# CODE SMELLS AND METRICS, REFACTORING, AND DESIGN PRINCIPLES

# Identifying What's Wrong: Smells, Metrics, SOFA

*http://pastebin.com/gtQ7QcHu*

# 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/*

# 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)
```

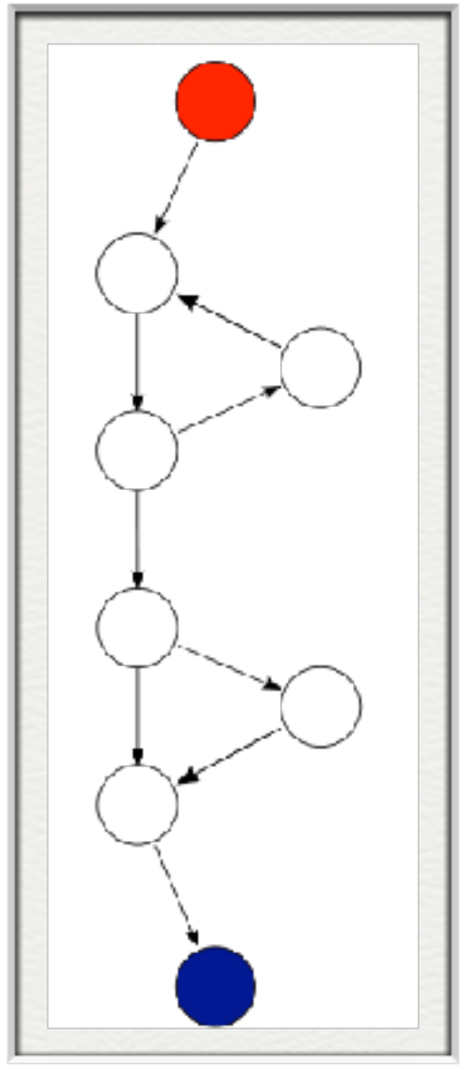
# QUANTITATIVE: ABC COMPLEXITY

- Counts Assignments, Branches, Conditions
- Score = Square Root($A^2 + B^2 + C^2$)
- NIST (Natl. Inst. Stds. & Tech.): ≤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

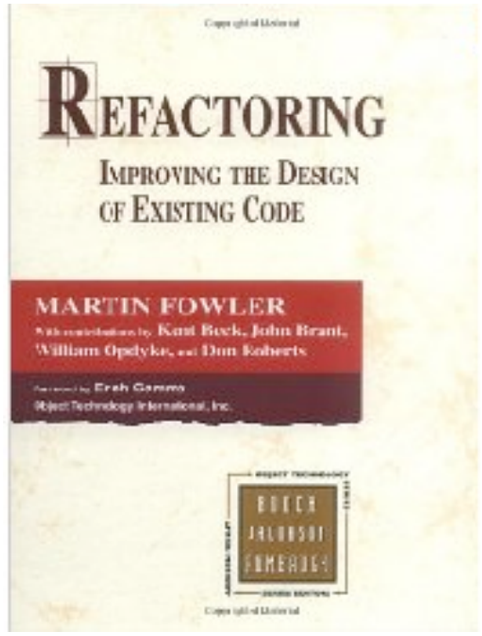
# Intro to Method-Level Refactoring

# Refactoring: Idea

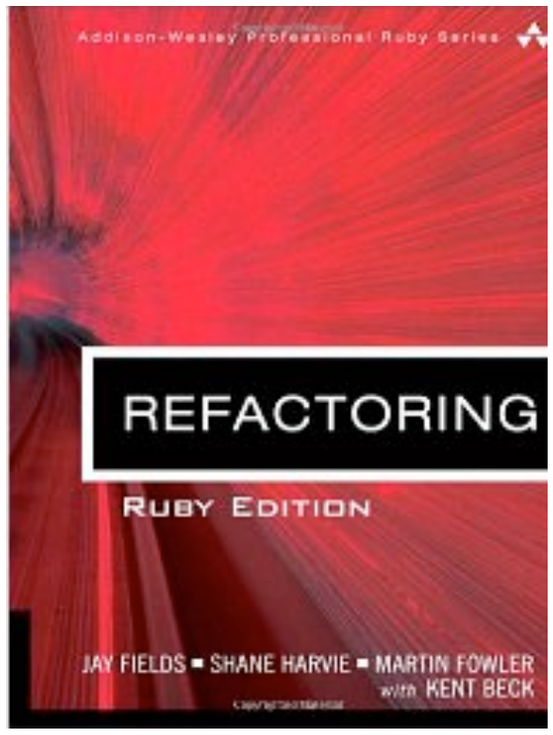
- Start with code that has 1 or more problems/ smells
- Through a series of *small steps*, transform to code from which those smells are absent
- Protect each step with tests
- *Minimize time during which tests are red*

# 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 TimeSetter

- Fix stupid names

http://pastebin.com/pYCfMQJp

- Extract method

http://pastebin.com/sXVDW9C6

- Extract method, encapsulate class

http://pastebin.com/zWM2ZqaW

- Test extracted methods
- 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*

*

# 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*<br>*Replace loop with collection method*<br>*Extract method*<br>*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<br>introduce assertion<br>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.

# 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”:

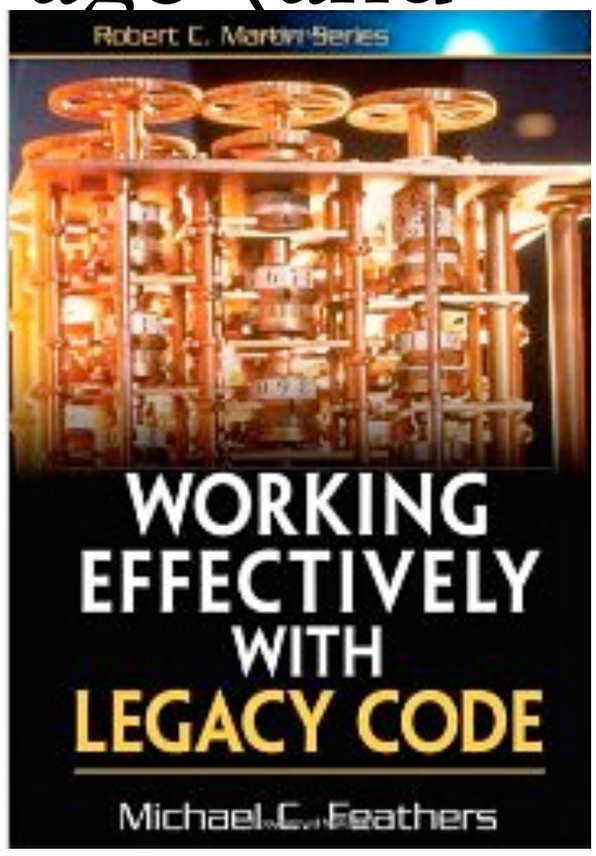
- 60% of software cost is maintenance
- 60% of maintenance cost is enhancements.

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."

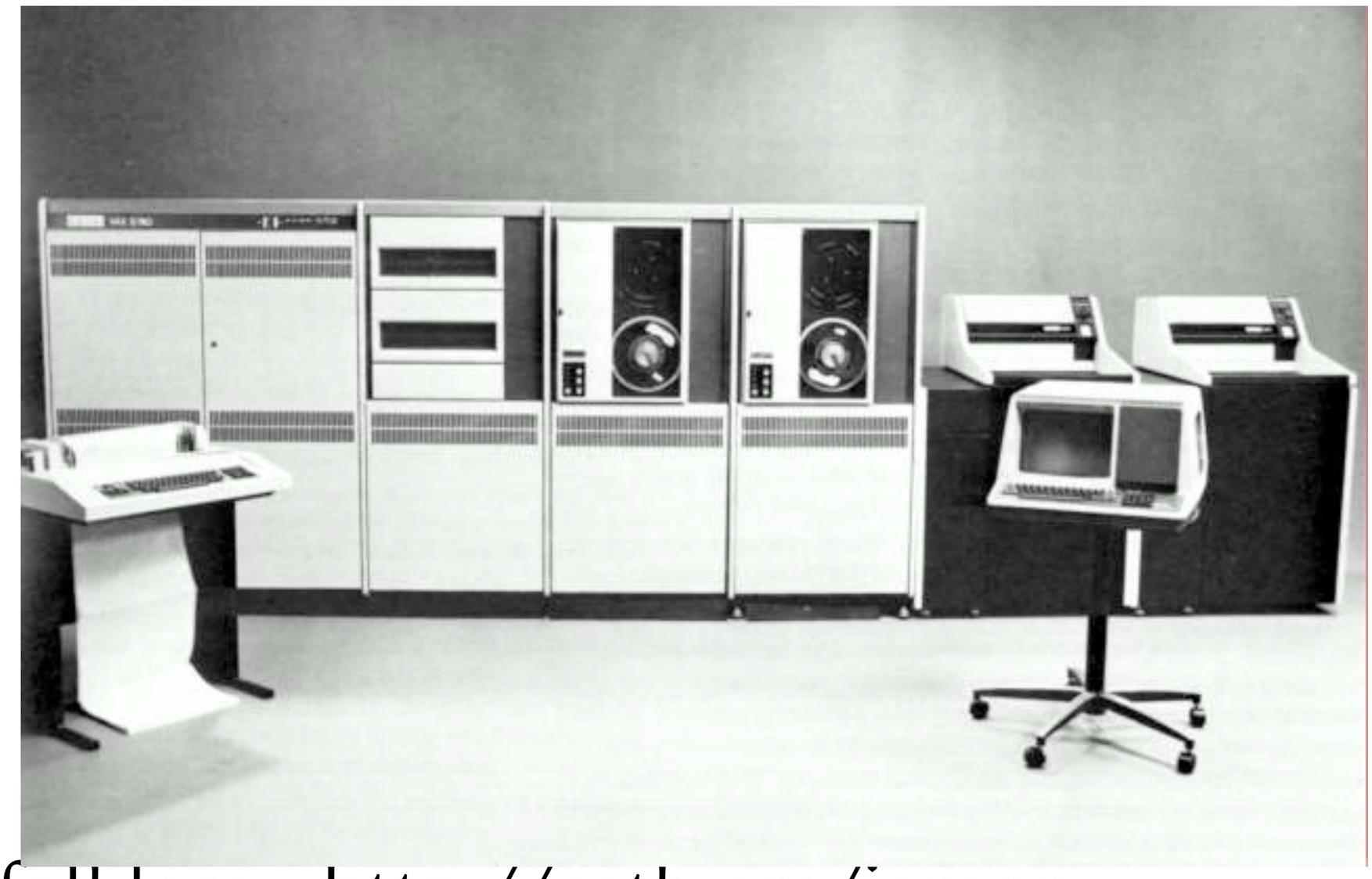
*Lord Robert Baden-Powell, founder of the Boy Scouts*

*

# 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

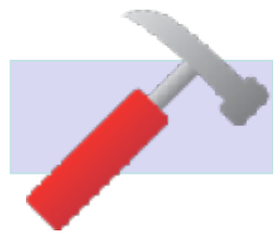
- Check out a *scratch branch* that won't be checked back in, and get it to run
  - *In a production-like setting or development-like setting*
  - *Ideally with something resembling a **copy** of production database*
  - *Some systems may be too large to clone*
- Learn the user stories: Get customer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

*http://pastebin.com/QARUzTnh*

*

Ruby RDoc Example | RDoc Documentation | +

| Files | Classes | Methods |
| --- | --- | --- |
| date_calculator.rb | DateCalculator | current_year_from_days (DateCalculator)<br>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 **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 (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*

*

```
it "should calculate sales tax" do
  order = Factory('order')
  order.compute_tax.should == -99.99
end
```

# Unit- and Functional-Level Characterization Tests

Cheat: write tests to learn as you go

```
# 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
it "should calculate sales tax" do
  order = Factory('order', :get_total => 100.00)
  order.compute_tax.should == 8.45
end
```

# 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?*

# Patterns, Antipatterns, and SOLID

# Design Patterns Promote Reuse

*"A pattern describes a problem that occurs often, along with a tried solution to the problem" - Christopher Alexander, 1977*

➤Christopher Alexander's 253 (civil) architectural patterns range from the creation of cities (2. distribution of towns) to particular building problems (232. roof cap)

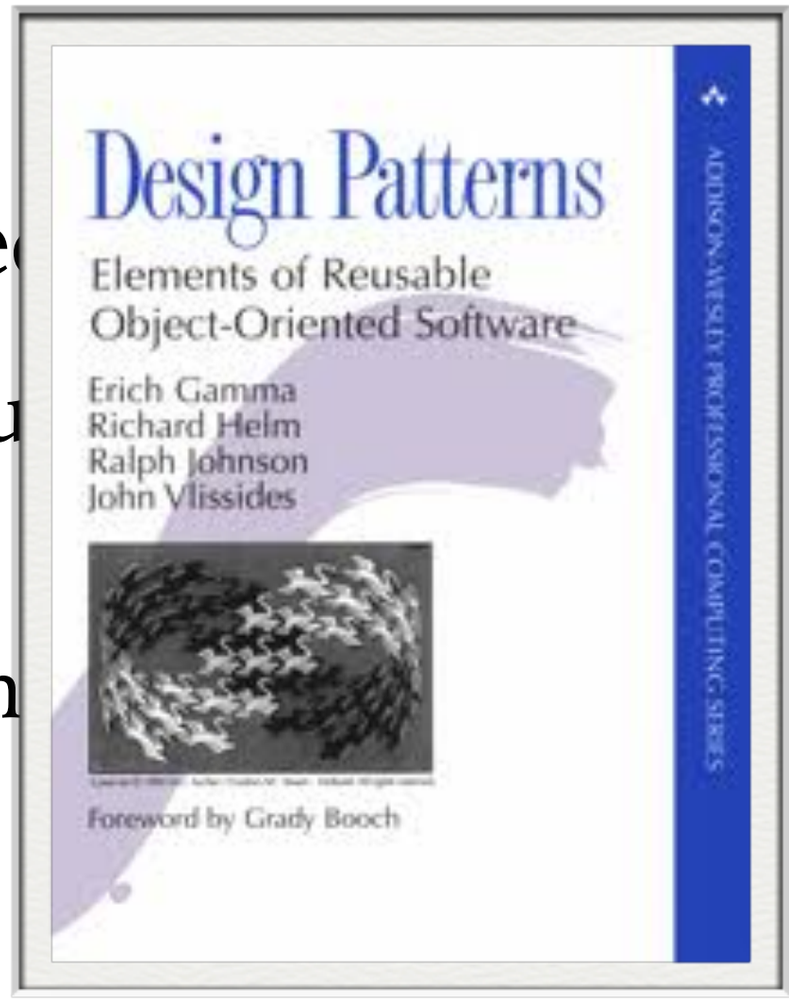
➤A pattern language is an organized  way of tackling an architectural problem using patterns

# Kinds of Patterns in Software

- Architectural ("macroscale") patterns
  - *Model-view-controller*
  - *Pipe & Filter (e.g. compiler, Unix pipeline)*
  - *Event-based (e.g. interactive game)*
  - *Layering (e.g. SaaS technology stack)*
- Computation patterns
  - *Fast Fourier transform*
  - *Structured & unstructured grids*
  - *Dense linear algebra*
  - *Sparse linear algebra*
- *GoF (Gang of Four) Patterns: structural, creational, behavior*

# The Gang of Four (GoF)

- 23 *structural* design patterns
  - description of communicating obje
  - captures common (and successful) solu of related problem instances
  - can be customized to solve a specific (n that category




- *Pattern ≠*
- *individual classes or libraries (list, hash, ...)*
- *full design—more like a blueprint for a design*

# The GoF Pattern Zoo

1. Factory
2. ***Abstract factory***
3. Builder
4. Prototype
5. ***Singleton/Null obj***
6. ***Adapter***
7. ***Composite***
8. ***Proxy***
9. Bridge
10. Flyweight
11. ***Façade***
12. ***Decorator***

*Creation*

*Behavioral*

*Structural*

13. ***Observer***
14. *Mediator*
15. *Chain of responsibility*
16. *Command*
17. *Interpreter*
18. ***Iterator***
19. *Memento (memoization)*
20. *State*
21. ***Strategy***
22. ***Template***
23. *Visitor*

# Meta-Patterns

Separate out the things that change from those that stay the same

1. Program to an Interface, not Implementation
2. Prefer composition & delegation over Inheritance
   - ➤ delegation is about interface sharing, inheritance is about implementation sharing

# Antipattern

- Code that looks like it should probably follow some design pattern, but it doesn't
- Symptoms:
  - *Viscosity (easier to do hack than Right Thing)*
  - *Immobility (can't DRY out functionality)*
  - *Needless repetition (comes from immobility)*
  - *Needless complexity from generality*

*Motivation: minimize cost of change*

- Single Responsibility principle
- Open/Closed principle
- Liskov substitution principle
- Injection of dependencies
  - *traditionally, Interface Segregation principle*
- Demeter principle

# Refactoring & Design Patterns

| Methods within a class | Relationships among classes |
|---|---|
| Code smells | Design smells |
| Many catalogs of code smells & refactorings | Many catalogs of design smells & design patterns |
| Some refactorings are superfluous in Ruby | Some design patterns are superfluous in Ruby |
| Metrics: ABC & Cyclomatic Complexity | Metrics: Lack of Cohesion of Methods (LCOM) |
| Refactor by extracting methods and moving around code within a class | Refactor by extracting classes and moving code between classes |
| SOFA: methods are **S**hort, do **O**ne thing, have **F**ew arguments, single level of **A**bstraction | SOLID: **S**ingle responsibility per class, **O**pen/closed principle, **L**iskov substitutability, **I**njection of dependencies, **D**emeter principle |

# Just Enough UML

*Armando Fox*

# Just Enough UML

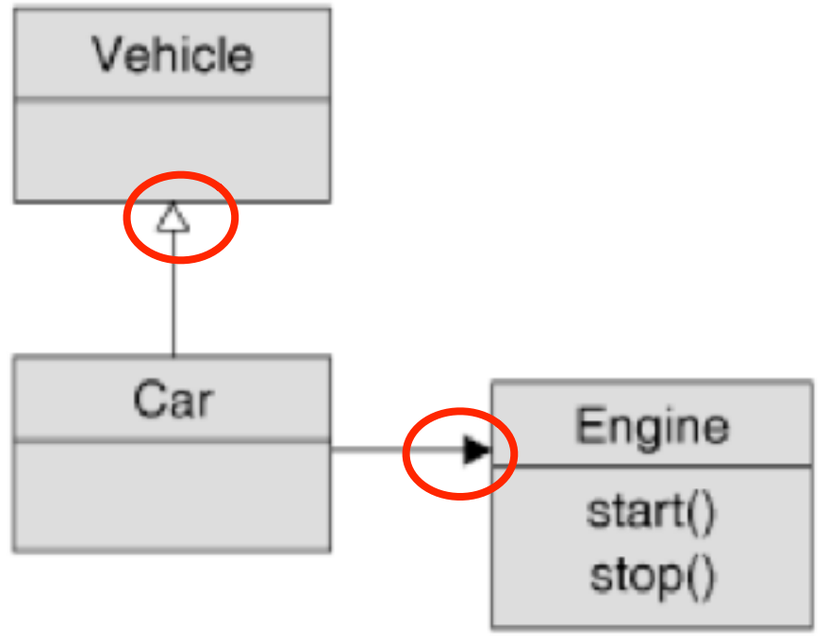
➤Unified Modeling Language: notation for describing various artifacts in OOP systems

➤One type of UML diagram is a *class diagram*, showing class relationships and principal methods:

➤Car is a subclass of Vehicle

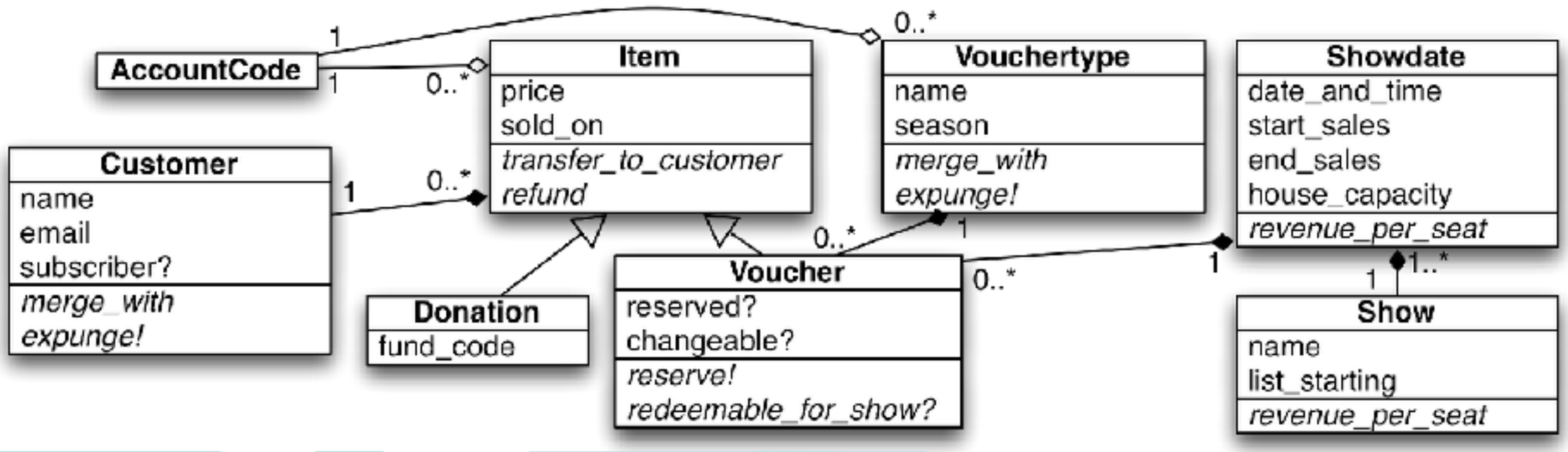
➤Engine is a *component* of Car

➤Engine class includes start(), stop() methods



# Relationships

*Aggregation*




*Composition*

*Inheritance*

# (Too Much UML)

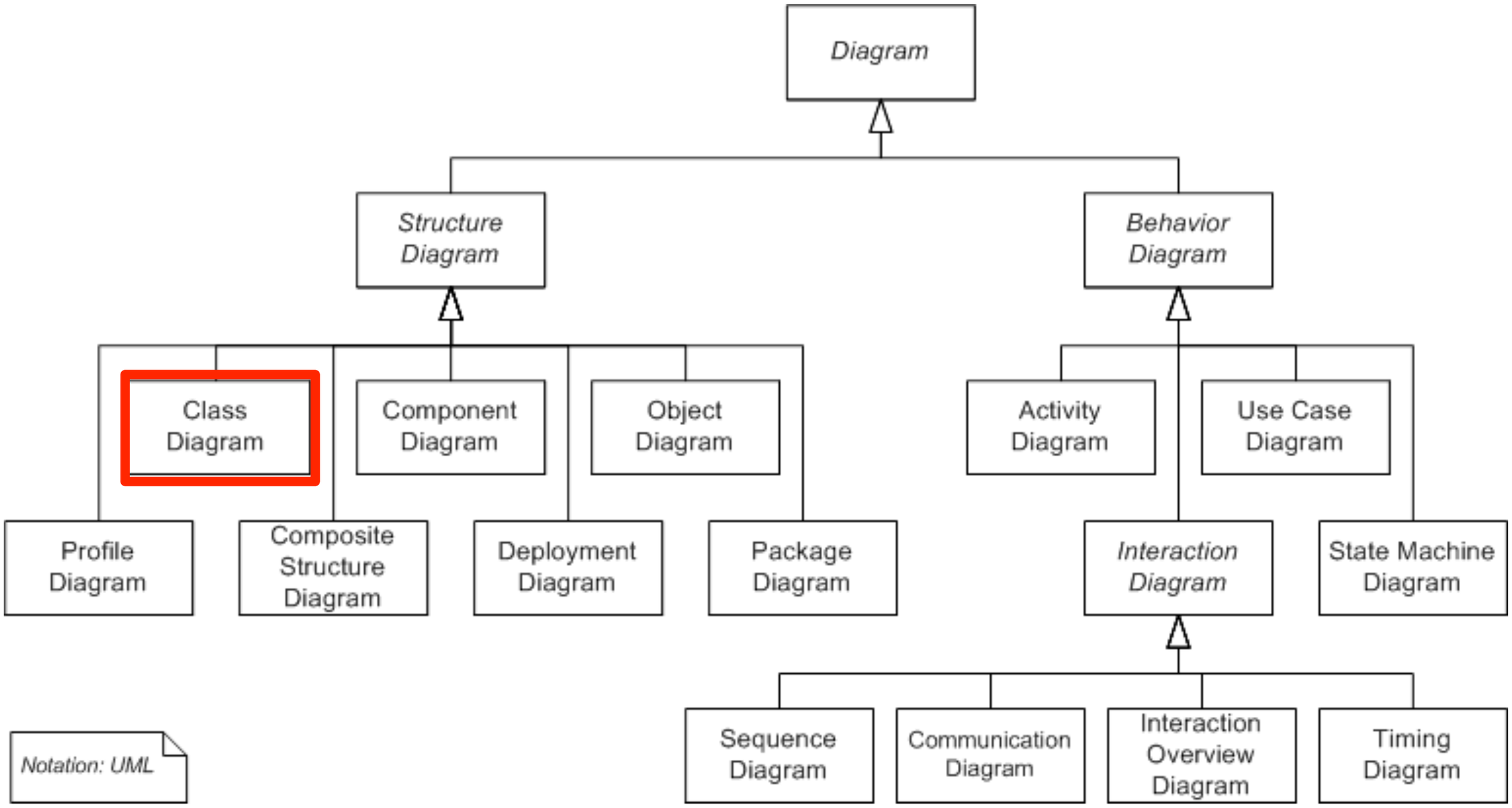


Diagram
Structure Diagram
Behavior Diagram
Class Diagram
Component Diagram
Object Diagram
Profile Diagram
Composite Structure Diagram
Deployment Diagram
Package Diagram
Activity Diagram
Use Case Diagram
Interaction Diagram
State Machine Diagram
Sequence Diagram
Communication Diagram
Interaction Overview Diagram
Timing Diagram
Notation: UML

# Single Responsibility Principle
## (ELLS §11.3)

*Armando Fox*

# Single Responsibility Principle (SRP)

- A class should have *one and only one* reason to change
  - Each *responsibility* is a possible *axis of change*
  - Changes to one axis shouldn't affect others
- What is class's responsibility, in ≤25 words?
  - Part of the craft of OO design is *defining* responsibilities and then sticking to them
- Models with many sets of behaviors
  - eg a user is a moviegoer, and an authentication principal, and a social network member, ...etc.
  - really big class files are a tipoff

# Lack of Cohesion of Methods

- Revised Henderson-Sellers

$LCOM = 1 - (sum(MV_i) / M*V)$ (between 0 and 1)

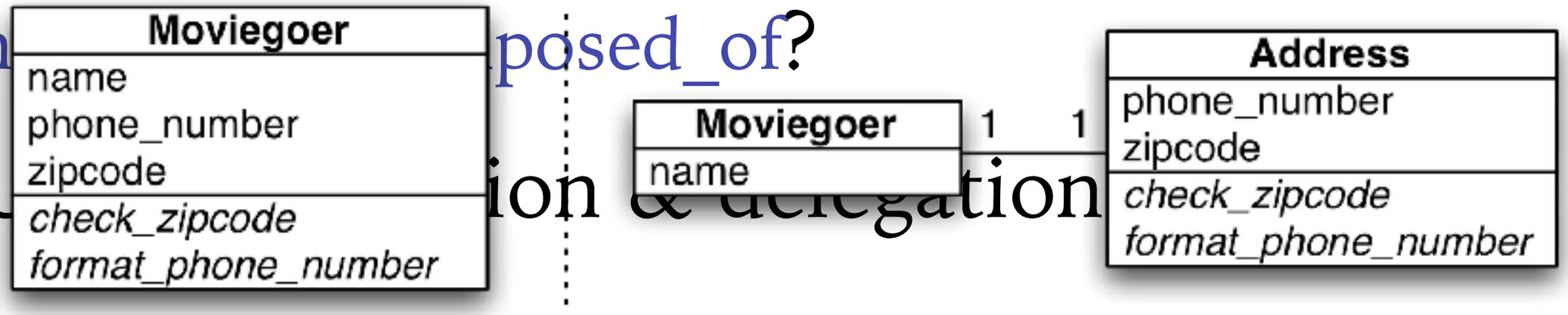
  - *M = # instance methods*
  - *V = # instance variables*
  - $MV_i$ *= # instance methods that access the i'th instance variable (excluding "trivial" getters/setters)*

- LCOM-4 counts # of connected components in graph where related methods are connected by an edge
- High LCOM suggests possible SRP violation

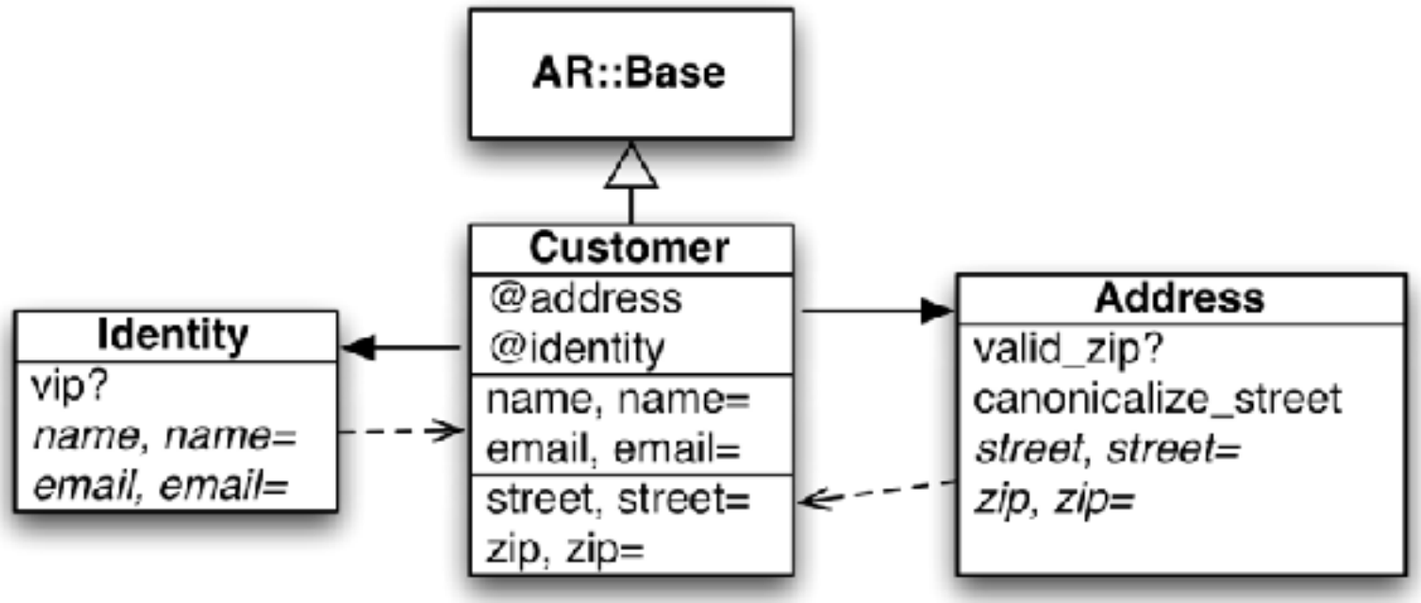
*

# Extract a module or class

- h…posed_of?
- U…ion & delegation




http://pastebin.com/bjdaTWN8




http://pastebin.com/XESSSNb6

# Open/Closed Principle
## (ELLS §11.4)

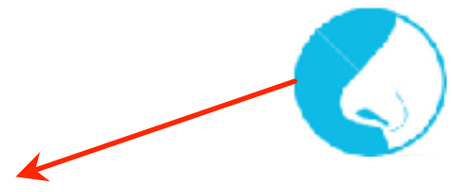
*Armando Fox*



# Open/Closed Principle

➤Classes should be *open for extension*, but *closed for* ***source modification***

```
class Report
  def output_report
    case @format
    when :html
      HtmlFormatter.new(self).output
    when :pdf
      PdfFormatter.new(self).output
```



➤Can’t extend (add new report types) without changing Report base class

➤Not as bad as in statically typed language….but still ugly

## Abstract Factory Pattern:

- How to avoid OCP violation in Report constructor, if output type isn't known until runtime?
- Statically typed language: *abstract factory* pattern
- Ruby has a particularly simple implementation of this pattern…

```
class Report
  def output
    formatter_class =
      begin
        @format.to_s.classify.constantize
      rescue NameError
        # ...handle 'invalid formatter type'
      end
    formatter = formatter_class.send(:new, self)
    # etc
  end
end
```

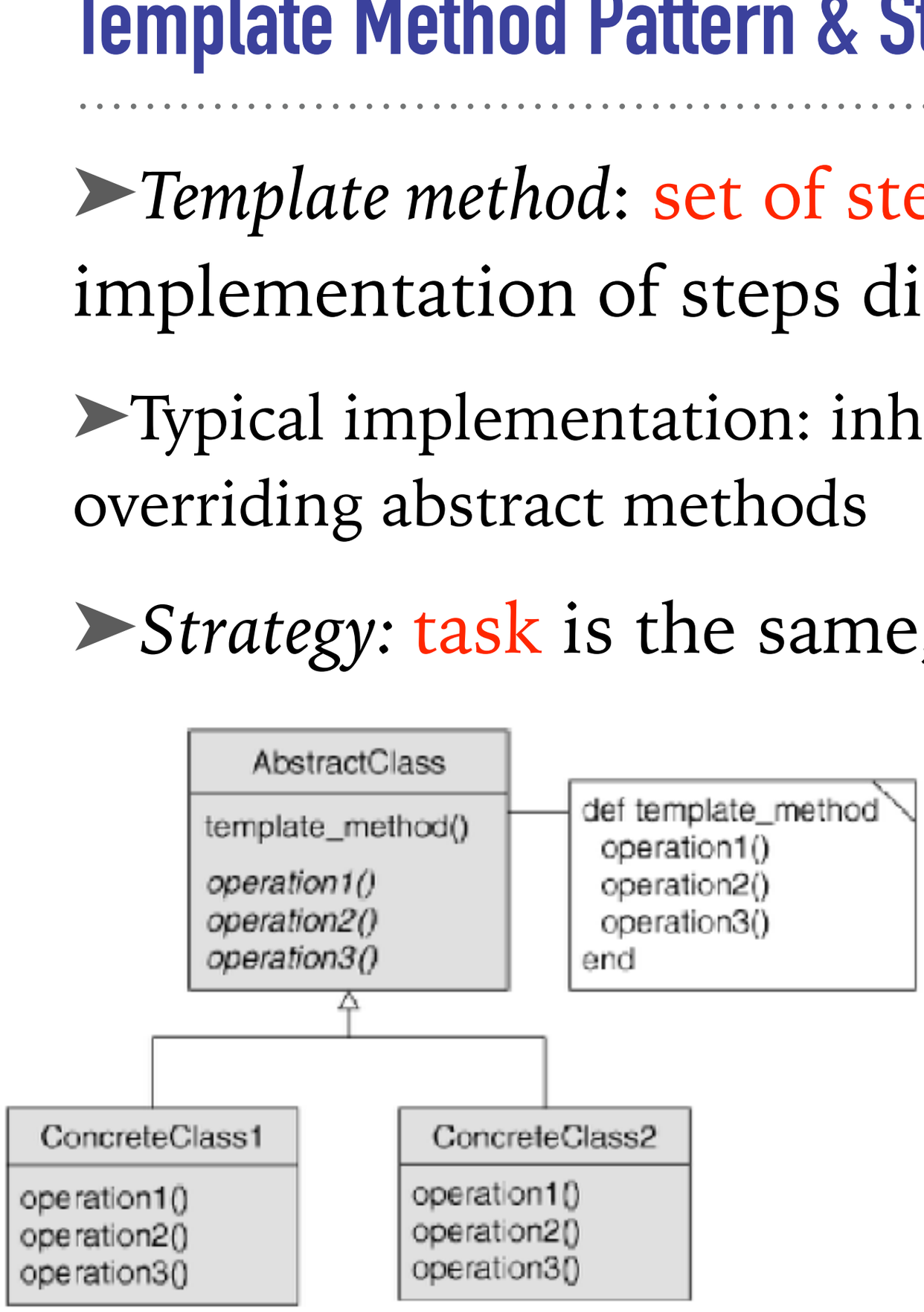
*http://pastebin.com/p3AHMqHZ*

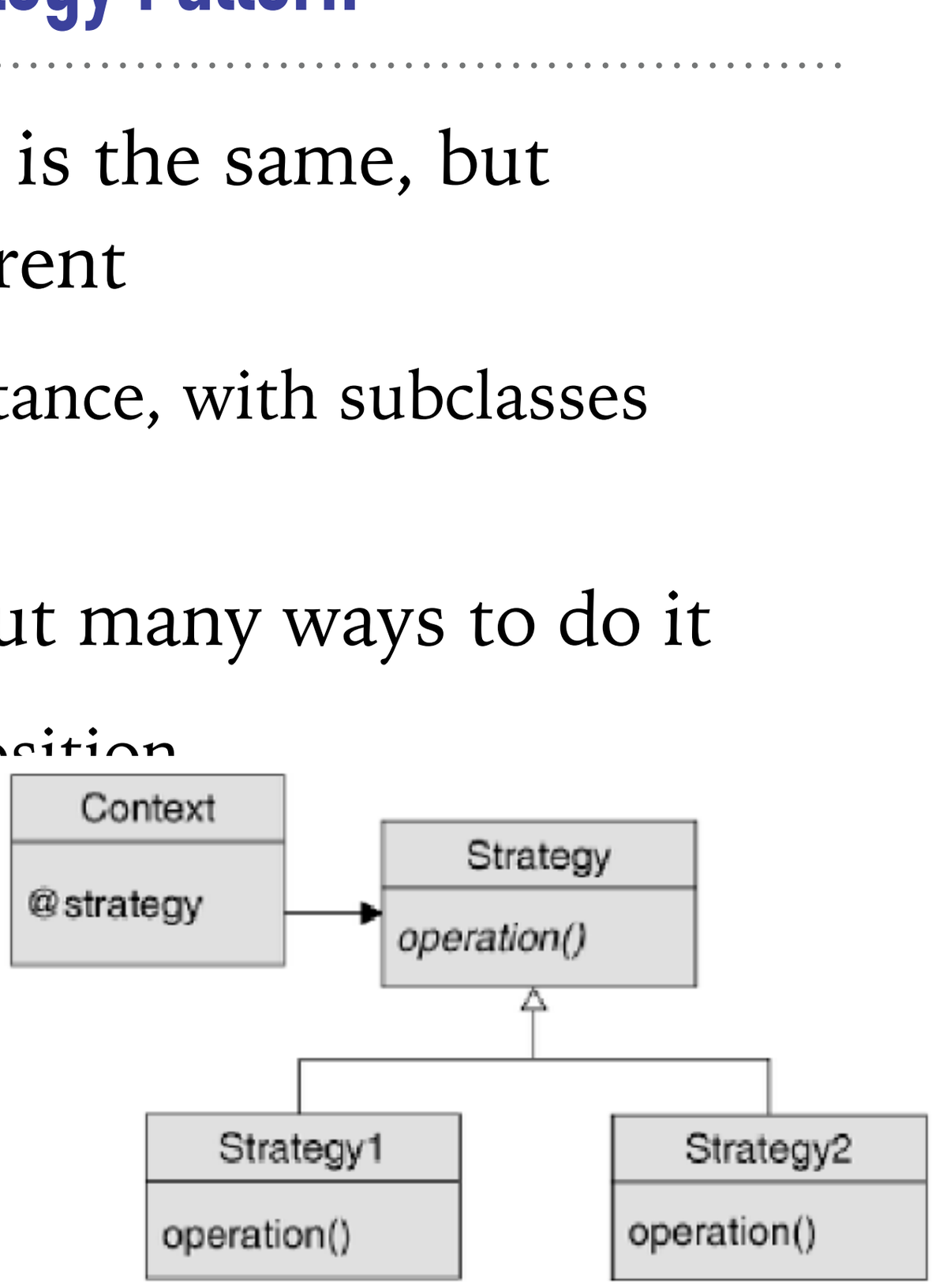
# Template Method Pattern & Strategy Pattern

➤*Template method*: set of steps is the same, but implementation of steps different

➤Typical implementation: inheritance, with subclasses overriding abstract methods

➤*Strategy*: task is the same, but many ways to do it



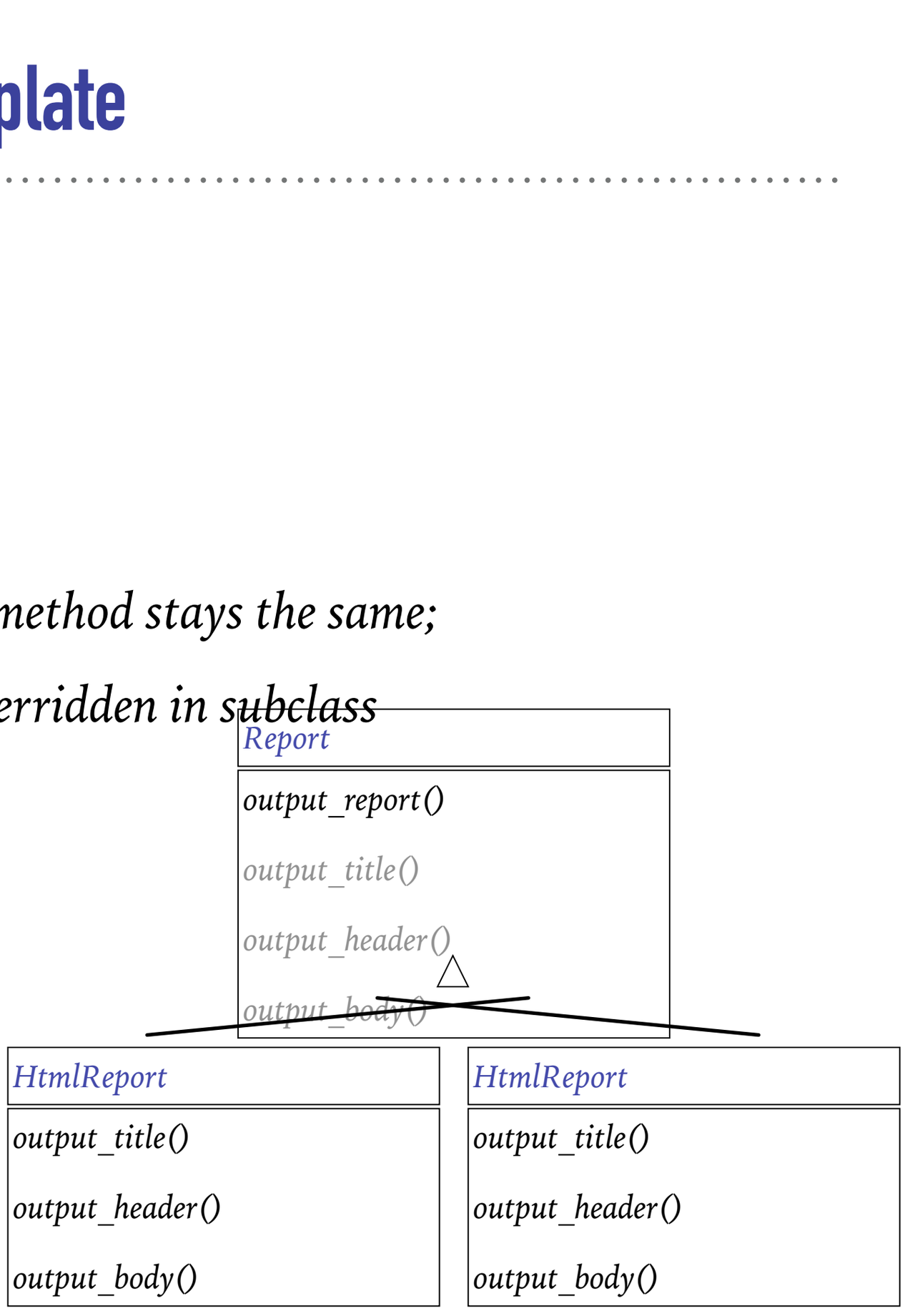
# Report Generation Using Template

```
class Report
 attr_accessor :title, :text
 def output_report
  output_title
  output_header
  output_body
 end
end
```

```
class HtmlReport < Report
 def output_title ... end
 def output_header ... end
end
class PdfReport < Report
 def output_title ... end
 def output_header ... end
end
```

*Template method stays the same;*

*helpers overridden in subclass*

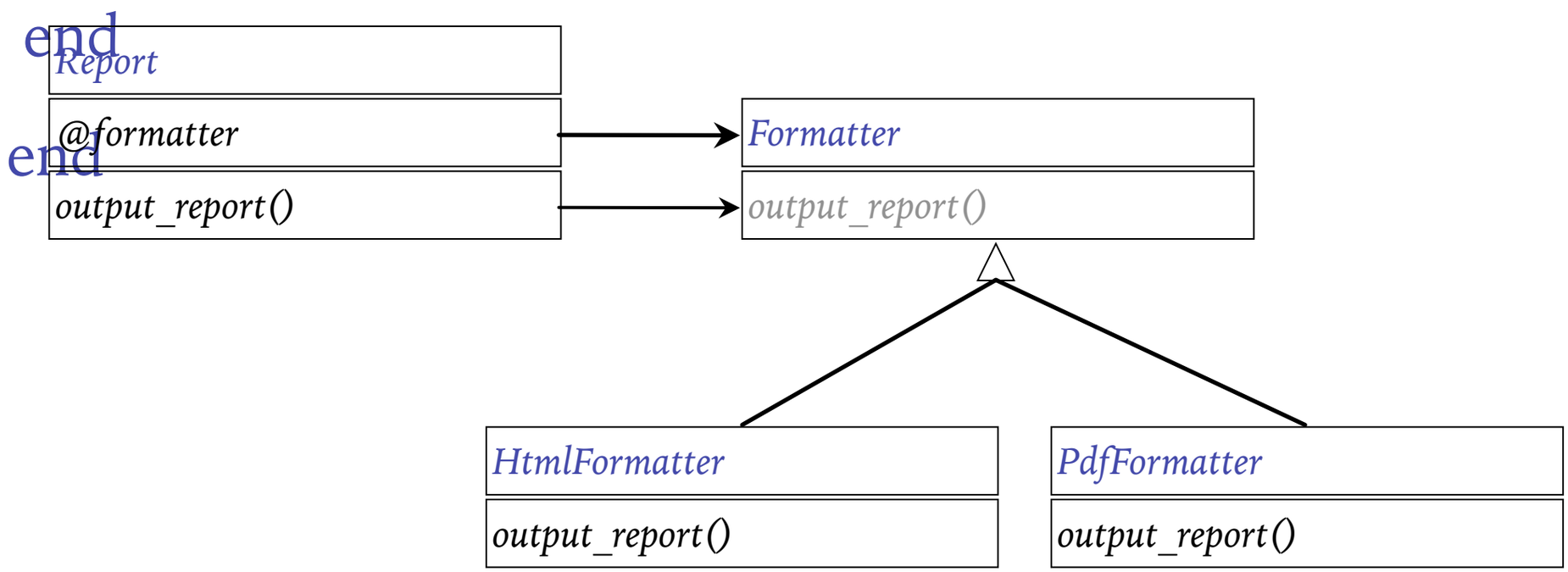


# Report Generation Using Strategy

```
class Report

 attr_accessor :title, :text, :formatter

 def output_report

  @formatter.output_report

 end

end
```

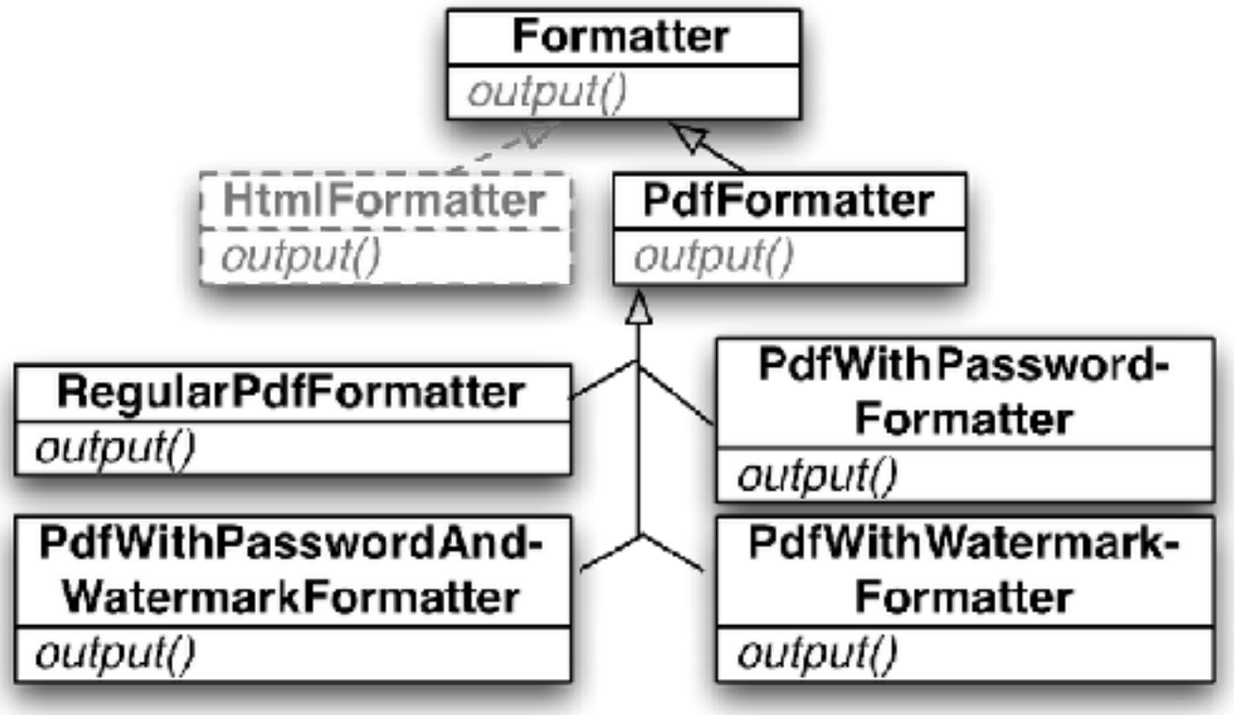
*Delegation*

*(vs. inheritance)*




*"Prefer composition over inheritance"*

# Decorator Pattern: DRYing Out Extension Points




*Example in Rails: ActiveRecord scopes*

*Movie.for_kids.with_good_reviews(3)*

*Movie.with_many_fans.recently_reviewed*

# OCP In Practice

- Can't close against *all* types of changes, so have to choose, and you might be wrong
- Agile methodology can help *expose important types of changes early*
  - *Scenario-driven design with prioritized features*
  - *Short iterations*
  - *Test-first development*
- Then you can try to close against *those types* of changes

# Liskov Substitution Principle

# Liskov Substitution: Subtypes can substitute for base types

➤Current formulation attributed to (Turing Award winner) Barbara Liskov

> *"A method that works on an instance of type T, should also work on any subtype of T"*

- *Type/subtype != class/subclass*

*With duck typing, substitutability depends on how collaborators interact with object*

➤Let's see an example

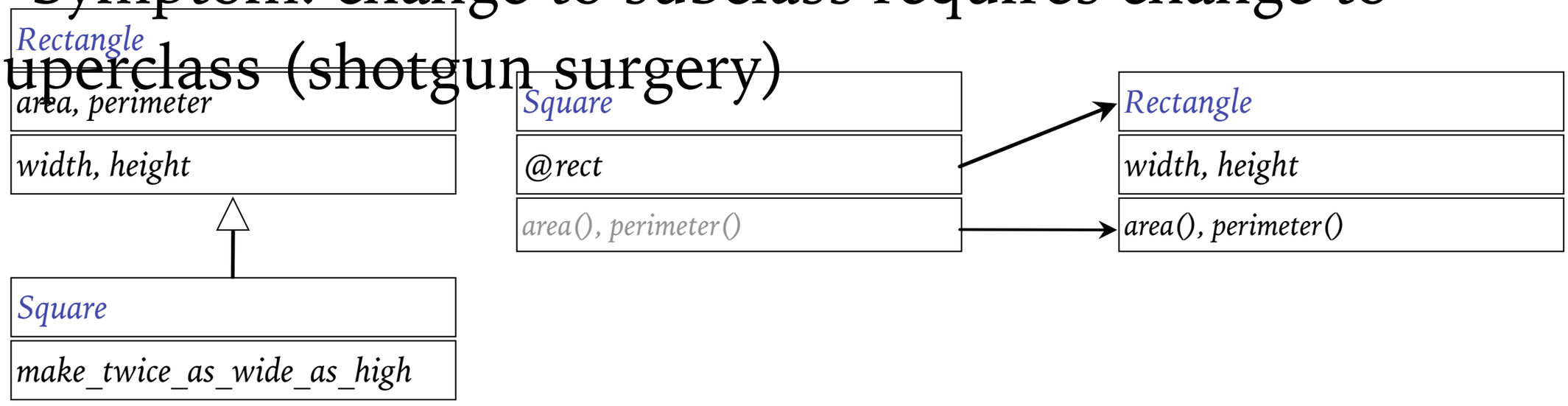
http://pastebin.com/nf2D9RYj

# Contracts

➤"Prefer composition & delegation over inheritance"

➤If can't express consistent assumptions about "contract" between class & collaborators, likely LSP violation

➤Symptom: change to subclass requires change to superclass (shotgun surgery)



# Dependency Injection
## (ELLS §11.6)



# ANNOUNCEMENTS

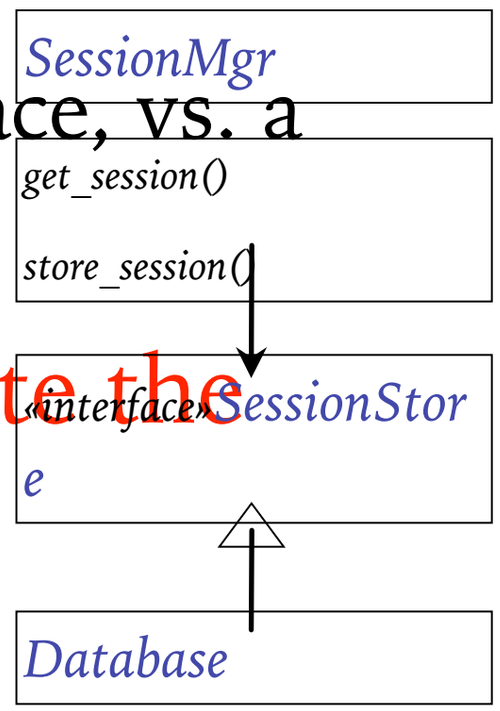
- Last Test: THURSDAY, April 22nd
  - Project Management (P&D versus agile)
  - Working in teams/git
    - What git tools can help you to avoid conflict?
    - When conflict occurs, what commands can help you fix things and see what went wrong?
  - Dealing with legacy code
    - Challenges, Methods to learn about it, Use of characterization tests
  - Comparing code and finding smells
    - Cyclomatic complexity, ABC scores, Other metrics and ways of measuring smells
  - SOFA smells (ew...)
    - What are the key method-level refactoring smells? How can we detect them? How can we deal with them? If you find one smell, are others likely to be evident also?
  - Design Patterns (This is probably the most important topic for you to really think about, especially since you haven't applied these much in practice!!!)
    - SOLID
    - For each, remember the principle and ways to deal with them. (Inheritence, Delegation, Compostion, etc...)
    - Patterns to keep in mind: Abstract Factory, Template, Strategy, Decorator, Facade, Adapter, Null Object, Proxy, Observer
    - When are these used?
    - What issues do each help solve?
    - How can you recognize them or implement them (at a high level)?

# ANNOUNCEMENTS

- ➤Sign up for group presentations by tonight!
  - ➤Presentations
    - ➤due 2 hours prior to the presentation
    - ➤submit under Final Presentation
  - ➤Group brochure- sell your product!
    - ➤due May 11th
    - ➤Submit under Group Writeups
- ➤Individual Writeups- due May 11th (instructions will appear in BB on April 28th)
  - ➤Submit under Individual Writeup
- ➤Homework 5 grades complete
  - ➤Regrades due before April 28th
- ➤Homework 6 (optional)- Blackboard is evil.
- ➤Iter 2-2 in progress
- ➤Iter 3-1 in progress

## Dependency Inversion

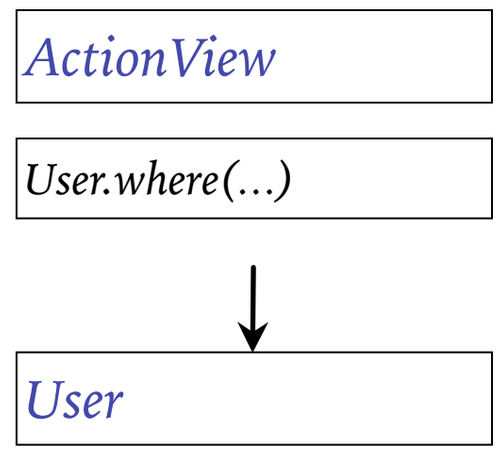
- Problem: *a* depends on *b*, but *b* interface & implementation can change, even if *functionality* stable
- Solution: "inject" an *abstract interface* that *a* & *b* depend on
  - If not exact match, Adapter/Façade
  - "inversion": now b (and a) depend on interface, vs. a depending on b
- Ruby equivalent: Extract Module to isolate the interface

# DIP in Rails: example

- What's wrong with this in a *view*:

```
- @vips = User.where('group="VIP"')
```

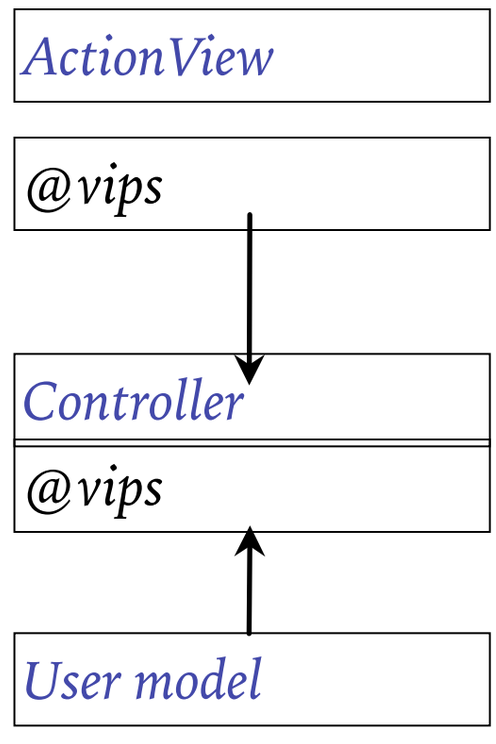



- A little better:

```
- @vips = User.find_vips
```

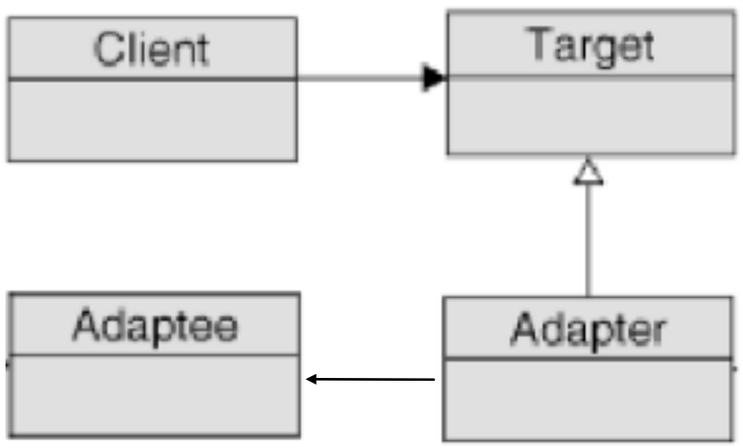
- Happiness:

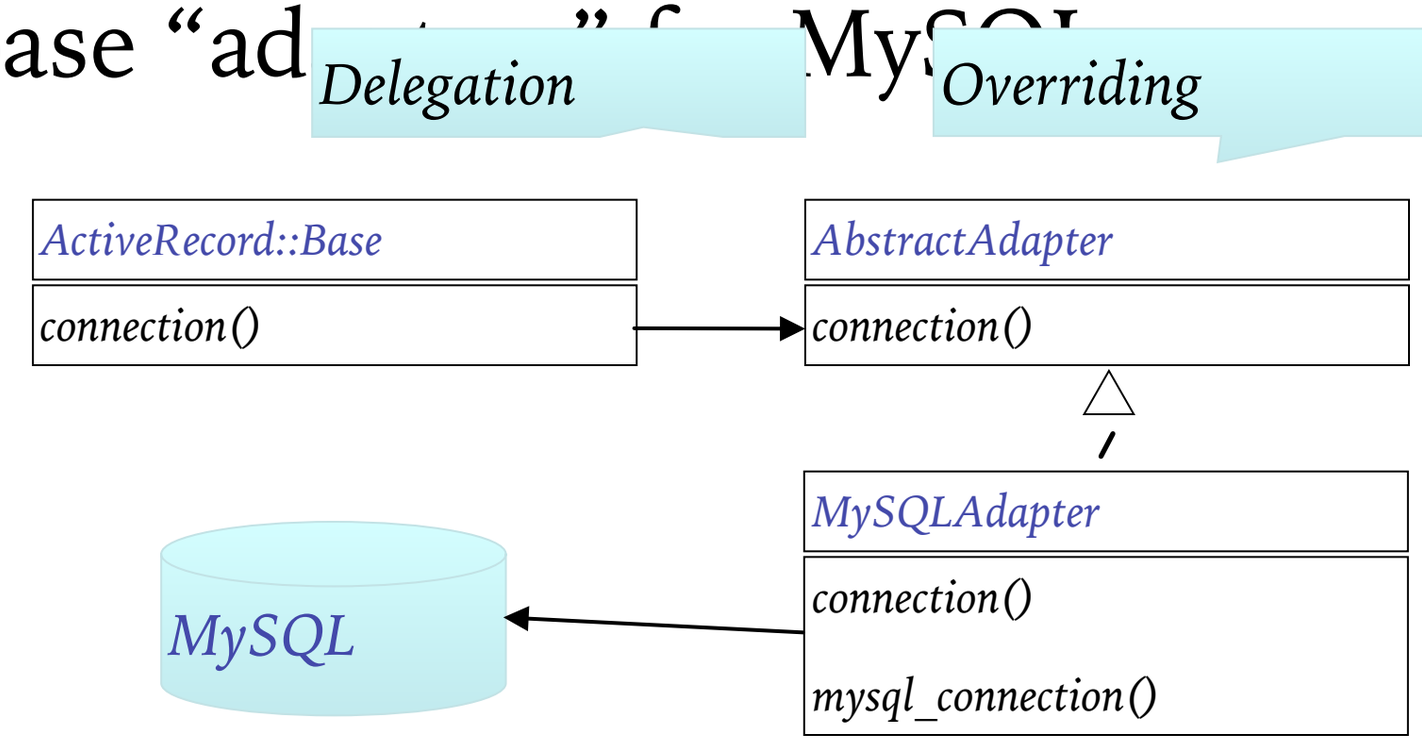
```
# in controller

@vips = User.find_vips
```



# Injecting Dependencies with the Adapter pattern

- Problem: client wants to use a "service"...
  - *service generally supports desired operations*
  - *but the API's don't match what client expects*
  - *and/or client must interoperate transparently with multiple slightly-different services*
- Rails example: database "ad[illegible]" f[illegible]MyS[illegible] Oracle, PostgreSQL, ...

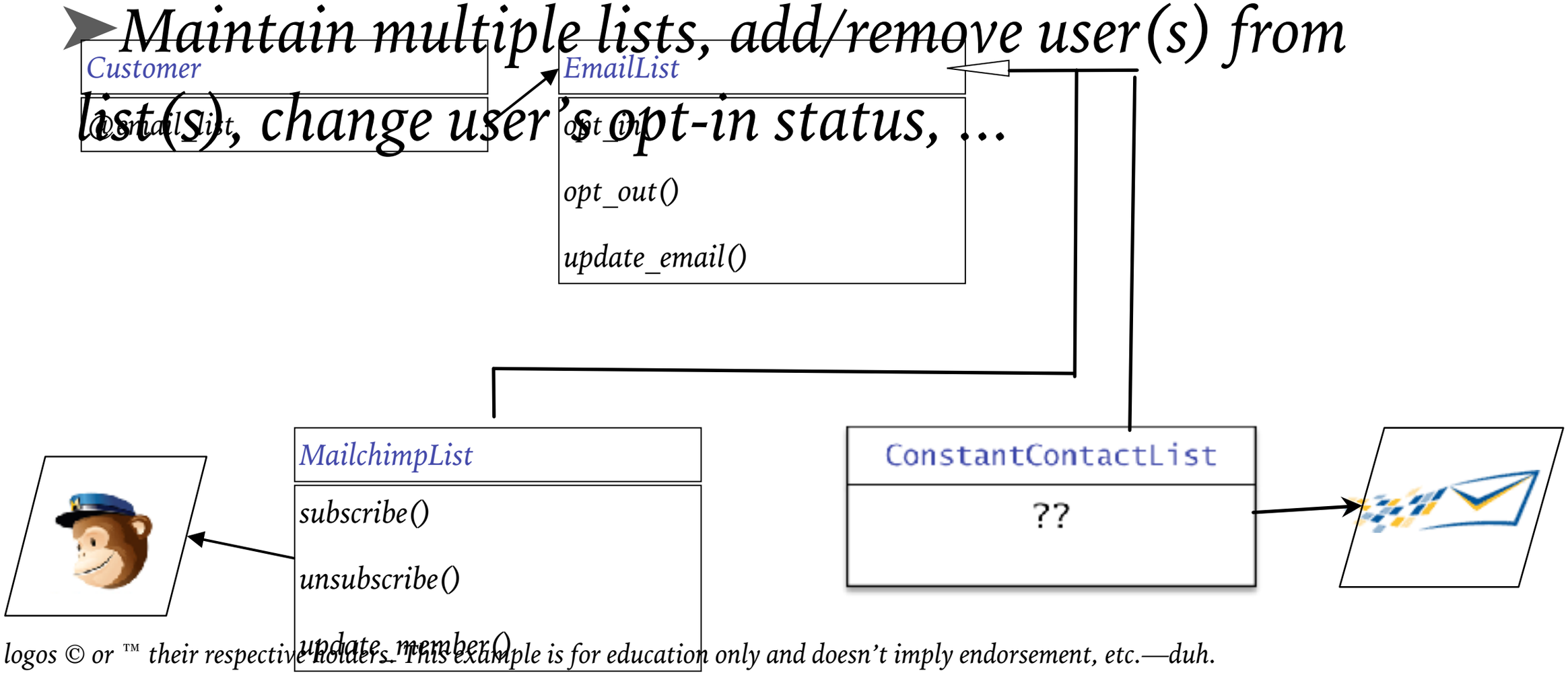


# Example: Supporting External Services

- External services for email marketing, both have RESTful API's
- Similar features
- *Maintain multiple lists, add/remove user(s) from list(s), change user's opt-in status, …*



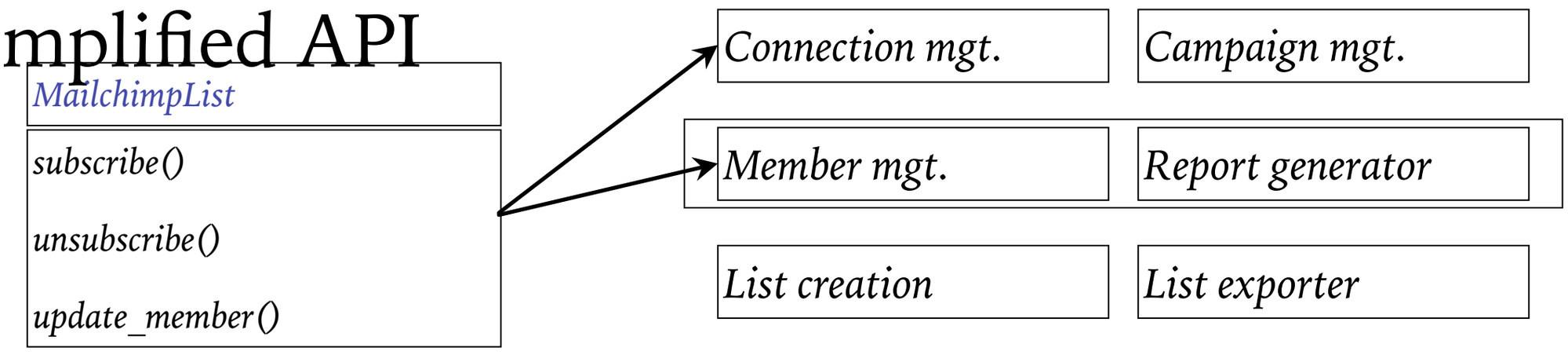



** Disclaimer: logos © or ™ their respective holders. This example is for education only and doesn't imply endorsement, etc.—duh.*

# Related: Façade

- In fact, we only use a *subset* of much more elaborate API's
  - Initialization, list management, start/stop campaign...

- So our adapter is also a *façade*
  - may *unify* distinct underlying API's into a single, simplified API



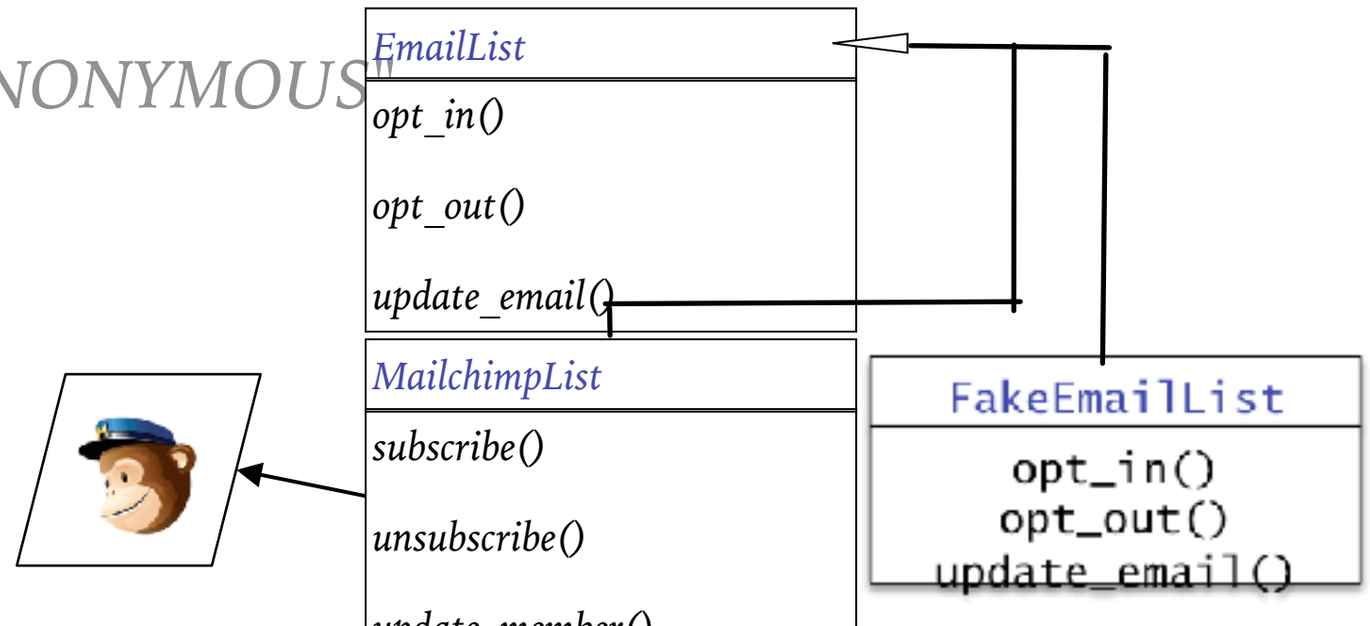
# Related: Null object

➤Problem: want *invariants* to simplify design, but app requirements seem to break this

➤*Null object:* stand-in on which "important" methods can be called

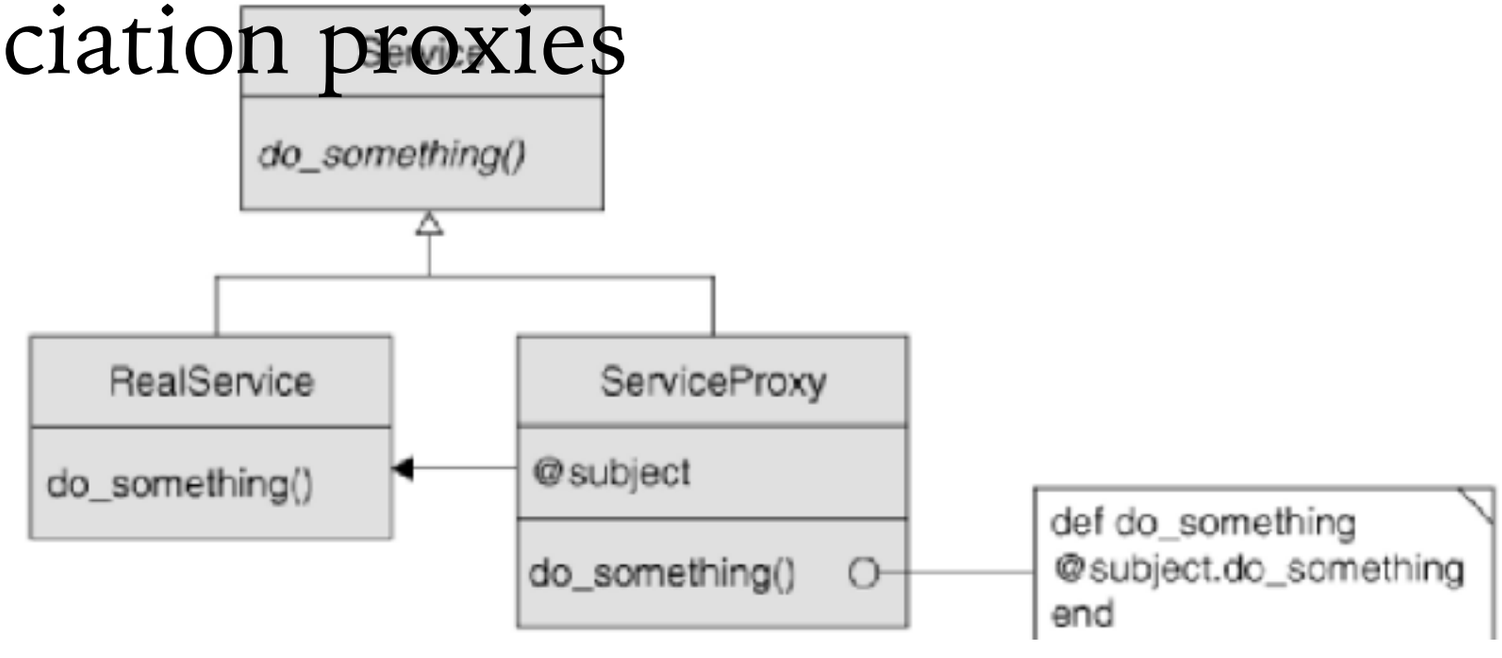
*http://pastebin.com/RBuvPMkR*

```
@customer = Customer.null_customer

@customer.logged_in?  # => false

@customer.last_name   # => "ANONYMOUS"

@customer.is_vip?     # => false
```



# Related: Proxy

- Proxy implements *same methods* as "real" service object, but "intercepts" each call
- do authentication/protect access
- hide remote-ness of a service
- defer work (be lazy)
- Rails example: association proxies (eg Movie.reviews)



# Demeter Principle
## (ELLS §11.7)



# Demeter Principle + Example

- Only talk to your friends...not strangers.
- You can call methods on
  - yourself
  - your own instance variables, if applicable
- But not on *the results returned by* them.
- Solutions:
  - replace method with delegate
  - Separate traversal from computation (Visitor)
  - Be aware of important events without knowing implementation details (Observer)

*http://pastebin.com/NRSkHstN*

# Observer

➤Problem: entity O (“observer”) wants to know when certain things happen to entity S (“subject”)

➤Design issues

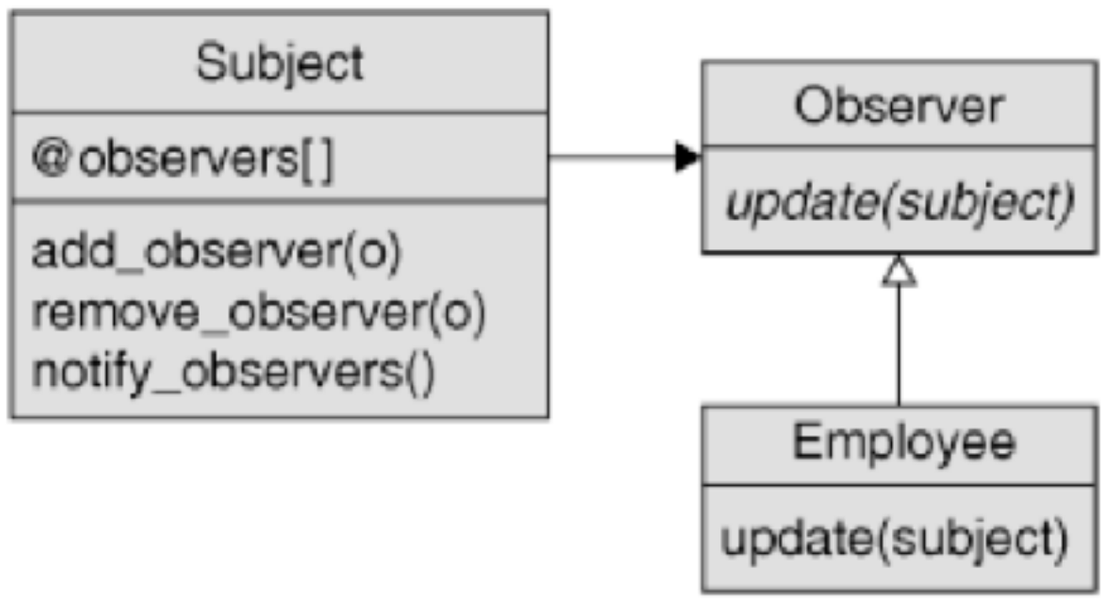
➤acting on events is O’s concern—don’t want to pollute S

➤any type of object could be an ob[illegible]

➤inheritance is awkward

*Example use cases*

- *full-text indexer wants to know about new post (e.g. eBay, Craigslist)*
- *auditor wants to know whenever “sensitive” actions are performed by an admin*



# Example: Maintaining Relational Integrity

- Problem: delete a customer who "owns" previous transactions (i.e., foreign keys point to her)
- My solution: merge with "the unknown customer"
- ActiveRecord provides built-in hooks for Observer design pattern

```
class CustomerObserver < ActiveRecord::Observer
  observe :customer # actually not needed (convention)
  def before_destroy ... end
end


# in config/environment.rb
config.active_record.observers = :customer_observer
```

# Design Patterns & SOLID wrapup
## (ELLS §11.8–11.10)

*Armando Fox*



# A Few Patterns Seen in Rails

- Adapter (database connection)
- Abstract Factory (database connection)
- Observer (caching)
- Proxy (AR association collections)
- Singleton (Inflector)
- Decorator (AR scopes, alias_method_chain)
- Command (migrations)
- Iterator (everywhere)
- Duck typing simplifies expressing and "plumbing" most of these by "weakening" the relative coupling of inheritance

# SOLID Caveat

- Designed for statically typed languages, so some principles have more impact there
  - "avoid changes that modify type signature" (often implies contract change)—but Ruby doesn't really use types
  - "avoid changes that require gratuitous recompiling"—but Ruby isn't compiled
- Use judgment: goal is *deliver working & maintainable code quickly*

# Summary

- Design patterns represent *successful solutions* to classes of problems
- Reuse of design rather than code/classes
- A few patterns "redefined" in Rails since useful to SaaS
- Can apply at many levels: architecture, design (GoF patterns), computation
- Separate what changes from what stays the same
  - *program to interface, not implementation*
  - *prefer composition over inheritance*
  - *delegate!*
  - *all 3 are made easier by duck typing*
- Much more to read & know—this is just an intro