[IF977] Engenharia de Software

Prof. Vinicius Cardoso Garcia
vcg@cin.ufpe.br :: @vinicius3w :: assertlab.com

AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
Universidade Federal de Pernambuco

Licença do material

Este Trabalho foi licenciado com uma Licença

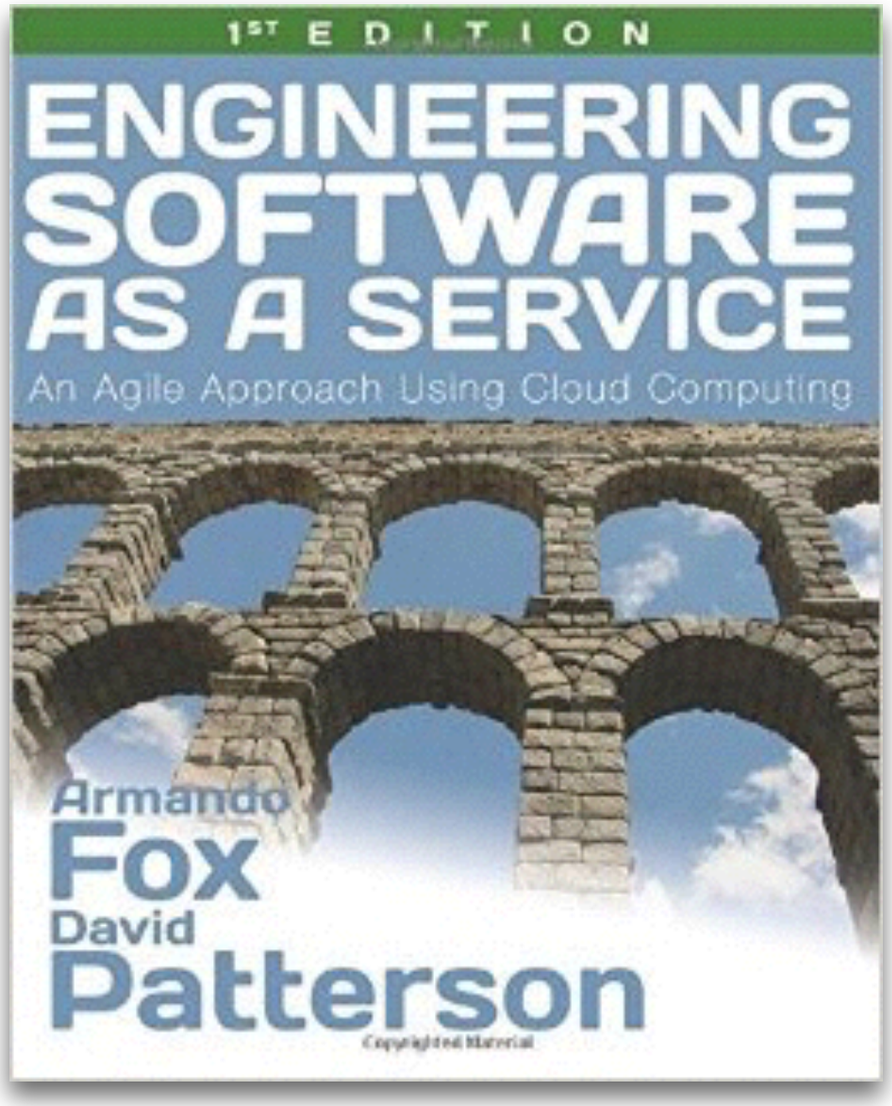
Creative Commons - Atribuição-NãoComercial-Compartilhalgual 3.0 Não Adaptada.

Mais informações visite

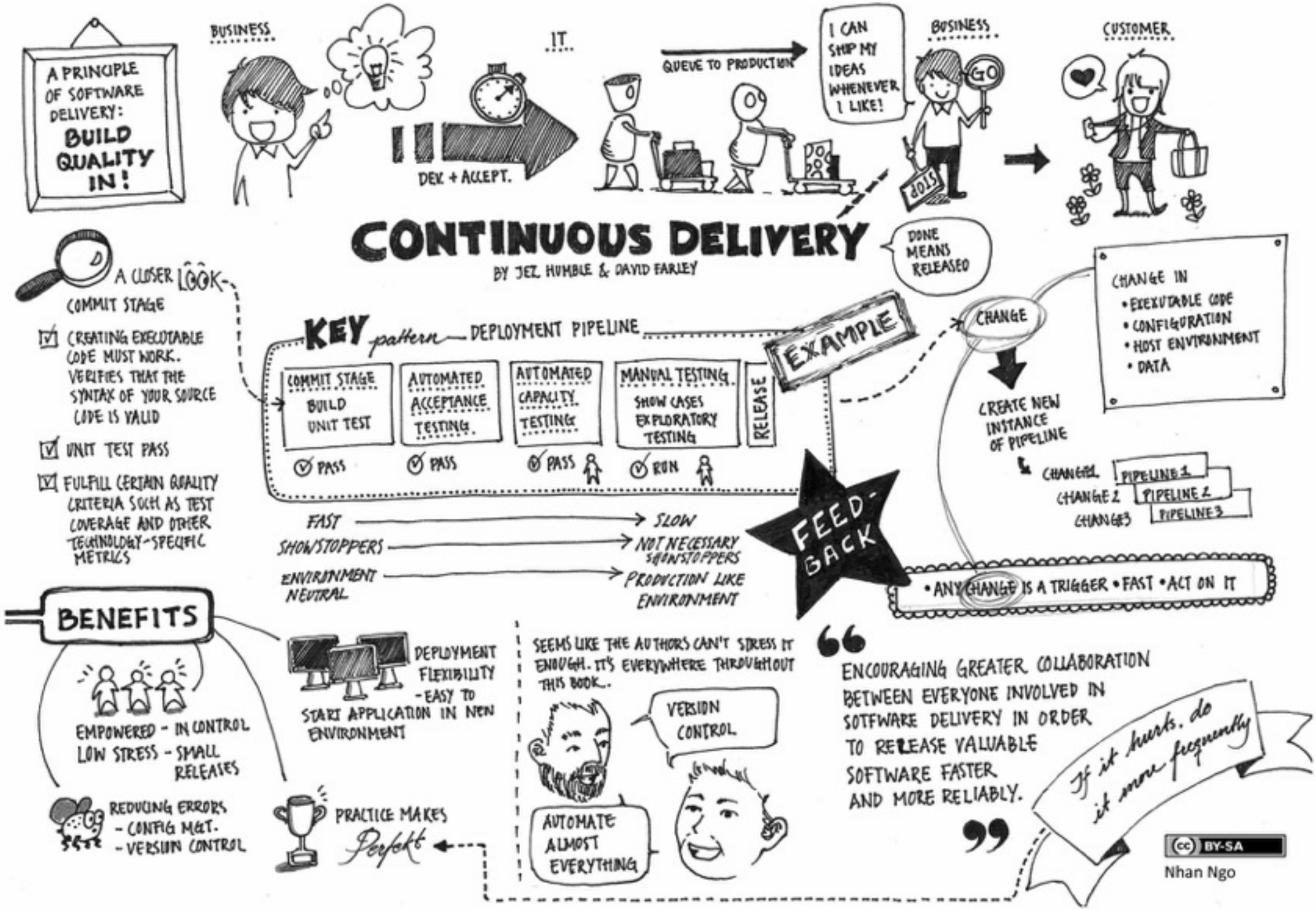
http://creativecommons.org/licenses/by-nc-sa/3.0/deed.pt

Referências

- Continuous Delivery
  - https://continuousdelivery.com
- A biblioteca do Desenvolvedor de Software dos dias de hoje
  - http://bit.ly/TDOA5L
- SWEBOK
  - Guide to the Software Engineering Body of Knowledge (SWEBOK): http://www.computer.org/web/swebok
- Engineering Software as a Service: An Agile Approach Using Cloud Computing (Beta Edition)
  - http://www.saasbook.info/



Releases Then and Now: Windows 95 Launch Party



Continuous Integration & Continuous Deployment

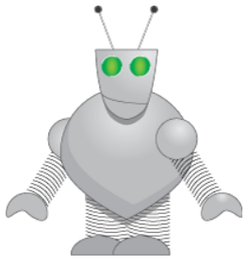
(ESaaS §12.3)

Releases Then and Now

- Facebook: master branch pushed once a week, aiming for once a day (Bobby Johnson, Dir. of Eng., in late 2011)
- Amazon: several deploys per week
- StackOverflow: multiple deploys per day (Jeff Atwood, co-founder)
- GitHub: tens of deploys per day (Zach Holman)
- Rationale: risk == # of engineer-hours invested in product since last deploy!

Like development and feature check-in, deployment should be a **non-event** that happens all the time

Successful Deployment



- **Automation: consistent deploy process**
  - PaaS sites like Heroku, CloudFoundry already do this
  - Use tools like Capistrano for self-hosted sites
- **Continuous integration: integration-testing the app beyond what each developer does**
  - Pre-release code checkin triggers CI
  - Since frequent checkins, CI always running
  - Common strategy: integrate with GitHub

Why CI?

- Differences between dev & production envs
- Cross-browser or cross-version testing
- Testing SOA integration when remote services act wonky
- Hardening: protection against attacks
- Stress testing/longevity testing of new features/code paths
- Example: Salesforce CI runs 150K+ tests and automatically opens bug report when test fails

Continuous Deployment

- Push => CI => deploy several times per day
  - deploy may be auto-integrated with CI runs
- So are releases meaningless?
  - Still useful as customer-visible milestones
  - “Tag” specific commits with release names

```
git tag 'happy-hippo' HEAD
git push --tags
```

- Or just use Git commit ID to identify release

Quiz

RottenPotatoes just got some new AJAX features. Where does it make sense to test these features?

A. Using autotest with RSpec+Cucumber

B. In CI

C. In the staging environment

D. All of these

Quiz

RottenPotatoes just got some new AJAX features. Where does it make sense to test these features?

A. Using autotest with RSpec+Cucumber

B. In CI

C. In the staging environment

D. All of these

What Makes Code "Legacy" and How Can Agile Help?

(ESaaS §9.1)

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

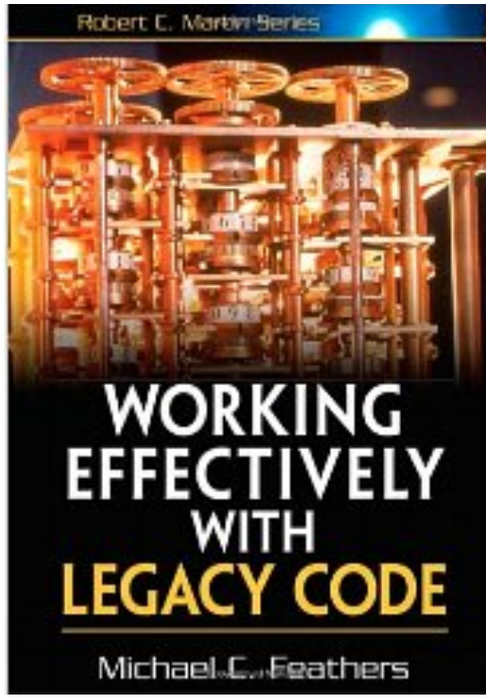
- Enhancements: 60% of maintenance costs
- Bug fixes: 17% of maintenance costs

- 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

Quiz

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

A. "It was originally developed using Agile techniques"

B. "It is well covered by tests"

C. "It's nicely structured and easy to read"

D. "Many of the original design documents are available"

Quiz

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

A. "It was originally developed using Agile techniques"

B. "It is well covered by tests"

C. "It's nicely structured and easy to read"

D. "Many of the original design documents are available"

FIGHT LEGACY CODE

WRITE UNIT TESTS

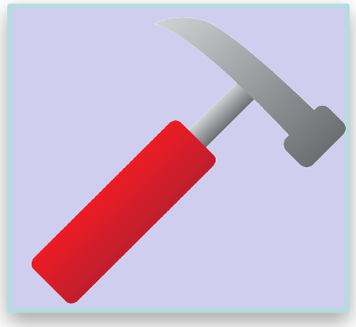
Approaching & Exploring Legacy Code

(ESaaS §9.2)

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*?

Codebase & “informal” docs

- Overall codebase gestalt
  - Subjective code quality? (`rake metrics` after installing metric-fu gem, or CodeClimate)
  - 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, RDoc documentation

Codebase & “informal” docs

```ruby
# This class calculates the current year given an ori
# supplied by a clock chip.
#
# Author:: Armando Fox
# Copyright:: Copyright(C) 2011 by Armando Fox
# License::  Distributed under the BSD License
#
class DateCalculator
  #
  #  Create a new DateCalculator initialized to the o
  # * +origin_year+ - days will be calculated from Ja
  #
  def initialize(origin_year)
    @origin_year = origin_year
  end
  #
  # Returns current year, given days since origin yea
  # * +days_since_origin+ - number of days elapsed si
  #
  def current_year_from_days(days_since_origin)
    return @origin_year
  end
end
```

http://pastebin.com/QARUzTnh

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

Quiz

“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 don't apply to donations, such as when they expire."

Which statement is LEAST compelling for this design?

A. Donation has at least 2 collaborator classes.

B. Donations and Tickets should subclass from a common ancestor.

C. Donations and Tickets should implement a common interface such as “Purchasable”.

D. Donations and Tickets should implement a common interface such as “Reportable”.

Quiz

“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 don't apply to donations, such as when they expire."

Which statement is LEAST compelling for this design?

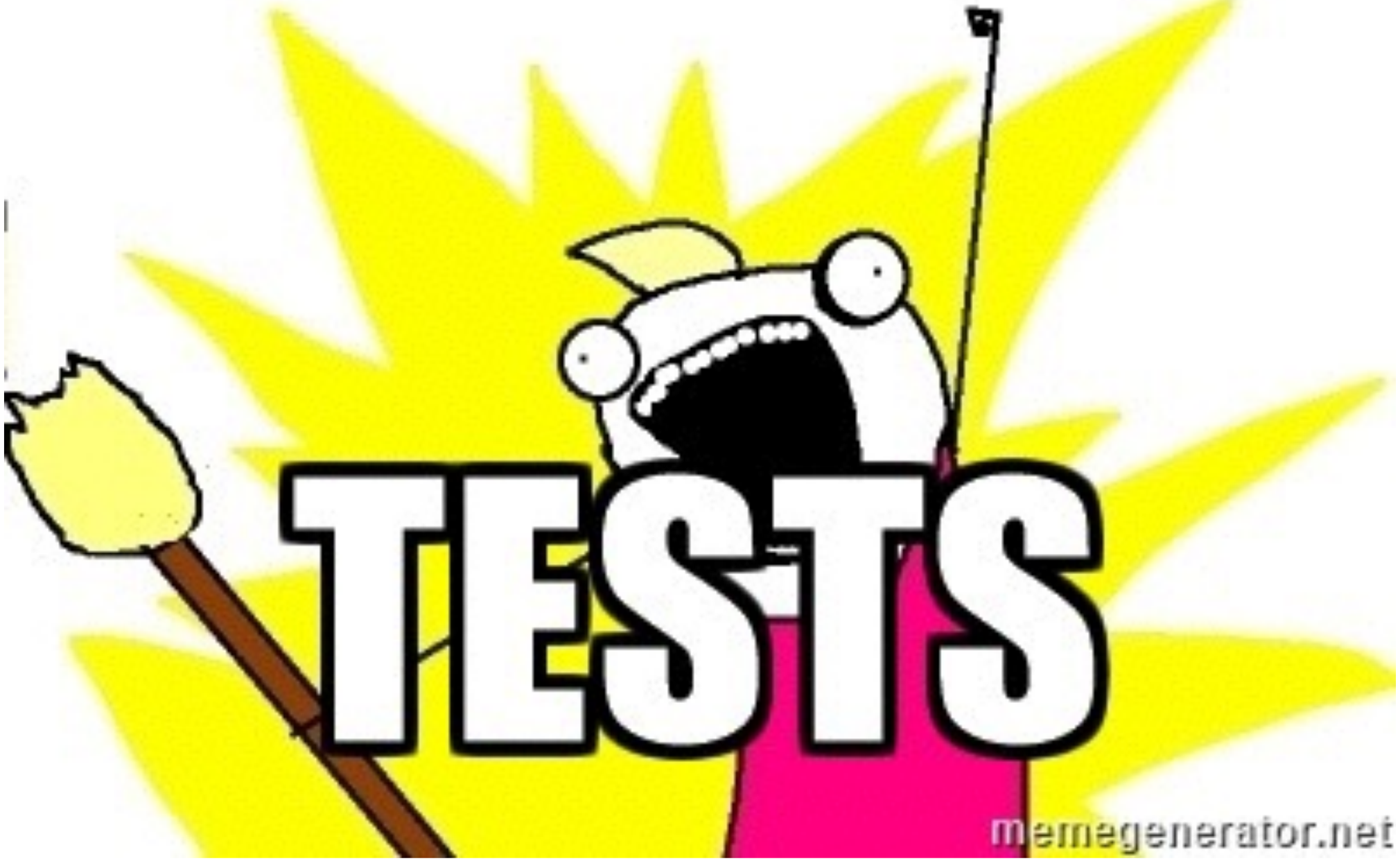
A. Donation has at least 2 collaborator classes.

B. Donations and Tickets should subclass from a common ancestor.

C. Donations and Tickets should implement a common interface such as “Purchasable”.

D. Donations and Tickets should implement a common interface such as “Reportable”.

CHARACTERIZATION



Establishing Ground Truth
With Characterisation 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?

Characterisation Tests

- Establish ground truth about how the app works today, as basis for coverage
  - Makes known behaviours Repeatable
  - Increase confidence that you’re not breaking anything

- **Pitfall: don’t try to make improvements at this stage!**

Integration-Level Characterisation 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

Unit- and Functional-Level Characterization Tests

Cheat: write tests to learn as you go
https://vimeo.com/34754876

```
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
```

Quiz

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

A. They are based on fewer assumptions about how the code works

B. They are just as likely to be unexpectedly dependent on the production database

C. They rely less on detailed knowledge about the code's structure

D. If a customer can do the action, you can create a simple characterization test by mechanizing the action by brute force

Quiz

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

A. They are based on fewer assumptions about how the code works

B. They are just as likely to be unexpectedly dependent on the production database

C. They rely less on detailed knowledge about the code's structure

D. If a customer can do the action, you can create a simple characterization test by mechanizing the action by brute force

Intro to Method-Level Refactoring

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
  - use **`metric_fu`** gem
  - or use *CodeClimate*!
- Take metrics with a grain of salt
  - Like coverage, better for *identifying where improvement is needed* than for *signing off*

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

- *CodeClimate* runs both *qualitative & quantitative* metrics

Single Level of Abstraction

- Complex tasks need divide & conquer
  - or: *Code should tell a story*
  - Like a news story, should read “top down”!
- 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

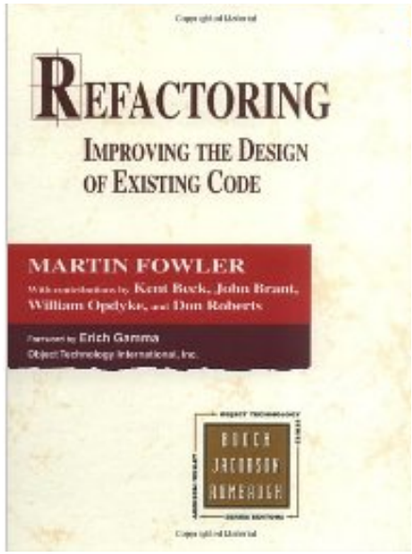
http://pastebin.com/AFQAKxbR

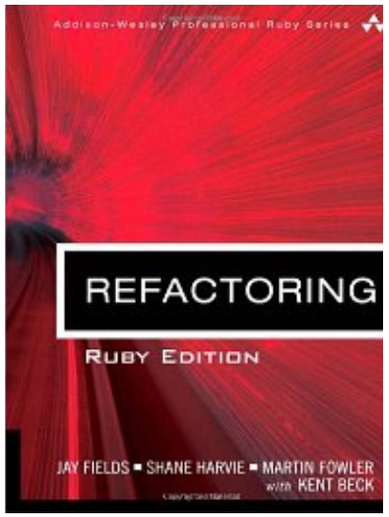
Refactoring: Idea

- 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

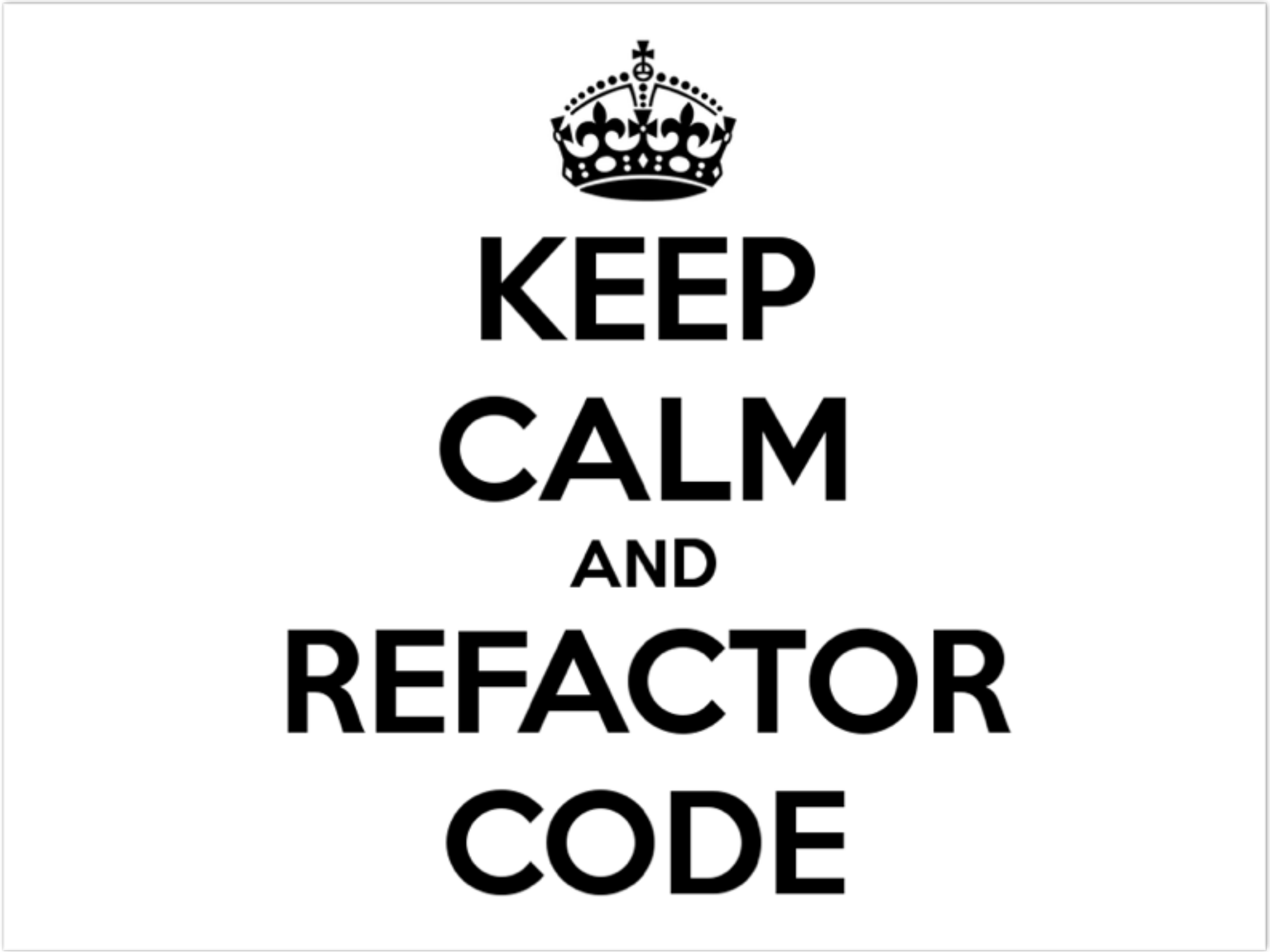
- Fowler et al. developed mostly definitive catalog of refactorings
  - Adapted to various languages
  - Method- and class-level refactorings
- Each refactoring consists of:
  - Name
  - Summary of what it does/when to use
  - Motivation (what problem it solves)
  - Mechanics: step-by-step recipe
  -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





Refactoring TimeSetter

- Fix stupid names http://pastebin.com/gtQ7QcHu
- Extract method http://pastebin.com/pYCfMQJp
- Extract method, encapsulate class http://pastebin.com/sXVDW9C6
- Test extracted methods http://pastebin.com/yrmyVd7R
- Some thoughts on unit testing http://pastebin.com/vNw66mn9
  - Glass-box testing can be useful while refactoring
  - Common approach: test critical values and representative noncritical values



Wrapup of Refactoring example

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	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

Quiz

Which is NOT a goal of method-level refactoring?

A. Reduce code complexity

B. Eliminate code smells

C. Eliminate bugs

D. Improve testability

Quiz

Which is NOT a goal of method-level refactoring?

A. Reduce code complexity

B. Eliminate code smells

C. Eliminate bugs

D. Improve testability

Legacy Code & Refactoring: Reflections, Fallacies, Pitfalls, etc.

(ESaaS §9.8-9.10)

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
- *Technical Debt*: Waiting too long to do a “big refactor” (vs. continuous refactoring)

Quiz

Which is TRUE regarding refactoring?

A. Refactoring usually results in fewer total lines of code

B. Refactoring should not cause existing tests to fail

C. Refactoring addresses explicit (vs. implicit) customer requirements

D. Refactoring often results in changes to the test suite

Quiz

Which is TRUE regarding refactoring?

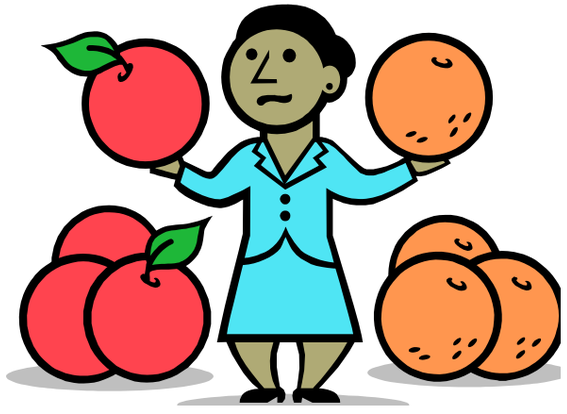
A. Refactoring usually results in fewer total lines of code

B. Refactoring should not cause existing tests to fail

C. Refactoring addresses explicit (vs. implicit) customer requirements

D. Refactoring often results in changes to the test suite

Maintenance: P&D vs. Agile



Tasks	*In Plan and Document*	*In Agile*
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
Roles		
	Maintenance Manager	n.a.
	Maintenance SW Engineers	Development team
	QA team	
	Documentation teams	
	Customer support group	