WHAT THE USER WANTS…? MOVING FROM IDEAS TO ACTION

ANNOUNCEMENTS

- Project Iter 0-2 due SUNDAY
- Homework 2 due Monday 2/20— **start early** and check out github accounts!

INTRODUCTION TO BEHAVIOR-DRIVEN DESIGN AND USER STORIES

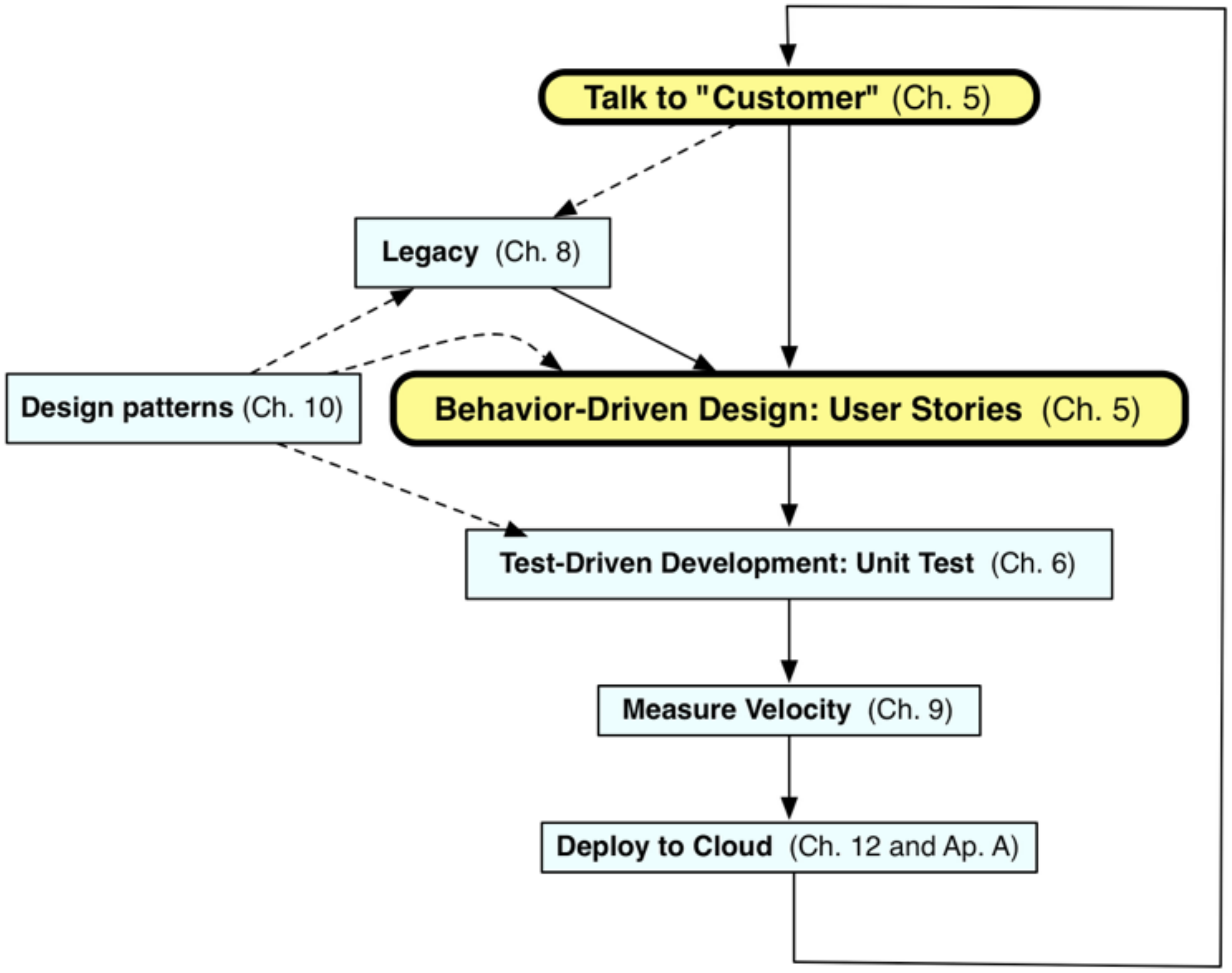
WHY DO SW PROJECTS FAIL?

- Don't do what customers want
- Or projects are late
- Or over budget
- Or hard to maintain and evolve
- Or all of the above
- Inspired Agile Lifecycle

AGILE LIFECYCLE

- Work closely, continuously with stakeholders to develop requirements, tests
  - Users, customers, developers, maintenance programmers, operators, project managers, …
- Maintain working prototype while deploying new features every **iteration**
  - Typically every 1 or 2 weeks
  - Instead of 5 major phases, each months long
- Check with stakeholders on what's next,
  to validate building right thing (vs. verify)

AGILE ITERATION

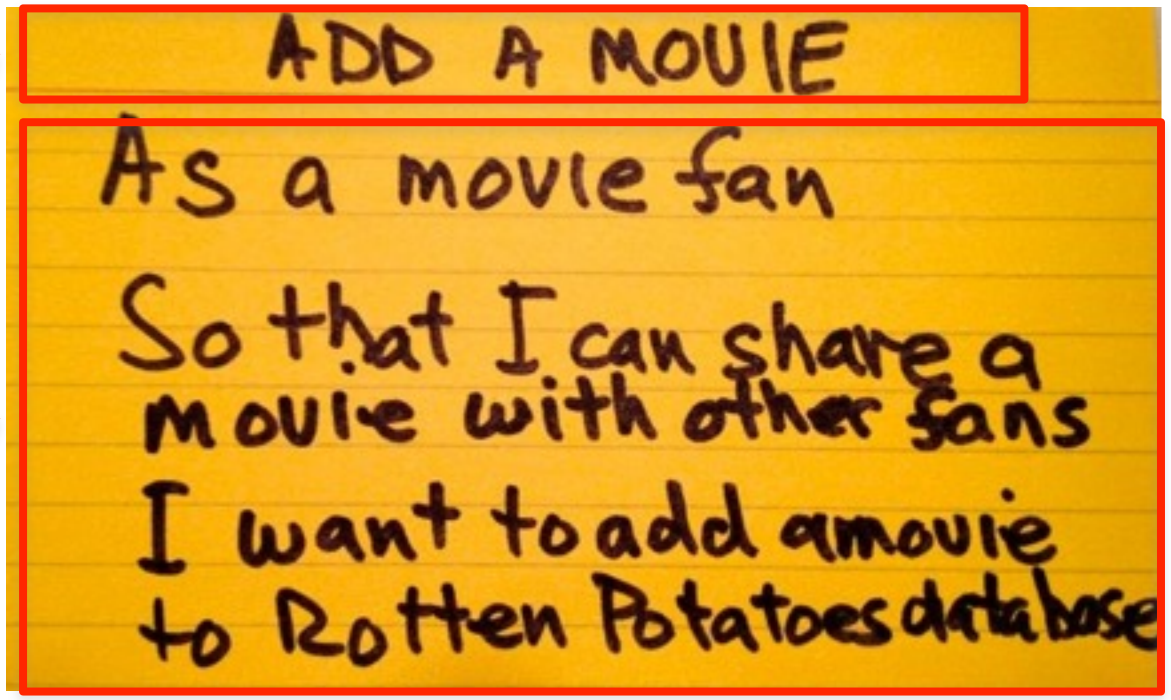

Talk to "Customer" (Ch. 5)
Legacy (Ch. 8)
Design patterns (Ch. 10)
Behavior-Driven Design: User Stories (Ch. 5)
Test-Driven Development: Unit Test (Ch. 6)
Measure Velocity (Ch. 9)
Deploy to Cloud (Ch. 12 and Ap. A)

BEHAVIOR-DRIVEN DESIGN (BDD)

- BDD asks questions about behavior of app *before and during development* to reduce miscommunication
- Requirements written down as *user stories*
  - Lightweight descriptions of how app used
- BDD concentrates on *behavior* of app vs. *implementation* of app
  - Test Driven Design or TDD (next chapter) tests implementation

USER STORIES

- 1-3 sentences in everyday language
  - Fits on 3” x 5” index card
  - Written by/with customer
- “Connextra” format:
  - Feature name
  - As a [kind of stakeholder],
    So that [I can achieve some goal],
    I want to [do some task]
  - 3 phrases must be there, can be in any order
- Idea: user story can be formulated as *acceptance test before* code is written


ADD A MOVIE

As a movie fan
So that I can share a movie with other fans
I want to add a movie to Rotten Potatoes database

WHY 3X5 CARDS?

- (from User Interface community)
- Nonthreatening => all stakeholders participate in brainstorming
- Easy to rearrange => all stakeholders participate in prioritization
- Since stories must be short, easy to change during development
  - As often get new insights during development

DIFFERENT STAKEHOLDERS MAY DESCRIBE BEHAVIOR DIFFERENTLY

- *See which of my friends are going to a show*
  - As a theatergoer
  - So that I can enjoy the show with my friends
  - I want to see which of my Facebook friends are attending a given show
- *Show patron's Facebook friends*
  - As a box office manager
  - So that I can induce a patron to buy a ticket
  - I want to show her which of her Facebook friends are going to a given show

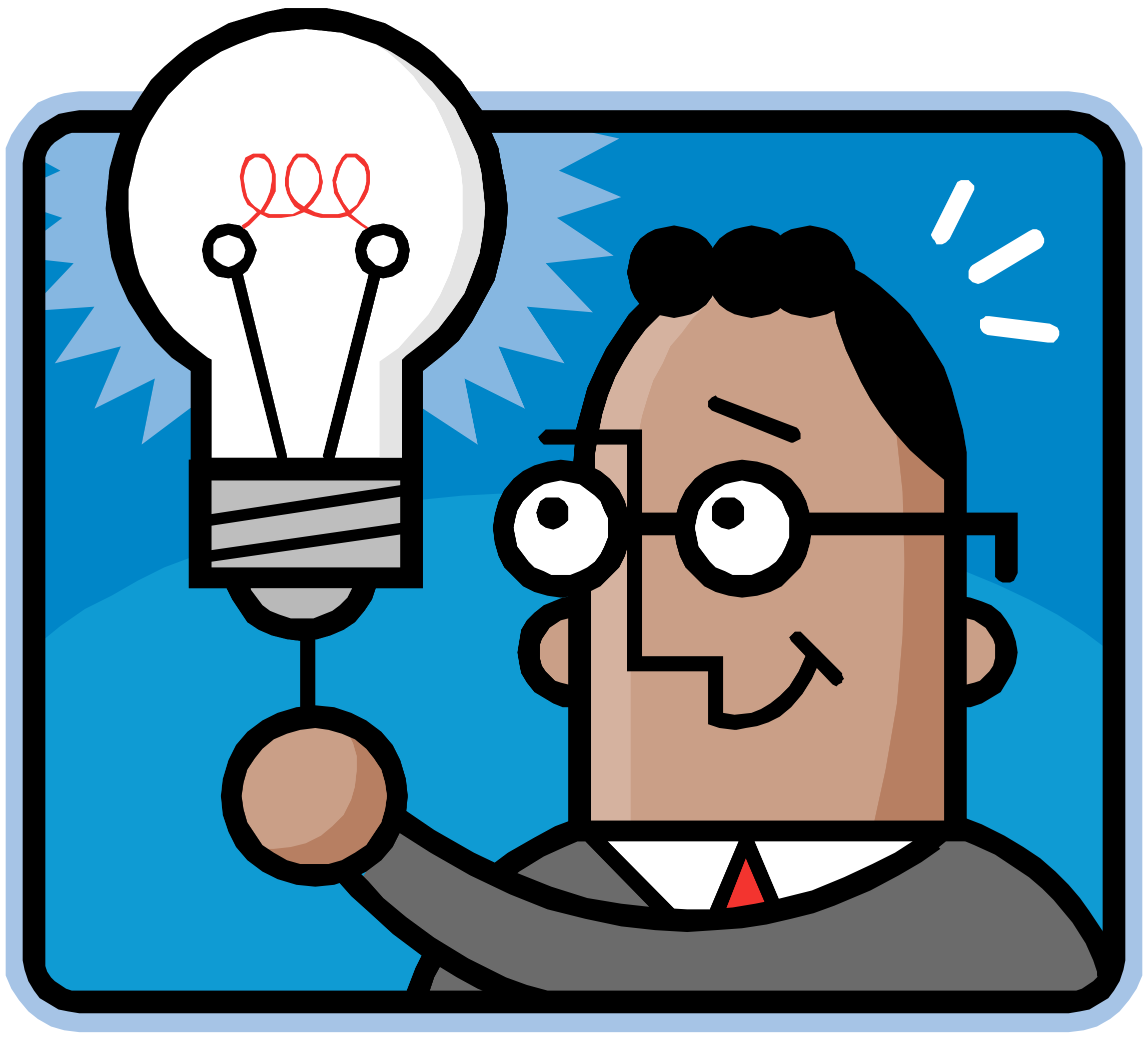
PRODUCT BACKLOG

- Real systems have 100s of user stories
- *Backlog*: User Stories not yet completed
  - (We'll see Backlog again with Zenhub)
- Prioritize so most valuable items highest
- Organize so they match SW releases over time

SMART USER STORIES

SMART STORIES

- **S**pecific
- **M**easurable
- **A**chievable
  (ideally, implement in 1 iteration)
- **R**elevant
  ("the 5 why's")
- **T**imeboxed
  (know when to give up)



SPECIFIC & MEASURABLE

- Each scenario testable
  - Implies known good input and expected results exist
- Anti-example: "UI should be user-friendly"
- Example: Given/When/Then.
  - *Given* some specific starting condition(s),
  - *When* I do X,
  - *Then* one or more specific thing(s) should happen

ACHIEVABLE

- Complete in 1 iteration
- If can’t deliver feature in 1iteration, deliver subset of stories
  - Always aim for working code @ end of iteration

TIMEBOXED

- Estimate what's achievable using *velocity*
  - Each story assigned *points* (1-3) based on progress amount
  - Velocity = Points completed / iteration
  - Use measured velocity to plan future i points per story
- Pivotal Tracker (later) tracks velocity

RELEVANT: "BUSINESS VALUE"

- Ask "Why?" recursively until discover business value, or kill the story:
  - Protect revenue
  - Increase revenue
  - Manage cost
  - Increase brand value
  - Making the product remarkable
  - Providing more value to your customers

STORIES ARE SMART— BUT FEATURES SHOULD BE RELEVANT

- Specific & Measurable: can I test it?
- Achievable? / Timeboxed?
- Relevant? use the "5 whys"
- *Show patron's Facebook friends*

As a box office manager

So that I can induce a patron to buy a ticket

I want to show her which Facebook friends are going to a given show

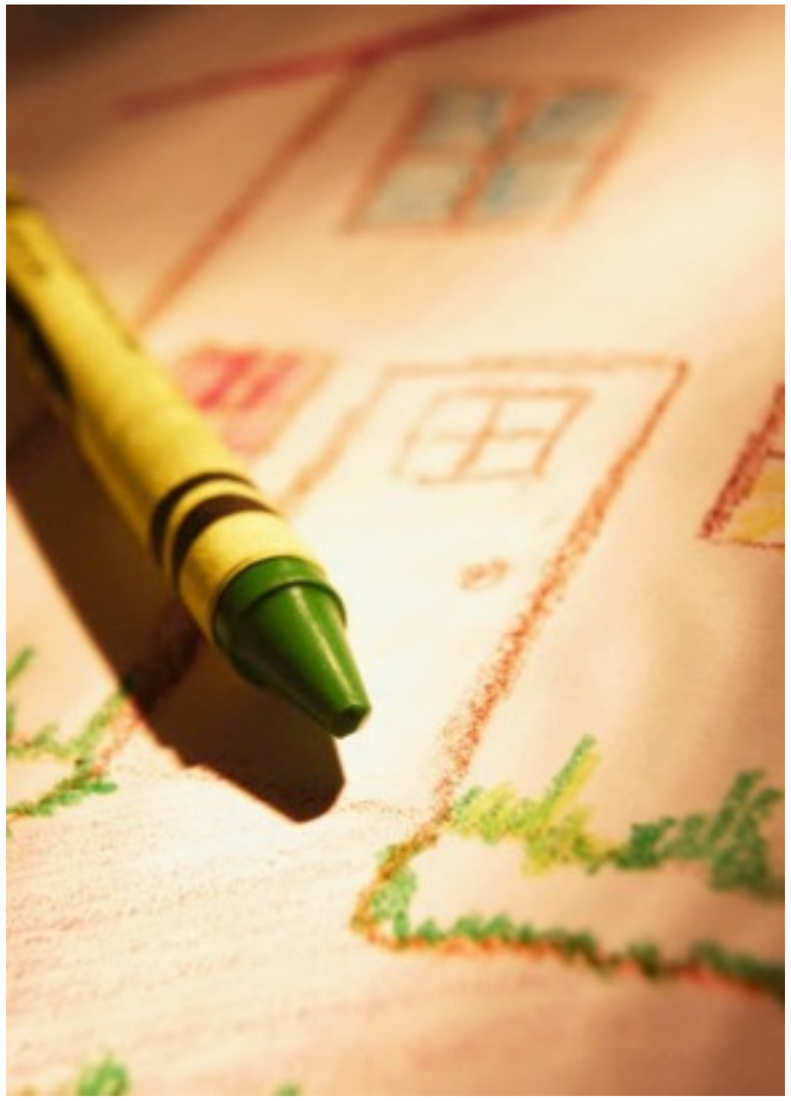
LO-FI UI SKETCHES AND STORYBOARDS

SAAS USER INTERFACE DESIGN

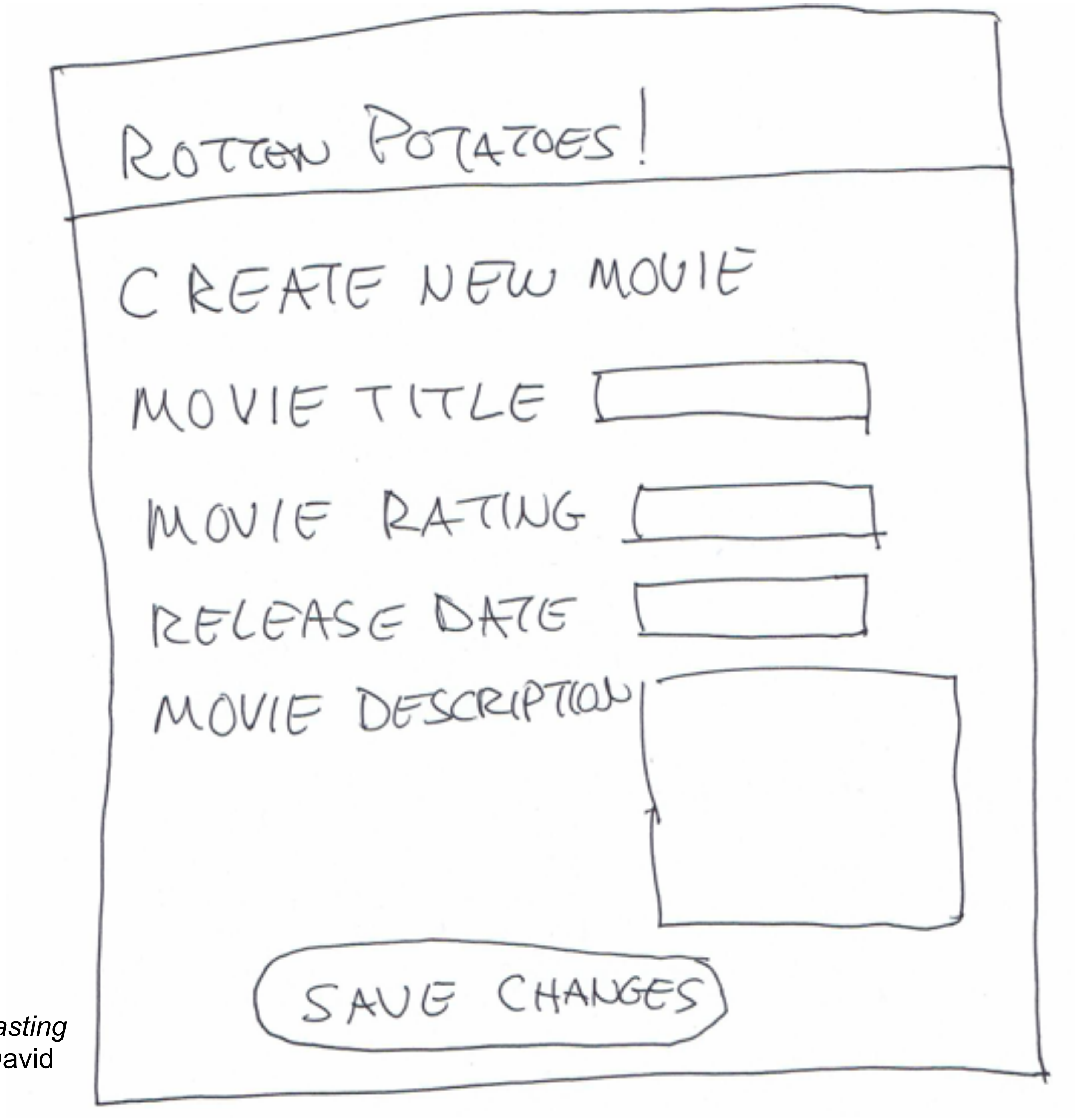
- SaaS apps often faces users

⇒User stories need User Interface (UI)

- Want *all* stakeholders involved in UI design
  - Don’t want UI rejected!
- Need UI equivalent of 3x5 cards
- Sketches: pen and paper drawings or “Lo-Fi UI”



LO-FI UI EXAMPLE

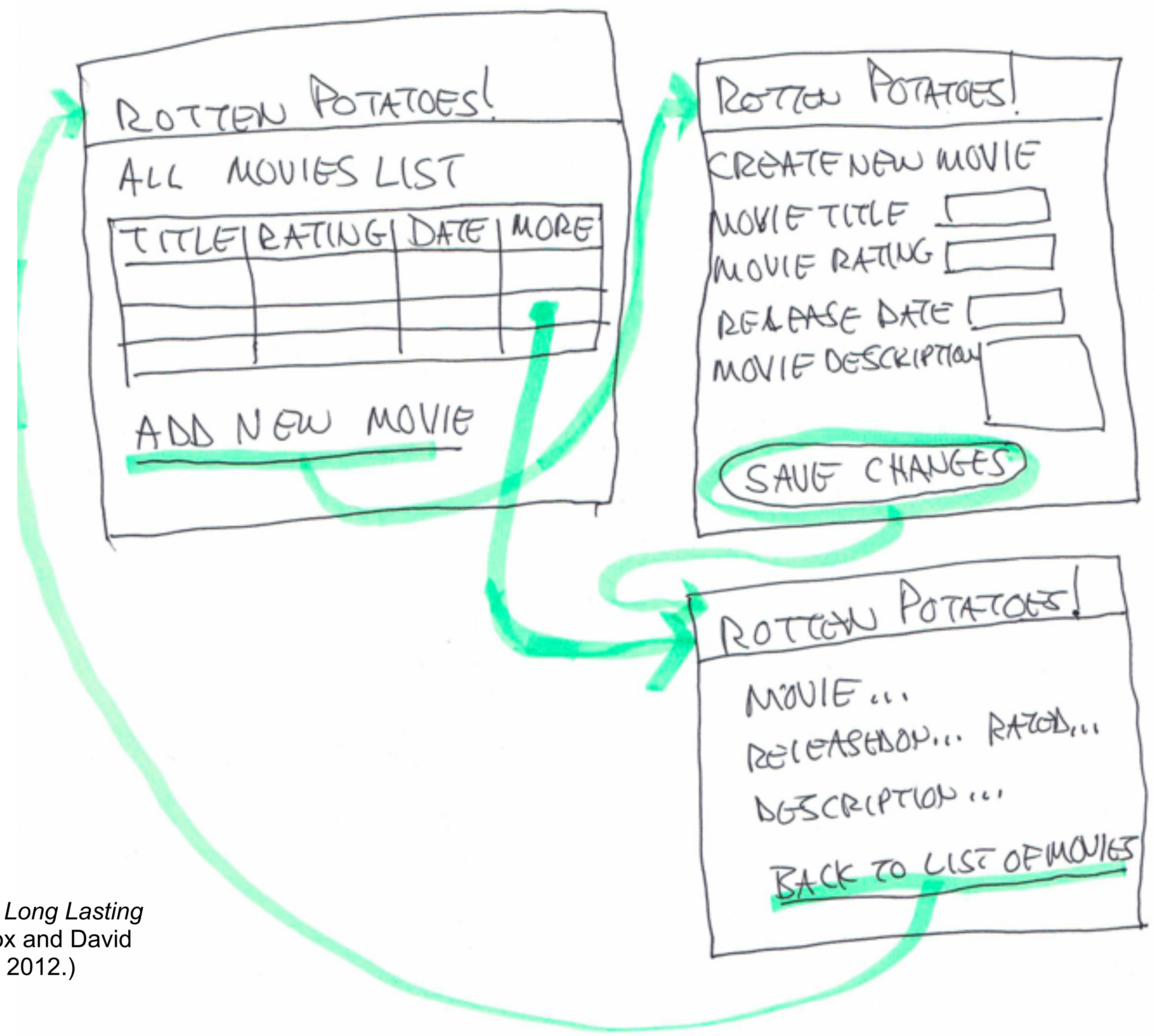


(Figure 4.3, *Engineering Long Lasting Software* by Armando Fox and David Patterson, Alpha edition, 2012.)

STORYBOARDS

- Need to show how UI changes based on user actions
- HCI => "storyboards"
- Like scenes in a movie
- But not linear

EXAMPLE STORYBOARD



(Figure 4.4, *Engineering Long Lasting Software* by Armando Fox and David Patterson, Alpha edition, 2012.)

LO-FI TO HTML

- Tedious to do sketches and storyboards,
  but easier than producing HTML!
  - Also less intimidating to nontechnical stakeholders =>
    More likely to suggest
    changes to UI if not code behind it
  - More likely to be happy with ultimate UI
- Next steps: More on CSS (Cascading Style Sheets) and Haml
  - Make it pretty *after* it works

MODELS, DATABASES, AND ACTIVE RECORD

Reminder: CS Colloquium Talk

TODAY, 5pm in Eng 103

Topic: Making Security Usable

Speaker: Dr. Kami Vaniea

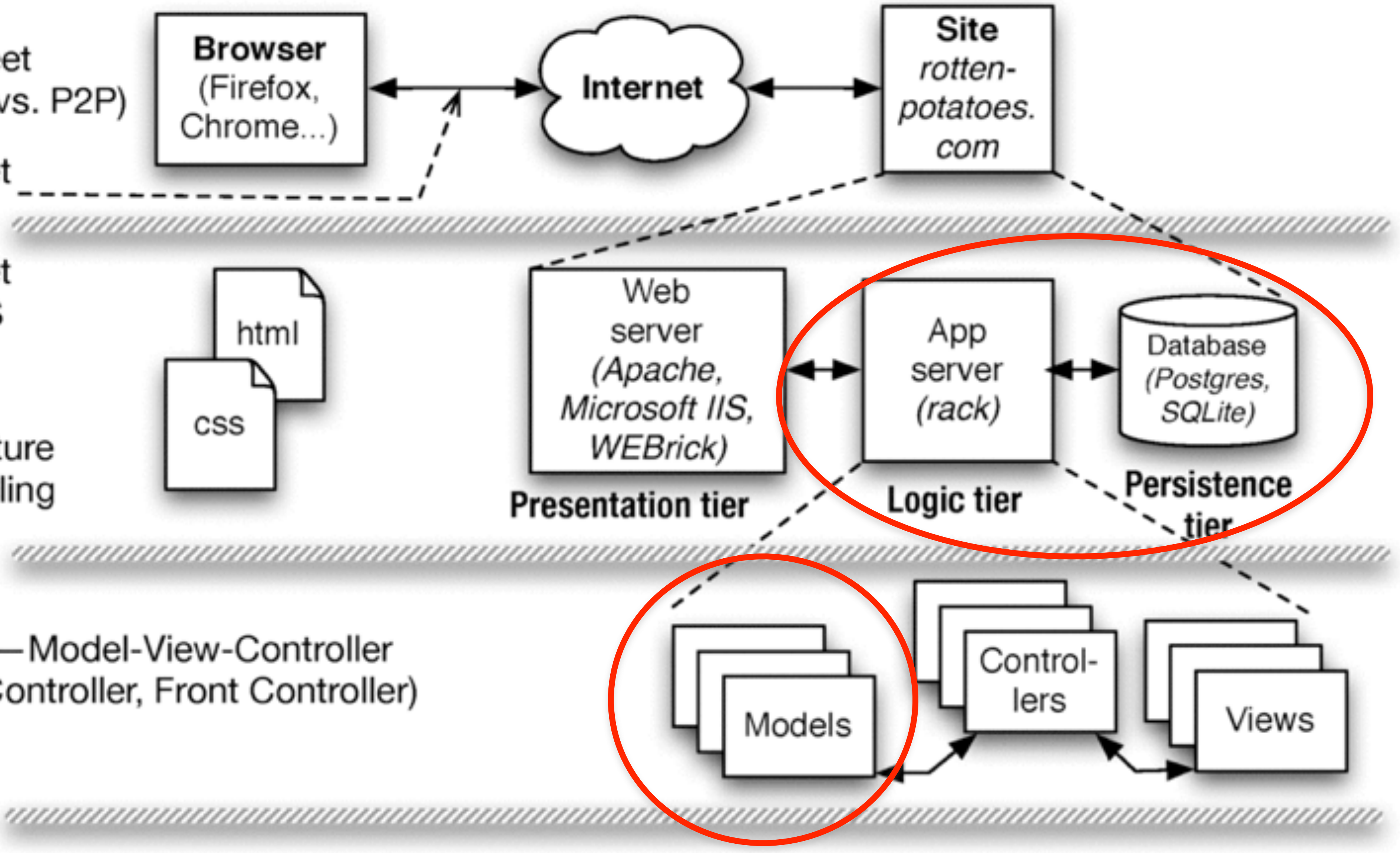
§2.1 100,000 feet
- Client-server (vs. P2P)

§2.2 50,000 feet
- HTTP & URIs

§2.3 10,000 feet
- XHTML & CSS

§2.4 5,000 feet
- 3-tier architecture
- Horizontal scaling

§2.5 1,000 feet—Model-View-Controller (vs. Page Controller, Front Controller)



§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)

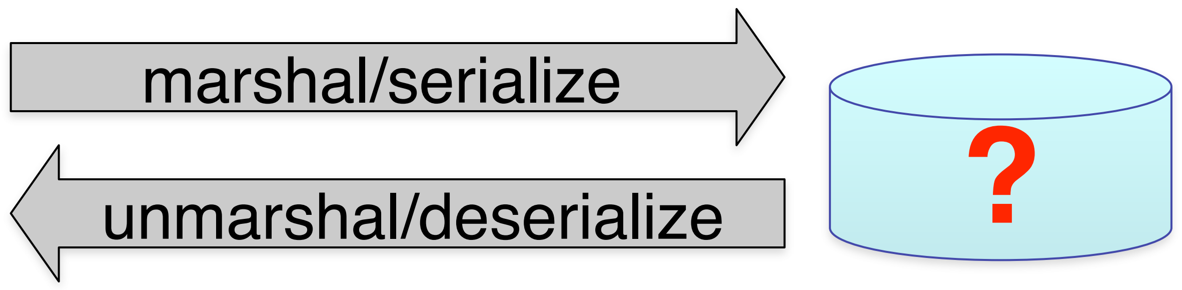
- **Active Record**
- Data Mapper

- **REST**

- **Template View**
- Transform View

IN-MEMORY VS. IN-STORAGE OBJECTS

```
#<Movie:0x1295580>
m.name, m.rating, ...
 #<Movie:0x32ffe416>
 m.name, m.rating, ...
```



- How to represent persisted object in storage
  - Example: Movie and Reviews
- Basic operations on object: CRUD (Create, Read, Update, Delete)
- ActiveRecord: every model knows how to CRUD itself, using common mechanisms

RAILS MODELS STORE DATA IN RELATIONAL DATABASES

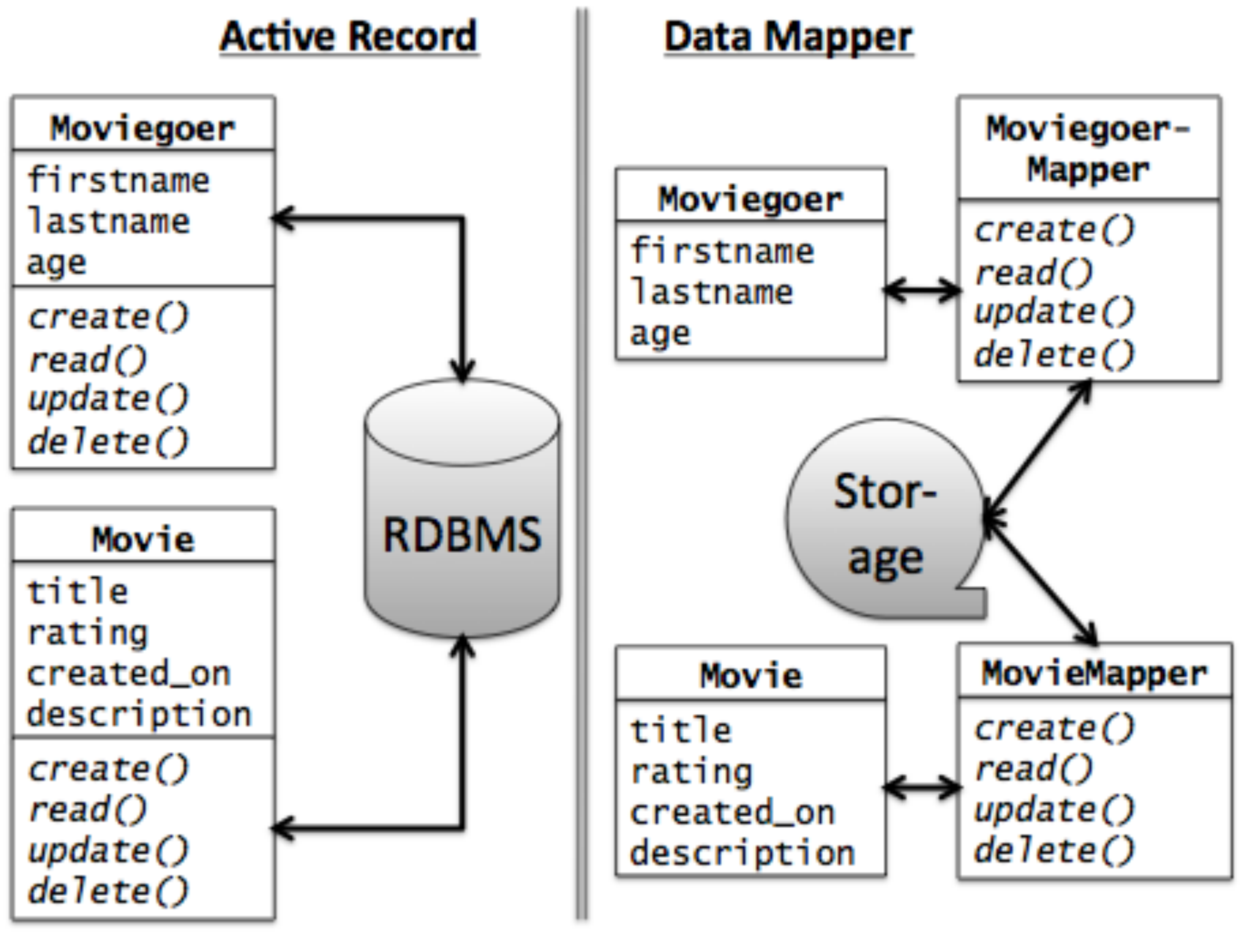
- Each type of model gets its own database *table*
  - All rows in table have identical structure
  - 1 row in table == one model instance
  - Each column stores value of an *attribute* of the model
  - Each row has unique value for *primary key* (by convention, in Rails this is an integer and is called *id*)

id	rating	title	release_date
2	G	Gone With the Wind	1939-12-15
11	PG	Casablanca	1942-11-26
...	...	...	...
35	PG	Star Wars	1977-05-25

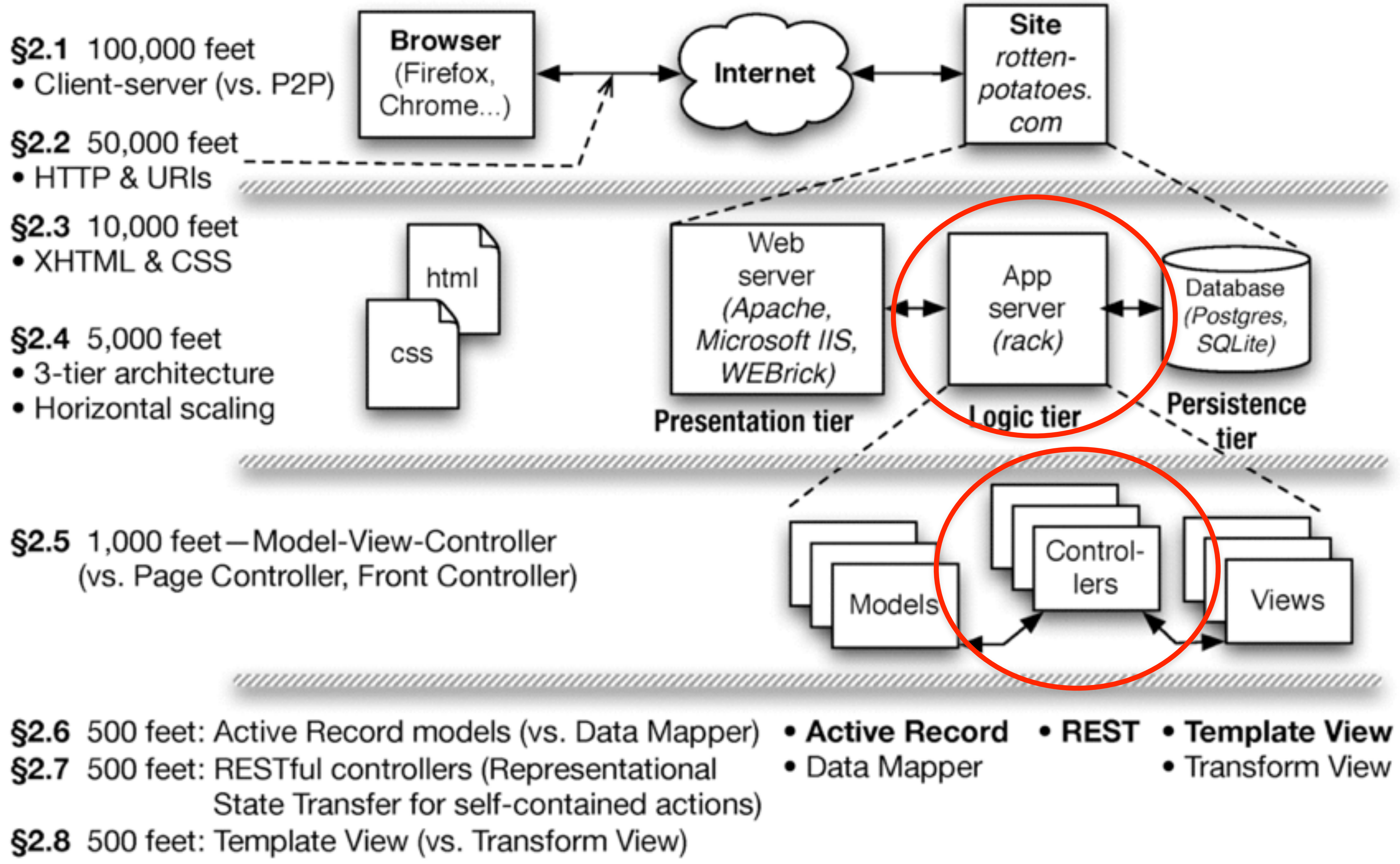
- *Schema:* Collection of all tables and their structure

ALTERNATIVE: DATAMAPPER

- Data Mapper associates separate *mapper* with each model
  - Idea: keep mapping *independent* of particular data store used => works with more types of databases
  - Used by Google AppEngine
  - Con: can't exploit RDBMS features to simplify complex queries & relationships
- We'll revisit when talking about *associations*



CONTROLLERS, ROUTES, AND RESTFULNESS



§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

§2.5 1,000 feet—Model-View-Controller (vs. Page Controller, Front Controller)

§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

ROUTES

- In MVC, each interaction the user can do is handled by a *controller action*
  - Ruby method that handles that interaction
- A *route* maps `<HTTP method, URI>` to controller action
-

Route	Action
`GET /movies/3`	Show info about movie whose ID=3
`POST /movies`	Create new movie from attached form data
`PUT /movies/5`	Update movie ID 5 from attached form data
`DELETE /movies/5`	Delete movie whose ID=5

BRIEF INTRO TO RAILS' ROUTING SUBSYSTEM

- dispatch <method,URI> to correct controller action
- provides *helper methods* that generate a <method,URI> pair given a controller action
- parses query *parameters* from both URI and form submission into a convenient hash
- Built-in shortcuts to generate all CRUD routes (though most apps will also have other routes)

rake routes

```
I    GET /movies             {:action=>"index", :controller=>"movies"}
C   POST /movies             {:action=>"create", :controller=>"movies"}
     GET /movies/new         {:action=>"new", :controller=>"movies"}
     GET /movies/:id/edit    {:action=>"edit", :controller=>"movies"}
R    GET /movies/:id         {:action=>"show", :controller=>"movies"}
U    PUT /movies/:id         {:action=>"update", :controller=>"movies"}
D DELETE /movies/:id         {:action=>"destroy", :controller=>"movies"}
```

GET /MOVIES/3/EDIT HTTP/1.0

- Matches route:

```
GET /movies/:id/edit {:action=>"edit", :controller=>"movies"}
```

- Parse wildcard parameters: `params[:id] = "3"`
- Dispatch to `edit` method in `movies_controller.rb`
- To include a URI in generated view that will submit the form to the update controller action with `params[:id]==3,` call helper:

rake routes `_movie_path(3)` *# => PUT /movies/3*

```
I     GET /movies              {:action=>"index", :controller=>"movies"}
C    POST /movies              {:action=>"create", :controller=>"movies"}
      GET /movies/new          {:action=>"new", :controller=>"movies"}
      GET /movies/:id/edit     {:action=>"edit", :controller=>"movies"}
R     GET /movies/:id          {:action=>"show", :controller=>"movies"}
U     PUT /movies/:id          {:action=>"update", :controller=>"movies"}
D  DELETE /movies/:id          {:action=>"destroy", :controller=>"movies"}
```

REST (REPRESENTATIONAL STATE TRANSFER)

- Idea: *Self-contained* requests specify what *resource* to operate on and what to do to it
  - Roy Fielding's PhD thesis, 2000
  - Wikipedia: "a *post hoc description of the features that made the Web successful*"
- A service (in the SOA sense) whose operations are like this is a RESTful service
- Ideally, RESTful URIs name the operations
- Let's see an *anti-example:*

http://pastebin.com/edF2NzCF

NOT RESTFUL

http://pastebin.com/edF2NzCF

```
def get_kindle_sales(cs_user,cs_pass)
  session = Mechanize.new
  session.user_agent_alias = 'Mac Safari'
  session.get 'https://www.amazon.com/ap/signin?
openid.assoc_handle=amzn_dtp&openid.identity=' #...etc.
  form = session.get('https://www.amazon.com/ap/signin?
openid.assoc_handle=amzn_dtp&openid.=' + # ...etc.
       '...').form_with(:name => 'signIn')
  params = {'email' => cs_user,   'password' => cs_pass}
  %w(appActionToken appAction openid.pape.max_auth_age openid.ns).each do |field| # there's
more, actually
    params[field] = form[field]
  end
  session.post('https://www.amazon.com/ap/signin', params)
  response = session.get('https://kdp.amazon.com/self-publishing/reports/transactionReport?
_=1326589411161&previousMonthReports=false&marketplaceID=ATVPDKIKX0DER')
  # note non-RESTful concept of "previousMonthReports" in URI
  hash = JSON.parse(response.body)
  kindle_units = hash['aaData'][0][5]
end
```

TEMPLATE VIEWS AND HAML

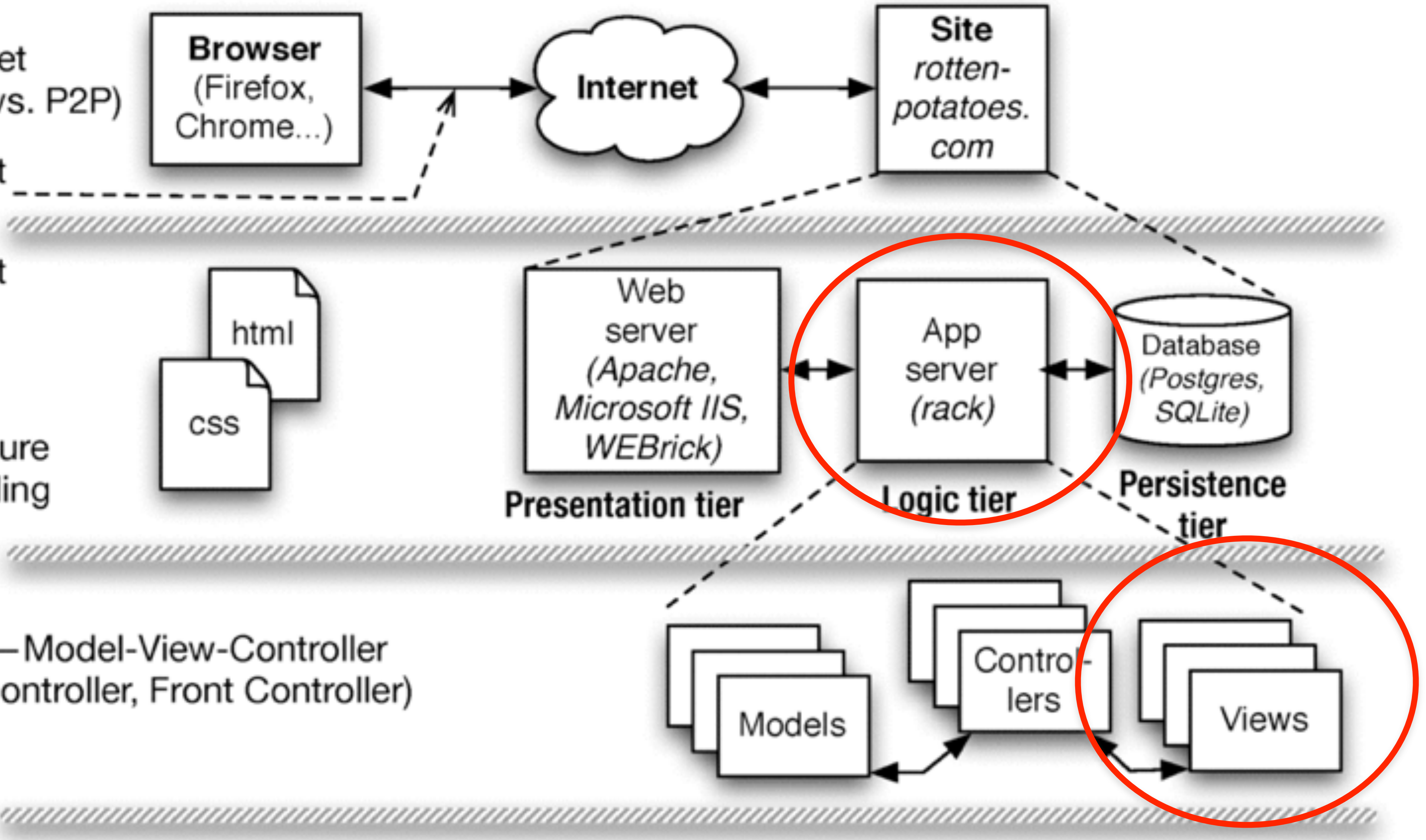
§2.1 100,000 feet
- Client-server (vs. P2P)

§2.2 50,000 feet
- HTTP & URIs

§2.3 10,000 feet
- XHTML & CSS

§2.4 5,000 feet
- 3-tier architecture
- Horizontal scaling

§2.5 1,000 feet—Model-View-Controller
(vs. Page Controller, Front Controller)



§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)

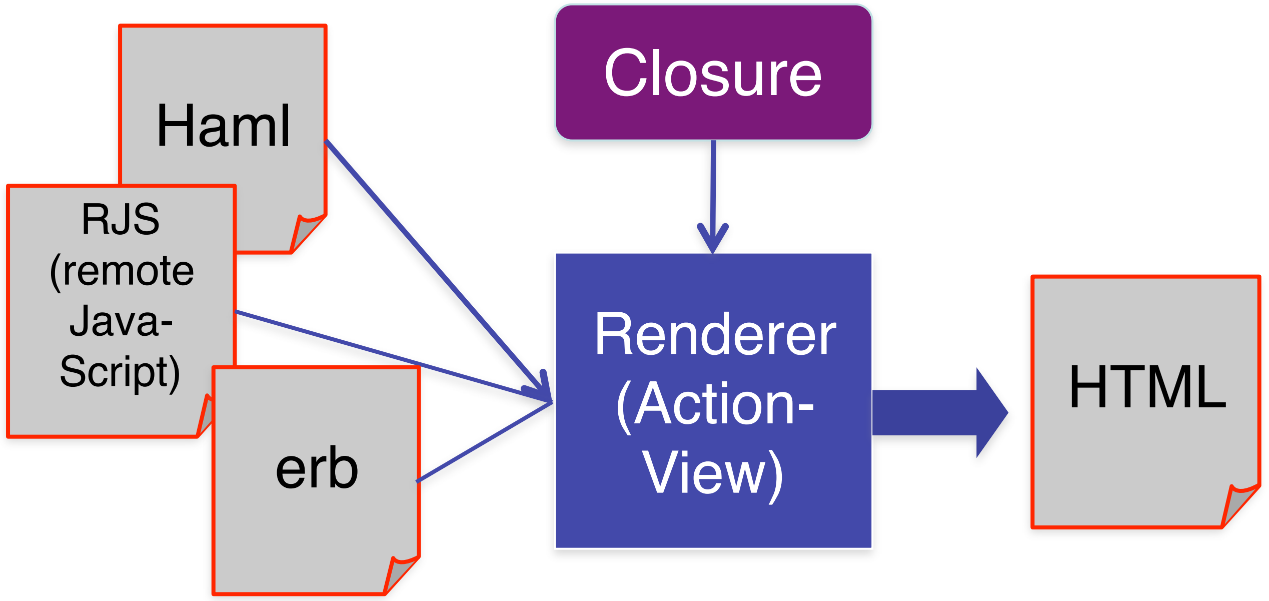
- **Active Record**
- Data Mapper

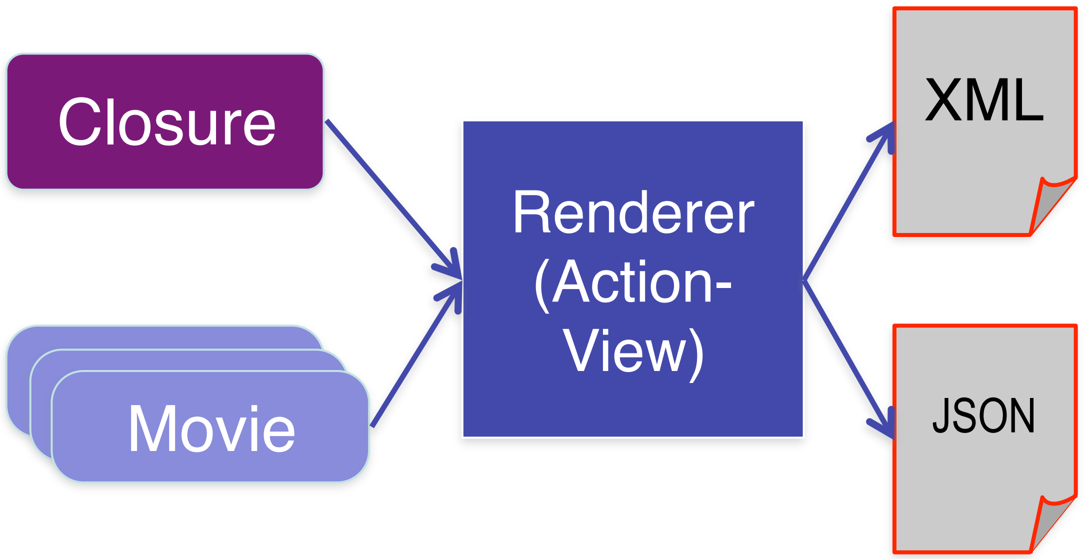
- **REST**

- **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
```

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.

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

Source & configuration management (SCM)

- What is it?
  - *Version* (snapshot) code, docs, config files, etc. at key points in time
  - Complete copy of every versioned file per snapshot
  - Implementation: deltas? complete file copy? symlink?
- Why do it?
  - Roll back if introduce bugs
  - Separate deployed from development version of code
  - Keep separate *branches* of development
  - Documented history of who did what when
  - Track what changed between revisions of a project

40 Years of Version Control

SCCS & RCS (1970s)

CVS (1986)

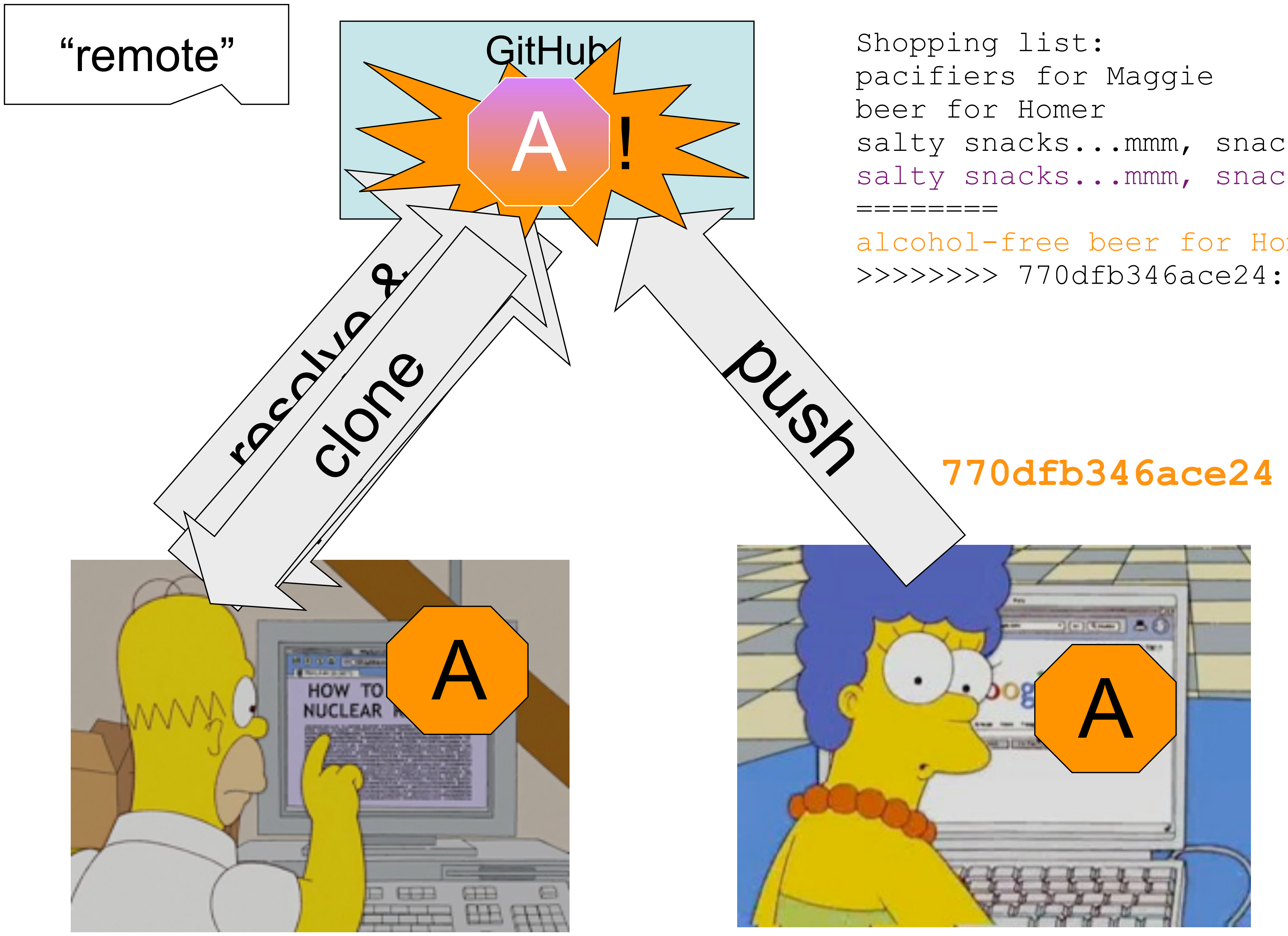
Subversion (2001)

Git (2005)

Image © TheSun.au

*

Merge Conflict



```
Shopping list:
pacifiers for Maggie
beer for Homer
salty snacks...mmm, snacks
salty snacks...mmm, snacks
=========
alcohol-free beer for Homer
>>>>>>>> 770dfb346ace24:shop_list.txt
```

Pull = Fetch + Merge

- ➤Merge two repos = try to apply commits in either one to both
  - ➤Conflict if different changes to same file “too close” together
  - ➤`git pull` = `git pull origin master`
- ➤Successful merge implies commit!
  - ➤Always commit before merging/pulling
  - ➤Commit early & often—small commits OK!

Commit: a tree snapshot identified by a commit-ID

- ➤40-digit hex hash (SHA-1), unique in the universe…but a pain
- ➤use unique (in this repo) prefix, eg 770dfb

HEAD: most recently committed version on current branch

ORIG_HEAD: right after a merge, points to pre-merged version

HEAD~***n***: n'th previous commit

770dfb~2: 2 commits before 770dfb

"master@{01-Sep-2012}": last commit prior to that date

*

Undo!

git reset --hard ORIG_HEAD

git reset --hard HEAD

git checkout *commit-id* -- *files…*

➤Comparing/sleuthing:

git diff *commit-id* -- *files…*

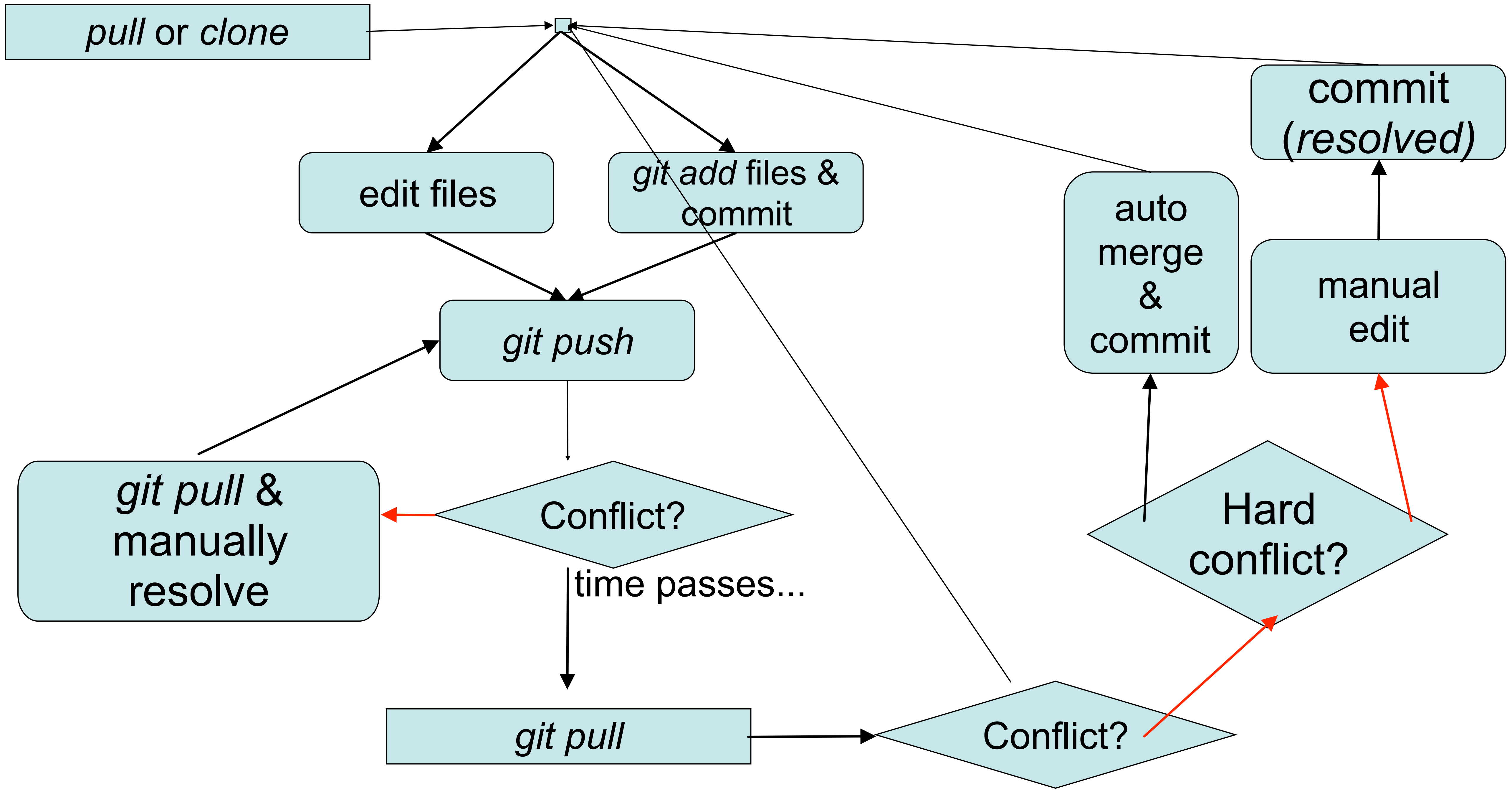
git diff "master@{01-Sep-12}" -- *files*

git blame *files*

git log *files*

*

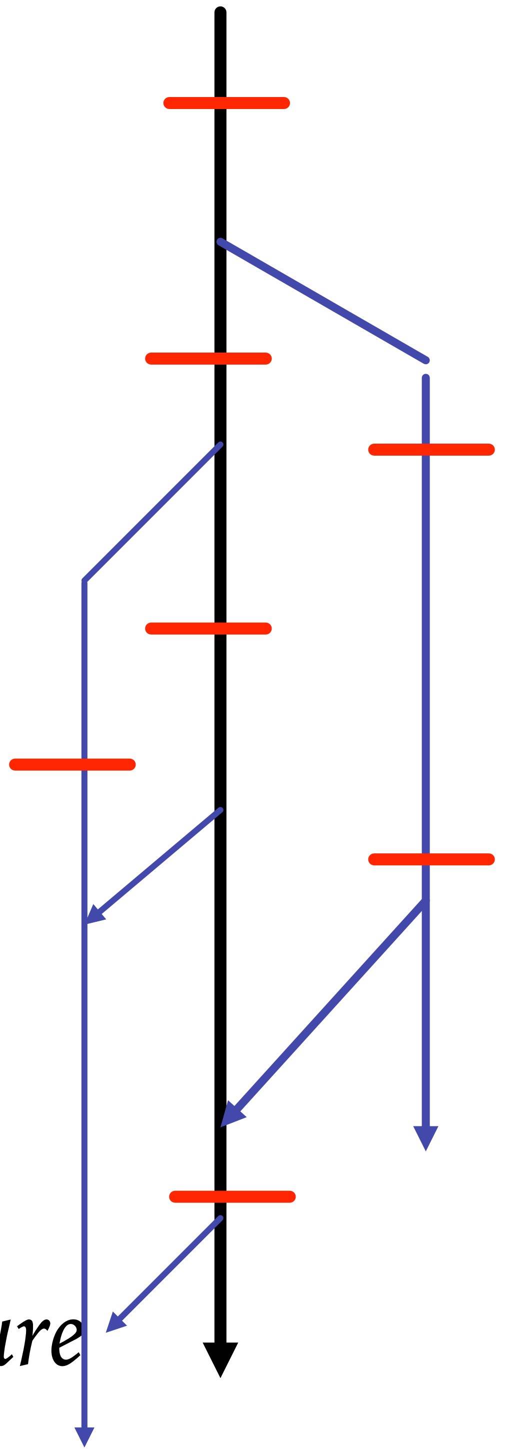
Version control with conflicts



Effective Branching

Branches

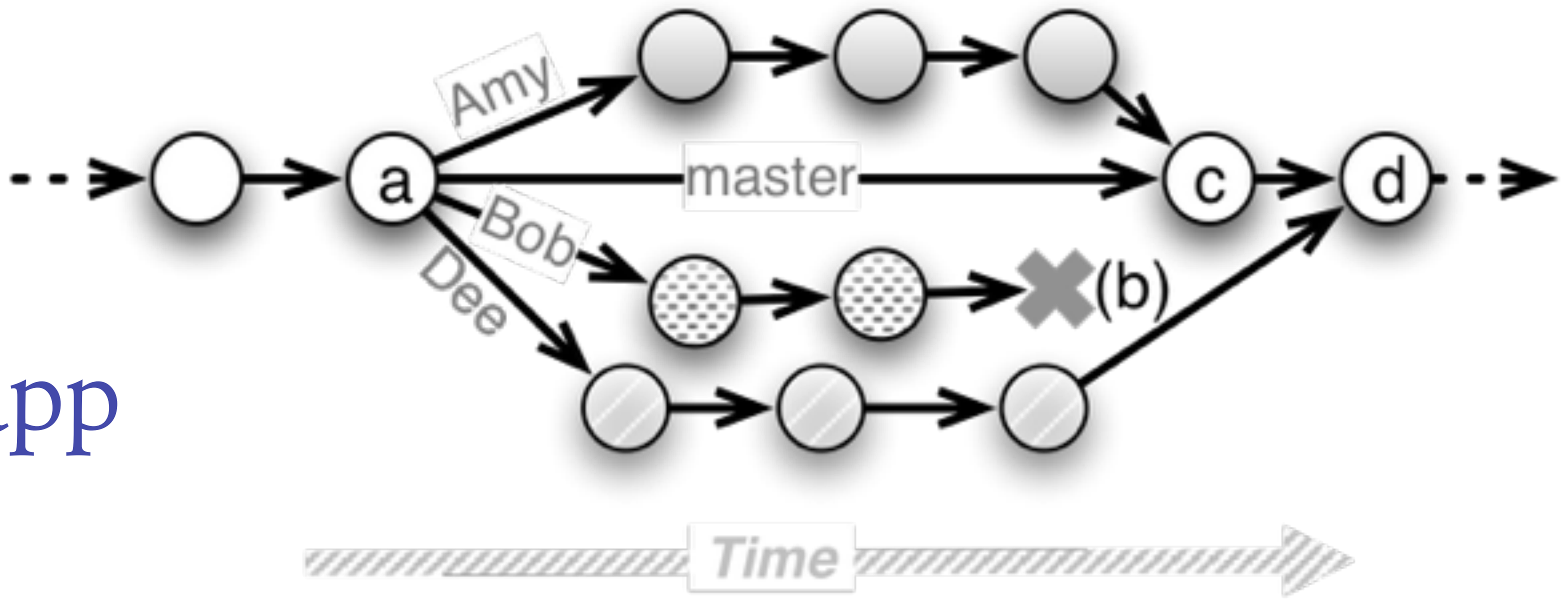
- Development **trunk** vs. **branches**
  - trunk is called “master branch” in Git
  - Creating branch is *cheap!*
  - switch among branches: *checkout*
- Separate commit histories per *branch*
- *Merge* branch back into trunk
  - ...or with *pushing* branch changes
  - Most branches eventually die
- Killer use case for agile SaaS: *branch per feature*



Creating new features without disrupting working code

1. To work on a new feature, create new branch *just for that feature*
   - ➤many features can be in progress at same time
2. Use branch *only* for changes needed for *this feature*, then merge into trunk
3. Back out this feature ⇔ undo this merge

In well-factored app, 1 feature shouldn't touch many parts of app



Mechanics

- Create new branch & switch to it

```
git branch CoolNewFeature
git checkout CoolNewFeature
```
←*current branch*

- Edit, add, make commits, etc. on branch
- Push branch to origin repo (optional):

```
git push origin CoolNewFeature
```

    - creates *tracking branch* on remote repo

- Switch back to master, and merge:

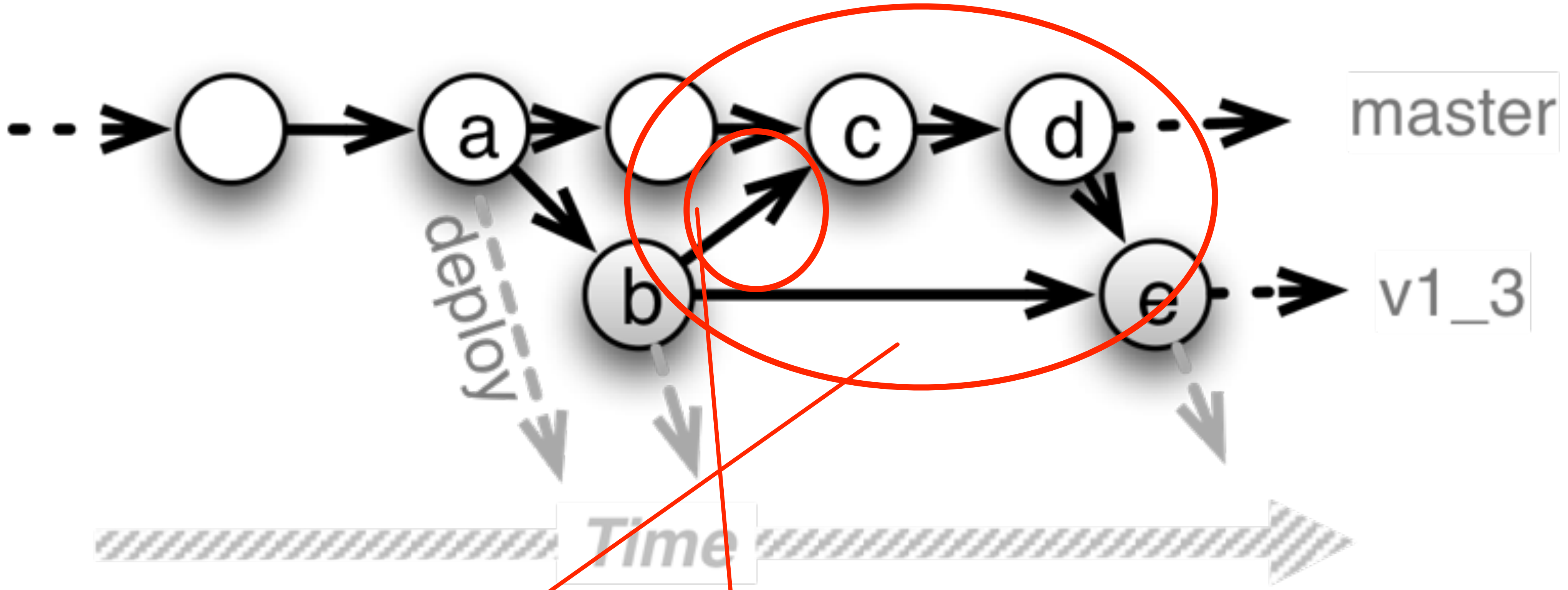
```
git checkout master
git merge CoolNewFeature
```
← *warning!!*

*

Branches & Deployment

- Feature branches should be short-lived
  - otherwise, drift out of sync with master, and hard to reconcile
  - git rebase can be used to “incrementally” merge
  - git cherry-pick can be used to merge only specific commits
- “Deploy from master” is most common

Release/bugfix branches and cherry-picking commits



criss-cross merge

git cherry-pick *commit-id*

Rationale: release branch is a stable place to do incremental bug fixes

*

Branch vs. Fork

- Git supports *fork & pull* collaboration model
  - branch: create temporary branch in *this repo*
  - merge: fold branch changes into master (or into another branch)
  - fork: clone *entire repo*
  - pull request: I ask you to pull specific commits from my forked repo

*