Design Patterns & Software Architecture
Software Architecture
MVC

dr. Joost Schalken-Pinkster

Windesheim University of Applied Science

The Netherlands

The contents of these course slides are based on:

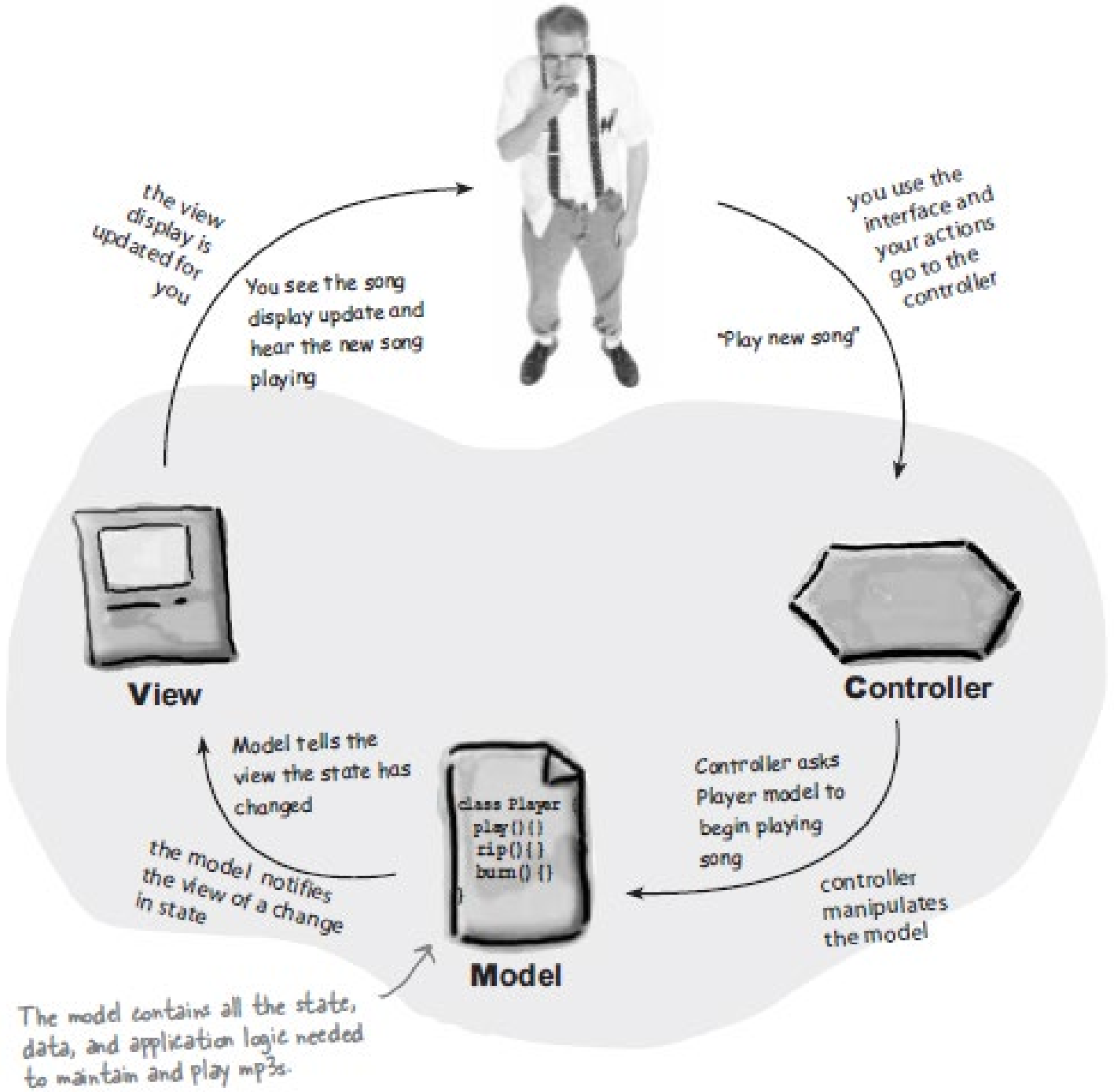
Jonathan Simon (2010) Design Patternsn: Week 6: MVC and Other Patterns. www.slideshare.net/jonsimon2/mvc-and-other-design-patterns
Jeroen Weber & Christian Köppe, *Course on Patterns and Frameworks*. Hogeschool Utrecht, 2013.
Leo Pruijt, *Course on Software Architecture*. Hogeschool Utrecht, 2010-2013.

Session overview

- Model View Controller
- Model 2 – MVC in a web-based world

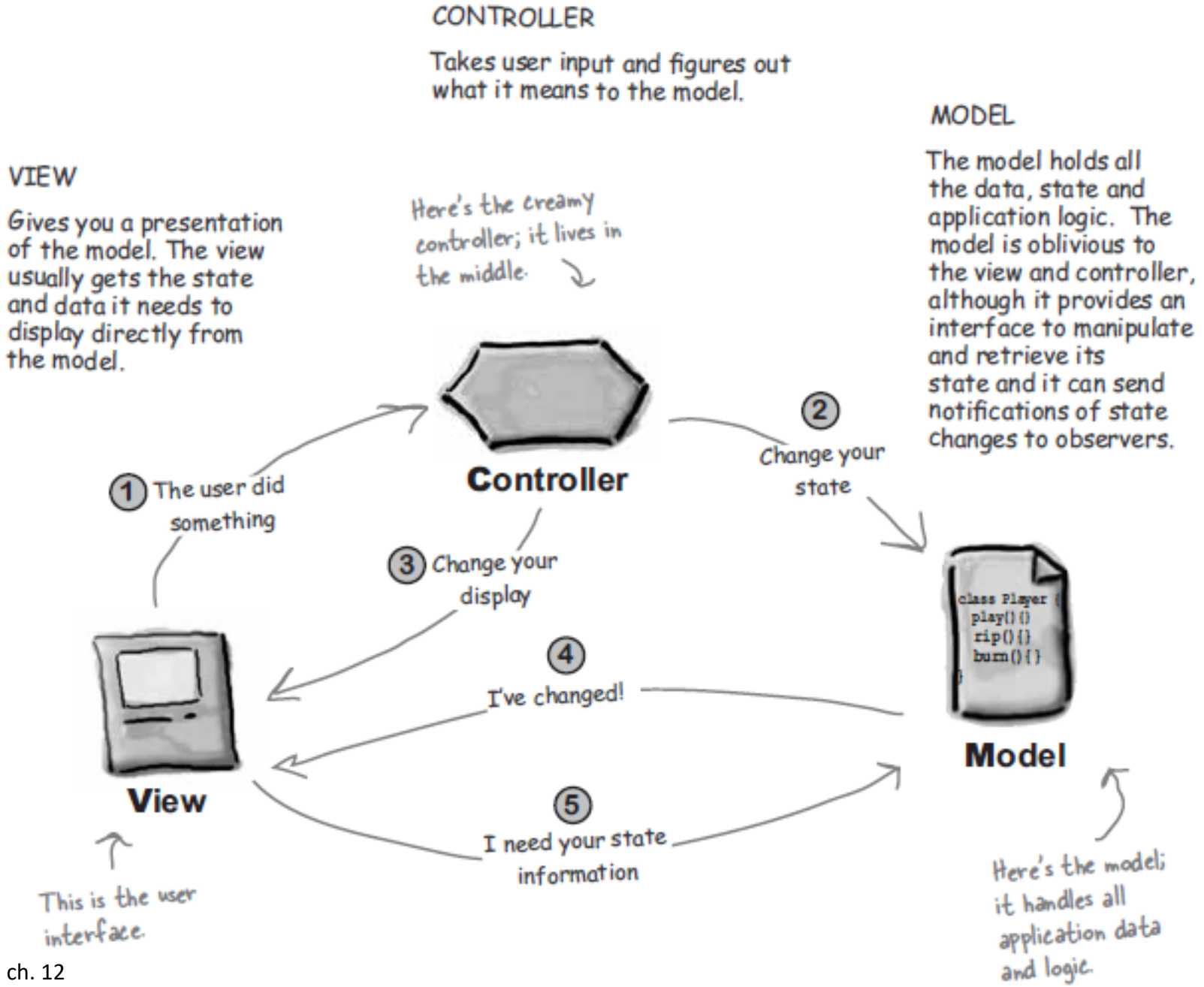
Model View Controller

MVC overview



Freeman & freeman (1995), Head First Design Patterns, ch. 12

MVC in detail

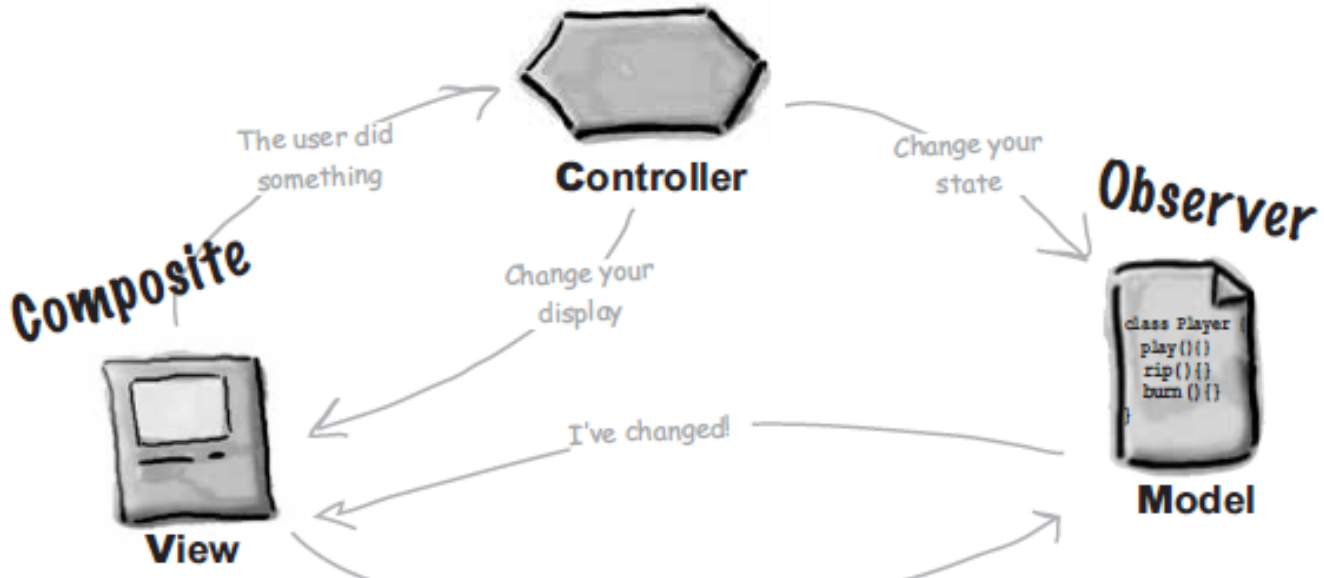


Freeman & freeman (1995), Head First Design Patterns, ch. 12

MVC & design patterns

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.

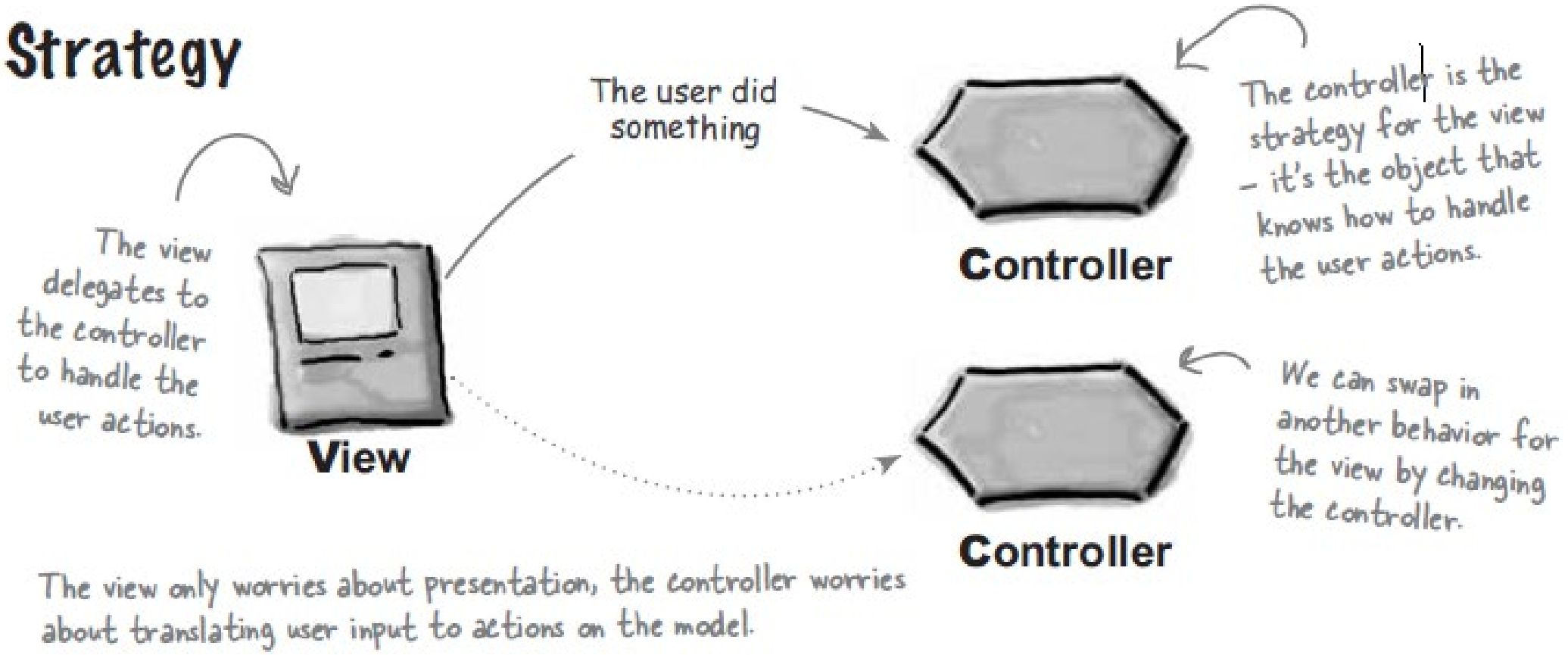


The display consists of a nested set of windows, panels, buttons, text labels and so on. Each display component is a composite (like a window) or a leaf (like a button). When the controller tells the view to update, it only has to tell the top view component, and Composite takes care of the rest.

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.

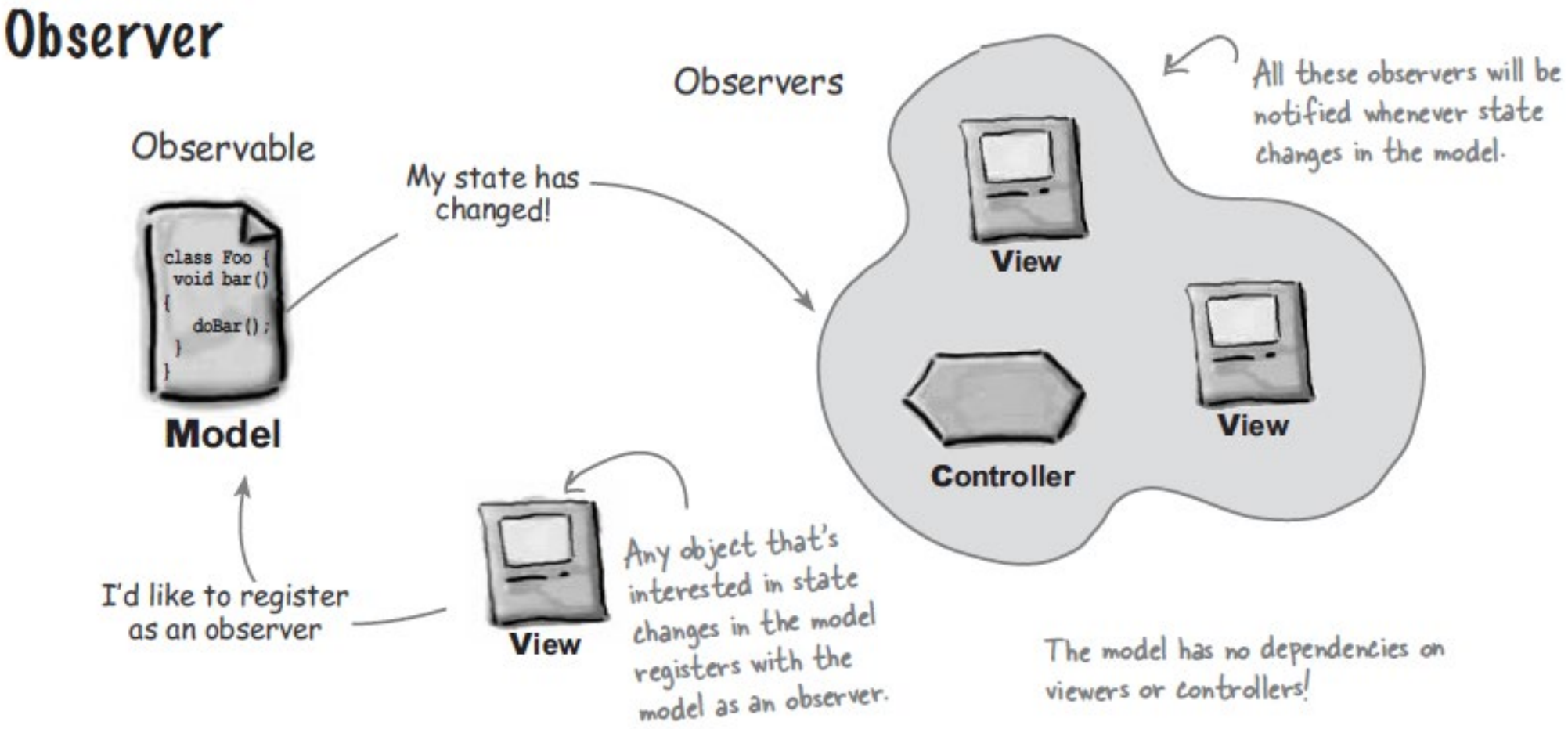
Freeman & freeman (1995), Head First Design Patterns, ch. 12

MVC & Strategy pattern



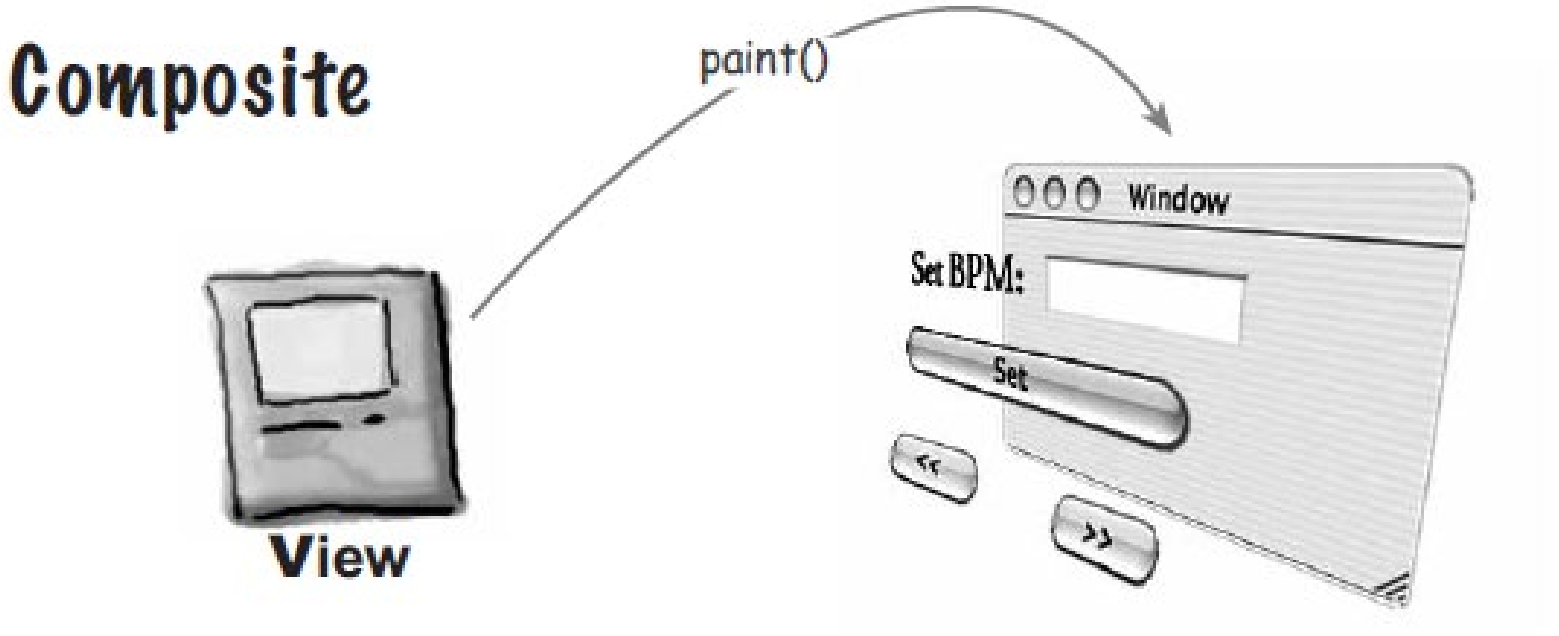
Freeman & freeman (1995), Head First Design Patterns, ch. 12

MVC &
Observer pattern



Freeman & freeman (1995), Head First Design Patterns, ch. 12

MVC & Composite pattern



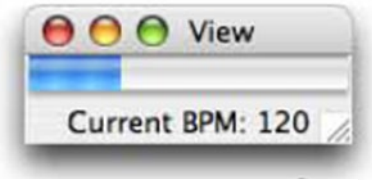
The view is a composite of GUI components (labels, buttons, text entry, etc). The top level component contains other components, which contain other components and so on until you get to the leaf nodes.

Freeman & freeman (1995), Head First Design Patterns, ch. 12

DJView

Two Different User Interfaces

- View – Shows BPM and Beat in real time
- Control – Enter BPM and click start button. Increase and Decrease buttons.



- Start and Stop buttons

Jonathan Simon (2010) Design Patternsn: Week 6: MVC and Other Patterns. www.slideshare.net/jonsimon2/mvc-and-other-design-patterns

DJView: Version 1

```
 //Fields
int bpm;
Sequencer sequencer;

public DJView() {
        //Display UI Elements
        CreateDisplayView();
        CreateControlView();

        //Enable/Disable UI State
        DisableStopMenuItem();
        EnableStartMenuItem();

        sequencer = new Sequencer();
}
```

DJView: Version 1

```
 public void StartButtonPushed() {
                sequencer.start();
                bpm = 90;
                sequencer.setTempoInBpm(bpm);
                DisableStartMenuItem();
                EnableStopMenuItem();
                DisplayBpm(bpm);
}
public void StopButtonPushed() {
                sequencer.turnOff();
                bpm = 0;
                EnableStartMenuItem();
                DisableStopMenuItem();
                DisplayBpm(bpm);
}
```

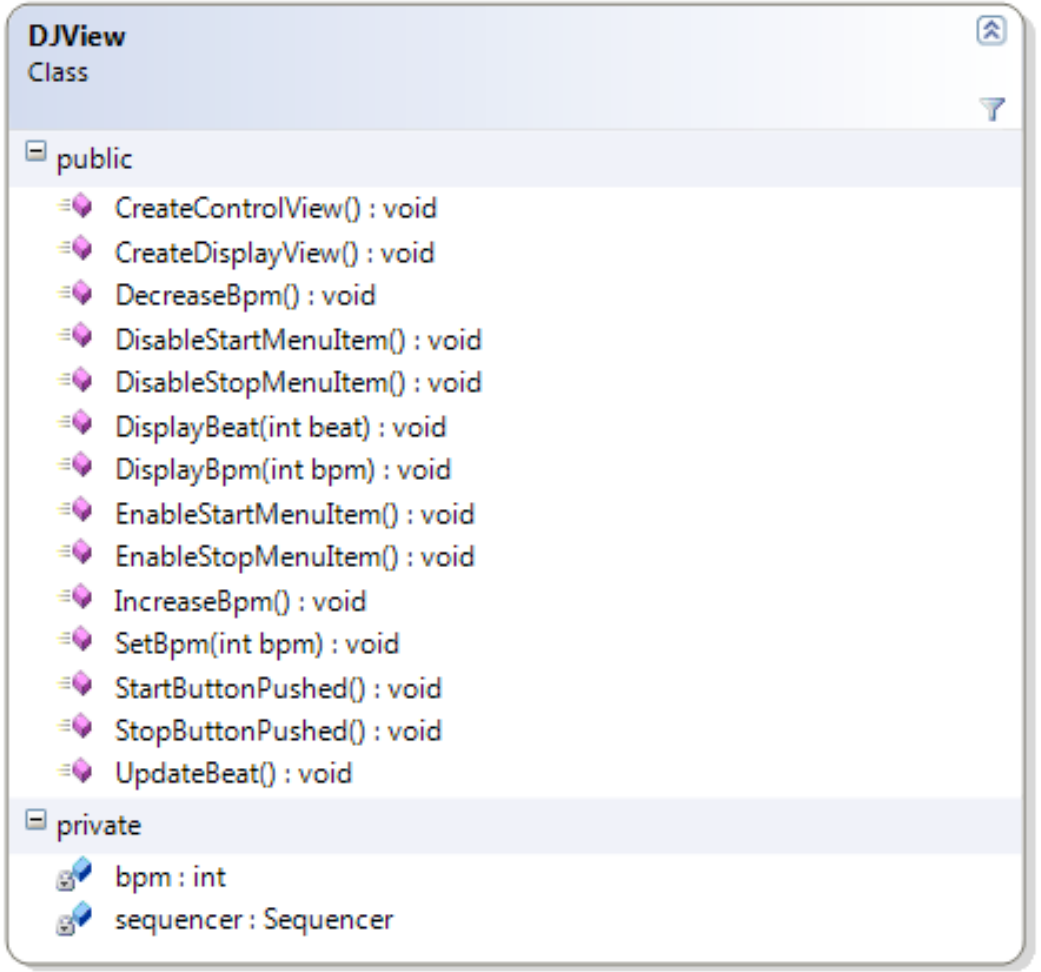
DJView: Version 1

```
public void SetBpm(int bpm) {
        this.bpm = bpm;
        sequencer.setTempoInBpm(bpm);
        DisplayBpm(bpm);
}
public void IncreaseBpm() {
        this.bpm = bpm + 1;
        sequencer.setTempoInBpm(bpm);
        DisplayBpm(bpm);
}
public void DecreaseBpm() {
        this.bpm = bpm - 1;
        sequencer.setTempoInBpm(bpm);
        DisplayBpm(bpm);
}
```

DJView: Version 1

```
public void UpdateBeat()  {

        while (true) {
                if (sequencer.BeatChanged)
                        DisplayBeat(sequencer.getBeat());
        }

}
```

DJView: Version 1



Requirement Change…

- Now we need another screen (ArtistView) to display BPM. (as well other information just as Song Name, Artist, Album Name, year, etc).
  - Note: ArtistView is not interested in a change in Beat.

- How can we display BPM?
  - We can modify function DisplayBpm() to update ArtistView and DJView whenever a change is made to BPM.
  - What happens when we create another screen, say AlbumView?

So..

- We have a few different screens (DJView, ArtistView, AlbumView) that are all interested whenever we have a change in data state (BPM or Beat).
  - Some screens are interested in change in BPM, others change Beat, and some changes in both.

- Currently, we have to update DJView whenever we want to add another screen that is interested in changes in BPM or Beat.

- What pattern is screaming to be used?

Creating a Subject

- We need a Subject that contains the data state that we know is changing.
- In this example, what should fields should go in our Subject?
- How many different events should our Subject expose?
- How do we “glue” our Observers to this Subject?

Subject = BeatModel


```
public class BeatModel  {
      int bpm;
      Sequencer sequencer;
      public void initialize() {
         sequencer = new Sequencer();
      }
      public void RegisterObserver(BeatObserver o) { }
      public void RemoveObserver(BeatObserver o) { }
      public void NotifyBeatObservers() { }

      public void RegisterObserver(BPMObserver o) { }
      public void RemoveObserver(BPMObserver o) { }
      public void NotifyBpmObservers() { }
}
```

BeatObserver and BPMObserver


```
public interface BeatObserver {
        void updateBeat();
}


public interface BPMObserver {
        void updateBPM();
}
```

DJView: Version 2

```
 //Fields
int bpm;
Sequencer sequencer;
public DJView() {
        //Display UI Elements
        CreateDisplayView();
        CreateControlView();

        //Enable/Disable UI State
        DisableStopMenuItem();
        EnableStartMenuItem();

        sequencer = new Sequencer();
}
```

Changes???

DJView: Version 2


```
//Fields
BeatModel model;
public DJView() {
        //Display UI Elements
        CreateDisplayView();
        CreateControlView();
        //Enable/Disable UI State

        DisableStopMenuItem();
        EnableStartMenuItem();

        model = new BeatModel();
        model.initialize();
}
```

DJView: Version 2

```
public void StartButtonPushed() {
        sequencer.start();
        bpm = 90;
        sequencer.setTempoInBpm(bpm);
        DisableStartMenuItem();
        EnableStopMenuItem();
        DisplayBpm(bpm);
}
public void StopButtonPushed() {
        sequencer.turnOff();
        bpm = 0;
        EnableStartMenuItem();
        DisableStopMenuItem();
        DisplayBpm(bpm);
}
```

Does DJView have a direct reference :
To sequencer?
To bpm field?

DJView: Version 2

```
public void StartButtonPushed() {
        model.on();

        DisableStartMenuItem();
        EnableStopMenuItem();

        DisplayBpm(bpm);
}

public void StopButtonPushed() {
        model.off();

        EnableStartMenuItem();
        DisableStopMenuItem();

        DisplayBpm(bpm);
}
```

What should we do with this DisplayBpm function?

What interfaces does DJView need to implement? Remember BeatModel…

DJView: Version 2

```
public class DJView implements BeatObserver, BPMObserver {
        public void updateBeat() {
                //Display change in Beat in the UI
        }

        public void updateBpm() {
                int bpm = model.getBpm();

                if (bpm == 0)
                        //Display "offline"
                else
                        //Display "Current BPM" with value
    }
  }
}
```

What are we missing?

DJView: Version 2

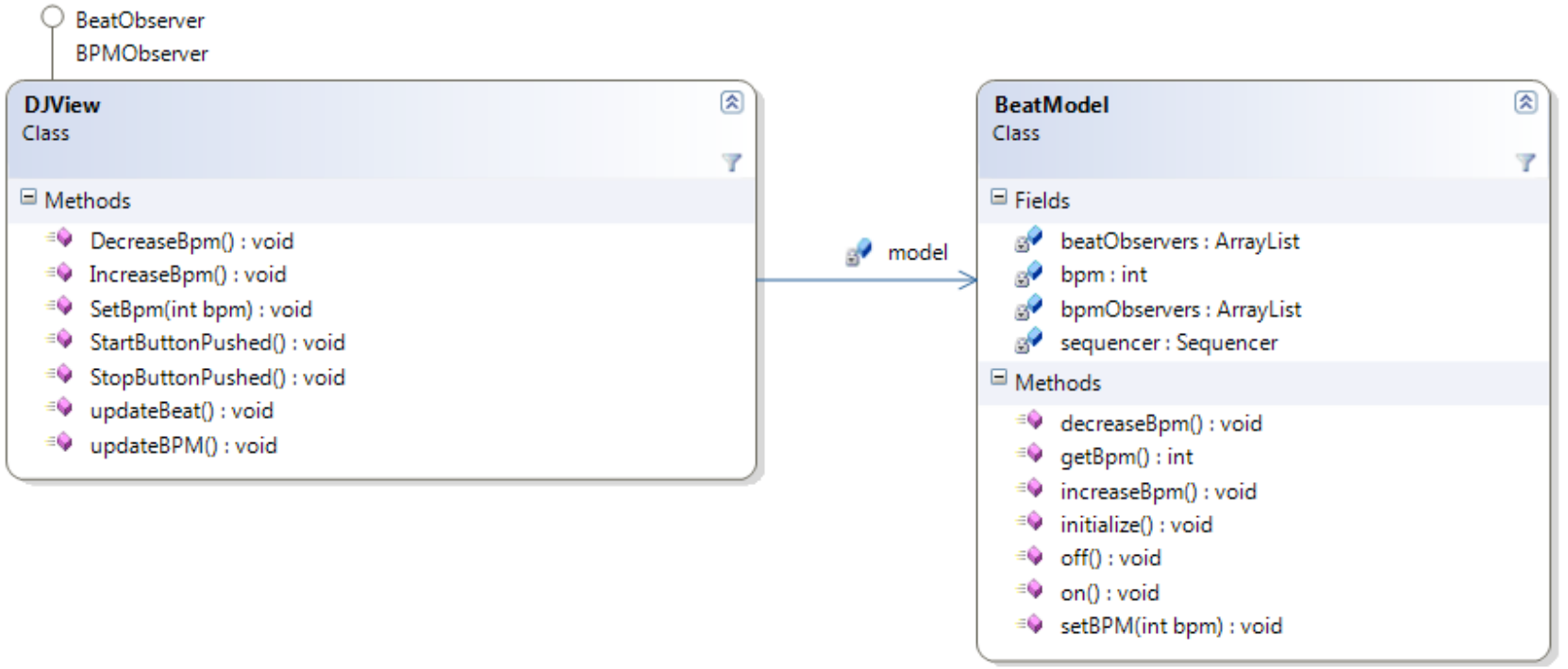
```
public DJView() implements BeatObserver, BPMObserver {

        // Display UI Elements
        CreateDisplayView();
        CreateControlView();

        //Enable/Disable UI State
        DisableStopMenuItem();
        EnableStartMenuItem();

        model = new BeatModel();
        model.initialize();

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

Jonathan Simon (2010) Design Patterns: Week 6: MVC and Other Patterns. www.slideshare.net/jonsimon2/mvc-and-other-design-patterns

DJView: Version 2

```
public void StartButtonPushed() {
        model.on();
        DisableStartMenuItem();
        EnableStopMenuItem();
}
public void StopButtonPushed() {
        model.off();
        EnableStartMenuItem();
        DisableStopMenuItem();
}
public void SetBpm(int bpm) {
        model.setBPM(bpm);
}
public void IncreaseBpm() {
        model.increaseBpm();
}
public void DecreaseBpm() {
        model.decreaseBpm();
}
```

Version 2



Notice beatObservers and bpmObservers in BeatModel.

View and Model

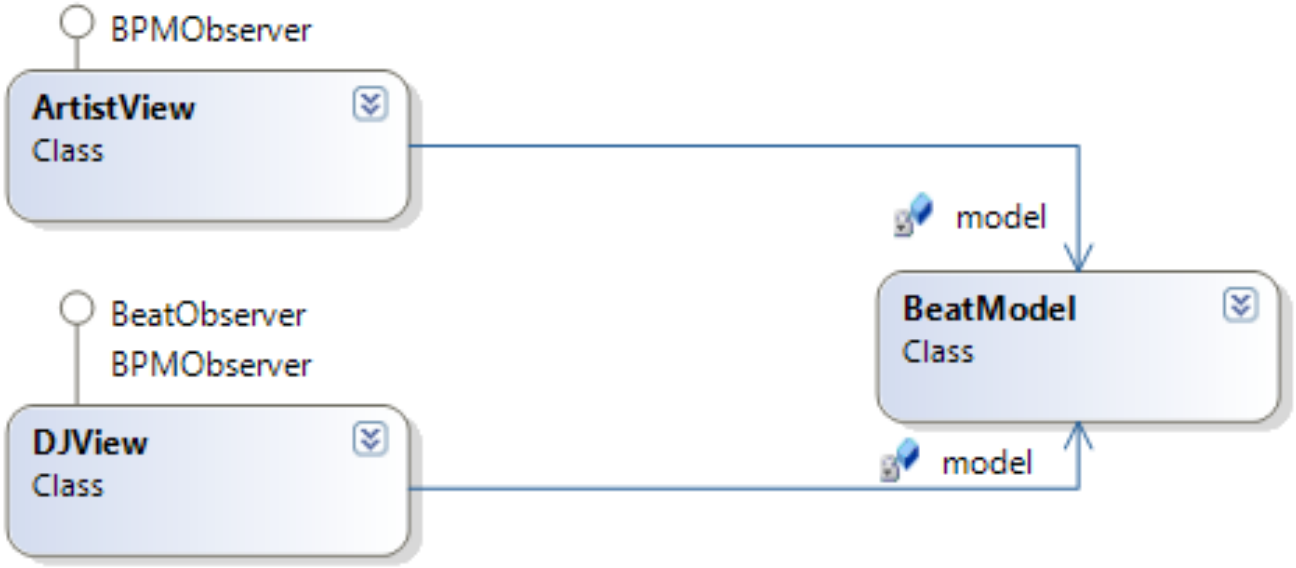
We now have a View and a Model

The View is also an Observer. It is interested in changes in the Model.

When the Model has a data change, the View gets notified (updateBPM). The View then gets data from the Model (getBPM).

We can have multiple Views all interested in this Model.

Multiple Views with one model



Question: Does the Model know any specific details about the UI elements on the View?

Definition (Partial): Model

- Contains the data, state, and application logic.
- State changes can occur internally or externally. (more on this later).
- When a change of state occurs, the Model sends a notification message.
- The Model doesn’t have any direct references to the Views that need it. (Since the Views registers themselves with the Model).

Definition (Partial): View

- Graphic User Interface.
- It is the “window” to the model.
- View registers with the Model for notification of certain events.
- When notified by the Model, the View asks the Model for state information.
- View must have a direct reference to the Model in order to get data. (can also be an interface the Model uses)

Requirement Change

We need to change all of these views from a Windows application to a Web application.

We would like to minimize the amount of code that we need to re-write for the new user interface.

Let’s look at the DJView code again..

DJView: Version 2

```
public DJView() {
        CreateDisplayView();
        CreateControlView();
        DisableStopMenuItem();
        EnableStartMenuItem();
        //Hidden: Create model, initialize and register observers
}
 public void StartButtonPushed() {
        model.on();
        DisableStartMenuItem();
        EnableStopMenuItem();
}
public void updateBeat() { ... }
public void updateBPM()  { ... }
```

We know that we cannot re-use UI elements when we change Views.

Challenge

We want to separate elements from DJView that vary from the things that do not vary.

- What varies?
- What does not vary?

What varies

- Internals of CreateDisplayView() + CreateControlView()
- Internals of all of the Disable/Enable methods
- The internals of the updateBeat and updateBPM that has to do with updating UI
- How events are associated with UI element

What does not vary

- The sequence of events. For example, StartButtonPushed has a specific set of steps.
- Calling the Model to do something (on(), off())
- Registering View with the Model
- The internals of the updateBPM that has to do with getting data from the Model

Inheritance

- Base View class that contains:
  - Reference to BeatModel
  - Template Pattern to define known set of steps; use abstract methods for pieces that need to be implemented by sub class.
- Each subclass just defines the UI specific elements.
- What problems will we encounter?

Inheritance (cont)

- We can’t inherit form the base View and the specific UI form (which contains a lot of functionality for interacting with UI elements)
- We would need to inherit from the Base View and wrap the specific UI element (Windows or Web form) and then expose what we need. (Adapter Pattern!)
- This would be painful…

Composition

- View just contains the UI elements and functions that manipulate the UI elements (What Varies)
- View associates UI element with a handler. (What Varies)
- The UI handler may do a special sequence of events (which may manipulate a combination of model changes and UI updates). This is What Not Varies.
- Here is our separation of View and Controller (the New Guy).

What about the Model?

The View interacts with the Model in two ways:

- Sends a message to the Controller to update the Model. (setBPM, increase, decrease)
- Gets notified when Model changes (updateBeat, updateBPM)

DJView: Version 3 (Pt 1)


```
public class DJView implements BeatObserver, BPMObserver {
    BeatModel model;
    BeatController controller;

    public DJView(BeatModel model, BeatController controller)
    {
       this.model = model;
       this.controller = controller;
       model.RegisterObserver((BeatObserver)this);
       model.RegisterObserver((BPMObserver)this);
    }

    public void updateBeat()  { . . }
    public void updateBPM() { . . }
```

Note: The only reason that View needs Model is for registering of events and getting state from Model.

DJView: Version 3 (Pt 2)

```
public void StartButtonPushed() {
        controller.start();
}

public void StopButtonPushed() {
        controller.stop();
}

public void SetBpm(int bpm) {
        controller.setBPM(bpm);
}

public void IncreaseBpm() {
        controller.increaseBPM();
}

public void DecreaseBpm() {
        controller.decreaseBPM();
}
```

The View delegates all actions to the Controller.

DJView: Version 3 (Pt 3)

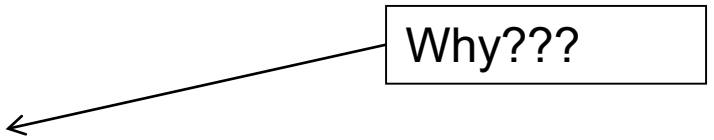
Lastly, we have all of the functions that directly manipulate UI elements in the View. These functions are called externally by the Controller:

```
public void CreateDisplayView() { }
public void CreateControlView() { }
public void DisableStopMenuItem() { }
public void EnableStopMenuItem() { }
public void DisableStartMenuItem() { }
public void EnableStartMenuItem() { }
```

NOTE: If we have to port this View from Windows to Web, here is where the crux of our changes will occur.

BeatController

```
public class BeatController {
  BeatModel model;
  DJView view;

  public BeatController(BeatModel model)
    this.model = model;
    view = new DJView(model, this);
    view.CreateDisplayView();
    view.CreateControlView();

    view.DisableStopMenuItem();
    view.EnableStartMenuItem();

     model.initialize();
}
```



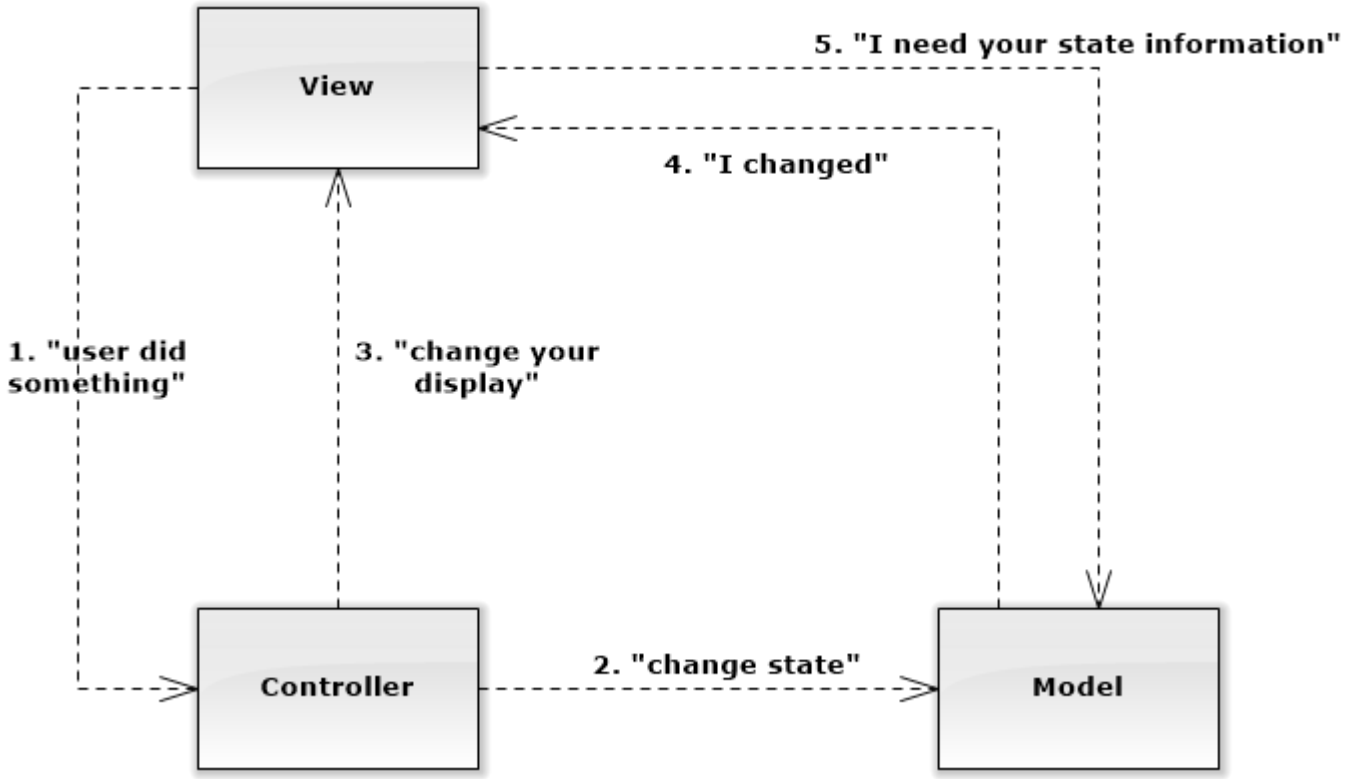
BeatController

```
public void start() {
        model.on();
        view.DisableStartMenuItem();
        view.EnableStopMenuItem();
}

public void stop() {
        model.off();
        view.EnableStartMenuItem();
        view.DisableStopMenuItem();
}

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

Controller sends a message to Model to update data…and sends a message back to the View.

Note: Not showing the Model here. It is exactly the same!

Model View Controller in a birds eye view



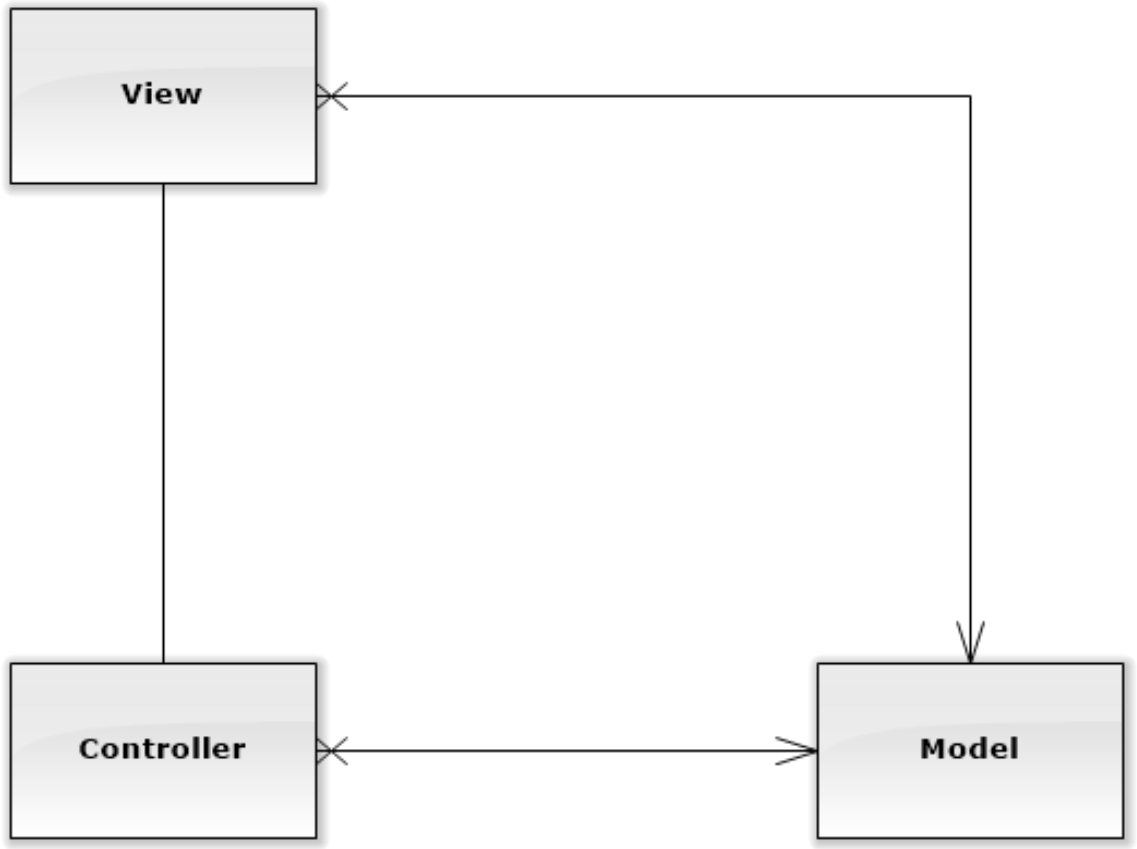
Eric Freeman et al. (2004), *Head First Design Patterns*, O'Reilly 2004. p 529-548.

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:**
   then 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.

Model View Controller (MVC)

- **Intent:**
  The MVC is a pattern of patterns that allows you to create a visual presentation (user interface) of information & business processes in the system while, separating the presentation from the underlying information & business processes.
  MVC separates, at the same time, the presentation into a view (the visual part) and the controller (the interaction) part.

- **Structure:**



- **Participants**:
  - View
    - A view is a (visual) representation of its model.
      - It would ordinarily highlight certain attributes of the model and suppress others. It is thus acting as a presentation filter.
    - A view is attached to its model and gets the data from the model by asking questions.
    - A view forwards all interaction between the user and the view to the controller.
  - Controller
    - Receives the input from the user
    - Decides what to do with the input from the user.
  - Model
    - Holds all data, state and application logic.
    - Does not know anything about Controller or View.
    - Provides interface to manipulate and retriev state.
    - Provides interface to alert other classes about state change

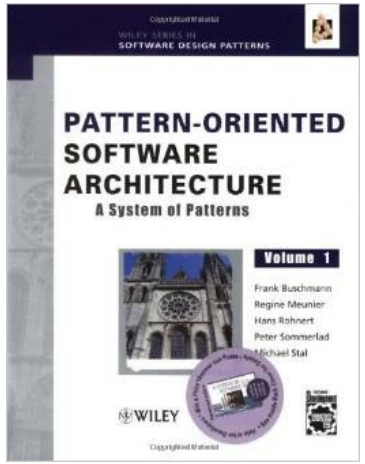
- **Consequences**:
  - Benefits:
    - Multiple views of the same model.
    - Synchronized views.
    - ‘Pluggable’ views and controllers.
    - Exchangeability of ‘look and feel’.
    - Framework potential.

Model-View-Controller history

- Created as "*an obvious solution to the general problem of giving users control over their information as seen from multiple perspectives*." by Trugve Reenskaug in 1979.

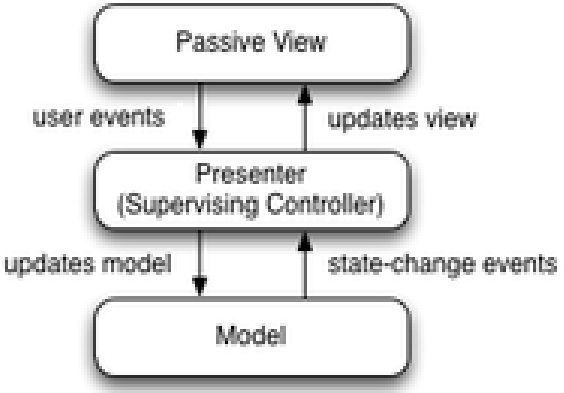
- First used in the Smalltalk-80 programming language in 1979.

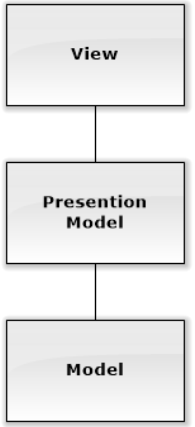
- MVC was first described as pattern in 'Pattern-Oriented Software Architecture' by Buschmann in 1996.



Trygve M. H. Reenskaug (1979), Models Views Controllers. Xerox Parc, December 10th, 1979.
Buschmann , et al. (1996) *Pattern-Oriented Software Architecture. : A System of Patterns*, Volume 1, Wiley, August 16th, 1996.

- **Implementation:**
  A number of variations on the MVC pattern exist.
  - Model View Presenter
    - all presentation logic is pushed to the presenter
    - the view cannot talk to the model any longer
  - Presentation Model (als ModelView View Model)
    - Presentation Model puts the state and behavior of the view into a model class that is part of the presentation.
    - The Presentation Model coordinates with the domain layer and provides an interface to the view that minimizes decision making in the view.
    - The view either stores all its state in the Presentation Model or synchronizes its state with Presentation Model frequently

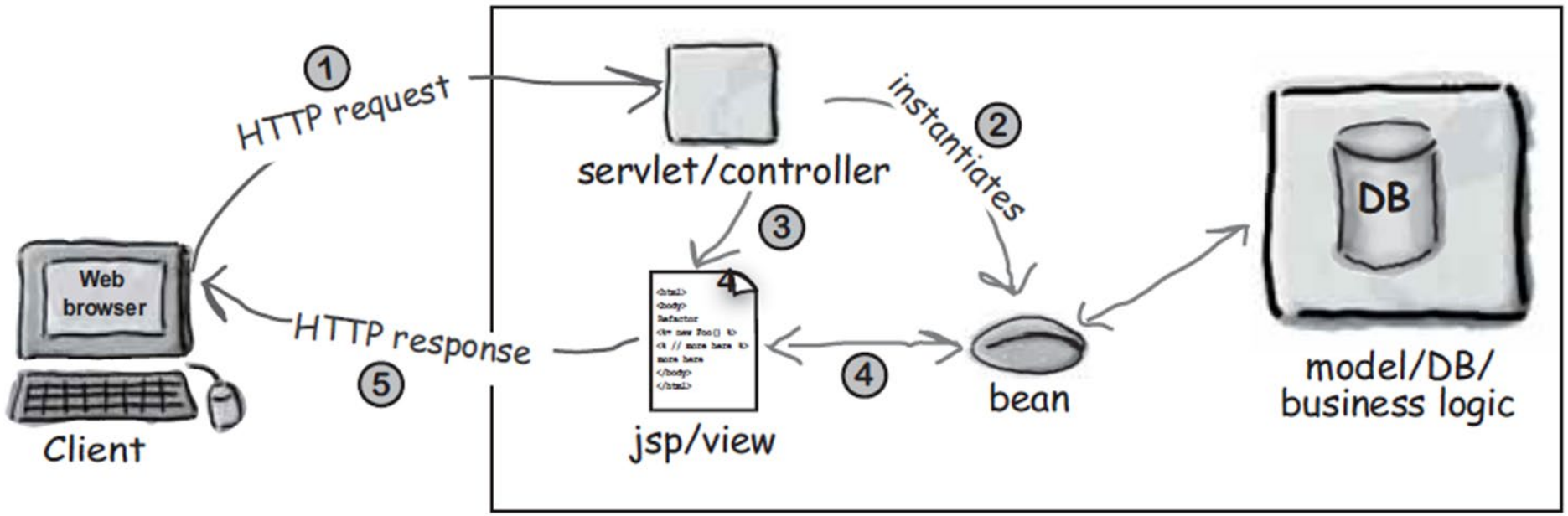




Wikipedia (2014), Model View Presenter. Wikipedia 2014. http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter
Martin Fowler. Presentation Model, 2004. http://martinfowler.com/eaaDev/PresentationModel.html

Model 2

Model View Controller in a web-based world


1
HTTP request
servlet/controller
instantiates
2
3
DB
Web
browser
HTTP response
5
4
bean
model/DB/
business logic
Client
jsp/view

1. **You make an HTTP request, which is received by a servlet.**
   Using your web browser you make an HTIP 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 u p 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 tor further actions.
5. **The view returns a page to the browser via HTTP.**
   A page is retumed to the browser, where it is displayed as the view. The user submits further requests, which are processed in the same fashion.

Advantages Model 2

Separate production responsibilities:

- **JSP**s (views) are created by user interface designers

- **Servlet**s (controllers) and the model are created by the software engineers.

Reading

Required reading for next lecture:

- Chapter 12 (Patterns of Patterns) of Head First Design Patterns.