

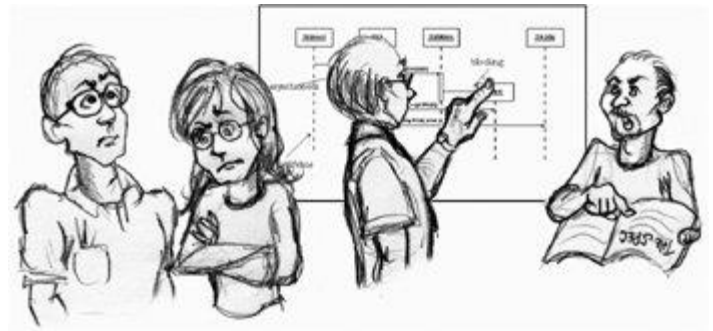


# הנדסת תוכנה

## 10. קוד קיים – Legacy Code (מבוא לעקרונות תיכון ותבניות מונחי 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](#)



# מה היום?

- ראינו: בדיקות ברמות שונות כדרך למוצר איכותי
  - מה עם אבטחה?... (מצגת)
  - בהמשך עוד על **איכות תוכנה**: עקרונות תיכון מונחה עצמים  
Object Oriented Design Principles, תבניות תיכון
  - משימת סבב - סקר שיפורי קוד
- היום: מבוא: עבודה עם קוד קיים
  - סיום משימה 4
  - משימה אישית 5 (בנוס) – Refactoring
- הרצאה 3\תרגיל:
  - מצגות סבב

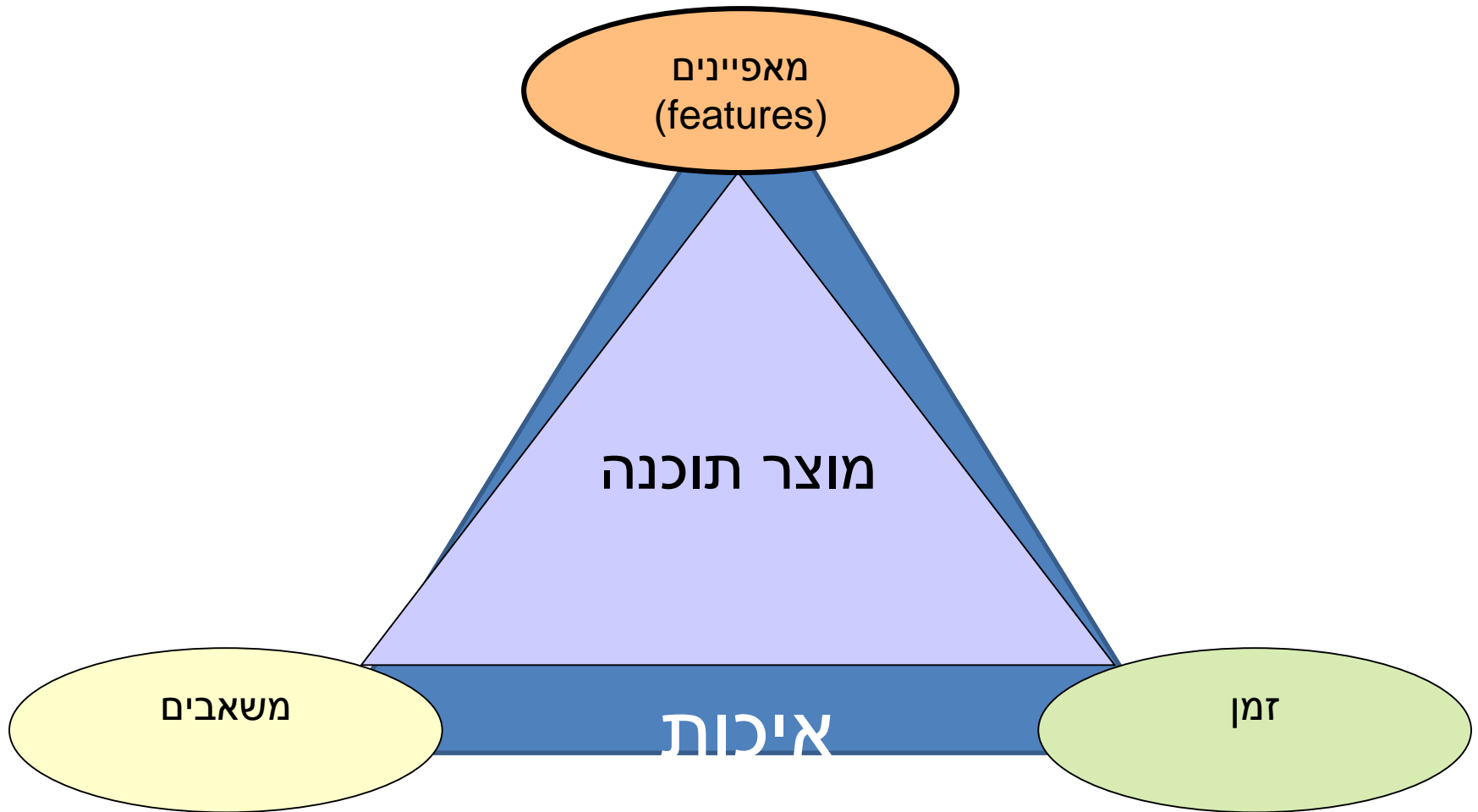
# מקורות

- Feathers, Working Effectively with Legacy Code
- Fowler, Refactoring
- Berkeley/Edx SAAS course

# קישורים להרחבה

- [How about learning f\\*\\*\\*ing programming?](#)  
Short video on testing/refactoring/design
- Fowler, [An Appropriate Use of Metrics](#), 2013
- Smell and Refactoring lists:
  - <http://www.soberit.hut.fi/mmantyla/BadCodeSmellsTaxonomy.htm>
  - <http://www.codinghorror.com/blog/2006/05/code-smells.html>
  - <http://users.csc.calpoly.edu/~jdalbey/305/Lectures/SmellsToRefactorings>

# תזכורת: פרויקט תוכנה:



# איכות תוכנה

- מרכיבים פנימיים

- נראים למפתחים

- דוגמאות:

- מודולריות

- עקיבות

- פשטות \ קריאות

- סגנון הקוד

- יכולת להשתנות ולהתפתח

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

- מרכיבים חיצוניים:

- נראים ללקוח\למשתמש

- דוגמאות:

- בעלי ערך ללקוח! (TTM)

- נכונות, יעילות

- שמישות

- עמידות

- הרחבתיות?

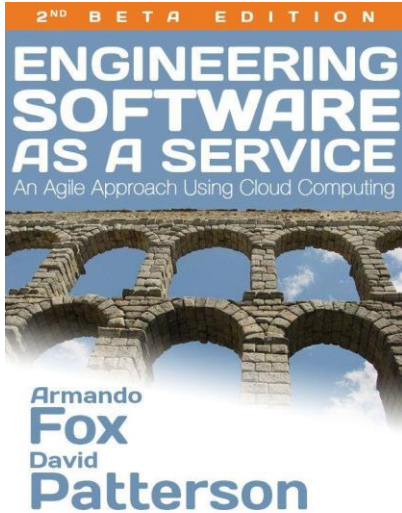
➤ אי אפשר את כולם ביחד

Butler W. Lampson, [Hints for Computer System Design](#), 1983  
(2013 [lecture](#))

# איכות תוכנה

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

# מתוך קורס הנדסת תוכנה בברקלי



- קורס ה"ת ותיק (וידאו, 2013)  
Fox & Patterson –  
Industry: legacy code –  
(obamacare) –
- קורס מקוון (I II coursera/edx) בשילוב תוכנה  
כשירות (SaaS)  $K100 <$  סטודנטים + ספר
- קורס בחירה דומה במכללה (הנדסת תוכנה  
למחשוב ענן)  
– דוגמאות קוד בשפת ruby ☹



# What Makes Code “Legacy” and How Can Agile Help? (ESaaS § 9.1)

Armando Fox

# 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)

*How do we understand and **safely** modify  
legacy code?*

# Maintenance != bug fixes

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

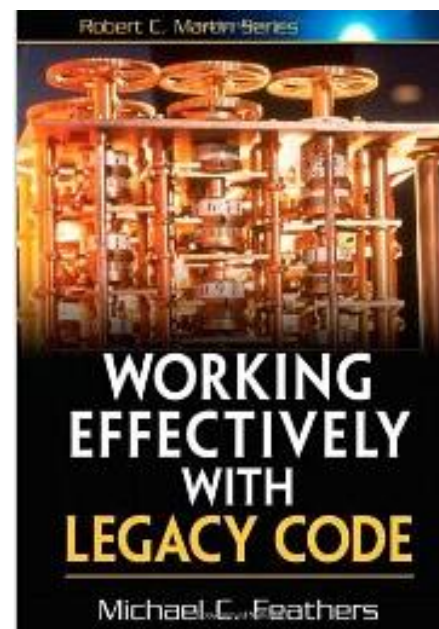
Hence the “60/60 rule”:

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

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

# 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 (*ESaaS § 9.2*)

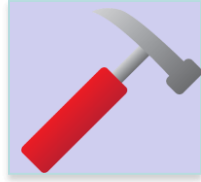
Armando Fox

# 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? (**reek**, **flog**, **flay**, **saikuro**)
  - Code to test ratio? Codebase size? (**rake stats**)
  - Major models/views/controllers?
  - Cucumber & Rspec tests
- 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](#) or [Basecamp](#))
  - Commit logs in version control system (**git log**)
  - RDoc documentation

Ruby RDoc Example	RDoc Documentation	+
Files	Classes	Methods
<a href="#">date_calculator.rb</a>	<a href="#">DateCalculator</a>	<a href="#">current_year_from_days</a> (DateCalculator) <a href="#">new</a> (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



# Establishing Ground Truth With Characterization Tests (*ESaaS § 9.3*)

# Why?

- You don't want to write code without tests
- You don't have tests
- You can't create tests without understanding the code

*How do you get started?*

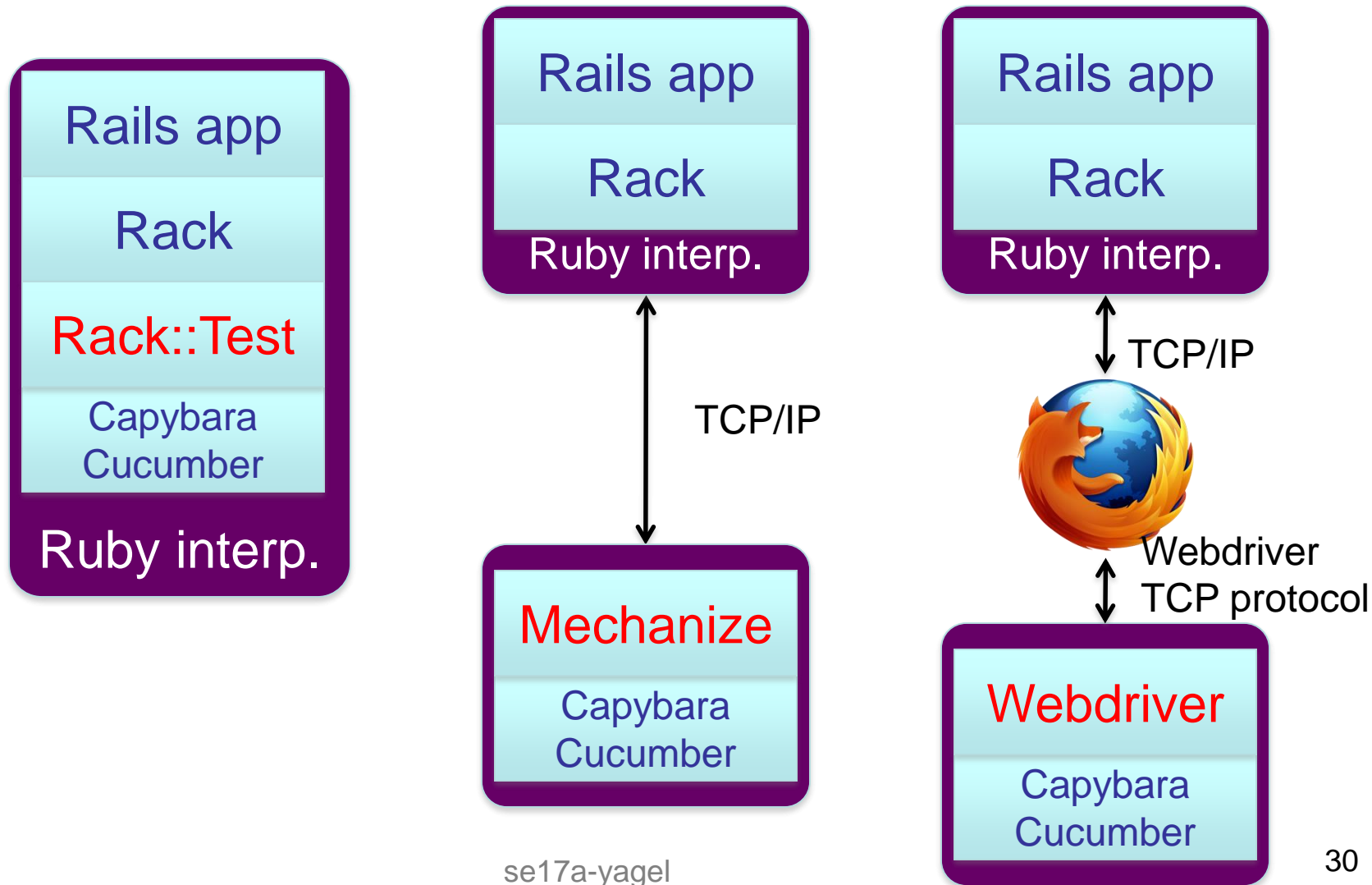
# Characterization Tests

- 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!**
- <https://github.com/github/scientist>

# Integration-Level Characterization Tests

- Natural first step: black-box/integration level
  - doesn't rely on 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](http://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 (ESaaS § 9.4)



<http://pastebin.com/gtQ7QcHu>



# Quantitative: Metrics

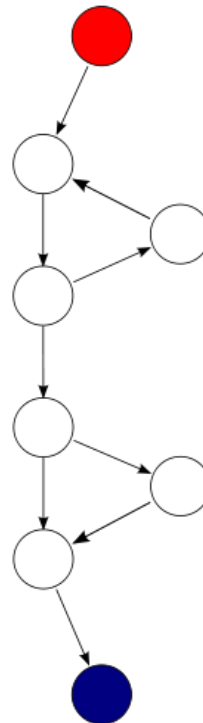
Metric	Tool	Target score
Code-to-test ratio	rake stats	$\leq 1:2$
C0 (statement) coverage	SimpleCov	90%+
Assignment-Branch-Condition score	flog	< 20 per method
Cyclomatic complexity	saikuro	< 10 per method (NIST)

- “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.)  $\leq 10$ /module

# Qualitative: Code Smells

**SOFA** captures symptoms that often indicate code smells:

- Be **s**hort
- Do **o**ne thing
- Have **f**ew arguments
- Consistent level of **a**bstraction

# 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
- Like a good news story, classes & methods should read “top down”!
- Example: encourage customers to opt in

<http://pastebin.com/AFQAKxbR>

# 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*
- Example: mailing address for **Moviegoer**

# 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	
<i>Responsibilities</i>	<i>Collaborators</i>
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	

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

(*ESaaS § 9.5*)

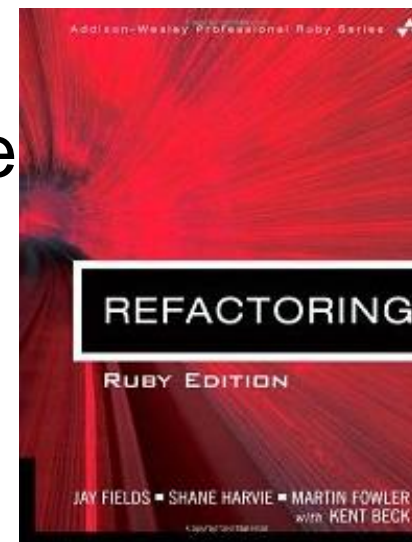
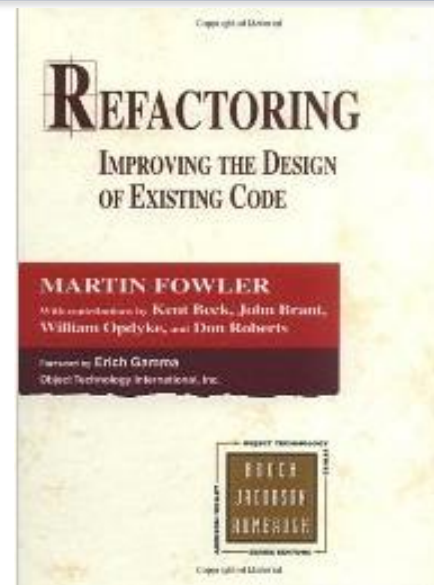


# Refactoring: Idea

- 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/pYCfMQJp>
- Extract method <http://pastebin.com/sXVDW9C6>
- Extract method, encapsulate class <http://pastebin.com/yrmyVd7R>
- Test extracted methods <http://pastebin.com/vNw66mn9>
- 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
- Observation: if we had developed method using TDD, might have gone easier!
- Improved our **flog** & **reek** scores

# Other Smells & Remedies

Smell	Refactoring that may resolve it
Large class	Extract class, subclass or module
Long method	<b>Decompose conditional</b> Replace loop with collection method <b>Extract method</b> Extract enclosing method with <code>yield()</code> 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	<b>Extract methods &amp; classes</b>

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.

*(ESaaS § 9.8-9.10)*



# 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 fewer total lines of code
- ☐ Refactoring should not cause existing tests to fail
- ☐ Refactoring addresses explicit (vs. implicit) customer requirements
- ☐ Refactoring often results in changes to the test suite



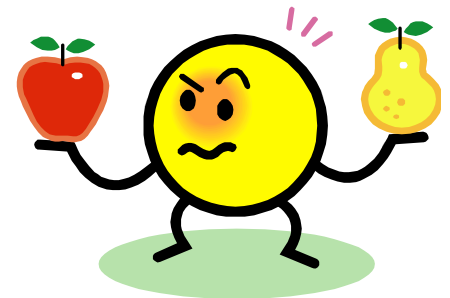
# Plan-And-Document Perspective on Software Maintenance:

*(Engineering Software as a Service § 9.7)*

David Patterson

# P&D Maintenance?

- How much spent on P&D development vs. P&D maintenance?
  - How does this compare to Agile?
- Agile developers maintain code
  - Does P&D use same or different people for maintenance?
- What does the P&D Maintenance Documentation look like?



# P&D Maintenance

- P&D spends 1/3 on development, 2/3 on maintenance
  - Customers pay 10%/year SW maintenance fee
- Development  $\neq$  Maintenance Team
  - Maintenance Managers
  - Maintenance SW Engineers
  - Typically less prestigious



# Maintenance Manager

- Like Development Manager
  - Estimate costs, maintain schedule, evaluate risks & overcomes them
  - Recruits maintenance team
  - Evaluate software engineers performance, which sets salary
  - Document project maintenance plan (maintain documents & code)
    - IEEE standard to follow
  - Blamed if upgrade takes too long or too expensive





# P&D Maintenance Process

- Differences vs. Development Process:

1. Working SW in field

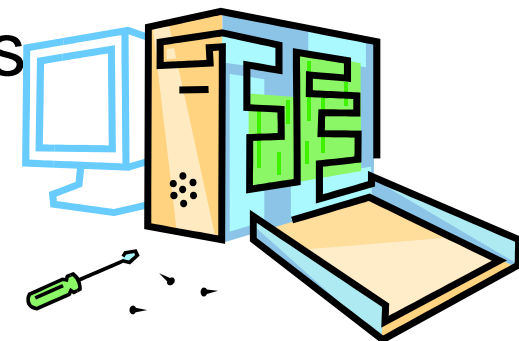
- New releases can't break features

2. Customer collaboration

- Work with customer to improve in next release vs. meet contract spec

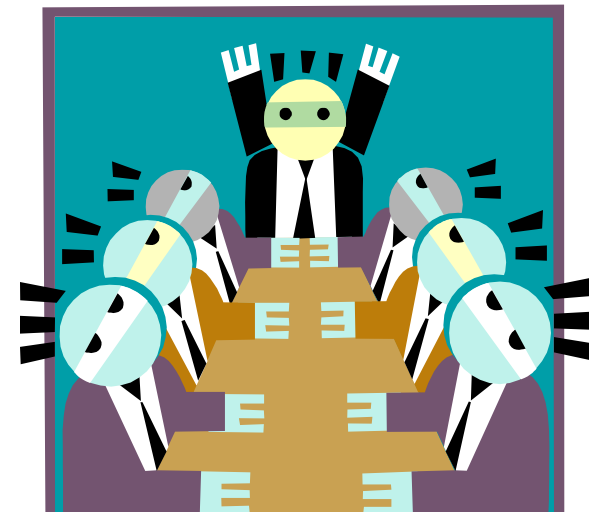
3. Responding to change

- Customers send *change requests*, which SW engineers must prioritize
- *Change request forms* have ticket tracking



# Change Control Board

- Board (not Manager) decides
- Manager estimates cost/time per change request
- QA team gives cost of testing for change request, including regression testing + new tests
- Documentation teams gives cost of updating docs
- Customer support group decides if urgent or workaround



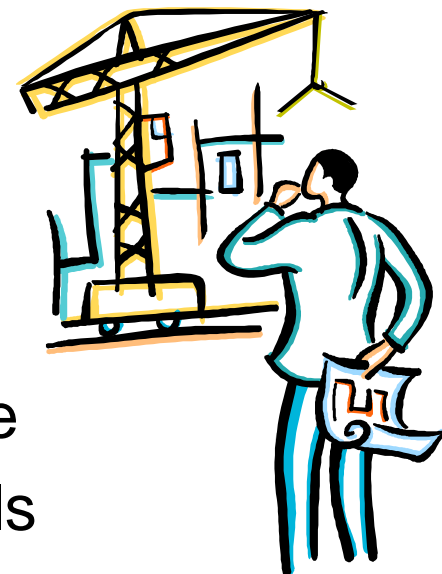
# Urgent Change Request

- No time to update docs, plans & code
  - Software product crashes
  - New laws affect product
  - Security hole => data vulnerable
  - New releases of underlying OS/library
  - Must match competitor's new feature
- Synch after emergency?
  - Emergencies too frequent to catch up?
- Time to refactor to improve maintainability
  - Too expensive for Change Control Board?

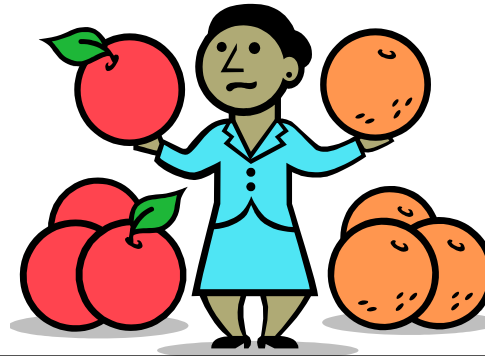


# Time to Re-Engineer?

- Time to refactor to improve maintainability?
  - Refactor continuous during dev & main
- Re-Engineer to Improve vs. Replace?
  - Use automated tools to upgrade as SW ages and maintenance hard
    - Change database schema
    - Improve documentation by reverse engineering
    - Code analysis tools to point to bad code
    - Programming Language translation tools



# Maintenance: P&D vs. Agile



<i>Tasks</i>	<i>In Plan and Document</i>	<i>In Agile</i>
Customer change request	Change request forms	User story on 3x5 cards in Connextra format
Change request cost/time estimate	By Maintenance Manager	Points by Development Team
Triage of change requests	Change Control Board	Development team with customer participation
<i>Roles</i>		
	Maintenance Manager	n.a.
	Maintenance SW Engineers	Development team
	QA team	
	Documentation teams	
	Customer support group	

Which statement regarding P&D maintenance is FALSE?

1. The cost of maintenance usually exceeds the cost of development in P&D
2. The Agile equivalent to P&D change requests is user stories; equivalent of change request cost estimates is points; P&D releases are iterations
3. The Agile lifecycle is similar to the P&D maintenance lifecycle: enhancing working software product, collaborating with customer vs. negotiating by contract, continuously responding to change
4. All the above are true

# Provocative Question

- If 2/3 cost of product are in the maintenance phase, why not use same maintenance-compatible software development process for whole lifecycle (Agile) vs. separate processes (and teams) for development and maintenance?



# משימה אישית 5 – בונוס

## Gilded-Rose Kata

- מטרות

- עבודה עם קוד קיים
- העמקה בבדיקות (איכות, יחידה מול פונקציונליות, ...)
- Refactoring ועקרונות תיכון מונחה עצמים וכדו'
- מקור [כאן](#) מבוסס על [שם](#).

- הוראות:

- Fork&clone: <https://github.com/jce-il/GildedRose-Refactoring-Kata>
  - [Requirements](#)
  - Do the JS/Jasmine part. Other languages: ask TA
- Add tests to cover existing features
- Refactor
- Add the new feature (with TDD)
- Commit every phase and stage (incl. selected [Refactoring](#))
- Submit a PR (by last lecture), get some feedback, etc.
- Extra Bonus: Hebrew or node.js + mocka forks



# בפעם הבאה - ?

- המשך עקרונות תיכון מונחה עצמים
- מימוש מקובל של עקרונות:
- תבניות **עיצוב (תיכון)** Design Patterns
- עוד על Refactoring
- עבודה על תרגיל 5
- עוד: חווית משתמש, כלים 2

# לסיכום

- קוד קיים \ Legacy Code
- Refactoring \ [שיפרוק?](#)
- תיכון ואיכות מתמשכים, הקשר לבדיקות
- האם הקוד שלכם כבר legacy?
- משימת סבב – סקר שיפורי קוד