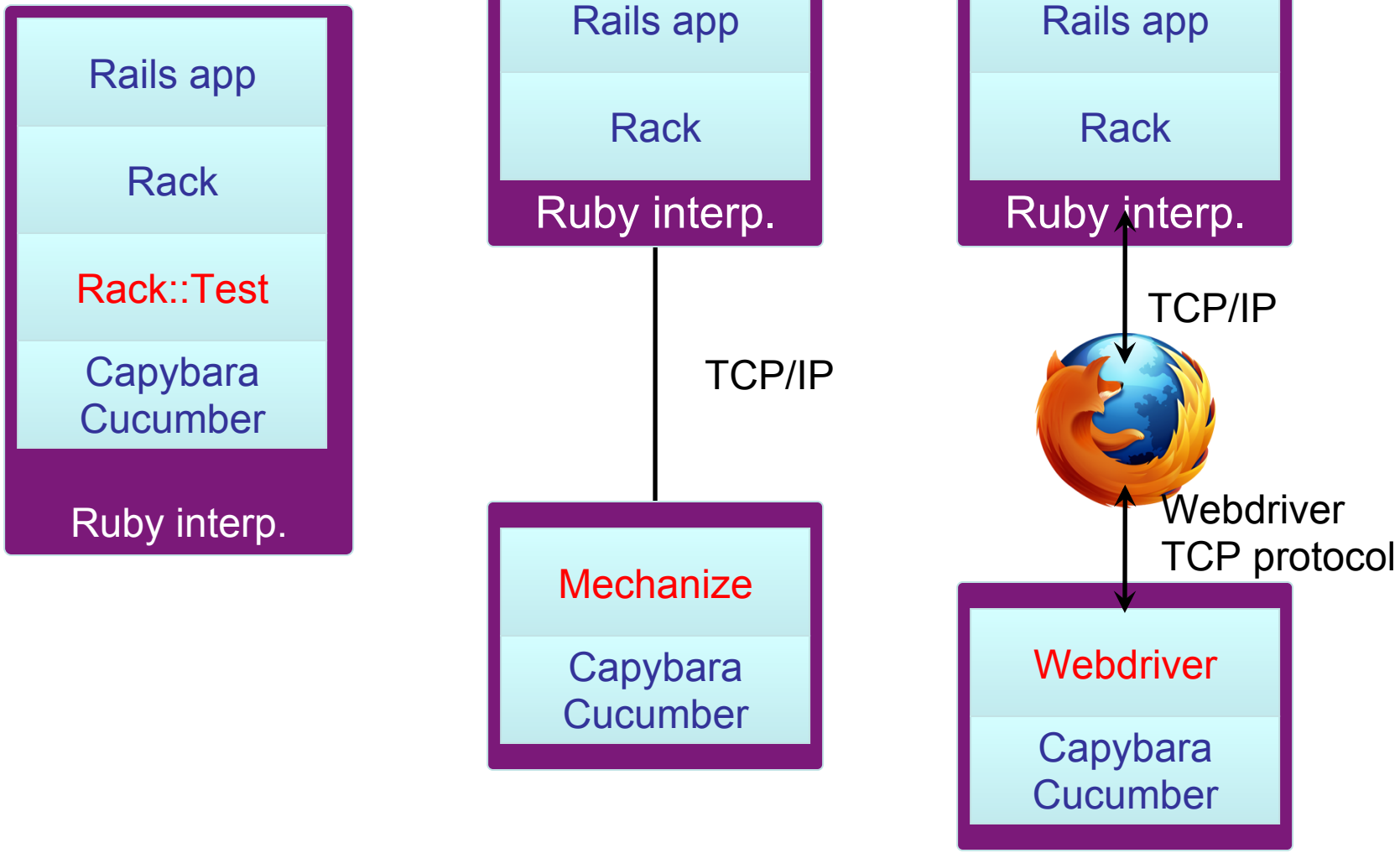
Characterization Tests

- Establish *ground truth about how the app works today*, as basis for coverage
  - Makes known behaviors **R**epeatable
  - Increase confidence that you're not breaking anything
- **Pitfall: don't try to make improvements at this stage!**

Integration-Level Characterization Tests

- Natural first step: black-box/integration level
  - don't rely on your understanding app structure
- Use the Cuke, Luke
  - Additional Capybara back-ends like Mechanize make almost everything scriptable
  - Do imperative scenarios now
  - Convert to declarative or improve `Given` steps later when you understand app internals

In-process vs. out-of-process



*

Unit- and Functional-Level Characterization Tests

- Cheat: write tests to learn as you go
  - *See Screencast 8.3.1* at screencast.saasbook.info

```
it "should calculate sales tax" do
  order = mock('order')
  order.compute_tax.should == -99.99
end
# object 'order' received unexpected message 'get_total'
it "should calculate sales tax" do
  order = mock('order', :get_total => 100.00)
  order.compute_tax.should == -99.99
end
#   expected compute_tax to be -99.99, was 8.45
it "should calculate sales tax" do
  order = mock('order', :get_total => 100.00)
  order.compute_tax.should == 8.45
end
```

Which is FALSE about integration-level characterization tests vs. module- or unit-level characterization tests?

- They are based on fewer assumptions about how the code works
- They are just as likely to be unexpectedly dependent on the production database
- They rely less on detailed knowledge about the code's structure
- If a customer can do the action, you can create a simple characterization test by mechanizing the action by brute force

*

Identifying What's Wrong: Smells, Metrics, SOFA *(ELLS §8.4)*

http://pastebin.com/gtQ7QcHu

Quantitative: Metrics

Metric	Tool	Target score
Code-to-test ratio	rake stats	≤ 1:2
C0 (statement) coverage	SimpleCov	90%+
Assignment-Branch-Condition score	flog	< 20 per method
Cyclomatic complexity	saikuro	< 10 per method (NIST)

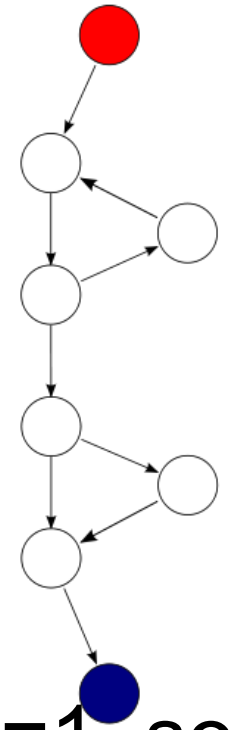
- "Hotspots": places where *multiple metrics* raise red flags
- add `require 'metric_fu'` to **`Rakefile`**
- **`rake metrics:all`**
- Take metrics with a grain of salt
- Like coverage, better for *identifying where improvement is needed* than for *signing off*

*

Cyclomatic complexity (McCabe, 1976)

- # of linearly-independent paths thru code = E– N+2P (edges, nodes, connected components)

```
def mymeth
  while(...)
    ....
  end
  if (...)
    do_something
  end
end
```



- Here, E=9, N=8, P=1, so CC=3
- NIST (Natl. Inst. Stds. & Tech.): ≤10 /module

Qualitative: Code Smells

SOFA captures symptoms that often indicate code smells:

- Be short
- Do one thing
- Have few arguments
- Consistent level of abstraction

Why Lots of Arguments is Bad

- Hard to get good testing coverage
- Hard to mock/stub while testing
- Boolean arguments should be a yellow flag
  - If function behaves differently based on Boolean argument value, maybe should be 2 functions
- If arguments “travel in a herd”, maybe you need to *extract a new class*

Single Level of Abstraction

- Complex tasks need divide & conquer
- Yellow flag for "encapsulate this task in a method":
  - line N of function says *what to do*
  - but line N+1 says *how to do* something
- Example: encourage customers to opt in

http://pastebin.com/AFQAKxbR

Example: AvailableSeat

- A real example
  - Shows have seat inventory for sale, at different prices and for different sections (premium vs. regular, eg)
  - Some seats only available to "VIP" customers
  - Some seat types only sold during certain date ranges, or have limited inventory

AvailableSeat	
Responsibilities	*Collaborators*
Knows rules for computing availability	Showdate Customer ValidVoucher VoucherType
Computes availability of each seat type given show & customer	
Provides explanation when a certain seat type is unavailable	

A good method is like a good news story

What makes a news article easy to read?

Good: start with a high level summary of key points, then go into each point in detail

Good: each paragraph deals with 1 topic

Bad: ramble on, jumping between “levels of abstraction” rather than progressively refining

Which SOFA guideline is most important for unit-level testing?

- Short
- Do one thing
- Have few arguments
- Stick to one level of abstraction

*

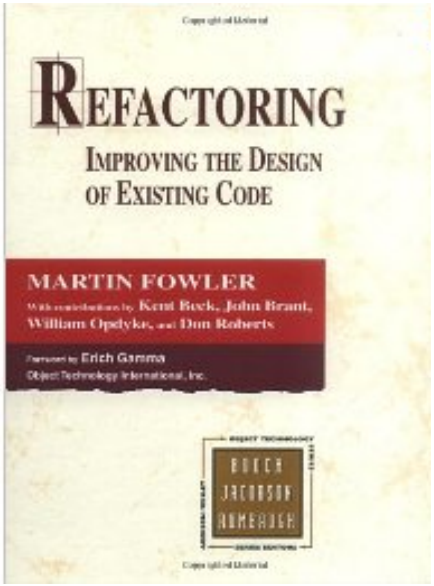
Intro to Method-Level Refactoring *(ELLS §8.5)*

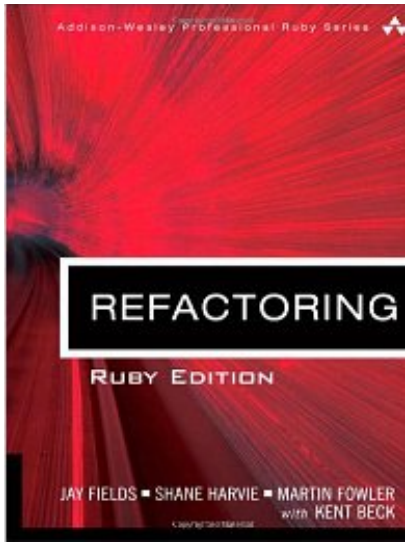
Refactoring: Idea

- Start with code that has 1 or more problems/ smells
- Through a series of *small steps,* transform to code from which those smells are absent
- Protect each step with tests
- *Minimize time during which tests are red*

History & Context

- Fowler et al. developed mostly definitive catalog of refactorings
  - Adapted to various languages
  - Method- and class-level refactorings
- Each refactoring consists of:
  - Name
  - Summary of what it does/when to use
  - Motivation (what problem it solves)
  - Mechanics: step-by-step recipe
  - Example(s)





*

Refactoring TimeSetter

- Fix stupid names
- Extract method
- Extract method, encapsulate class
- Test extracted methods
- Some thoughts on unit testing
- Glass-box testing can be useful while refactoring
- Common approach: test *critical values* and *representative noncritical values*

http://pastebin.com/pYCfMQJp

http://pastebin.com/sXVDW9C6

http://pastebin.com/zWM2ZqaW

http://pastebin.com/DRpNPzpT

*

What did we do?

- Made date calculator easier to read and understand using simple *refactorings*
- Found a bug

http://pastebin.com/0Bu6sMYi

- Observation: if we had developed method using TDD, might have gone easier!
- Did we improve our **`flog`** & **`reek`** scores?

Other Smells & Remedies

Smell	Refactoring that may resolve it
Large class	Extract class, subclass or module
Long method	Decompose conditional Replace loop with collection method Extract method Extract enclosing method with `yield()` Replace temp variable with query Replace method with object
Long parameter list/data clump	Replace parameter with method call Extract class
Shotgun surgery; Inappropriate intimacy	Move method/move field to collect related items into one DRY place
Too many comments	Extract method introduce assertion replace with internal documentation
Inconsistent level of abstraction	Extract methods & classes

*

Which is NOT a goal of method-level refactoring?

- Reduce code complexity
- Eliminate code smells
- Eliminate bugs
- Improve testability

*

Legacy Code & Refactoring: Reflections, Fallacies, Pitfalls, etc. *(ELLS §8.8-8.10)*

Armando Fox

First Drafts

When in the Course of human events, it becomes necessary for a people to advance from that subordination in which they have hitherto remained, & to assume among the powers of the earth the equal & independent station to which the Laws of Nature & of Nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the change.

We hold these truths to be sacred & undeniable…

First Drafts

When in the Course of human events, it becomes necessary for one people to dissolve the political bands which have connected them with another, & to assume among the powers of the earth, the separate & equal station to which the Laws of Nature & of Nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation.

We hold these truths to be self-evident…

Fallacies & Pitfalls

Most of your design, coding, and testing time will be spent refactoring.

- "We should just throw this out and start over"
- Mixing refactoring with enhancement
- Abuse of metrics
- Waiting too long to do a "big refactor" (vs. continuous refactoring)

Which is TRUE regarding refactoring?

- Refactoring usually results in more concise code (fewer total LOC)
- Refactoring should not cause existing tests to fail
- Refactoring addresses explicit (vs. implicit) customer requirements
- Refactoring often results in changes to the test suite

*