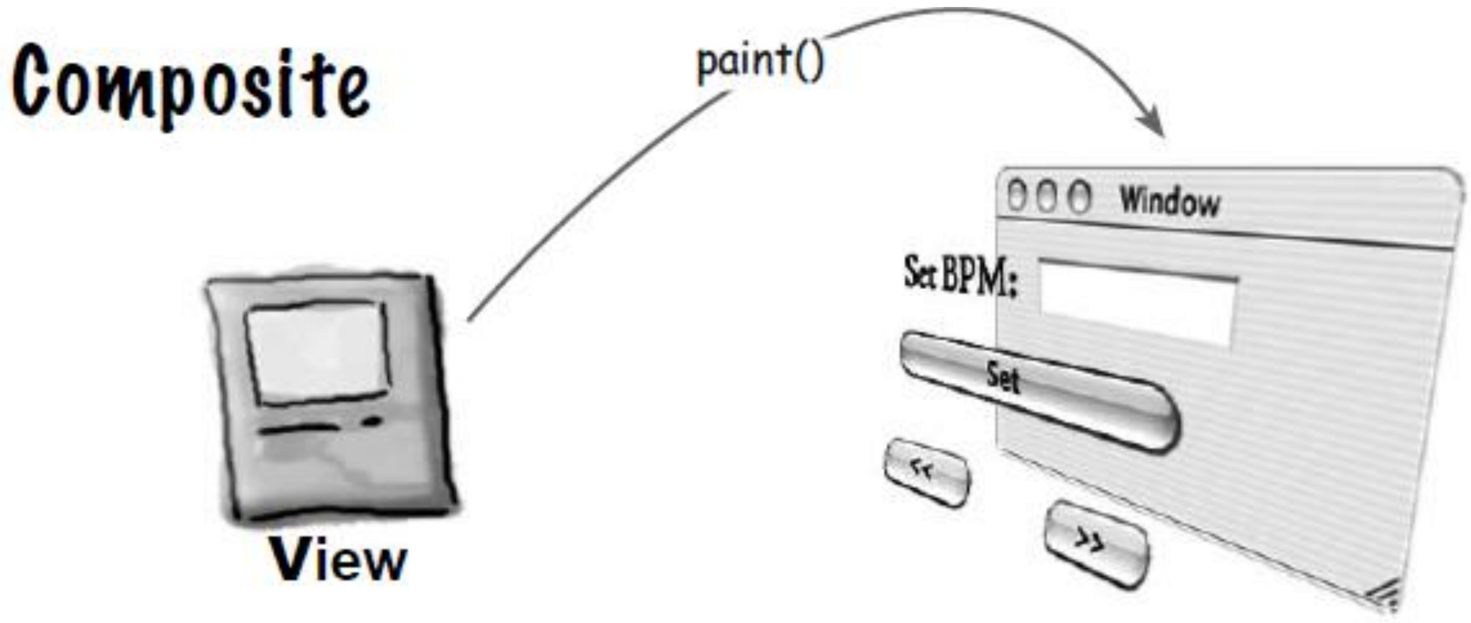

Observer
Observers
Observable
My state has changed!
All these observers will be notified whenever state changes in the model.
class Foo {
void bar()
{
doBar();
}
}
Model
View
Controller
View
I'd like to register as an observer
View
Any object that's interested in state changes in the model registers with the model as an observer.
The model has no dependencies on viewers or controllers!

Looking at MVC through patterns-colored glasses

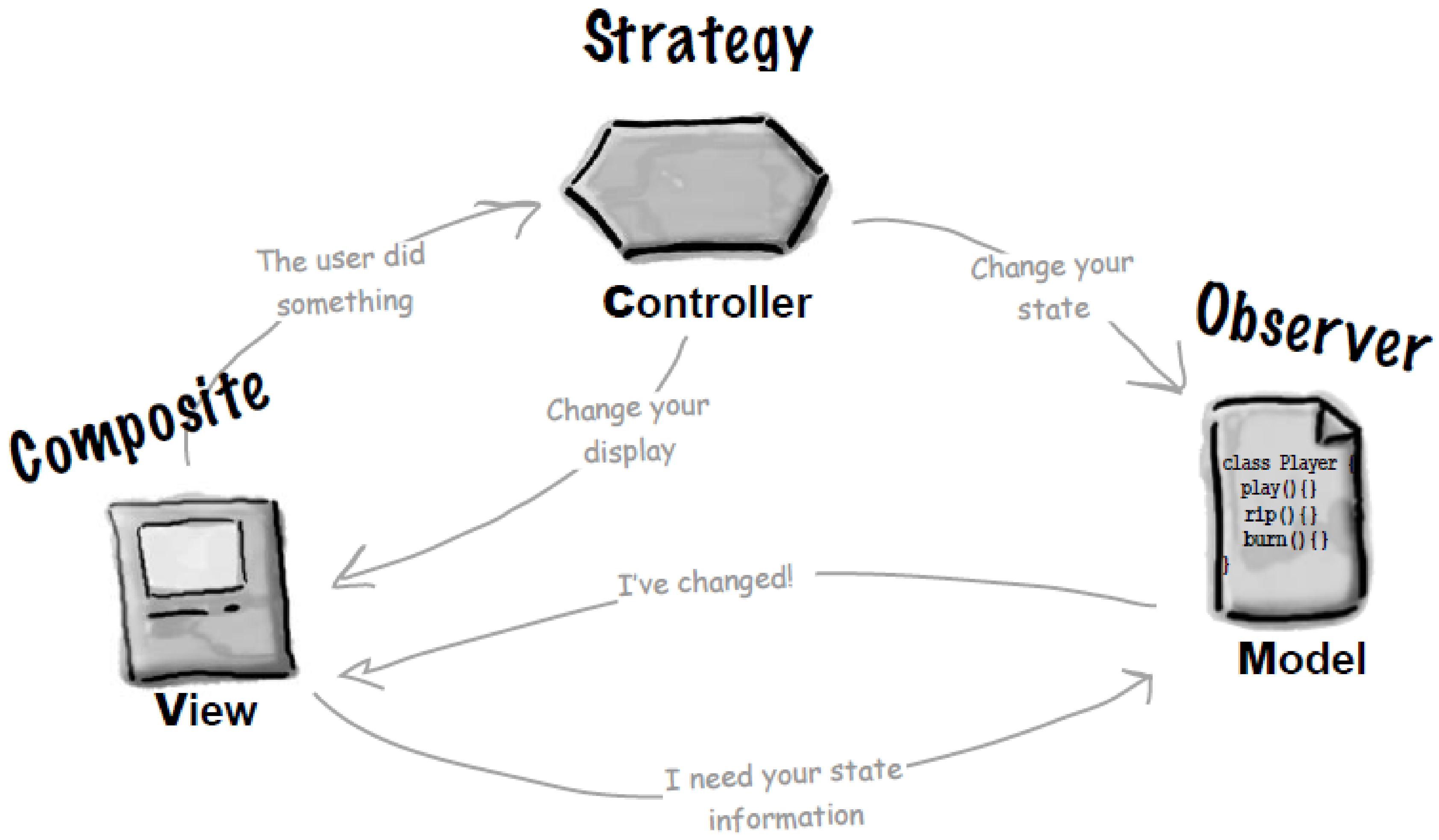

Strategy
The user did something
The view delegates to the controller to handle the user actions.
View
Controller
Controller


Duck
Flyable f
setFlyBehavior(Flyable f)
<<interface>>
Flyable

Looking at MVC through patterns-colored glasses



Looking at MVC through patterns-colored glasses



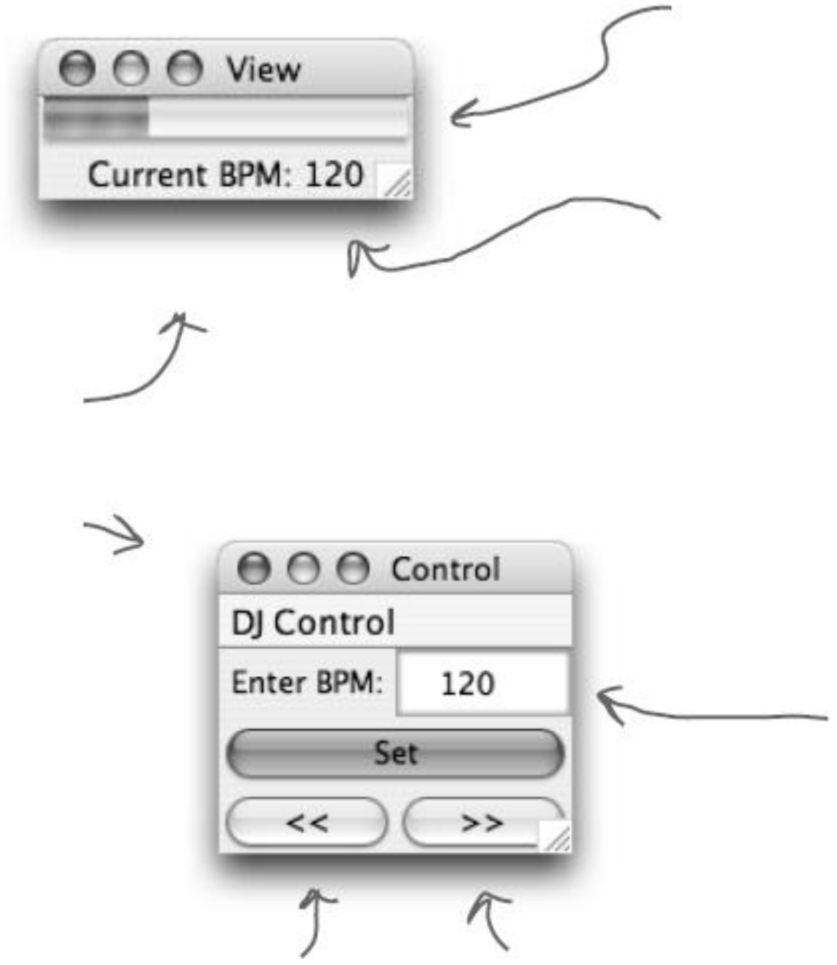
Q & A

- Does the controller ever implement any application logic?.
- I’ve seen descriptions of the MVC where the controller is described as a “mediator” between the view and the model?
- If I have more than one view, do I always need more than one controller?

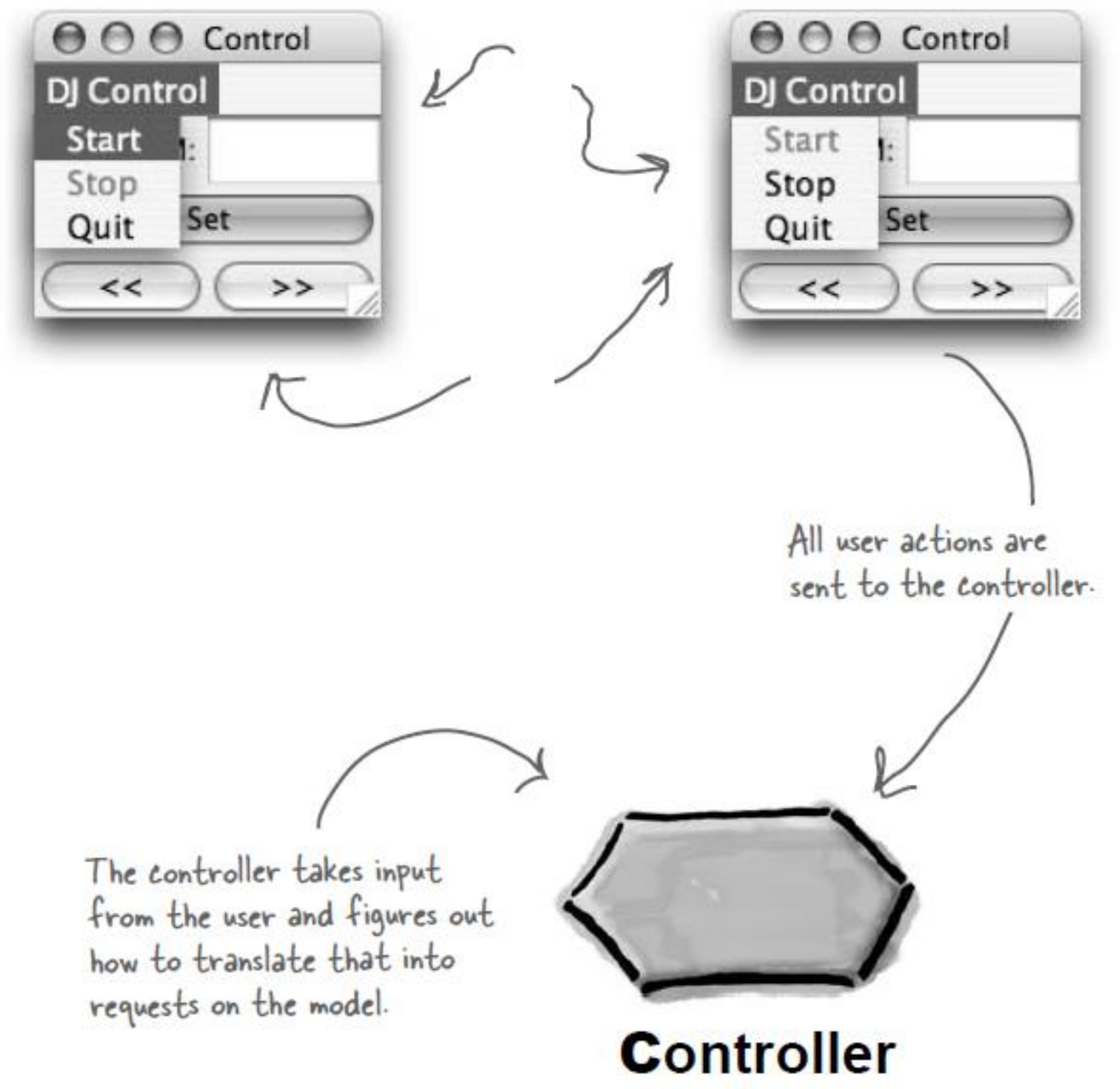
Using MVC to Control the Beat

- 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.

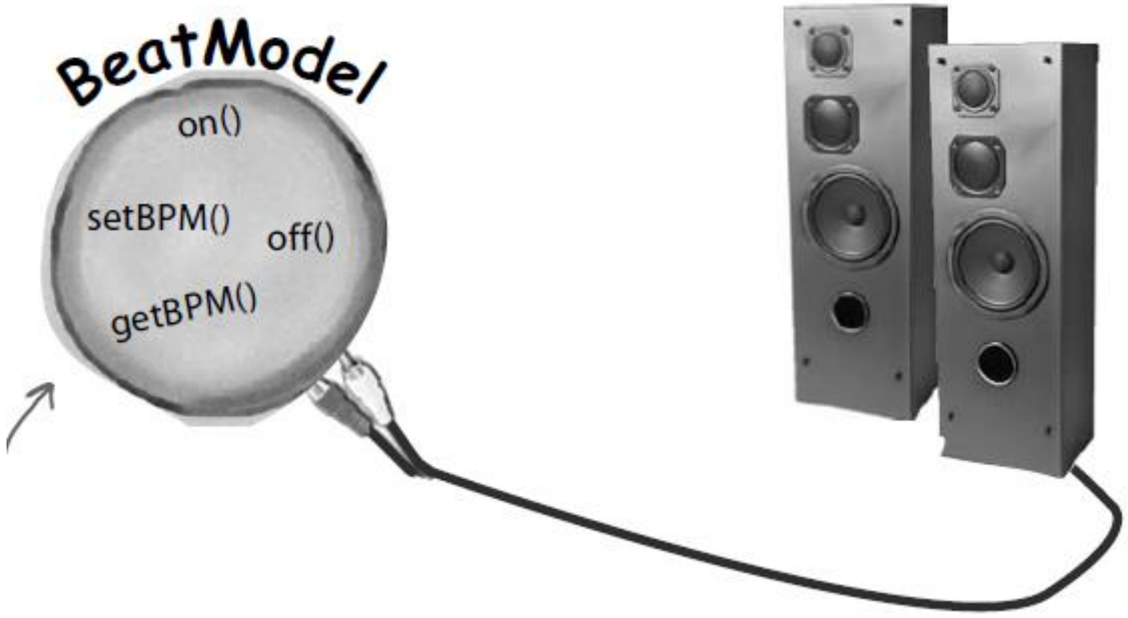
Meet the Java DJ View



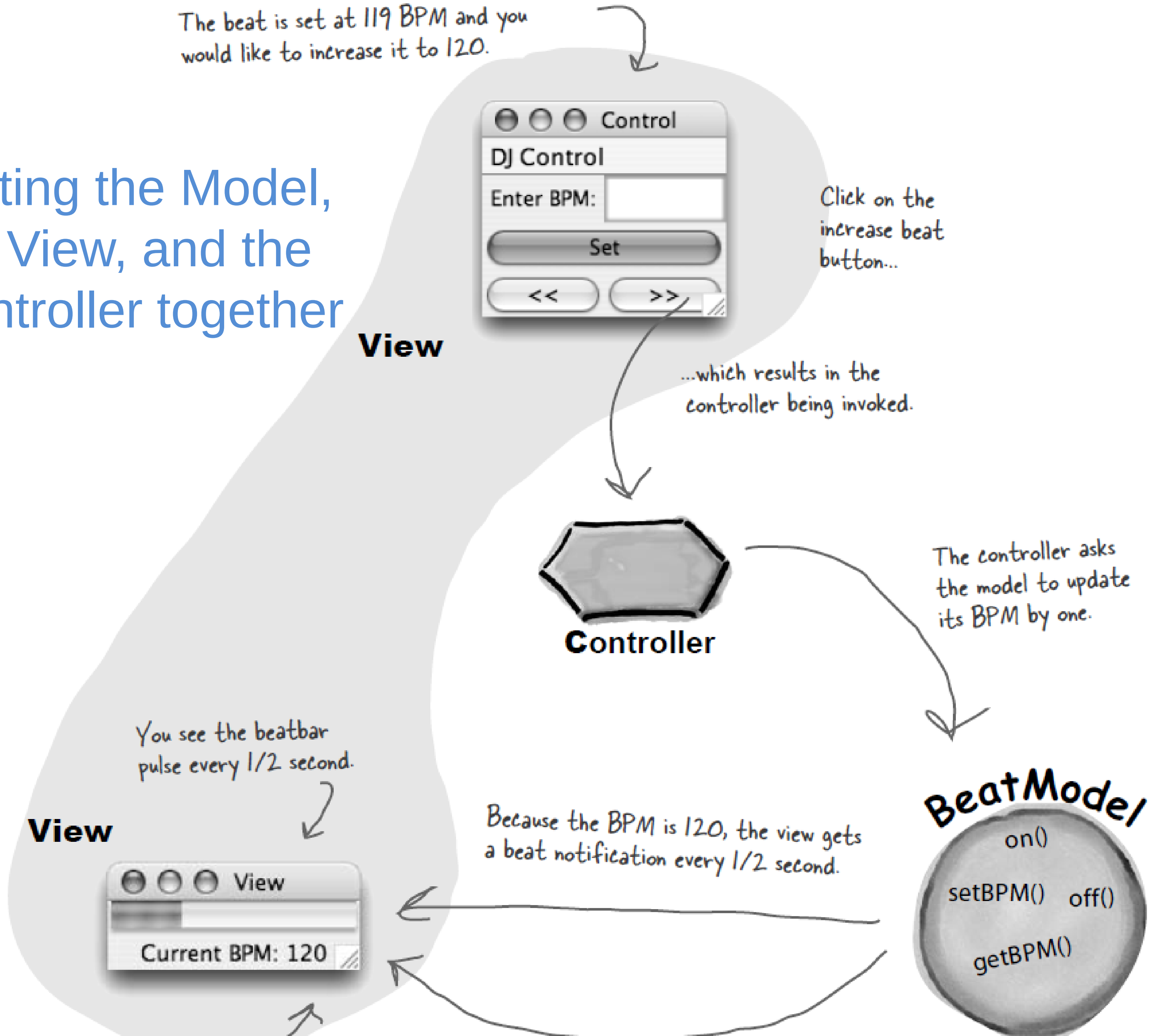
Meet the Java DJ View: The Views and The Controller


Control
DJ Control
Start
Stop
Quit
Set
<<
>>
Control
DJ Control
Start
Stop
Quit
Set
<<
>>
All user actions are sent to the controller.
The controller takes input from the user and figures out how to translate that into requests on the model.
Controller

Meet the Java DJ View: The Model



Putting the Model, the View, and the Controller together


The beat is set at 119 BPM and you would like to increase it to 120.
Control
DJ Control
Enter BPM:
Set
<<
>>
Click on the increase beat button...
View
...which results in the controller being invoked.
Controller
The controller asks the model to update its BPM by one.
BeatModel
on()
setBPM()
off()
getBPM()
Because the BPM is 120, the view gets a beat notification every 1/2 second.
You see the beatbar pulse every 1/2 second.
View
View
Current BPM: 120

Let Us Check the Model 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);
}
```

Let Us Check the Model Implementation

```
public class BeatModel implements
  BeatModelInterface, MetaEventListener {

// other instance variables here
public void initialize() {
   setUpMidi();
   buildTrackAndStart();
}
public void on() {
   sequencer.start();
   setBPM(90);
}
public void off() {
   setBPM(0);
   sequencer.stop();
}
```

Let Us Check the Model Implementation

```
public class BeatModel implements
  BeatModelInterface, MetaEventListener {

ArrayList beatObservers = new ArrayList();
ArrayList bpmObservers = new ArrayList();
int bpm = 90;

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

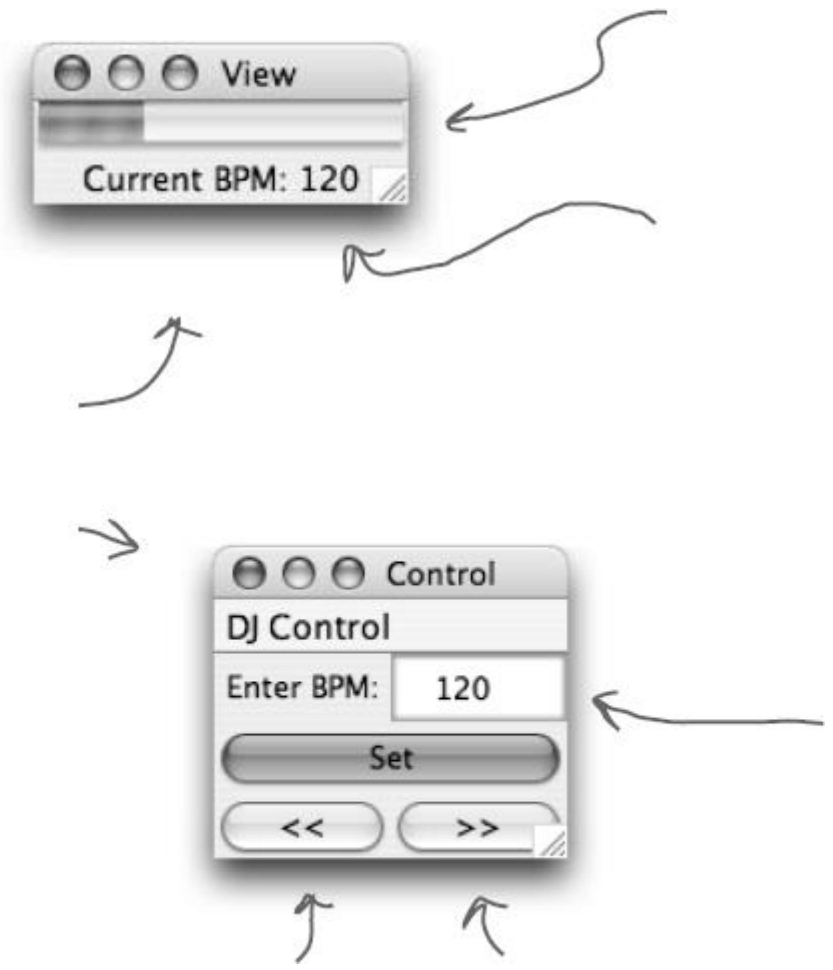
Let Us Check the Model Implementation

```
public class BeatModel implements
  BeatModelInterface, MetaEventListener {
…
public int getBPM() {
  return bpm;
}
// Code to register and notify observers
}
```

Let Us Check the View Implementation

- The view is displayed in two separate windows.

- One window contains the current BPM and the pulse; the other contains the interface controls.

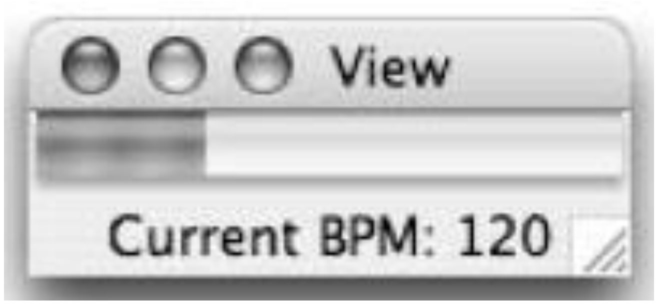


Let Us Check the View Implementation

```
public class DJView implements ActionListener,
   BeatObserver, BPMObserver {
    BeatModelInterface model;

    ControllerInterface controller;

    JFrame viewFrame;
    JPanel viewPanel;
    BeatBar beatBar;
    JLabel bpmOutputLabel;
```

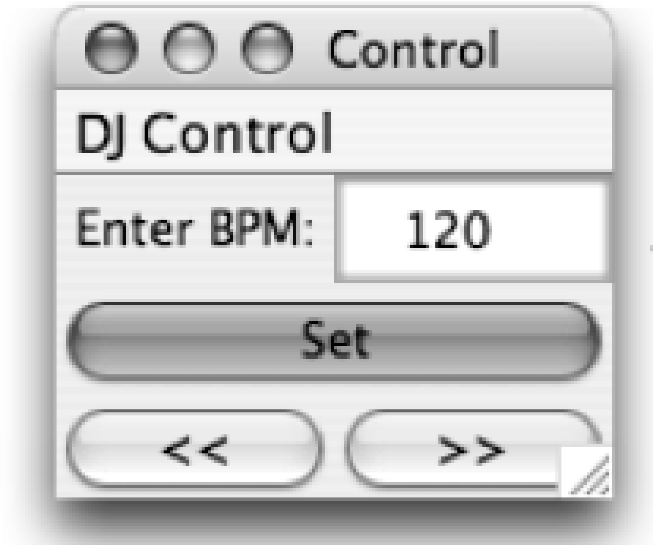


Let Us Check the View Implementation

```
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;
```



Let Us Check the View Implementation

```
public class DJView implements ActionListener
  BeatObserver, BPMObserver {
   public DJView(ControllerInterface controller,
     BeatModelInterface model) {
     this.controller = controller;
     this.model = model;

     model.registerObserver((BeatObserver)this);
     model.registerObserver((BPMObserver)this);
   }
```

Let Us Check the View Implementation

```
public class DJView implements ActionListener
  BeatObserver, BPMObserver {
   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);
   }
```

Let Us Check the View Implementation

```
ublic class DJView implements ActionListener{
  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(); }
  }
  }….
```

Let Us Check the View Implementation

```
ublic class DJView implements ActionListener{
  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(); }
  }
  }….
```

Let Us Check the Controller Implementation

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

Let Us Check the Controller Implementation

```
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();
}
```

Let Us Check the Controller Implementation

```
public class BeatController implements
  ControllerInterface {

  BeatModelInterface model;
  DJView view;

public void start() {
   model.on();
   view.disableStartMenuItem();
   view.enableStopMenuItem();
}
public void stop() {
   model.off();
   view.disableStopMenuItem();
   view.enableStartMenuItem();
}
```

Let Us Check the Controller Implementation

```
public class BeatController implements
  ControllerInterface {

  BeatModelInterface model;
  DJView view;

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);
}
```

Required Reading

- Chapter 12 from “Head First Design Patterns”.