Review

- Debugging: Read, Ask, Search, Post
- Rails Pitfalls: Too much code in Controller, Some extra code in View
- Agile – prototypes, iterate with customer
- BDD – Design of app before implementation
- User Story – all stakeholders write what features want on 3x5 cards
- Cucumber – magically turns 3x5 card user stories into acceptance tests for app
- Day Thought: Reasonable Person Principle

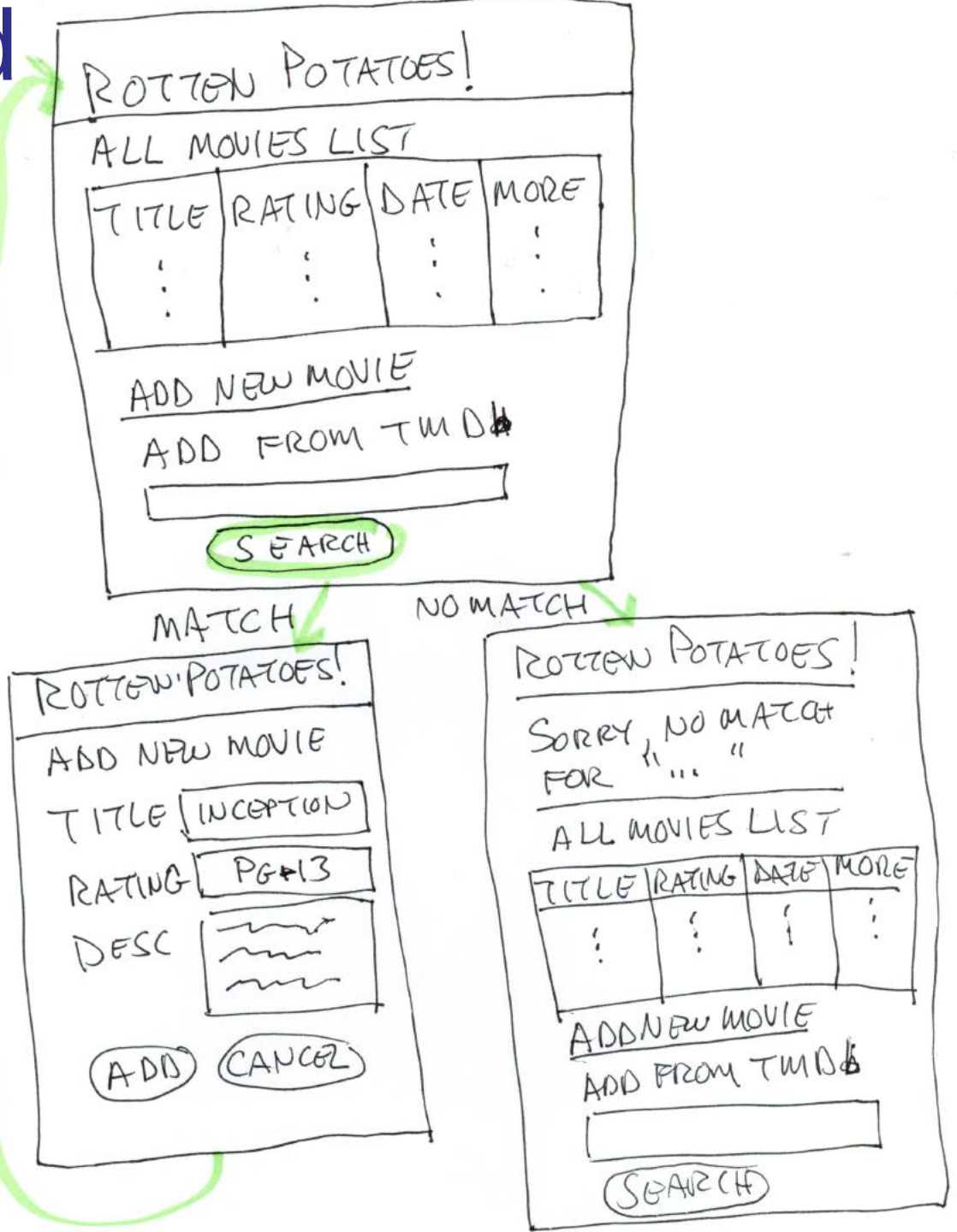
Outline

- Enhancing Rotten Potatoes Again (§ 5.6)
- Explicit vs. Implicit and Imperative vs. Declarative Scenarios (§ 5.7)
- Fallacies & Pitfalls, BDD Pros & Cons (§ 5.8- § 5.9)
- HTML+CSS (§ 2.3)
- Template Views and Haml (§ 2.8)
- Summary & Reflections: SaaS Architecture (§ 2.9)
- Comments on Comments: time?
- Day Thought: Building Credibility: time?

Integrated with The Movie Database (TMDb)

- New Feature: Populate from TMDb, versus enter information by hand
- Need to add ability to search TMDb from Rotten Potatoes home page
- Need LoFi UI and Storyboard

Storyboard TMDb



Search TMDb User Story (Figure 4.6 in Engineering Long Lasting SW)

Feature: User can add movie by searching in The Movie Database (TMDb)

As a movie fan

So that I can add new movies without manual tedium

I want to add movies by looking up their details in TMDb

Scenario: Try to add nonexistent movie (sad path)

Given I am on the RottenPotatoes home page

Then I should see "Search TMDb for a movie"

When I fill in "Search Terms" with "Movie That Does Not Exist"

And I press "Search TMDb"

Then I should be on the RottenPotatoes home page

And I should see "'Movie That Does Not Exist' was not found in TMDb."

Haml for Search TMDb page
(Figure 4.7 in Engineering Long Lasting SW)

```
-# add to end of app/views/movies/index.html.haml:

%h1 Search TMDb for a movie

= form_tag :action => 'search_tmdb' do

  %label{:for => 'search_terms'} Search Terms
  = text_field_tag 'search_terms'
  = submit_tag 'Search TMDb'
```

http://pastebin/18yYBVbC

Haml expansion last 2 lines

- Haml

  = text_field_tag 'search_terms'

  = submit_tag 'Search TMDb’

- Turns into

<label for='search_terms'>Search Terms</label>

<input id="search_terms" name="search_terms" type="text" />

- for attribute of label tag matches id attribute of input tag, from text_field_tag helper

Cucumber?

- If try Cucumber, it fails
- MoviesController#search_tmdb is controller action that should receive form, yet doesn't exist in movies_controller.rb
- Should use Test Driven Development (next Chapter) to implement method search_tmdb
- Instead, to let us finish sad path, add fake controller method that always fails

Fake Controller Method: Will Fail Finding Movie (Figure 4.8)

```
# add to movies_controller.rb, anywhere inside
#  'class MoviesController < ApplicationController':


def search_tmdb
  # hardwired to simulate failure
  flash[:warning] = "'#{params[:search_terms]}' was
    not found in TMDb."
  redirect_to movies_path
end
```

http:/pastebin/smwxv70i

Trigger Fake Controller when form is POSTed (Figure 4.8)

```
# add to routes.rb, just before or just after 'resources :movies' :


# Route that posts 'Search TMDb' form
post '/movies/search_tmdb'
```

http://pastebin/FrfkF6pd

- Try Cucumber now

Happy Path of TMDb

- Find an existing movie, should return to Rotten Potatoes home page
- But some steps same on sad path and happy path
- How make it DRY?
- Background means steps performed before *each* scenario

TMDb with 2 Scenarios

http://pastebin/icQGrYCV

Feature: User can add movie by searching for it in The Movie Database (TMDb)

As a movie fan

So that I can add new movies without manual tedium

I want to add movies by looking up their details in TMDb

Background: Start from the Search form on the home page

Given I am on the RottenPotatoes home page

Then I should see "Search TMDb for a movie"

Scenario: Try to add nonexistent movie (sad path)

When I fill in "Search Terms" with "Movie That Does Not Exist"

And I press "Search TMDb"

Then I should be on the RottenPotatoes home page

And I should see "'Movie That Does Not Exist' was not found in TMDb."

Scenario: Try to add existing movie

When I fill in "Search Terms" with "Inception"

And I press "Search TMDb"

Then I should be on the RottenPotatoes home page

And I should see "Inception"

And I should see "PG-13"

Demo

- Add feature to search for movie in TMDb
  - Note: This will be a sad path, in that won't find it
  - Will use fake method
    (until next week when implement it using TDD)
- (Or can look at screencast:

  http://vimeo.com/34754766)

Summary

- New feature => UI for feature, write new step definitions, even write new methods before Cucumber can color steps green
- Usually do happy paths first
- Background lets us DRY out scenarios of same feature
- BDD/Cucumber test behavior; TDD/RSpec in next chapter is how write methods to make all scenarios pass

Explicit vs. Implicit and Imperative vs. Declarative Scenarios

(Engineering Long Lasting Software § 5.7)

David Patterson

Explicit vs. Implicit Scenarios

- Explicit requirements usually part of acceptance tests => likely explicit user stories and scenarios
- Implicit requirements are logical consequence of explicit requirements, typically integration testing
  - Movies listed in chronological order or alphabetical order?

Imperative vs. Declarative Scenarios

- Imperative: specifying logical sequence that gets to desired result
  - Initial user stories usually have lots of steps
  - Complicated `When` statements and `And` steps
- Declarative: try to make a Domain Language from steps, and write scenarios declaratively
- Easier to write declaratively as create more steps and more Rails experience

Example Imperative Scenario

- Given I am on the RottenPotatoes home page
- When I follow "Add new movie"
- Then I should be on the Create New Movie page
- When I fill in "Title" with "Zorro"
- And I select "PG" from "Rating"
- And I press "Save Changes"
- Then I should be on the RottenPotatoes home page
- When I follow "Add new movie"
- Then I should be on the Create New Movie page
- When I fill in "Title" with "Apocalypse Now"
- And I select "R" from "Rating"
- And I press "Save Changes"
- Then I should be on the RottenPotatoes home page
- And I should see "Apocalypse Now" before "Zorro"

Only 1 step specifying behavior;
Rest are really implementation.
But BDD should be about design

Example Declarative Scenario

- Given I have added "Zorro" with rating "PG-13”
- And I have added "Apocalypse Now" with rating "R”
- And I am on the RottenPotatoes home page sorted by title
- Then I should see "Apocalypse Now" before "Zorro" on the Rotten Potatoes home page

Declarative Scenario Needs New Step Definitions

```
Given /I have added "(.*)" with rating
    "(.*)"/ do |title, rating|
  Given %Q{I am on the Create New
    Movie page}
  When  %Q{I fill in "Title" with
    "#{title}"}
  And   %Q{I select "#{rating}" from
    "Rating"}
  And   %Q{I press "Save Changes"}
end
Then /I should see "(.*)" before "(.*)"
    on (.*)/ do |string1, string2, path|
  Given %Q{I am on #{path}}
  regexp = Regexp.new
    ".*#{string1}.*#{string2}"
  page.body.should =~ regexp
end
```

- As app evolves, reuse steps from first few imperative scenarios to create more concise and descriptive declarative scenarios
- Declarative scenarios focus attention on feature being described and tested vs. steps needed to set up test

Which is TRUE about implicit vs. explicit and declarative vs. imperative scenarios?

- ☐ As you get more experience with user stories, you will write many more declarative scenarios
- ☐ Explicit scenarios usually capture integration tests
- ☐ Declarative scenarios usually capture implementation as well as behavior
- ☐ Explicit requirements are usually defined with imperative scenarios and implicit requirements are usually defined with declarative scenarios

Fallacies & Pitfalls, BDD Pros & Cons, End of Chapter 5

(Engineering Long Lasting Software §5.8- §5.9)

David Patterson

Pitfalls

- Customers who confuse mock-ups with completed features
  - May be difficult for nontechnical customers to distinguish a polished digital mock-up from a working feature
- Solution: LoFi UI on paper clearly *proposed* vs. implemented

Pitfalls

- Sketches without storyboards
  - Sketches are static
  - Interactions with SaaS app = sequence of actions over time
- "Animating" the Lo-Fi sketches helps prevent misunderstandings before turning stories are into tests and code
  - "OK, you clicked on that button, here's what you see; is that what you expected?"

Pitfalls

- Adding cool features that do not make the product more successful
  - Customers reject what programmers liked
  - User stories help prioritize, reduce wasted effort

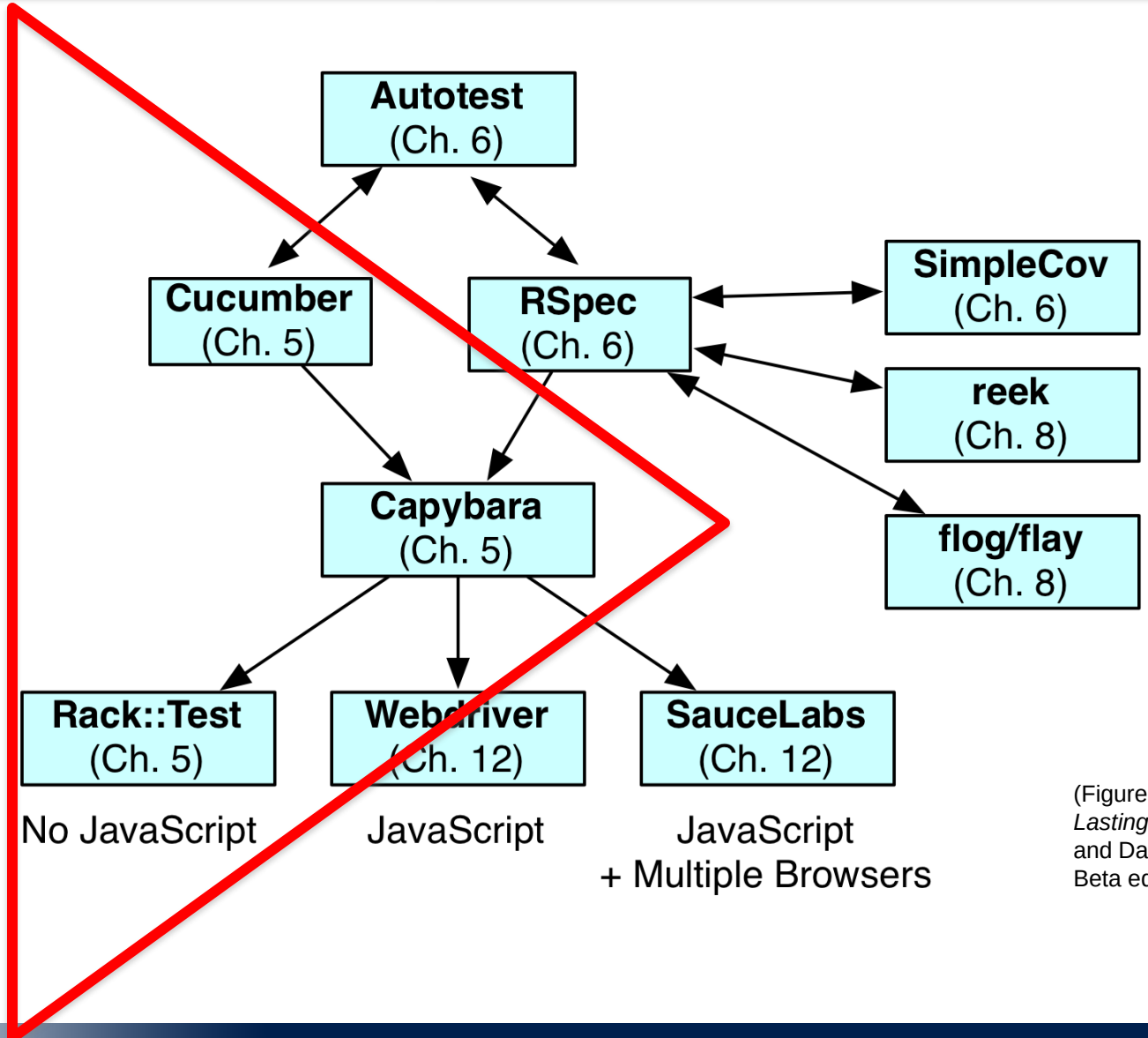
Pitfalls

- Trying to predict what code you need before need it
  - BDD: write tests *before* you write code you need, then write code needed to pass the tests
  - No need to predict, wasting development

Pitfalls

- Careless use of negative expectations
  - Beware of overusing “Then I should not see….”
  - Can’t tell if output is what want, only that it is not what you want
  - Many, many outputs are incorrect
  - Include positives to check results “Then I should see …”

Testing Tools in Book



(Figure 5.11, *Engineering Long Lasting Software* by Armando Fox and David Patterson, Beta edition, 2012.)

Which statement is FALSE about Lo-FI UI and BDD?

- ☐ The purpose of the Lo-Fi UI approach is to debug the UI before you program it
- ☐ A BDD downside is requiring continuous contact with customers, which may not be possible
- ☐ A BDD downside is that it may lead to a poor software architecture, since focus is on behavior
- ☐ None are false; all three above are true

Pros and Cons of BDD

- Pro: BDD/user stories - common language for all stakeholders, including nontechnical
  - 3x5 cards
  - LoFi UI sketches and storyboards
- Pro: Write tests before coding
  - Validation by testing vs. debugging
- Con: Difficult to have continuous contact with customer?
- Con: Leads to bad software architecture?
  - Will cover patterns, refactoring 2nd half of course

Behavior Driven Design

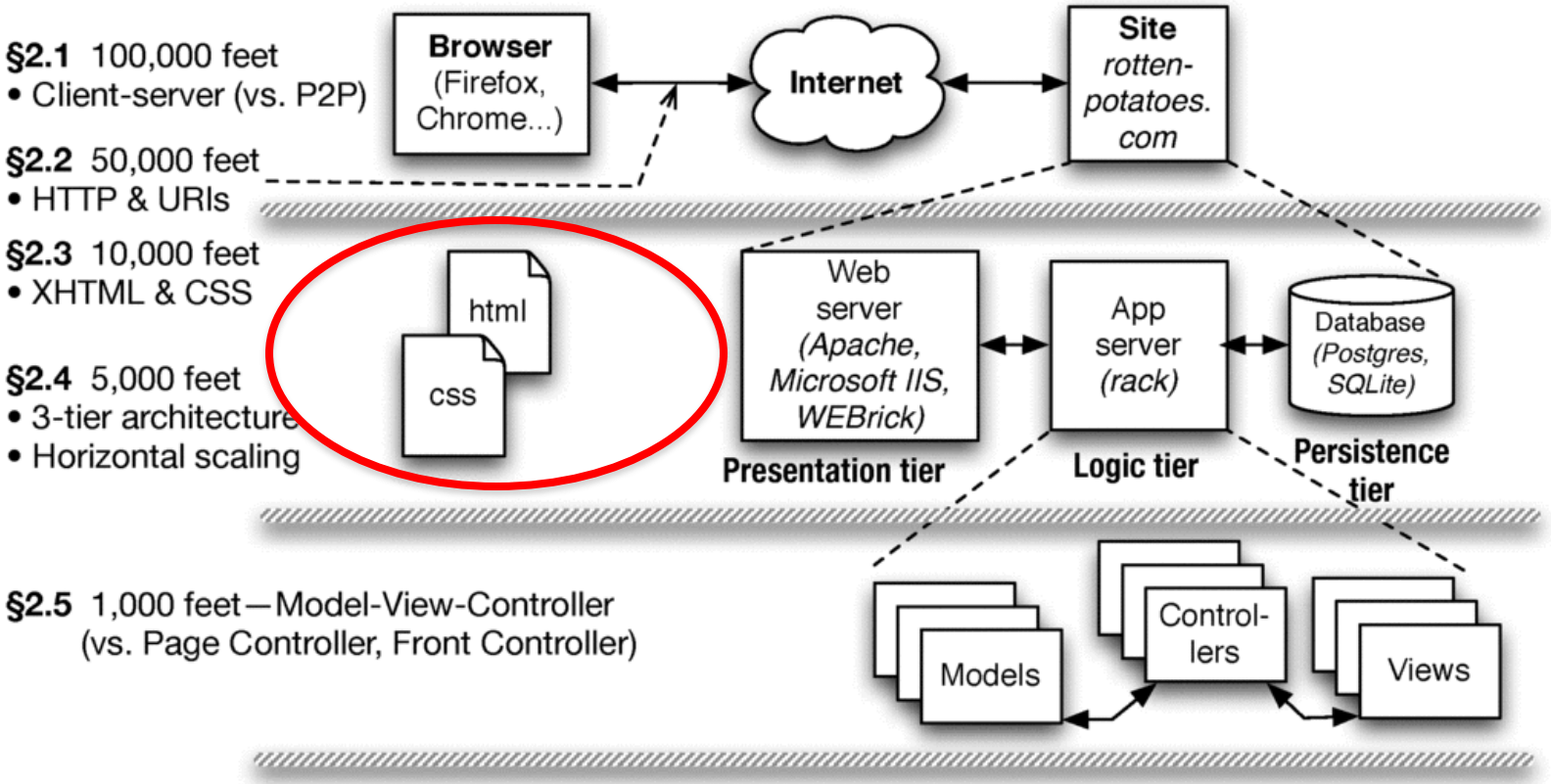
(Figure 5.12, *Engineering Long Lasting Software* by Armando Fox and David Patterson, Beta edition, 2012.)

- Doesn't feel natural at first
- Rails tools make it easier to follow BDD
- Once learned BDD and had success at it, no turning back
  - 2/3 Alumni said BDD/TDD useful in industry

HTML+CSS

Engineering Long Lasting Software § 2.3

Armando Fox


§2.1 100,000 feet
• Client-server (vs. P2P)
§2.2 50,000 feet
• HTTP & URIs
§2.3 10,000 feet
• XHTML & CSS
§2.4 5,000 feet
• 3-tier architecture
• Horizontal scaling
Browser (Firefox, Chrome...)
Internet
Site rotten-potatoes.com
html
css
Web server (Apache, Microsoft IIS, WEBrick)
App server (rack)
Database (Postgres, SQLite)
Presentation tier
Logic tier
Persistence tier
§2.5 1,000 feet—Model-View-Controller
(vs. Page Controller, Front Controller)
Models
Control-lers
Views
§2.6 500 feet: Active Record models (vs. Data Mapper)
§2.7 500 feet: RESTful controllers (Representational State Transfer for self-contained actions)
§2.8 500 feet: Template View (vs. Transform View)
• Active Record
• Data Mapper
• REST
• Template View
• Transform View

Introduction
This article is a review of the book Dietary Preferences of Penguins, by Alice Jones and Bill Smith. Jones and Smith's controversial work makes three hard-to-swallow claims about penguins:
First, that penguins actually prefer tropical foods such as bananas and pineapple to their traditional diet of fish
Second, that tropical foods give penguins an odor that makes them unattractive to their traditional predators

```
<h1>Introduction</h1>
<p>
  This article is a review of the book
  <i>Dietary Preferences of Penguins</i>,
  by Alice Jones and Bill Smith. Jones and Smith's
  controversial work makes three hard-to-swallow claims
  about penguins:
</p>
<ul>
  <li>
   First, that penguins actually prefer tropical foods
   such as bananas and pineapple to their traditional diet
   of fish
  </li>
  <li>
   Second, that tropical foods give penguins an odor that
   makes them unattractive to their traditional predators
  </li>
</ul>
...
```

Introduction

This article is a review of the book *Dietary Preferences of Penguins*, by Alice Jones and Bill Smith. Jones and Smith's controversial work makes two hard-to-swallow claims about penguins:

- First, that penguins actually prefer tropical foods such as bananas and pineapple to their traditional diet of fish
- Second, that tropical foods give penguins an odor that makes them unattractive to their traditional predators

...

```
<h1>Introduction</h1>
<p>
This article is a review of the book
<i>Dietary Preferences of Penguins</i>,
by Alice Jones and Bill Smith. Jones
and Smith's controversial work makes
three hard-to-swallow claims about
penguins:
<ul>
<li>
First, ...
```

HTML ~1.0

- Descendant of IBM's Generalized Markup Language (1960's) via SGML (Standard Generalized Markup Language, 1986)
- Document = Hierarchical collection of *elements*
  - inline (headings, tables, lists...)
  - embedded (images, JavaScript code...)
  - forms—allow user to submit simple input (text, radio/check buttons, dropdown menus...)
- Each element can have *attributes* (many optional) and some elements also have *content*
  - of particular interest: *id* and *class* attributes, for *styling*

Cascading Style Sheets

- Idea: *visual appearance* of page described in a separate document (*stylesheet*)
  - accessibility
  - branding/targeting
  - separate designers' & developers' concerns
- *Current best practice: HTML markup should contain **no** visual styling information*

How does it work?

- `<link rel="stylesheet" href="http://..."/>` (inside `<head>` element) says what stylesheet goes with this HTML page
- HTML `id` & `class` attributes important in CSS
  - *id* must be ***unique within this page***
  - same *class* can be attached to many elements

```
<div id="right" class="content">
  <p>
   I'm Armando.  I teach CS169 and do
   research in the AMP Lab and Par Lab.
  </p>
</div>
```

Selectors identify specific tag(s)

```
<div class="pageFrame" id="pageHead">
  <h1>
    Welcome,
    <span id="custName">Armando</span>
    <img src="welcome.jpg" id="welcome"/>
  </h1>
</div>
```

- tag name: `h1`
- class name: `.pageFrame`
- element ID: `#pageHead`

} both of these match the outer *div* above. Don't do this!

- tag name & class: `div.pageFrame`
- tag name & id: `img#welcome` (usually redundant)
- descendant relationship: `div .custName`
- Attributes *inherit* browser defaults unless overridden

Which CSS selector will select ***only*** the word “bar” for styling:

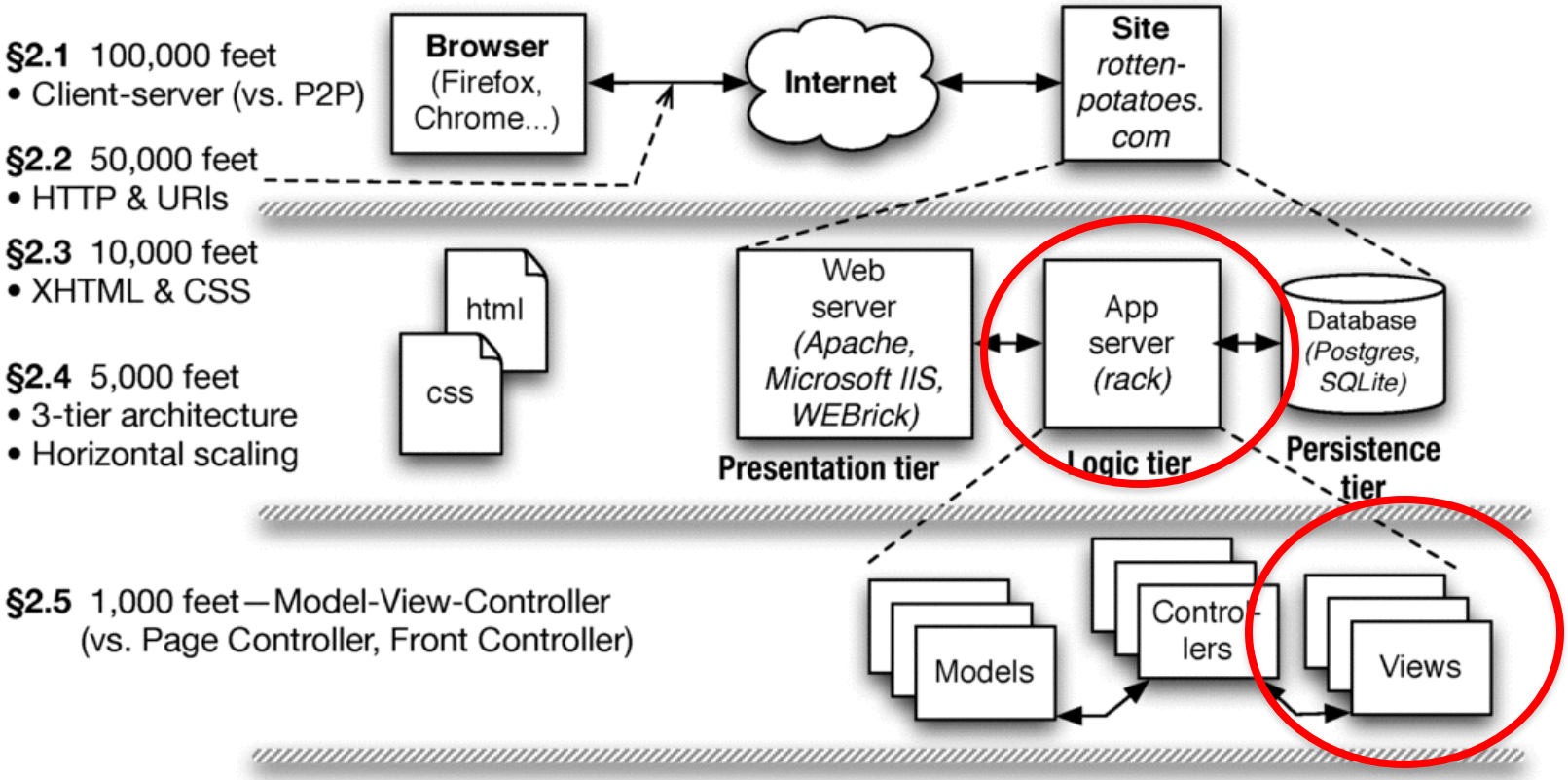
```
<p class="a">foo,
 <span class="a">bar<span></p>
```

- ☐ `span.a`
- ☐ `p  .a`
- ☐ `.a  span`
- ☐ All of these

Template Views and Haml

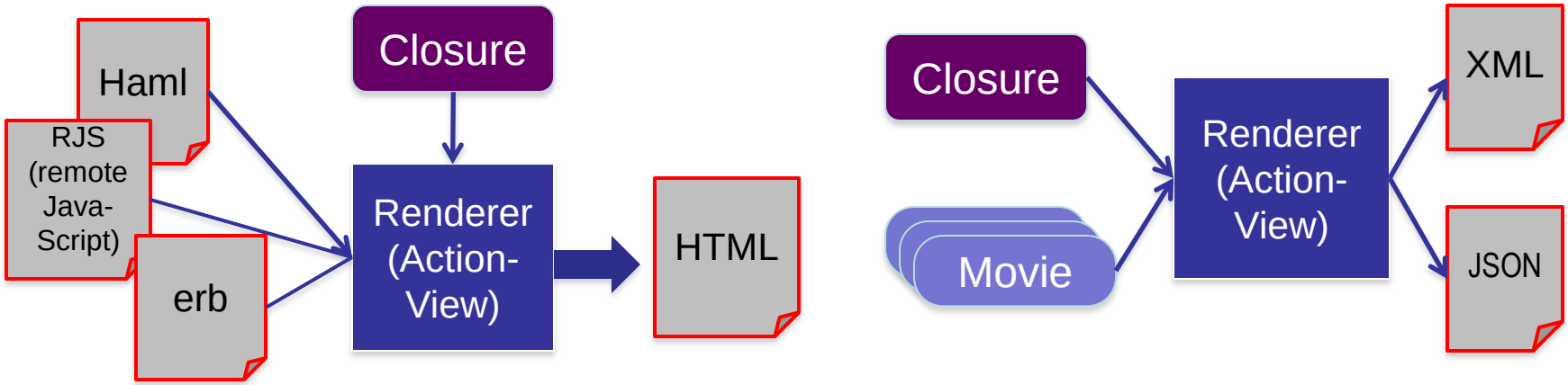
Engineering Long Lasting Software § 2.8

David Patterson


§2.1 100,000 feet
• Client-server (vs. P2P)
§2.2 50,000 feet
• HTTP & URIs
§2.3 10,000 feet
• XHTML & CSS
§2.4 5,000 feet
• 3-tier architecture
• Horizontal scaling
Browser (Firefox, Chrome...)
Internet
Site rotten-potatoes.com
html
css
Web server (Apache, Microsoft IIS, WEBrick)
App server (rack)
Database (Postgres, SQLite)
Presentation tier
Logic tier
Persistence tier
§2.5 1,000 feet—Model-View-Controller
(vs. Page Controller, Front Controller)
Models
Controllers
Views
§2.6 500 feet: Active Record models (vs. Data Mapper)
§2.7 500 feet: RESTful controllers (Representational State Transfer for self-contained actions)
§2.8 500 feet: Template View (vs. Transform View)
• Active Record
• Data Mapper
• REST
• Template View
• Transform View

Template View pattern

- View consists of markup with selected *interpolation* to happen at runtime
  - Usually, values of variables or result of evaluating short bits of code
- In Elder Days, this *was* the app (e.g. PHP)
- *Alternative:* Transform View



Haml is HTML on a diet

```
%h1.pagename All Movies
%table#movies
  %thead
    %tr
      %th Movie Title
      %th Release Date
      %th More Info
  %tbody
    - @movies.each do |movie|
      %tr
        %td= movie.title
        %td= movie.release_date
        %td= link_to "More on #{movie.title}",  |
          movie_path(movie) |
= link_to 'Add new movie', new_movie_path
```

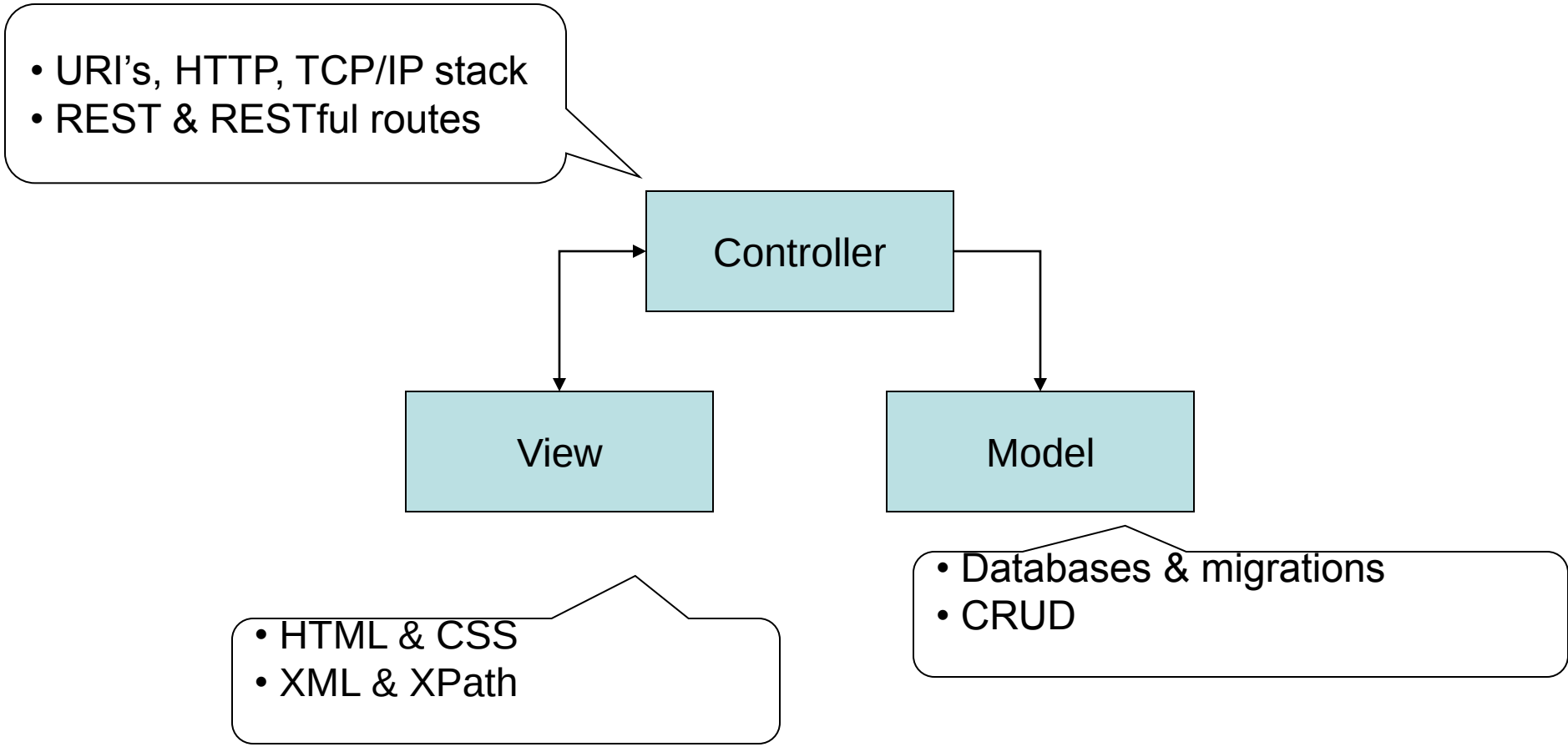
Don’t put code in your views

- Syntactically, you can put any code in view
- But MVC advocates thin views & controllers
  - Haml makes deliberately awkward to put in lots of code
- *Helpers* (methods that “prettify” objects for including in views) have their own place in Rails app
- Alternative to Haml: html.erb (Embedded Ruby) templates, look more like PHP

What happens if you embed code in your Rails views that directly accesses the model?

- ☐ It will work, but it's bad form and violates the MVC guidelines
- ☐ It will work when developing against a "toy" database, but not in production
- ☐ It won't work, because Views can't communicate directly with Models
- ☐ Behavior varies depending on the app

The big picture (technologies)



c. 2008: “Rails doesn’t scale”

- Scalability is an *architectural* concern—not confined to language or framework
- The stateless tiers of 3-tier arch *do scale*
  - With cloud computing, just worry about constants
- Traditional relational databases *do not scale*
- Various solutions combining relational and non-relational storage (“NoSQL”) *scale much better*
  - DataMapper works well with some of them
- Intelligent use of *caching* (later in course) can greatly improve the constant factors

Frameworks, Apps, Design patterns

- Many design patterns so far, more to come
- *In 1995, it was the wild west:* biggest Web sites were minicomputers, *not* 3-tier/cloud
- Best practices (patterns) “extracted” from experience and captured in frameworks
- But API’s transcended it: 1969 protocols + 1960s markup language + 1990 browser + 1992 Web server works in 2011

Architecture is about Alternatives

Pattern we're using	Alternatives
Client-Server	Peer-to-Peer
Shared-nothing (cloud computing)	Symmetric multiprocessor, shared global address space
Model-View-Controller	Page controller, Front controller, Template view
Active Record	Data Mapper
RESTful URIs (all state affecting request is explicit)	Same URI does different things depending on internal state

As you work on other SaaS apps beyond this course, you should find yourself considering different architectural choices and questioning the choices being made.

Summary: Architecture & Rails

- Model-view-controller is a well known *architectural pattern* for structuring apps
- Rails codifies SaaS app structure as MVC
- *Views* are Haml w/embedded Ruby code, transformed to HTML when sent to browser
- *Models* are stored in tables of a relational database, accessed using ActiveRecord
- *Controllers* tie views and models together via *routes* and code in controller methods

Other factors being equal, which statement is **NOT** true regarding SaaS scalability?

- ☐ Shared-nothing clusters scale better than systems built from mainframes
- ☐ Relational databases scale better than "NoSQL" databases
- ☐ The programming language used (Ruby, Java, etc.) isn't a main factor in scalability
- ☐ Scalability can be impeded by *any* part of the app that becomes a bottleneck

Comments Should Describe Things That Aren't Obvious From The Code: Why, not What

(from John Ousterhout)

Bad Comments

```
// Add one to i.
i++;


// Lock to protect against concurrent access.
SpinLock mutex;


// This function swaps the panels.
void swap_panels(Panel* p1, Panel* p2) {...}
```

Comments, cont'd

Comments should be at a higher abstract level than code:

```
# Scan the array to see if the symbol exists
```

not

```
# Loop through every array index, get the
# third value of the list in the content to
# determine if it has the symbol we are looking
# for. Set the result to the symbol if we
# find it.
```

And in Conclusion

- Debugging: Read, Ask, Search, Post
- Rails Pitfalls: Too much code in Controller, Some extra code in View
- Agile – prototypes, iterate with customer
- BDD – Design of app before implementation
- User Story – all stakeholders write what features want on 3x5 cards
- Cucumber – magically turns 3x5 card user stories into acceptance tests for app