המכללה האקדמית להנדסה ירושלים

הנדסת תוכנה

9. קוד קיים – Legacy Code
(מבוא ל-Refactoring ועקרונות תיכון מונחה עצמים OODP)

"Simplicity is prerequisite for reliability"
- E. W. Dijkstra

"Writing code a computer can understand is science.
Writing code other programmers can understand is an art.", Jason Gorman

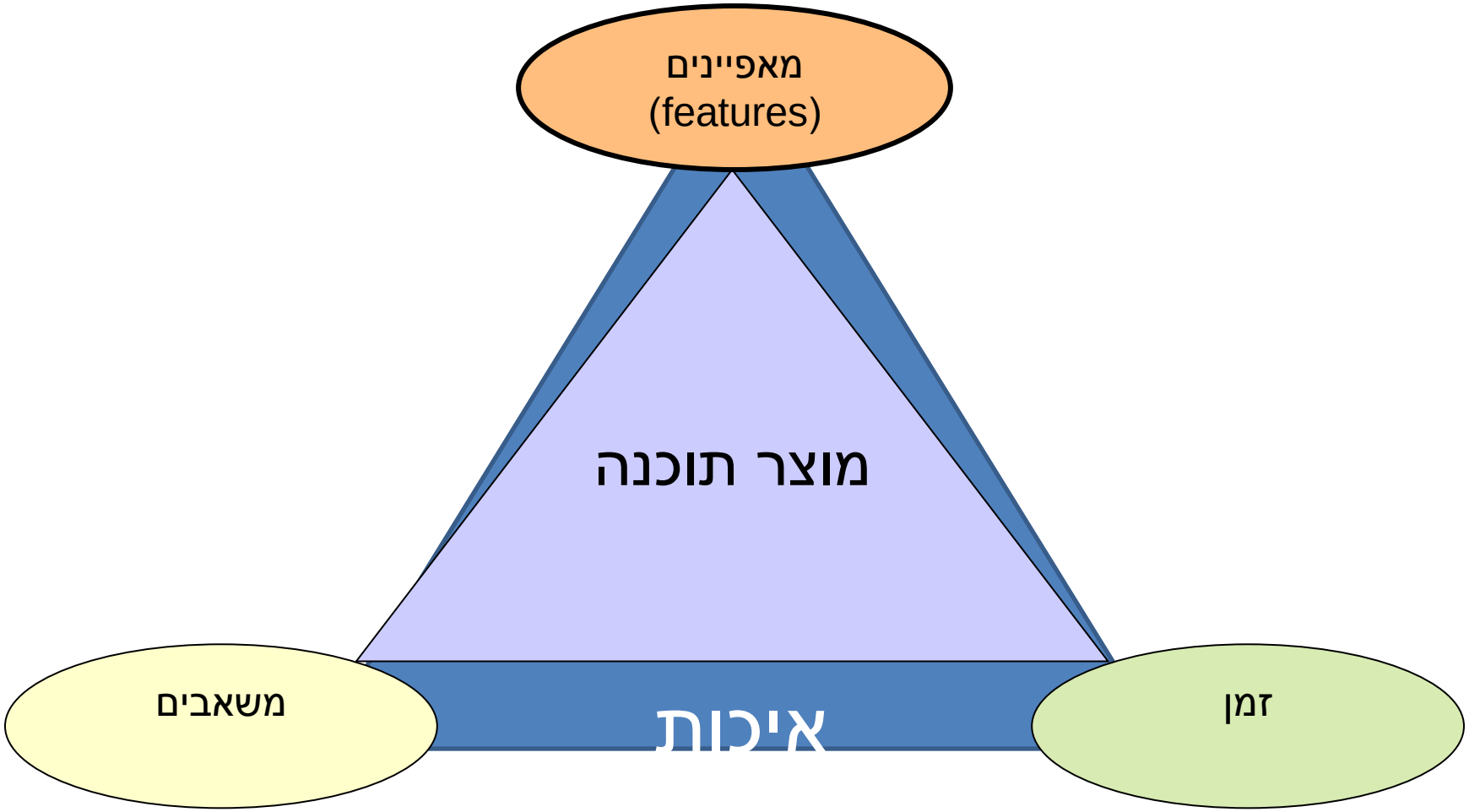
מקורות

- Feathers, Working Effectively with Legacy Code
- Pressman 8.13.11
- Robert Martin,
  Design Principles and Design Patterns
  http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf (pointers article)
  - Books: "Agile Software Development, Principles, Patterns, and Practices", "Agile Principles, Patterns, and Practices in C#"
- Metz, Practical Object Oriented Design in Ruby
  (motivation for design 1:40min, and here)
- Smith, http://www.pluralsight-training.net/microsoft/Courses#software-practices Head First OOA&D
- Motivation slides
  www.lostechies.com/blogs/derickbailey/archive/2009/02/11/solid-development-principles-in-motivational-pictures.aspx

מה היום?

- ראינו: בדיקות ברמות שונות כדרך למוצר איכותי
- בהמשך עוד על איכות תוכנה: עקרונות תיכון מונחה עצמים
  Object Oriented Design Principles
  – משימת סבב 3
- היום: מבוא דרך עבודה עם קוד קיים
- בפעם הבאה: הרצאת אורח – יזמות
- אח"כ: המשך העקרונות ותבניות תיכון
- הרצאה 3\תרגיל:
  – סקר בדיקות יחידה

תזכורת: פרויקט תוכנה:



איכות תוכנה

- מרכיבים חיצוניים:
  - נראים ללקוח\למשתמש
  - דוגמאות:
    - בעלי ערך ללקוח!
    - שמישות
    - נכונות
    - עמידות
    - הרחבתיות?
- מרכיבים פנימיים
  - נראים למפתחים
  - דוגמאות:
    - הסתרת מידע
    - עקיבות
    - פשטות \ קריאות
    - סגנון הקוד
    - יכולת להשתנות

[Begel & Simon 08], עובדים חדשים במיקרוסופט מבלים את רוב השנה הראשונה בקריאת קוד

איכות תוכנה

- איך משיגים תוכנה איכותית?
- ראינו יעדים כלליים: צימוד (coupling) נמוך, לכידות (cohesion) גבוהה
  כיצד משיגים אותם?
- עקרונות + תבניות + הרגלים = תוכנה איכותית
- פיתוח תוכנה מונחה עצמים שולט (?), לכן נדון מכיוון זה בהמשל

מתוך קורס הנדסת תוכנה בברקלי

- קורס בן כמה שנים מקביל לשלנו (וידאו)
  - Fox & Patterson –
  - peer instruction –
- לאחרונה כקורס מקוון (II I coursera/edx)בשילוב תוכנה כשירות (SaaS) > K100 סטודנטים + ספר
- קורס בחירה דומה במכללה (ענן וה"ת II)
  - נלמד הפעם כמה עקרונות כלליים יותר
  - לפי הזמן גם מונחה עצמים
  - גיוון והשוואה

What Makes Code "Legacy" and How Can Agile Help? *(ELLS §8.1)*

Armando Fox

Maintenance != bug fixes

- Enhancements: 60% of maintenance costs
- Bug fixes: 17% of maintenance costs

Hence the “60/60 rule”:

- 60% of software cost is maintenance
- 60% of maintenance cost is enhancements.

Glass, R. *Software Conflict.* Englewood Cliffs, NJ: Yourdon Press, 1991

*

Legacy Code Matters

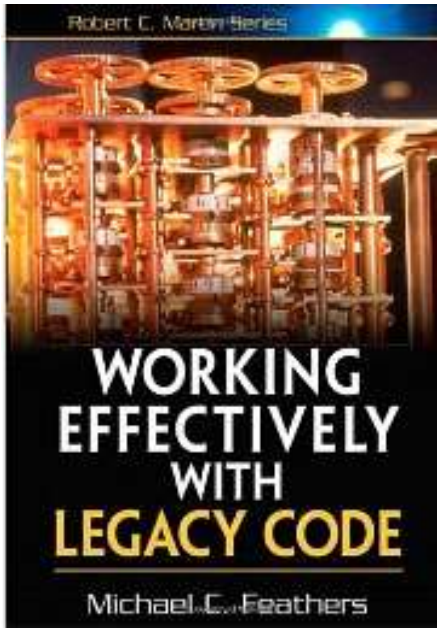
- Since maintenance consumes ~60% of software costs, *it is probably the most important life cycle phase of software . . .*

"Old hardware becomes obsolete;
old software goes into production every night."

Robert Glass, *Facts & Fallacies of Software Engineering (fact #41)*

What makes code “legacy”?

- Still meets customer need, **AND:**
- You didn’t write it, and it’s poorly documented
- You did write it, but a long time ago (and it’s poorly documented)
- *It lacks good tests (regardless of who wrote it)*—Feathers 2004



*

2 ways to think about modifying legacy code

- Edit & Pray
- "I kind of think I probably didn't break anything"

- Cover & Modify
- Let *test coverage* be your safety blanket

*

How Agile Can Help

1. **Exploration:** determine where you need to make changes (*change points)*
2. **Refactoring**: is the code around change points (a) tested? (b) testable?
   - (a) is true: good to go
   - !(a) && (b): apply BDD+TDD cycles to improve test coverage
   - !(a) && !(b): **refactor**

How Agile Can Help, cont.

3. Add tests to **improve coverage** as needed
4. **Make changes**, using tests as *ground truth*
5. **Refactor** further, to leave codebase better than you found it

- This is "embracing change" on long time scales

"Try to leave this world a little better than you found it."

Lord Robert Baden-Powell, founder of the Boy Scouts

If you've been assigned to modify legacy code, which statement would make you happiest if true?

- ☐ "It was originally developed using Agile techniques"
- ☐ "It is well covered by tests"
- ☐ "It's nicely structured and easy to read"
- ☐ "Many of the original design documents are available"

Approaching & Exploring Legacy Code*(ELLS §8.2)*

Armando Fox

Interlude/Armando's Computer History Minute

Always mount a scratch monkey

More folklore: http://catb.org/jargon

*

Get the code running in development

- Check out a *scratch branch* that won't be checked back in, and get it to run
- In a production-like setting or development-like setting
- Ideally with something resembling a **copy** of production database
- Some systems may be too large to clone
- Learn the user stories: Get customer to talk you through what they're doing

Understand database schema & important classes

- Inspect database schema (**rake db:schema:dump)**

•Create a model interaction diagram automatically (**gem install railroady**) or manually by code inspection

•What are the main (highly-connected) *classes*, their *responsibilities*, and their *collaborators?*

Class-Responsibility-Collaborator (CRC) Cards(Kent Beck & Ward Cunningham,OOPSLA 1989)

Showing	
Responsibilities	*Collaborators*
Knows name of movie	Movie
Knows date & time	
Computes ticket availability	Ticket

Ticket	
Responsibilities	*Collaborators*
Knows its price	
Knows which showing it's for	Showing
Computes ticket availability	
Knows its owner	Patron

Order	
Responsibilities	*Collaborators*
Knows how many tickets it has	Ticket
Computes its price	
Knows its owner	Patron
Knows its owner	Patron

CRC's and User Stories

Feature: Add movie tickets to shopping cart
As a patron
So that I can attend a showing of a movie
I want to add tickets to my order

Scenario: Find specific showing
Given a showing of "Inception" on Oct 5 at 7pm
When I visit the "Buy Tickets" page
Then the "Movies" menu should contain "Inception"
And the "Showings" menu should contain "Oct 5, 7pm"

Scenario: Find what other showings are available
Given there are showings of "Inception" today at 2pm,4pm,7pm,10pm
When I visit the "List showings" page for "Inception"
Then I should see "2pm" and "4pm" and "7pm" and "10pm"

Codebase & “informal” docs

- Overall codebase *gestalt*
- Subjective code quality? (We’ll show tools to check)
- Code to test ratio? Codebase size? (**rake stats**)
- Major models/views/controllers?
- Cucumber & Rspec tests
- RDoc documentation

http://pastebin.com/QARUzTnh

- Informal design docs
- Lo-fi UI mockups and user stories
- Archived email, newsgroup, internal wiki pages or blog posts, etc. about the project
- Design review notes (eg Campfire Design review notes (eg Campfire or Basecamp)

*

Ruby RDoc Example | RDoc Documentation

Files	Classes	Methods
date_calculator.rb	DateCalculator	current_year_from_days (DateCalculator) new (DateCalculator)

Class **DateCalculator**

In: date_calculator.rb

Parent: Object

This class calculates the current year given an origin day supplied by a clock chip.

Author: Armando Fox

Copyright: Copyright(C) 2011 by Armando Fox

License: Distributed under the BSD License

Methods

current_year_from_days new

Public Class methods

new(*origin_year*)

Create a new DateCalculator initialized to the origin year

- origin_year - days will be calculated from Jan. 1 of this year

Public Instance methods

current_year_from_days(*days_since_origin*)

Returns current year, given days since origin year

- days_since_origin - number of days elapsed since Jan. 1 of origin year

[Validate]

*

Summary: Exploration

- "Size up" the overall code base
- Identify key classes and relationships
- Identify most important data structures
- Ideally, identify place(s) where change(s) will be needed
- Keep design docs as you go
- diagrams
- GitHub wiki
- comments you insert using RDoc

“Patrons can make donations as well as buying tickets. For donations we need to track which fund they donate to so we can create reports showing each fund’s activity. For tickets we need to track what show they’re for so we can run reports by show, plus other things that aren’t true of donations, such as when they expire.”
Which statement is LEAST compelling for this design?

- ☐ Donation has at least 3 collaborator classes.
- ☐ Donations and Tickets should subclass from a common ancestor.
- ☐ Donations and Tickets should implement a common interface such as “Purchasable”.
- ☐ Donations and Tickets should implement a common interface such as “Reportable”.

*

Establishing Ground Truth With Characterization Tests *(ELLS §8.3)*

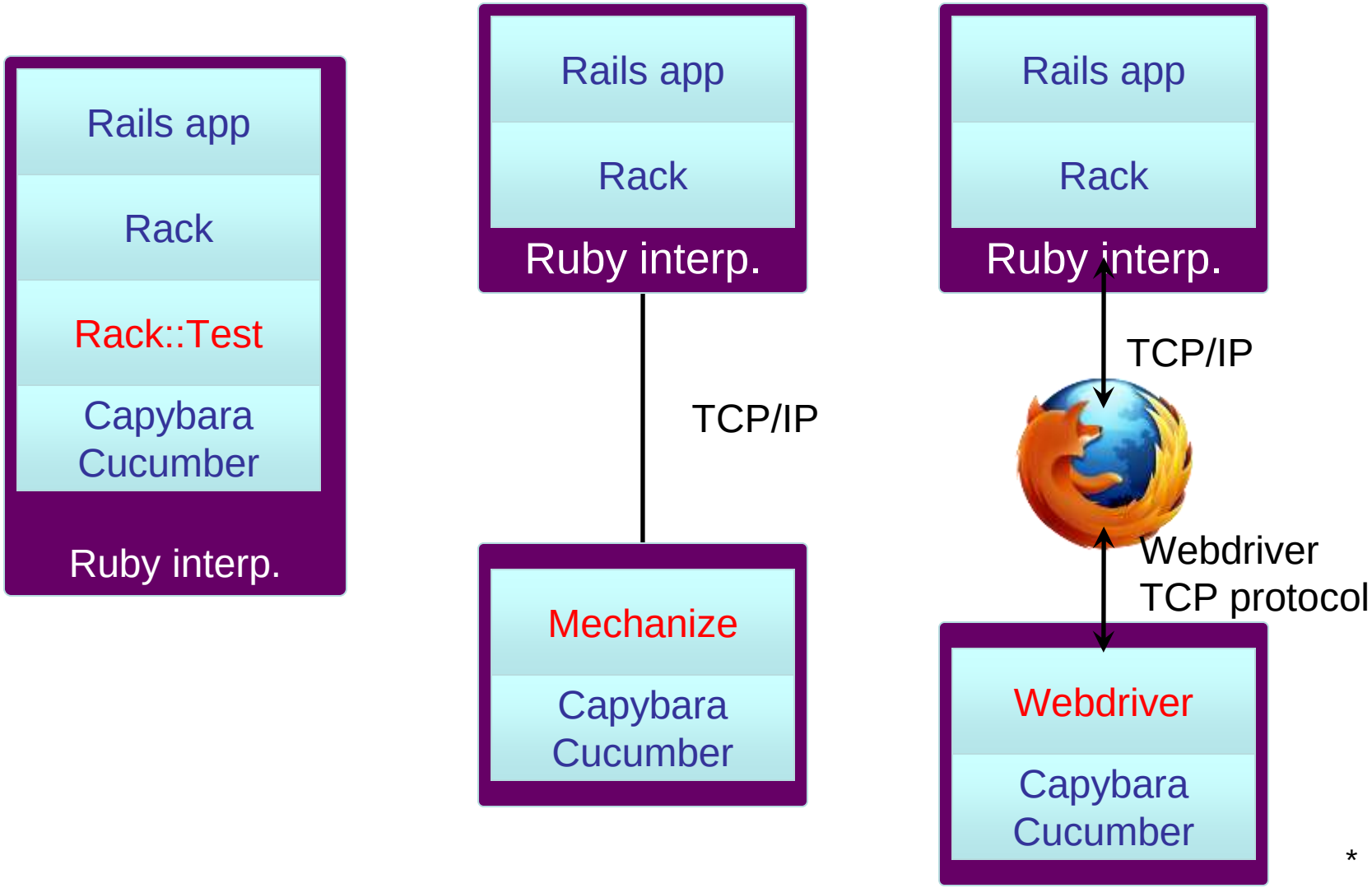
Characterization Tests

- 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)

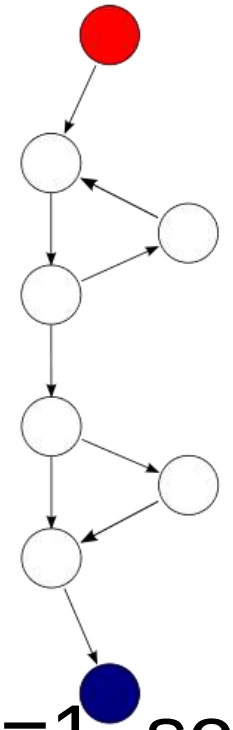
- "Hotspots": places where *multiple metrics* raise red flags
- add `require 'metric_fu'` to **Rakefile**
- **rake metrics:all**
- Take metrics with a grain of salt
- Like coverage, better for *identifying where improvement is needed* than for *signing off*

*

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

*

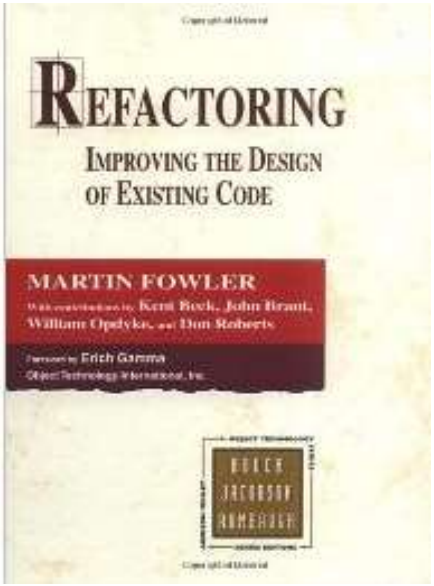
Intro to Method-Level Refactoring *(ELLS §8.5)*

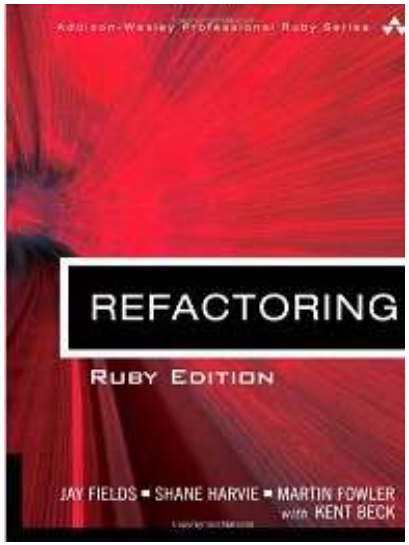
Refactoring: Idea

- 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 — http://pastebin.com/pYCfMOJp
- Extract method — http://pastebin.com/sXVDW9C6
- Extract method, encapsulate class — http://pastebin.com/zWM2ZqaW
- Test extracted methods — http://pastebin.com/DRpNPzpT
- Some thoughts on unit testing
- Glass-box testing can be useful while refactoring
- Common approach: test *critical values* and *representative noncritical values*

*

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 conditionalReplace loop with collection methodExtract methodExtract 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

*

בפעם הבאה

- אורח – יזמות
- המשך עקרונות תיכון מונחה עצמים
  - סקרי קוד בהתאם
- מימוש מקובל של עקרונות:
  תבניות עיצוב (תיכון) Design Patterns
- עוד על Refactoring
- קריאה:

"Separation of Concern vs Single Responsibility Principle (SoC vs SRP)"

שאלה: מהו ההבדל העיקרי בין שני העקרונות המוזכרים?

לסיכום

- קוד קיים \ Legacy Code ואיכות
- Refactoring \ שיפרוק?