CS 635 Advanced Object-Oriented Design & Programming
Fall Semester, 2019
Doc 18 Value Object & MVC
Nov 26, 2019

Copyright ©, All rights reserved. 2019 SDSU & Roger Whitney,
5500 Campanile Drive, San Diego, CA 92182-7700 USA.
OpenContent (http://www.opencontent.org/opl.shtml) license
defines the copyright on this document.

References

Pattern-Oriented Software Architecture: A System of Patterns v1, Buschman, Meunier, Rohnert, Sommerlad, Stal, 1996, pp 125-143

Patterns of Enterprise Application Architecture, Folwer, 2003, pp 330-386

Core J2EE Patterns: Best Practices and Design Strategies, 2nd, Alur, Crupi, Malks, 2003

Values in Object Systems, Baumer, Riehle

App Architecture: iOS Application Patterns in Swift , Eidhof, Gallagher, Kugler, 2018

Value Object

Values versus Objects

Values

integers,
real numbers,
strings

No alterable state
No side effects

One 5 is the same as all 5's

Objects

new Person("Roger")

Alterable state
Side effects

Pointer equality

Values in Programs

social security numbers
credit card numbers
money
date
account numbers
width
height
weight
colors

Model abstractions from problem domain

Often
- Measurements
- Identifiers

Can use primitive types (ints, float) for value, but ...

Money Example

```
int bankBalance = 5;
```

But what about

- Different currencies
- Rounding errors

Money Example

So make a Money class

But then have side effects

Value Object Pattern

For values in applications that need more that primitive types

Create a class for the abstraction

Make the objects immutable

Swift - Value Objects

struct

- Like a class
- Fields
- methods
- Default is immutable
- Copied on assignment

let

```
let x = Person() // x is immutable
```

MVC & Related Web Patterns

Model-View-Controller (MVC)

Context

Interactive application with human-computer interface

Forces

Same data may be displayed differently

Display & application must reflect data changes immediately

UI changes should be easy and even possible at runtime

Changing look & feel or port to other platforms should not affect core application code

Solution

Divide application into three parts:

Model (core application)
View (display, output)
Controller (user input)

Model

Core application code

Contains a list of observers (view or controller)

Has a broadcast mechanism to inform views of a change

Same mechanism as subject in Observer pattern

View

Displays information to user

Obtains data from model

Each view has a controller

Controller

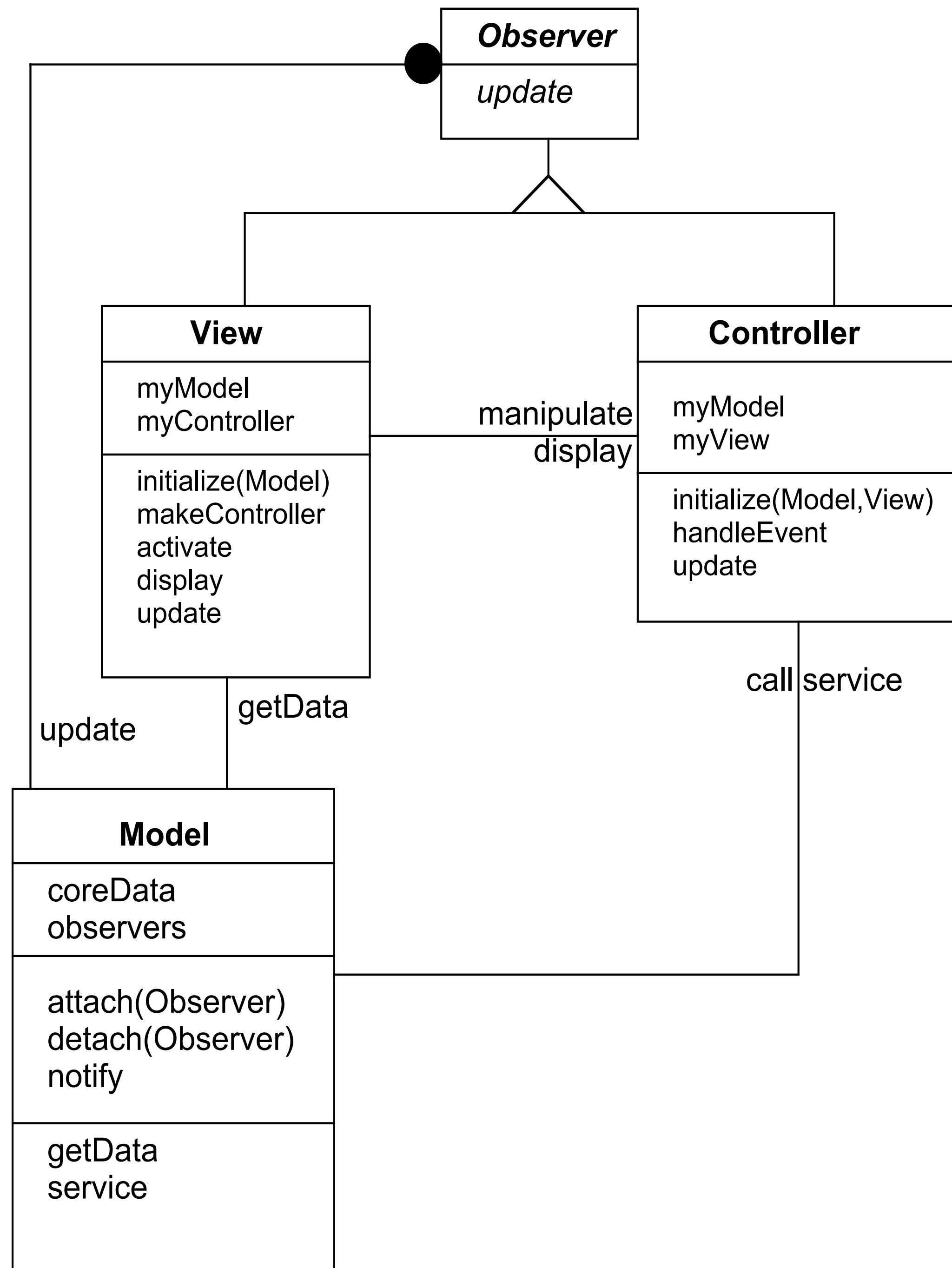
Handles input from user as events

- Keystrokes
- Mouse clicks
- Mouse movements

Maps each event to proper action on model and/or view

Many people misinterpret what a controller does

Structure



View + Controller

Make up the user interface

Some GUI frameworks combine these

VW Smalltalk contains both, but hides controller from programmer

Some Existing Smalltalk Controllers & Views

Controllers	Views
ApplicationDialogController	ActionButtonView
BasicButtonController	AutoScrollingView
ClickWidgetController	BasicButtonView
ColoredAreaController	BooleanWidgetView
ComboBoxButtonController	CheckButtonView
ComboBoxInputBoxController	ClickWidget
ComboBoxListController	ComboBoxButtonView
ControllerWithMenu	ComboBoxInputFieldView
ControllerWithSelectMenu	ComboBoxListView
DataSetController	ComposedTextView
DataSetControllerProxy	DataSetView
DelayingWidgetController	DefaultLookCheckButtonView
DrawingController	DefaultLookRadioButtonView
DropDownListController	EmulationScrollBar
EmulatedDataSetController	GeneralSelectionTableView
EmulatedSequenceController	HierarchicalSequenceView
EmulationScrollBarController	HorizontalTabBarView
HierarchicalSequenceController	HorizontalTopTabBarView
InputBoxController	InputFieldView

Architecture Patterns

How to structure an application

GOF patterns

- Not at the architecture level

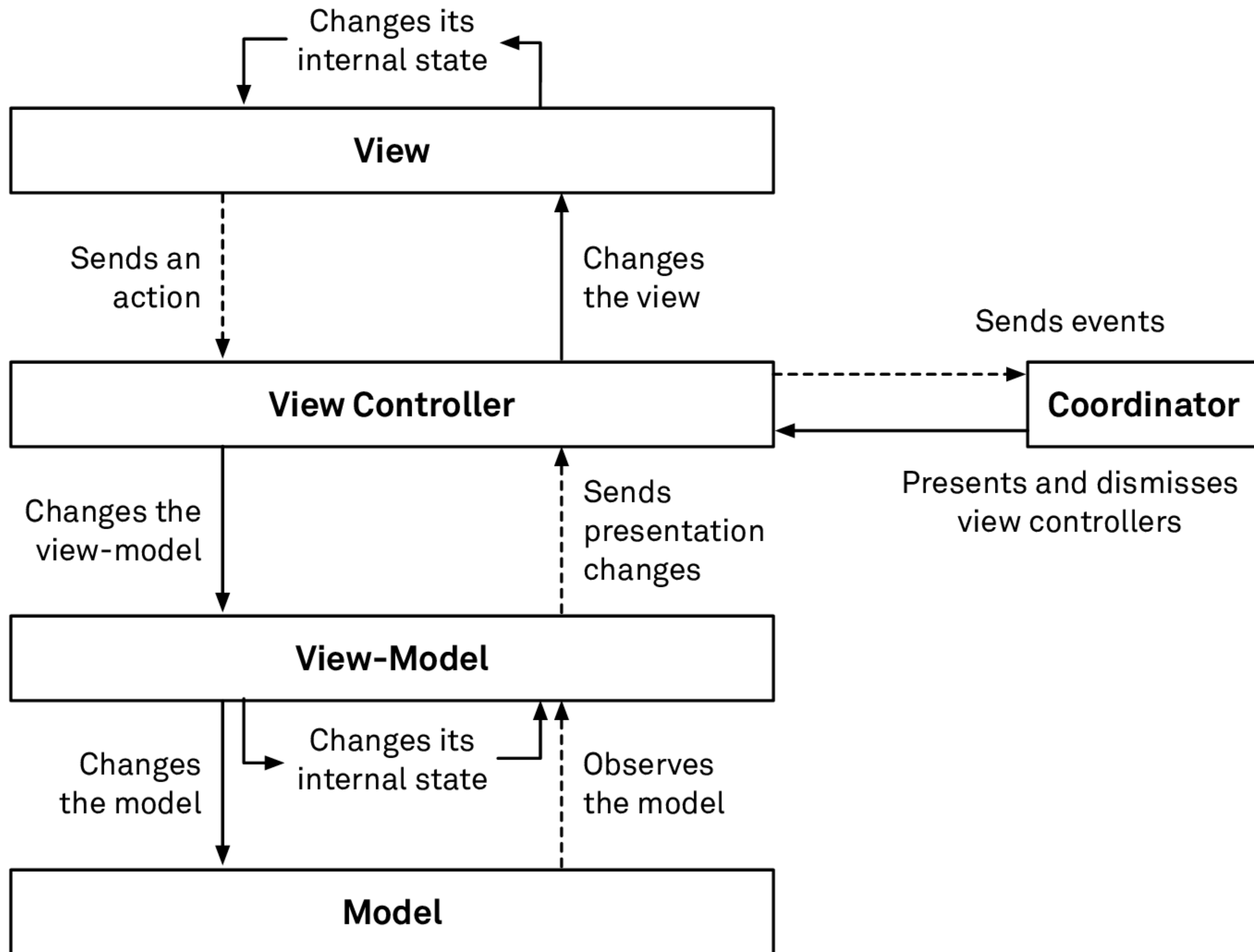
Some iOS Variants on MCV

Model-View-ViewModel+Coordinator

Model-View-Controller+ViewSate

ModelAdapter-ViewBinder

Model-View-ViewModel+Coordinator



Model-View-ViewModel+Coordinator

Each screen (scene) has
- View
- ViewController
  - No view state

Coordinator
- Logic to change from one scene to another

View-model
- No compile time references to views or controllers
- Has properties that will be displayed in views
- Properties are from model object via transformations (ReactiveX)

Web related Patterns

The Patterns

Template View
Page Controller
Front Controller
Intercepting Filter
Composite View
Transform View

Template View

Template View

Renders information into HTML by embedding markers in an HTML page

Server Pages

Java

```
<html>
<body>
<%! int x = 1; %>
<%! int y = 2; %>
If we add <%= x %> to <%= y %> we will get <%= x + y %>
</body>
</html>
```

PHP, Smalltalk Server pages

Template View

Advantage

- Graphic designers can generate view
- Rapid development for small projects

Disadvantages

- Poor module structure
- Leads to mixing model, controller and view logic
- Leads to repeated code in files
- Many programming tools do not work on template files

Template View - Some common Issues

Conditional display

```
<p>Please pay your bill
<If user.isDeadBeat()> <B> </IF>
now.
<IF use.isDeadBeat()> </B> </IF>
```

Iteration over collection

Given a list create a drop down menu

Use View Helper to separate out processing logic

Some Background

Servlets

Generate HTML in code

Return result

```
public class HelloWorld extends HttpServlet {

    public void doGet(HttpServletRequest request,
            HttpServletResponse response)
            throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<head>");
        out.println("<title>Hello World!</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Hello World!</h1>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

Smalltalk Example

```
doGet: aRequest response: aResponse

   aResponse write: '<HTML><BODY>GET<BR>
      Hello world</BODY></HTML>'.

doPost: aRequest response: aResponse

   aResponse write: '<HTML><BODY>POST<BR>
      Hello world</BODY></HTML>'.
```

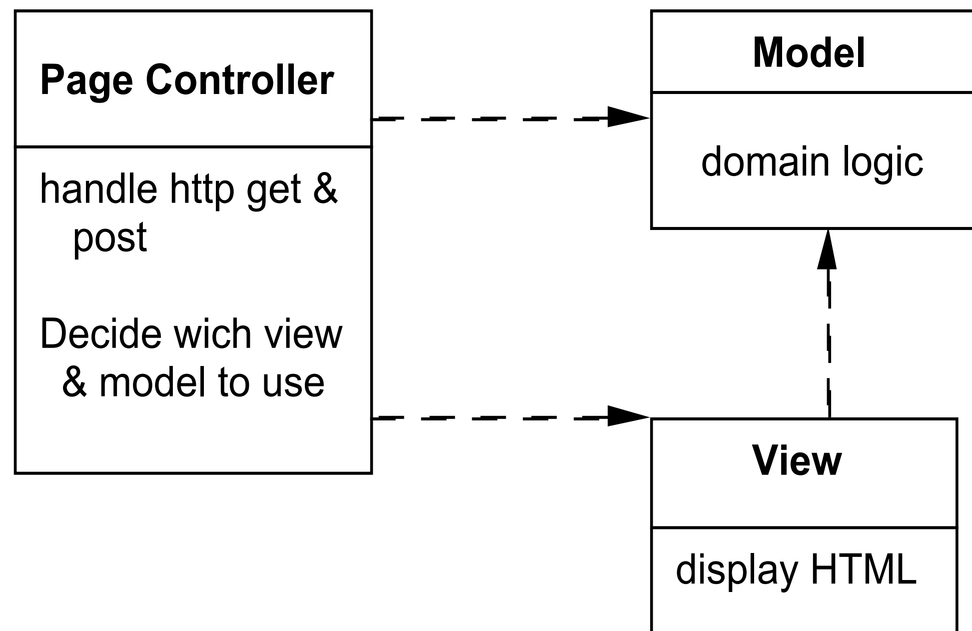
Clojure

```
(defn login-table-ui [redid pass]
  [:table
   [:thead
    [:tr
     [:th {:key :label-col} ]
     [:th {:key :data-col} ]
     ]]
   [:tbody
    [:tr
     [:td "Red ID"]
     [:td [textfield redid]]]
    [:tr
     [:td "Password"]
     [:td [passwordfield pass]]]
    ]])
```

Page & Front Controller

Page Controller

An object that handles a request for a specific page or action on a Web page



Decodes URL

Extracts all form data and gets all data for the action

Create and invoke model objects, pass all relevant data to model

Determine which view should display the result page and forward model information to it

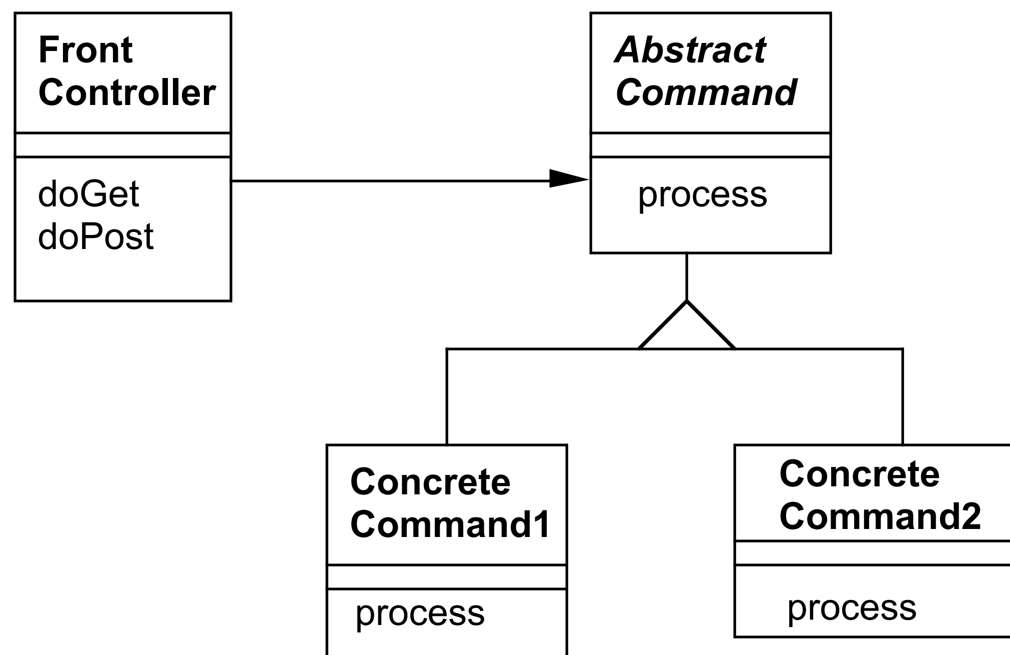
Each page or url on the site has a different page controller

Front Controller

A controller that handles all requests for a Web site

Forces

- Avoid duplicate control logic
- Apply common logic to multiple requests
- Separate system processing logic from view
- Have a centralized controlled access point into system



How it works

All requests to a Web site are directed to the FrontController

The FrontController

- Examines the URL & form data
- Determines the correct command to handle the request
- Create the correct command
- Forwards the request to the command

Command is part of controller so it uses a separate view

Pros & Cons

Disadvantage

More complex than Page Controller

Advantages

Only one controller has to be configured into the web server

A command object handles only one request so command does not have to be thread-safe

Commands can be added dynamically (if controller uses reflection to create a command object)

Factor out common code from multiple Page Controllers

```
(defapi service-routes
  {:swagger {:ui "/swagger-ui"
             :spec "/swagger.json"}}

  (context "/class/api" []

    (GET "/allCourses" []
      :return      [Object]
      (ok (data/public-courses)))

    (POST "/registerCourse" []
      :return      Object
      :body-params [redid :- String, password :- String, courseid :- Long]
      (log/info (str "registering " redid " in " courseid))
      (let [result (data/register-class redid password courseid)]
        (if (contains? result :ok)
          (ok (data/public-course-with-schedule courseid))
          (bad-request result))))
```

Intercepting Filter
Composite View
Transform View

Intercepting Filter

You want to manipulate a request and a response before and after the request is processed

Forces

You want

- Common processing across requests like
  - Logging
  - Compressing
  - Data encoding

Pre & post processing components loosely coupled with core request-handling services

Pre & post processing components independent of each other

Solution

Add a chain of decorators (filters) that end on the Front Controller

Composite View

Build a view from atomic components while managing content and layout independently

Forces

You want subview, such as headers, footers and tables reused in different pages

You want to avoid directly embedding and duplicating subviews in multiple pages

You have content in subviews that frequently change or are subject to access control

Solution

Use the composite pattern on views.

A page then is created as a composite object of views.

Transform View

A view that processes domain data elements by element and transforms them into HTML

Given a domain object, MusicAlbum, how to generate a web page for the object?

- Use Template View
- Convert object into html

Converting object into html

One could add toHtml to the object

```
MusicAlbum ragas = new MusicAlbum.find(“Passages”);
String html = ragas.toHtml();
```

But

- Domain object is coupled to view language
- Provides only one way to display object

Using Transforms

Use XML and XSLT

- Convert domain object to XML
- Use XSLT to convert XML into HTML

Now we can produce many different views of the object without changing the object

More complex that converting object to HTML

Transform View Verses Template View

- Template View
  - More tools support
  - No language to learn
- Transform View
  - Easier to avoid domain logic in view
  - Testing can be done without Web server
  - Easier to make global changes to Web site