Object-Oriented Design

Portland State UNIVERSITY

Why “Design”?

- Metz: “Immerse yourself in Objects”
  - everything else is commentary!
  - Sharp’s “Object-oriented thinking”
- Metz: Design is about *managing dependencies*
  - the fewer the dependencies, the more change the software can tolerate.

Design is the Art of arranging code

- Designs that anticipate specific future requirements almost always end badly.
  - Practical design does not anticipate what will happen to your application, it merely accepts that something will and that, in the present, you cannot know what
- The purpose of design is to allow you to do design later
  - its primary goal is to *reduce the cost of change*.

Design & Coding go together

- The problem is not one of technical knowledge but of organization;
  you *know* how to write the code but not where to put it.

Portland State UNIVERSITY

“Principles”

- You may have heard of:
  - SOLID: single responsibility, Open-Closed, Liskov Substitutability, Interface Segregation, and Dependency Inversion
  - DRY: Don't Repeat Yourself
  - Law of Demeter
  - Tell, don't Ask

Don't be fooled!

- These are not principles like Archimedes' or Heisenberg's!
- They are *heuristics* that capture what has been shown to be good practice.
    - You can violate the principles
    - But you better know why you did so!

Time-value of Code

Metz (p11):

“Sometimes the value of having the feature right now is so great that it outweighs any future increase in costs. If lack of a feature will force you out of business today it doesn’t matter how much it will cost to deal with the code tomorrow; you must do the best you can in the time you have.

Time-value of Code

"Making this kind of design compromise is like borrowing time from the future and is known as taking on technical debt. This is a loan that will eventually need to be repaid, quite likely with interest."

Agile design *vs.* BUFD

- Agile Design: arranging the code so that it can *Embrace Change*
  - recognizing that you will *never* know less about the code than before you have written any of it.
- BUFD: completely specifying and documenting the future workings of the whole program before you write any of it

Portland State UNIVERSITY

Application as Language

- Each OO application gradually becomes a unique programming language that is specifically tailored to your domain.
- Whether this language ultimately brings you pleasure or gives you pain is a matter of design.
  - See also Baniassad & Myers: “An Exploration of Program as Language”

Designing Objects
with a
Single Responsibility

Portland State UNIVERSITY

Keep it Simple

- Beck:
  - Is the simplest design the one with the fewest classes? This would lead to objects that were too big to be effective. Is the simplest design the one with the fewest methods? This would lead to big methods and duplication. Is the simplest design the one with the fewest lines of code? This would lead to compression for compression's sake and a loss of communication.

Four Constraints, in Priority Order:

- The simplest system (code and tests together) is one that:

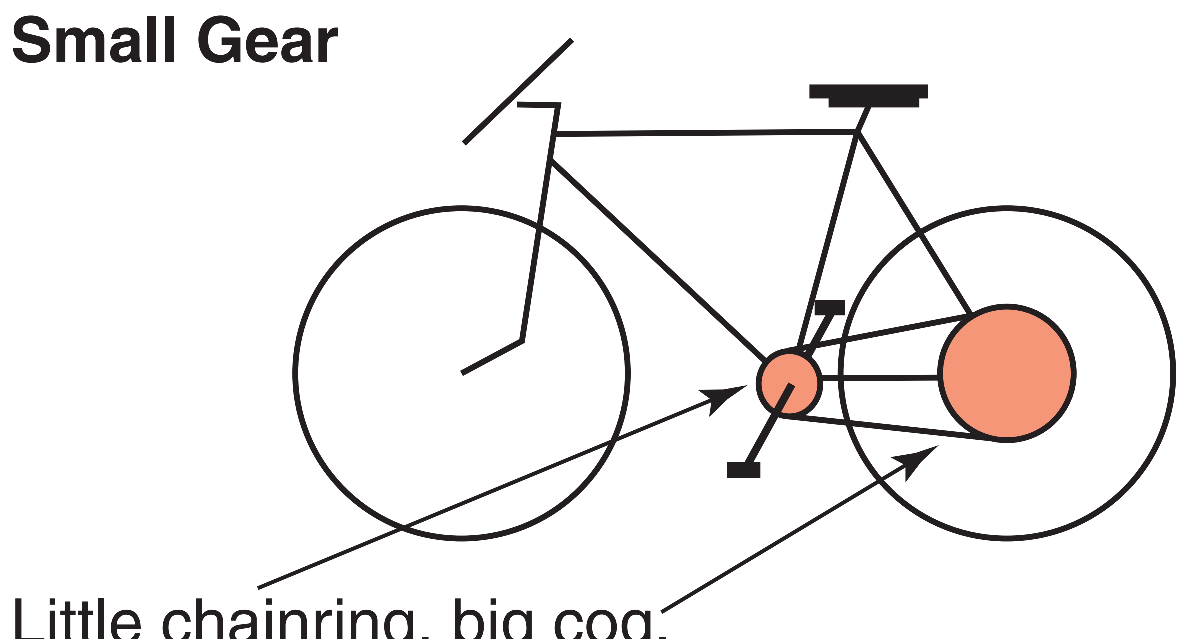
1. communicates everything you want to communicate,
2. contains no duplicate code,
3. has the fewest possible classes, and
4. has the fewest possible methods.

  - (1) and (2) together constitute the "Once and Only Once" rule.

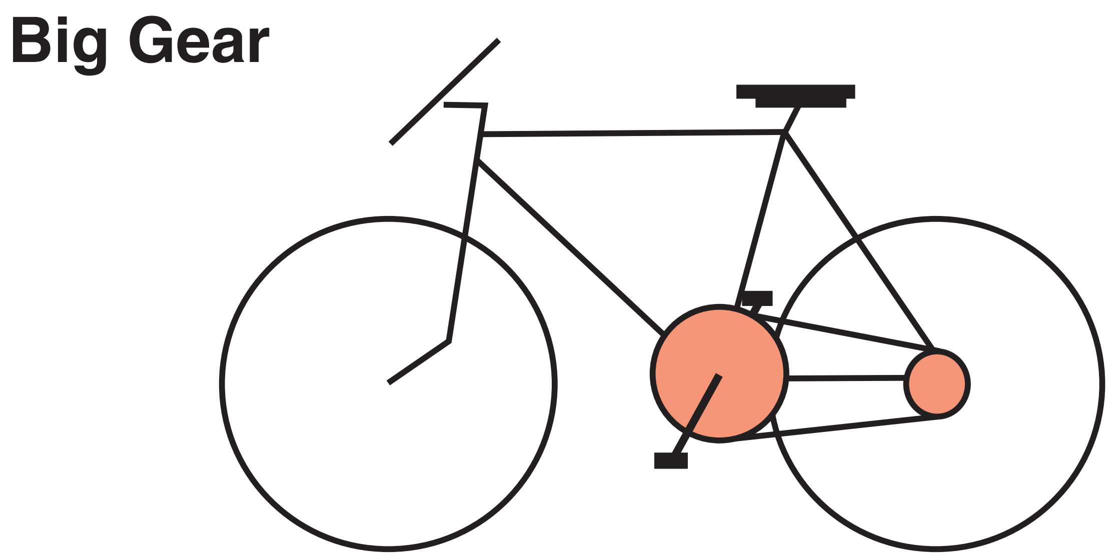
What belongs in a Class

- Metz:
  - Your goal is to model your application, using [objects], such that it does what it is supposed to do right now, and is also easy to change later.
  - Design is more the art of preserving changeability than it is the act of achieving perfection.
- An object with a single responsibility:
  - can be reused wherever that responsibility is needed
  - solves the design problem, because *it* tells *you* what code should be in it.

Bicycle Example



Little chainring, big cog.
Feet go around many times, wheel goes around just once.



A big chainring and little cog.
Feet go around once; wheel goes around many times.

Figure 2.1 Small versus big bicycle gears.

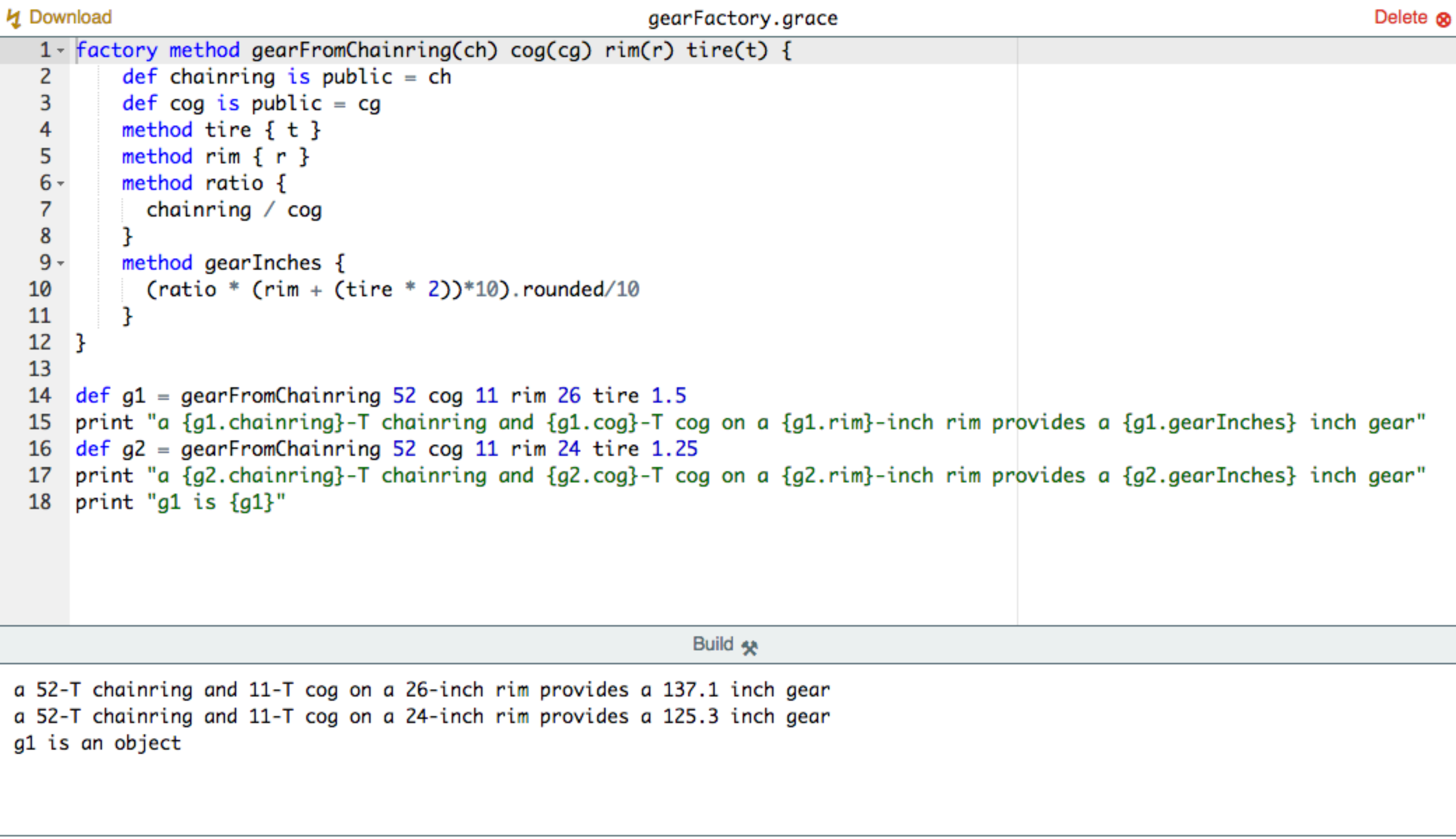
Ruby code:

```
chainring = 52                              # number of teeth
cog       = 11
ratio     = chainring / cog.to_f
puts ratio                                  # -> 4.7272727272727273

chainring = 30
cog       = 27
ratio     = chainring / cog.to_f
puts ratio                                  # -> 1.1111111111111
```

```ruby
class Gear
   attr_reader :chainring, :cog, :rim, :tire
   def initialize(chainring, cog, rim, tire)
    @chainring = chainring
    @cog       = cog
    @rim       = rim
    @tire      = tire
  end

  def ratio
    chainring / cog.to_f
  end

  def gear_inches
      # tire goes around rim twice for diameter
    ratio * (rim + (tire * 2))
  end
end

puts Gear.new(52, 11, 26, 1.5).gear_inches
# -> 137.090909090909

puts Gear.new(52, 11, 24, 1.25).gear_inches
# -> 125.272727272727
```

Portland State UNIVERSITY

Grace version



```
factory method gearFromChainring(ch) cog(cg) rim(r) tire(t) {
    def chainring is public = ch
    def cog is public = cg
    method tire { t }
    method rim { r }
    method ratio {
      chainring / cog
    }
    method gearInches {
      (ratio * (rim + (tire * 2))*10).rounded/10
    }
}

def g1 = gearFromChainring 52 cog 11 rim 26 tire 1.5
print "a {g1.chainring}-T chainring and {g1.cog}-T cog on a {g1.rim}-inch rim provides a {g1.gearInches} inch gear"
def g2 = gearFromChainring 52 cog 11 rim 24 tire 1.25
print "a {g2.chainring}-T chainring and {g2.cog}-T cog on a {g2.rim}-inch rim provides a {g2.gearInches} inch gear"
print "g1 is {g1}"
```

```
a 52-T chainring and 11-T cog on a 26-inch rim provides a 137.1 inch gear
a 52-T chainring and 11-T cog on a 24-inch rim provides a 125.3 inch gear
g1 is an object
```