WHAT MAKES CODE “LEGACY” AND HOW CAN AGILE HELP?

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”:

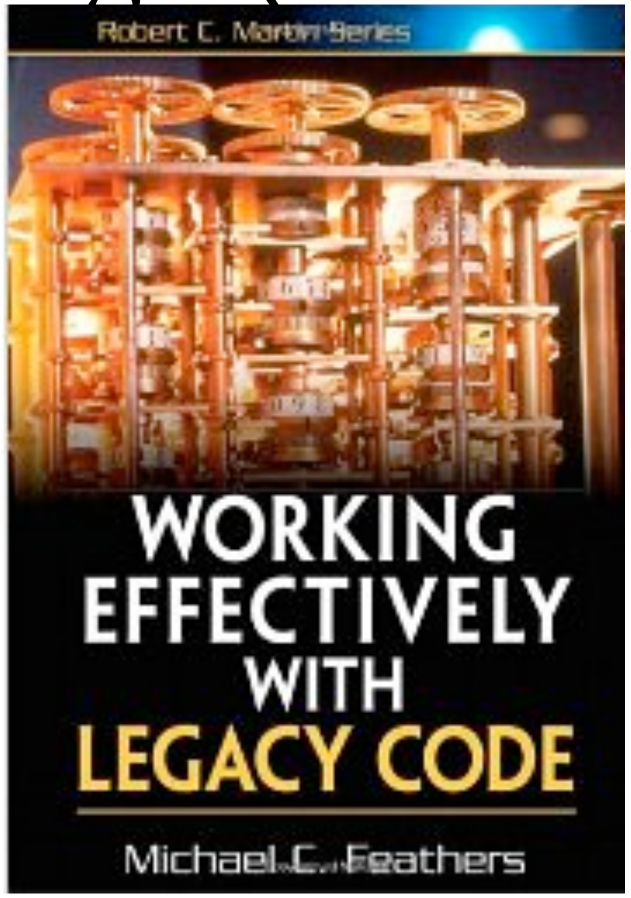
- ➤60% of software cost is maintenance
- ➤60% of maintenance cost is enhancements.

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

Approaching & Exploring Legacy Code

Interlude/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

LEARNING ABOUT LEGACY CODE AND HOW TO DEAL WITH IT

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"

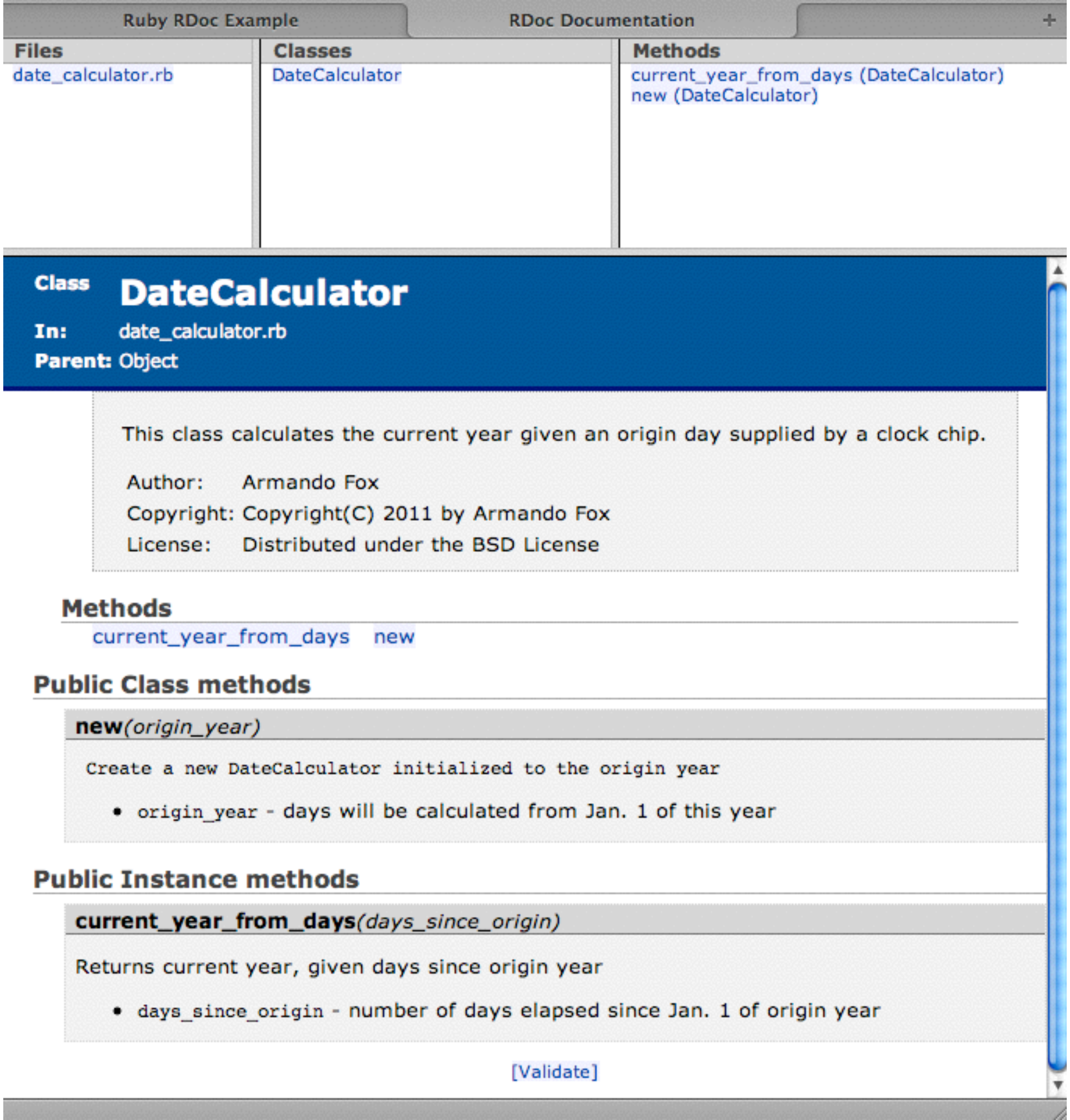
Codebase & "informal" docs

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

*



Ruby RDoc Example | RDoc Documentation

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*

Establishing Ground Truth With Characterization Tests

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 Repeatable*
  - *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 (bad pun- sorry.)
  - *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

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

*

Identifying What's Wrong: Smells, Metrics, SOFA

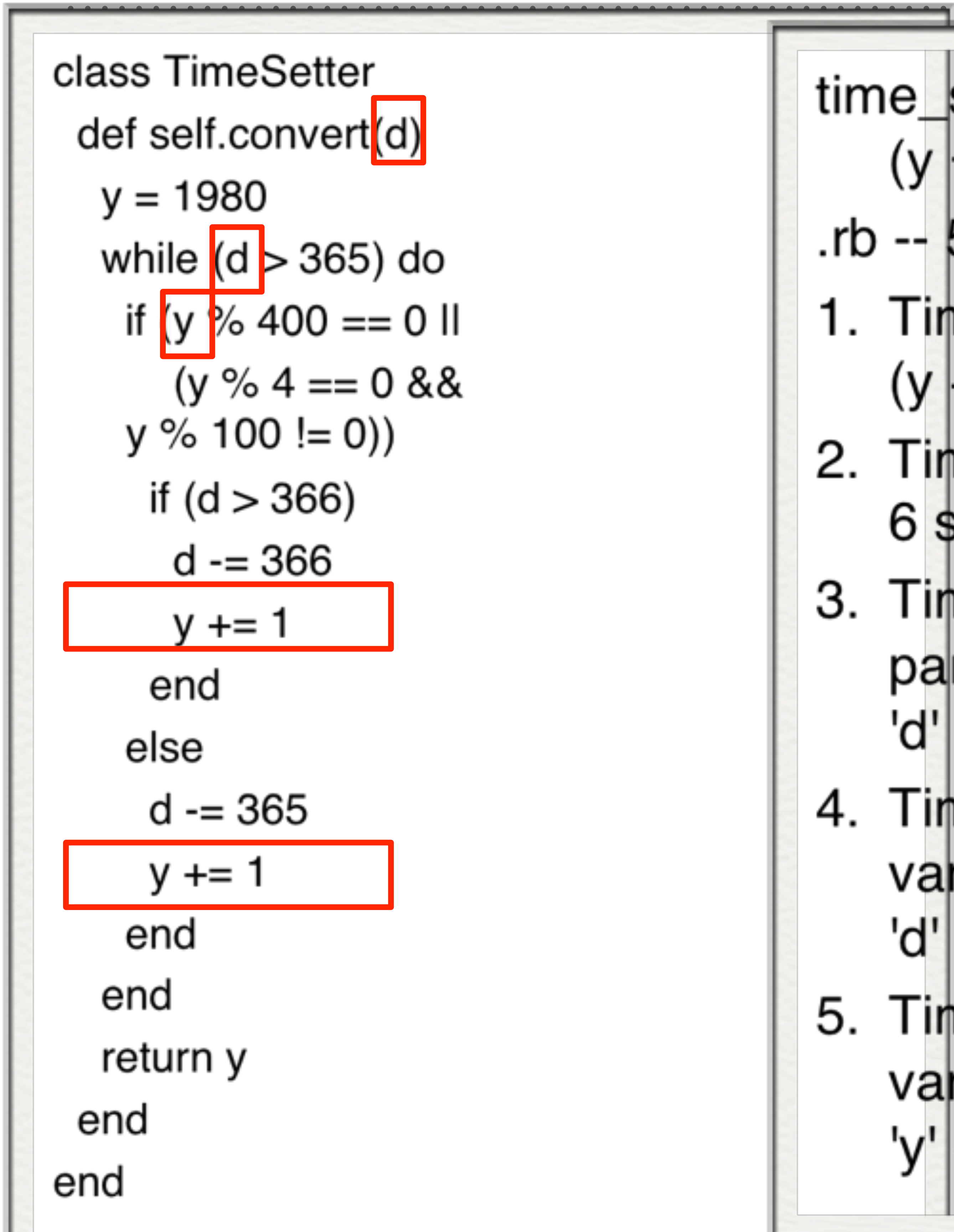
http://pastebin.com/gtQ7QcHu

BEYOND CORRECTNESS

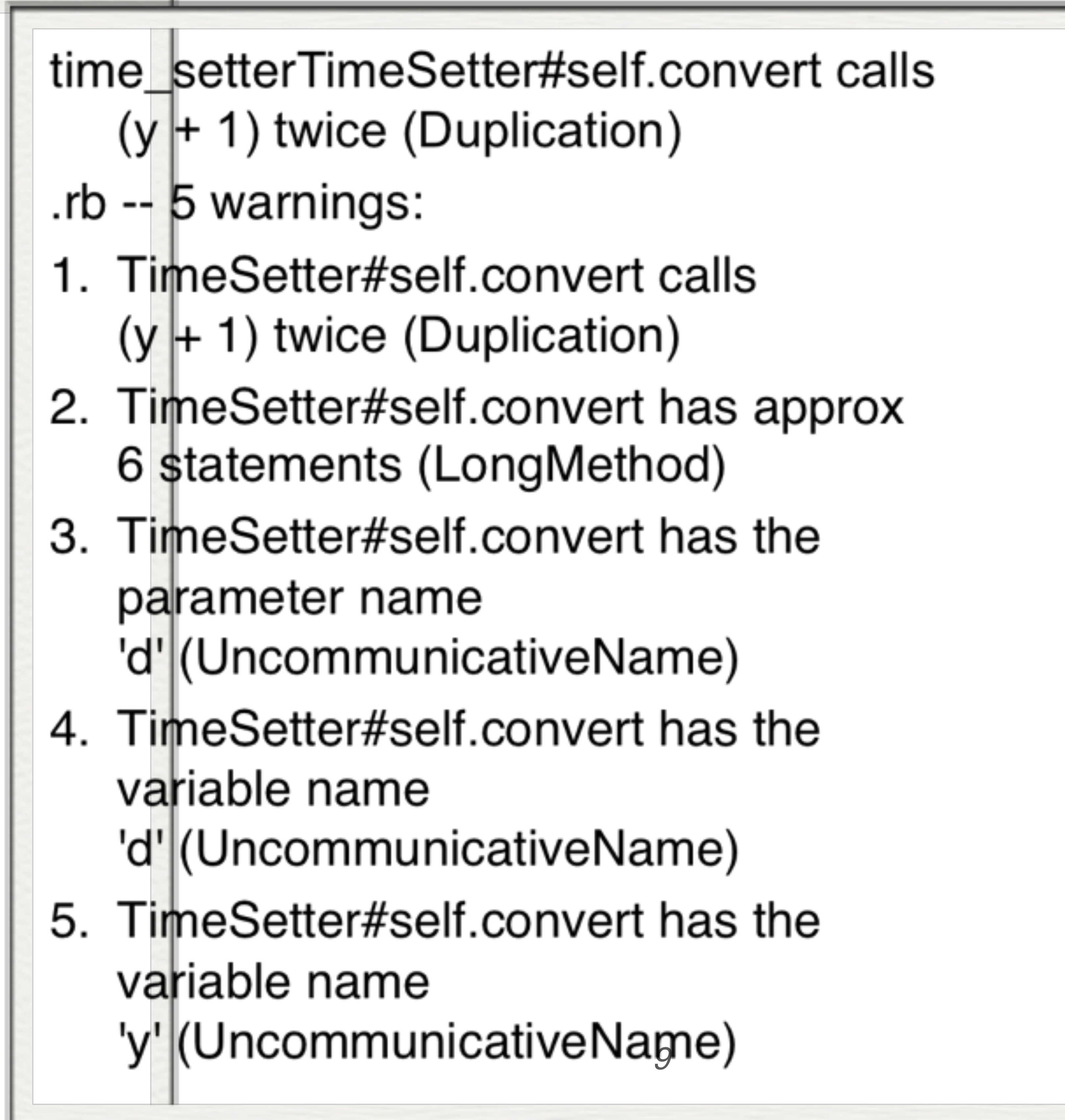
- Can we give feedback on software *beauty*?
  - Guidelines on what is beautiful?
  - Qualitative evaluations?
  - Quantitative evaluations?
  - If so, how well do they work?
  - *And does Rails have tools to support them?*

CODE SMELLS AND METRICS, REFACTORING, AND DESIGN PRINCIPLES

PROGRAM X & SMELLS



```
class TimeSetter
 def self.convert(d)
  y = 1980
  while (d > 365) do
   if (y % 400 == 0 ||
     (y % 4 == 0 &&
   y % 100 != 0))
    if (d > 366)
     d -= 366
     y += 1
    end
   else
    d -= 365
    y += 1
   end
  end
  return y
 end
end
```



```
time_setterTimeSetter#self.convert calls
    (y + 1) twice (Duplication)
.rb -- 5 warnings:
1.  TimeSetter#self.convert calls
    (y + 1) twice (Duplication)
2.  TimeSetter#self.convert has approx
    6 statements (LongMethod)
3.  TimeSetter#self.convert has the
    parameter name
    'd' (UncommunicativeName)
4.  TimeSetter#self.convert has the
    variable name
    'd' (UncommunicativeName)
5.  TimeSetter#self.convert has the
    variable name
    'y' (UncommunicativeName)
```

Quantitative: Metrics

- "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*

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)

*

https://www.ruby-toolbox.com/categories/c

Qualitative: Code Smells

SOFA captures symptoms that often indicate code smells:

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

- Ruby tool: reek

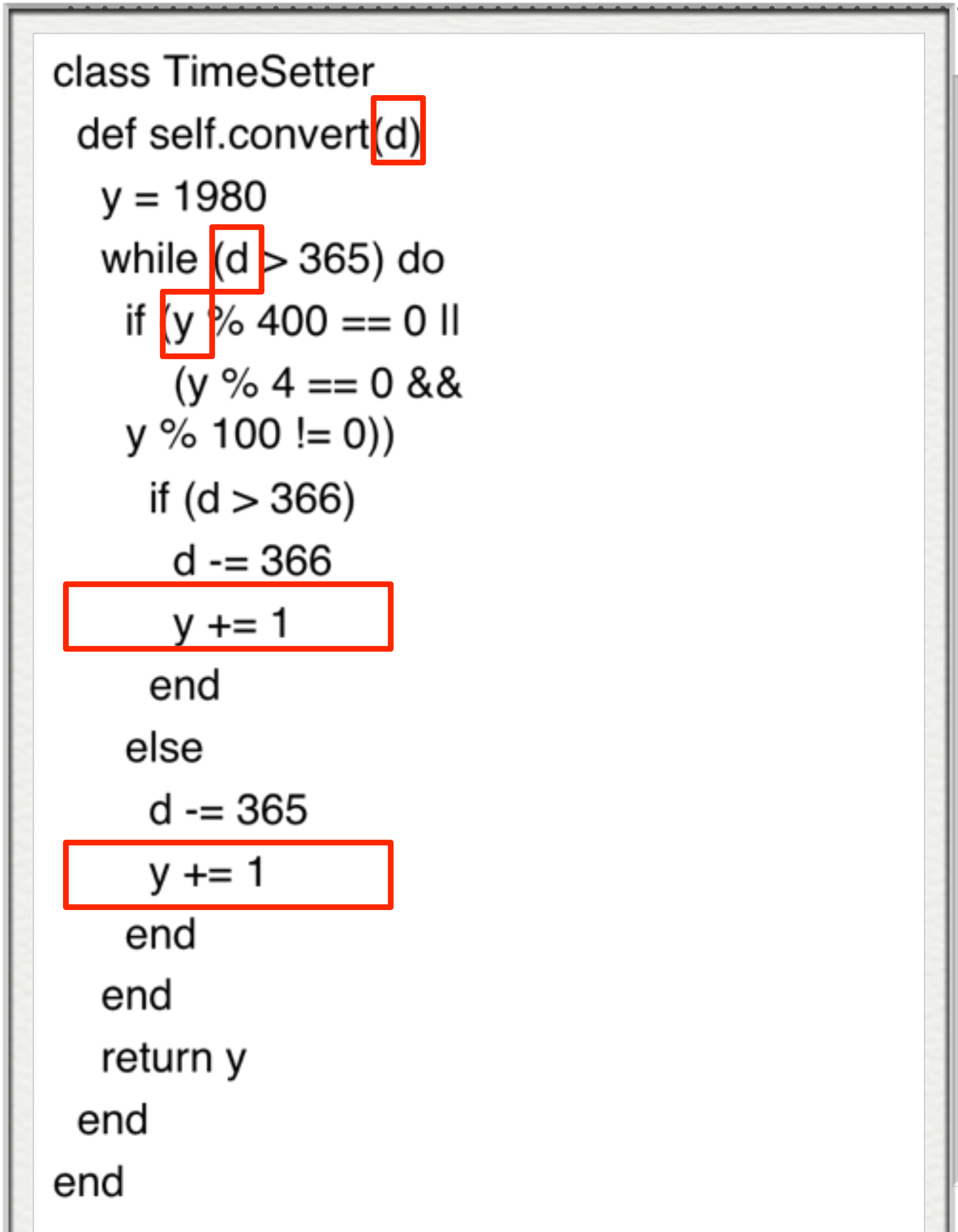
SINGLE LEVEL OF ABSTRACTION

- Complex tasks need divide & conquer
- Yellow flag for “encapsulate this task in a method”
- Like a good news story, classes & methods should read “top down”!
  - 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

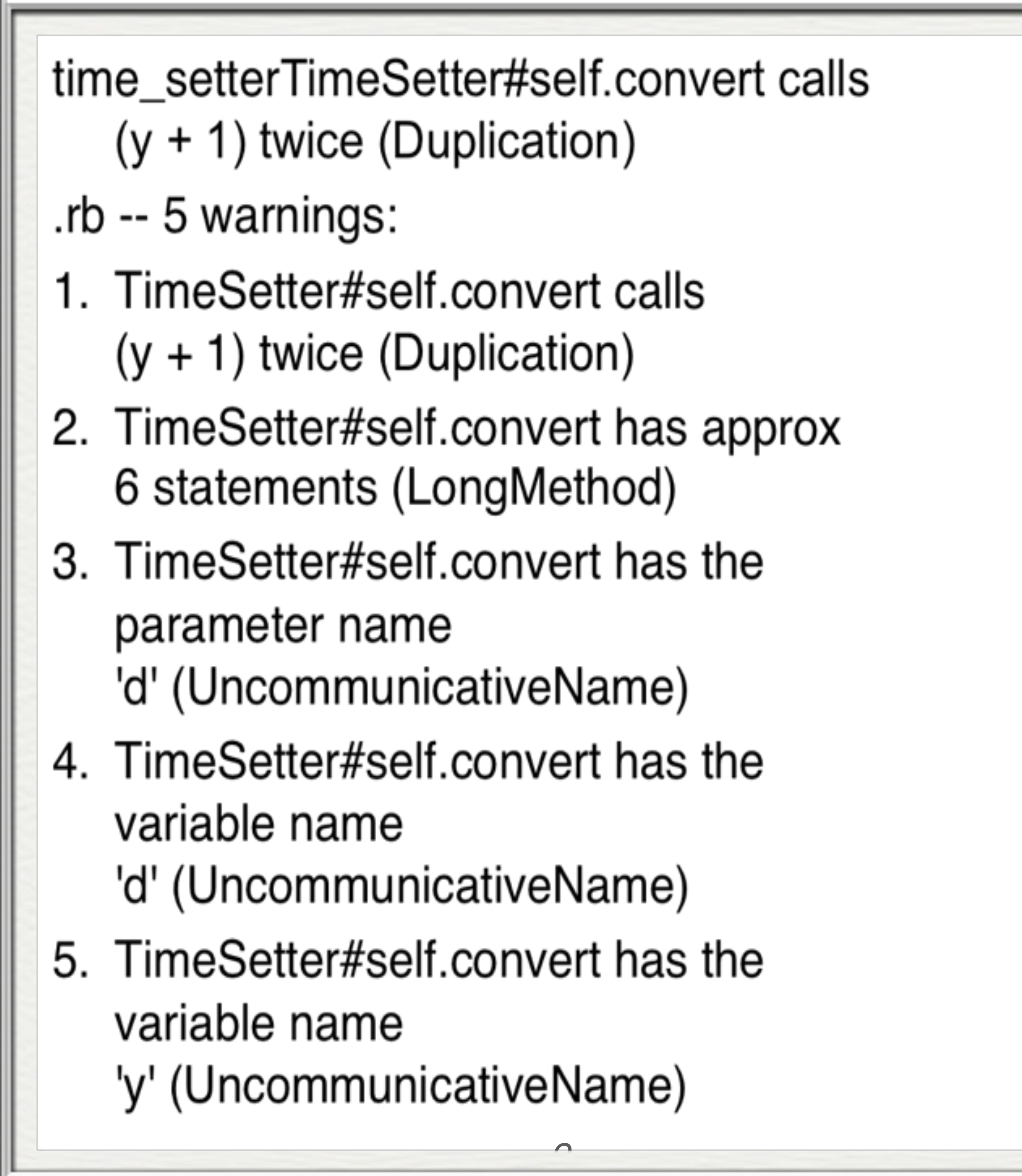
WHY LOTS OF ARGUMENTS IS BAD

- 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 pack", maybe you need to *extract a new class*
  - Same set of arguments for a lot of methods

PROGRAM X & SMELLS



```
class TimeSetter
 def self.convert(d)
  y = 1980
  while (d > 365) do
   if (y % 400 == 0 ||
      (y % 4 == 0 &&
   y % 100 != 0))
    if (d > 366)
     d -= 366
     y += 1
    end
   else
    d -= 365
    y += 1
   end
  end
  return y
 end
end
```



```
time_setterTimeSetter#self.convert calls
    (y + 1) twice (Duplication)
.rb -- 5 warnings:
1.  TimeSetter#self.convert calls
    (y + 1) twice (Duplication)
2.  TimeSetter#self.convert has approx
    6 statements (LongMethod)
3.  TimeSetter#self.convert has the
    parameter name
    'd' (UncommunicativeName)
4.  TimeSetter#self.convert has the
    variable name
    'd' (UncommunicativeName)
5.  TimeSetter#self.convert has the
    variable name
    'y' (UncommunicativeName)
```

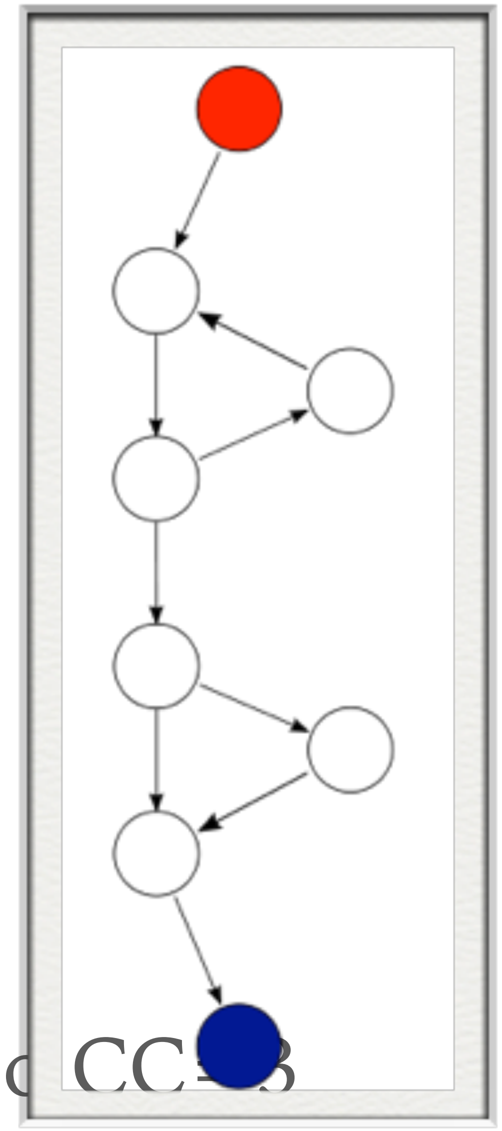
QUANTITATIVE: ABC COMPLEXITY

- Counts Assignments, Branches, Conditions
- Score = Square Root($A^2 + B^2 + C^2$)
- NIST (Natl. Inst. Stds. & Tech.): $\leq$20 /method
- Rails tool `flog` checks ABC complexity

QUANTITATIVE: CYCLOMATIC COMPLEXITY

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

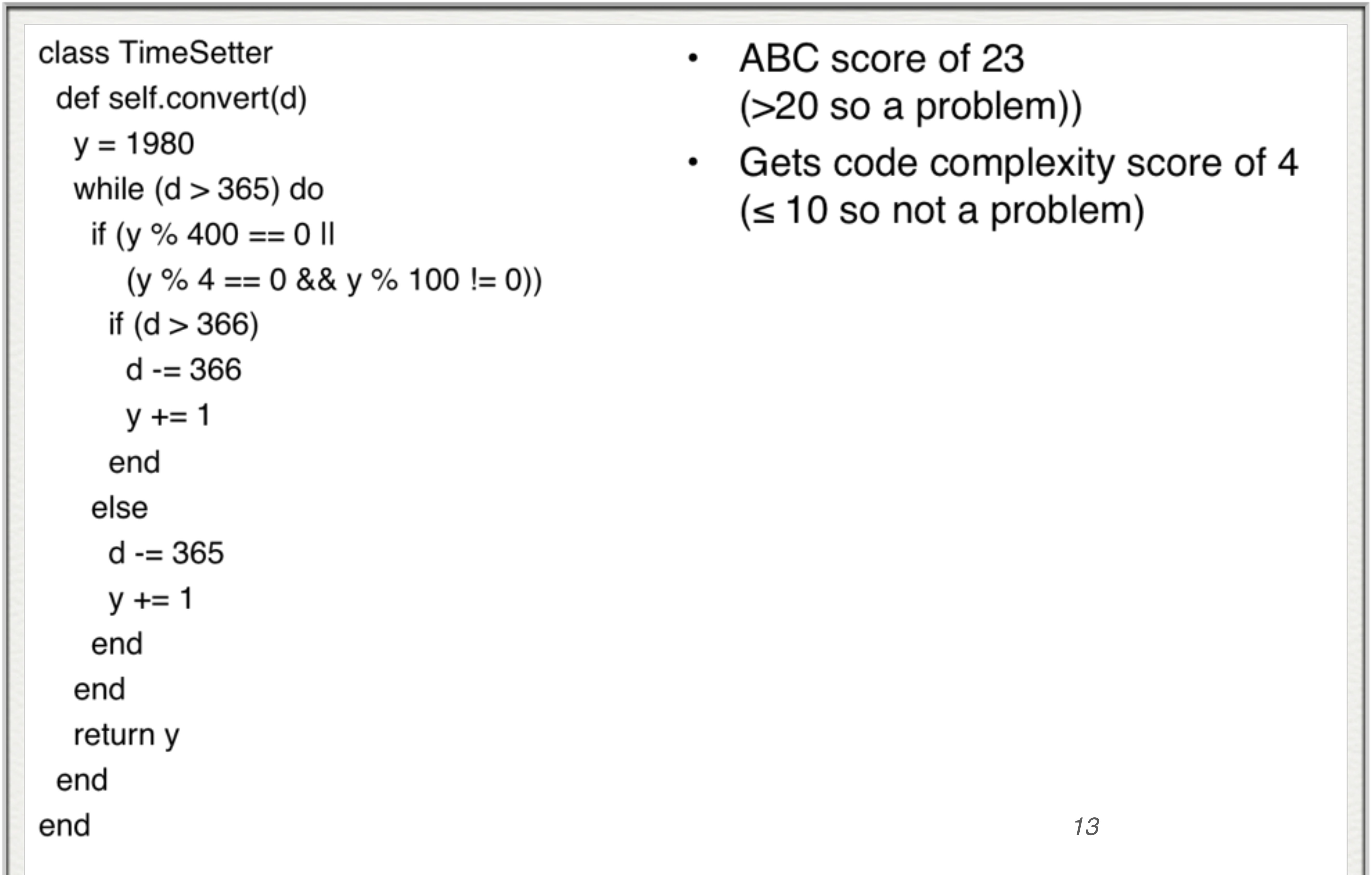
```
def mymeth
  while(...)
    ....
  end
  if (...)
    do_something
  end
end
```



Rails tool `saikuro` *calculates cyclomatic complexity*

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

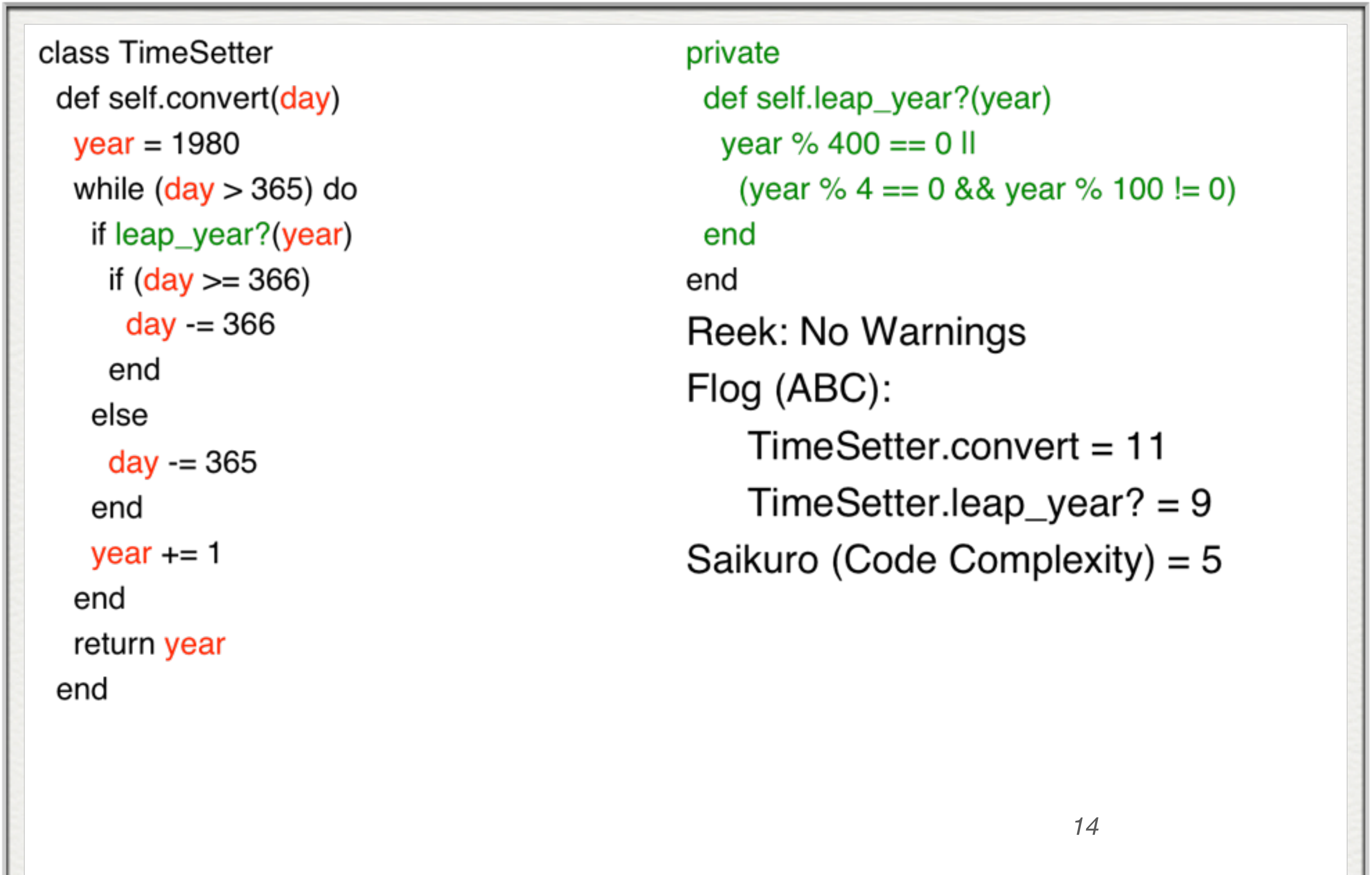
LEAP YEAR & QUANTITATIVE



```
class TimeSetter
  def self.convert(d)
    y = 1980
    while (d > 365) do
      if (y % 400 == 0 ||
          (y % 4 == 0 && y % 100 != 0))
        if (d > 366)
          d -= 366
          y += 1
        end
      else
        d -= 365
        y += 1
      end
    end
    return y
  end
end
```

- ABC score of 23 (>20 so a problem))
- Gets code complexity score of 4 (≤ 10 so not a problem)

REVISED LEAP YEAR & METRICS



```
class TimeSetter
  def self.convert(day)
    year = 1980
    while (day > 365) do
      if leap_year?(year)
        if (day >= 366)
          day -= 366
        end
      else
        day -= 365
      end
      year += 1
    end
    return year
  end
```

```
private
  def self.leap_year?(year)
    year % 400 == 0 ||
      (year % 4 == 0 && year % 100 != 0)
  end
end
```

Reek: No Warnings

Flog (ABC):

TimeSetter.convert = 11

TimeSetter.leap_year? = 9

Saikuro (Code Complexity) = 5

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

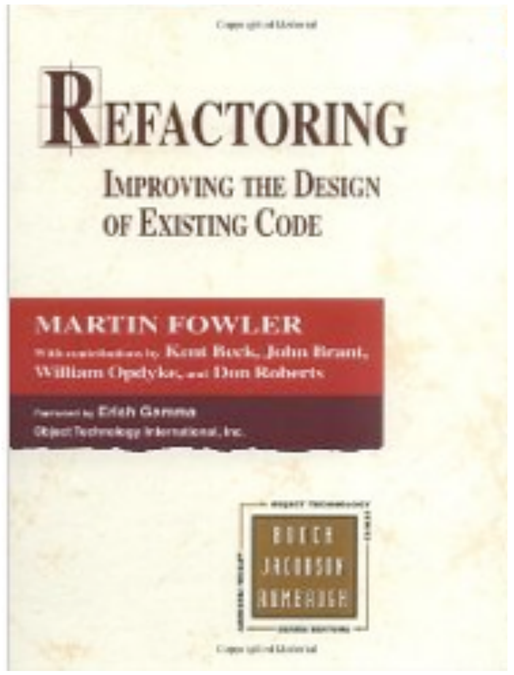
Intro to Method-Level Refactoring

Refactoring: Idea

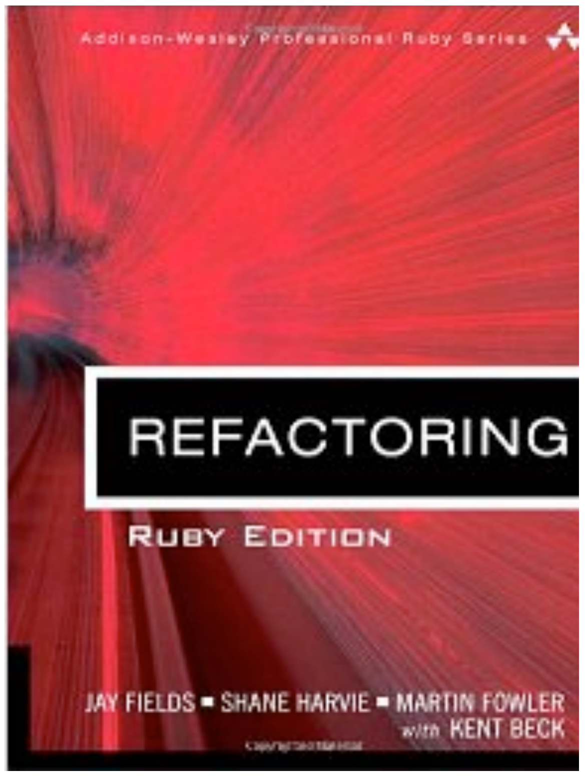
- Start with code that has 1 or more problems/ smells
- Through a series of *small steps*, transform to code from which those smells are absent
- Protect each step with tests
- *Minimize time during which tests are red*

History & Context

- ➤Fowler et al. developed mostly definitive catalog of refactorings
  - ➤*Adapted to various languages*
  - ➤*Method- and class-level refactorings*
  - ➤*Sidenote: Martin Fowler is awesome!!*
- ➤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
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER
With contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International, Inc.



Addison-Wesley Professional Ruby Series
REFACTORING
RUBY EDITION
JAY FIELDS ▪ SHANE HARVIE ▪ MARTIN FOWLER with KENT BECK

Refactoring TimeSetter

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

- Creating characterization tests and doing exploratory testing
  - *http://vimeo.com/47043669*

*

zune

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

Refactored TimeSetter -> Date Calculator:

http://pastebin.com/0Bu6sMYi

*

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()*
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*

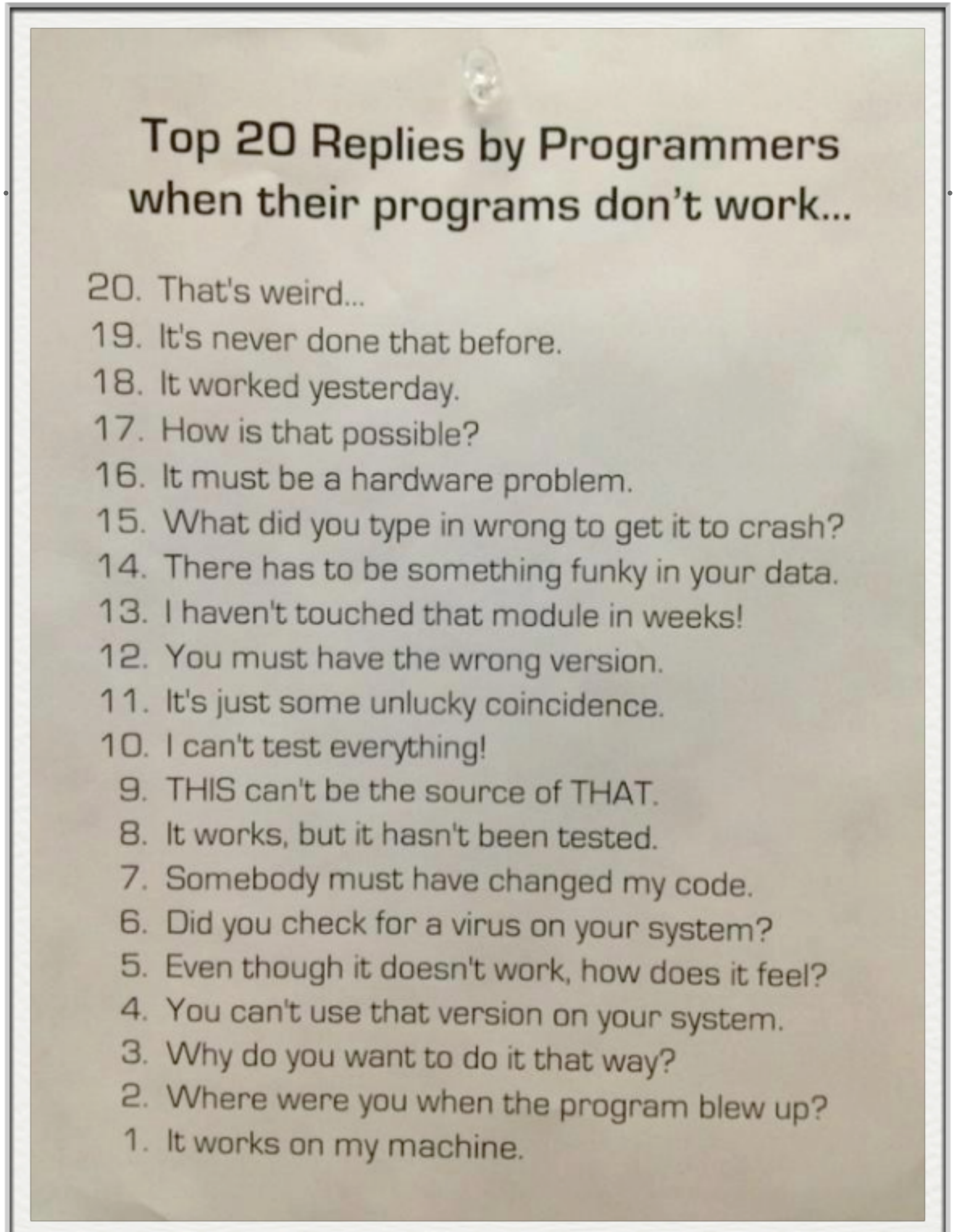
*

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)

*



Top 20 Replies by Programmers when their programs don't work...

20. That's weird...
19. It's never done that before.
18. It worked yesterday.
17. How is that possible?
16. It must be a hardware problem.
15. What did you type in wrong to get it to crash?
14. There has to be something funky in your data.
13. I haven't touched that module in weeks!
12. You must have the wrong version.
11. It's just some unlucky coincidence.
10. I can't test everything!
9. THIS can't be the source of THAT.
8. It works, but it hasn't been tested.
7. Somebody must have changed my code.
6. Did you check for a virus on your system?
5. Even though it doesn't work, how does it feel?
4. You can't use that version on your system.
3. Why do you want to do it that way?
2. Where were you when the program blew up?
1. It works on my machine.