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)*

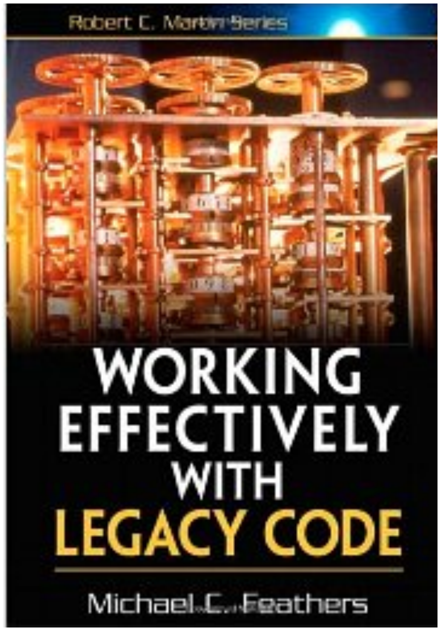
What makes code “legacy”?

•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

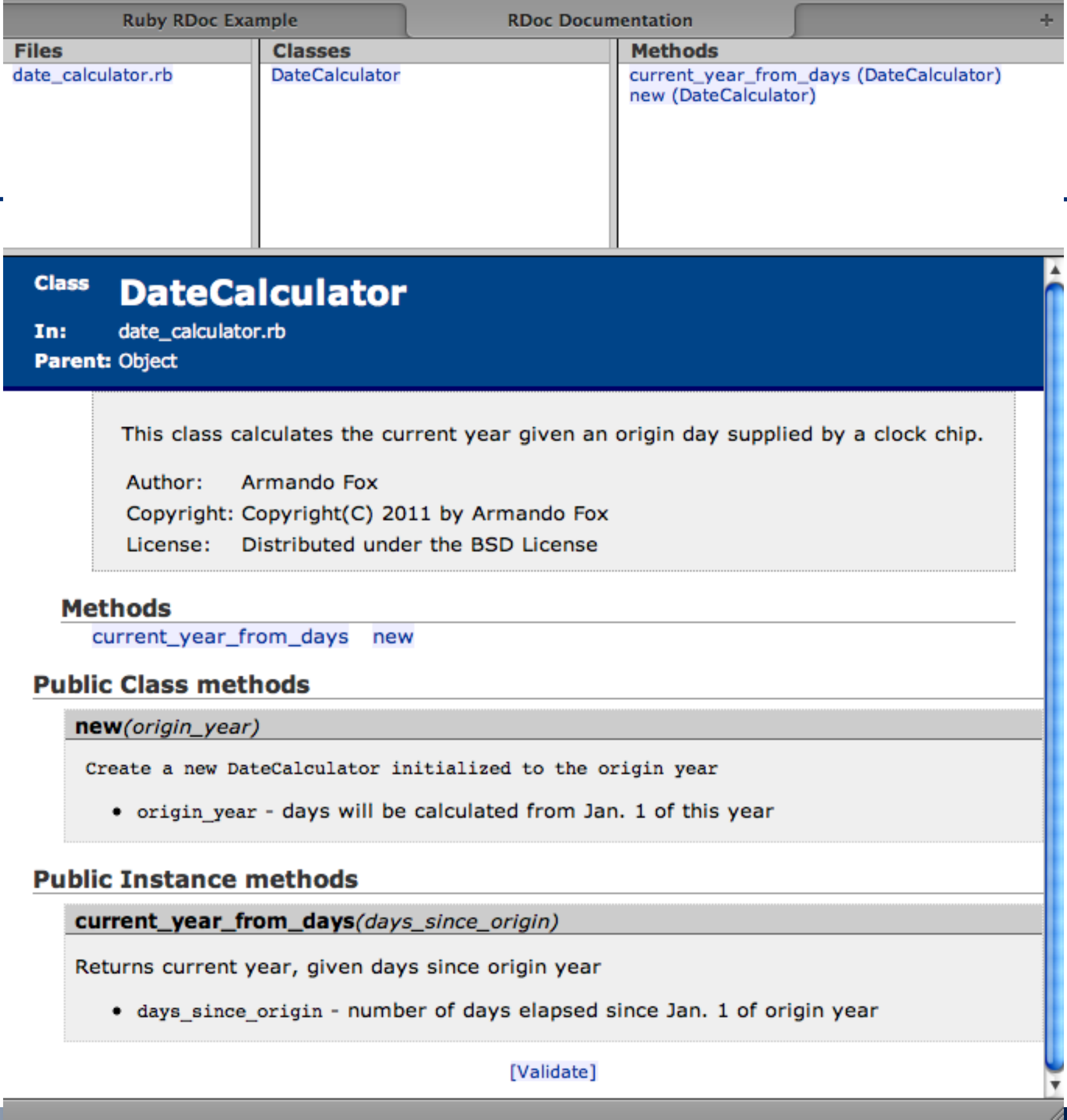
Scenario: Find specific showing
  Given a showing of "Inception" on Oct 5 at 7pm
  When I visit the "Buy Tickets" page
  Then the "Movies" menu should contain "Inception"
  And the "Showings" menu should contain "Oct 5, 7pm"

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
- 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)
- Commit logs in version control system (`git log`)

http://pastebin.com/QARUzTnh



Files
date_calculator.rb

Classes
DateCalculator

Methods
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".