Designing Objects with a Single Responsibility

Andrew P. Black

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.

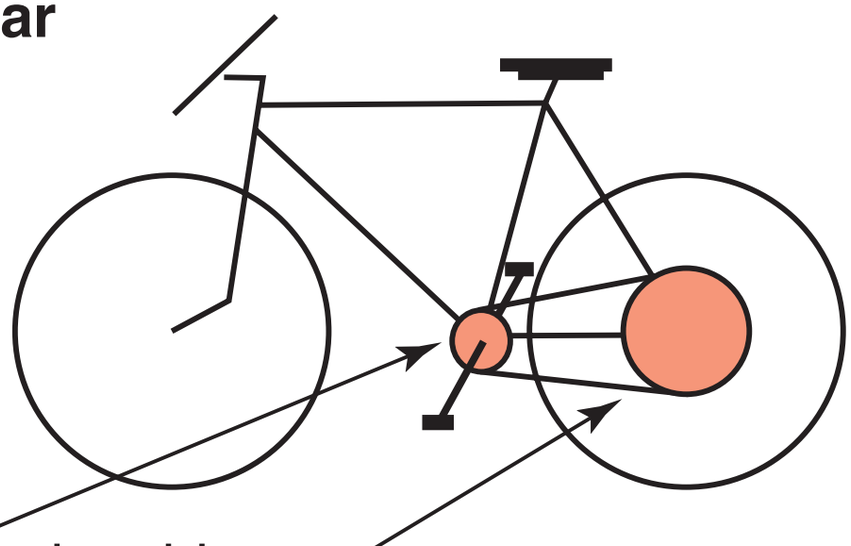
- (1) and (2) together constitute the “Once and Only Once” rule.

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.

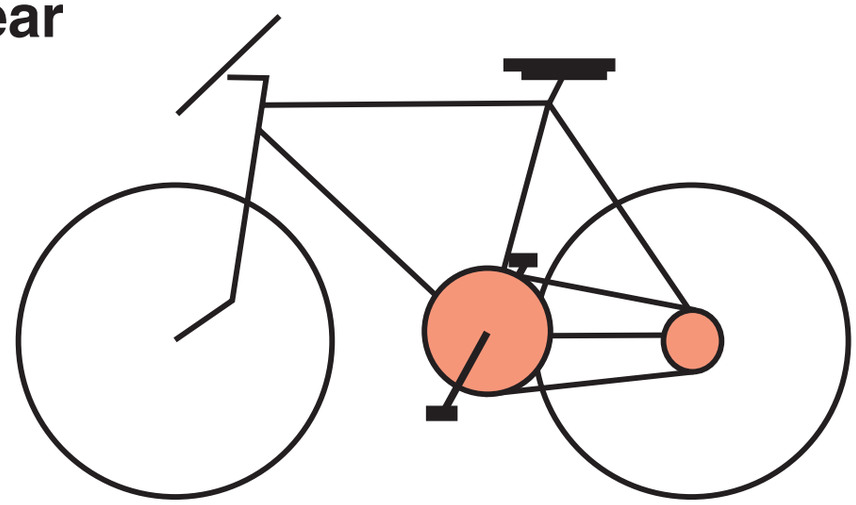
Bicycle Example

Small Gear



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

Big Gear



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:

```ruby
chainring = 52                          # number of teeth
cog       = 11
ratio     = chainring / cog.to_f
puts ratio                              # -> 4.72727272727273

chainring = 30
cog       = 27
ratio     = chainring / cog.to_f
puts ratio                              # -> 1.11111111111111
```

```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
```

Grace version

Download gearFactory.grace Delete

```
1  factory method gearFromChainring(ch) cog(cg) rim(r) tire(t) {
2      def chainring is public = ch
3      def cog is public = cg
4      method tire { t }
5      method rim { r }
6      method ratio {
7        chainring / cog
8      }
9      method gearInches {
10       (ratio * (rim + (tire * 2))*10).rounded/10
11     }
12 }
13
14 def g1 = gearFromChainring 52 cog 11 rim 26 tire 1.5
15 print "a {g1.chainring}-T chainring and {g1.cog}-T cog on a {g1.rim}-inch rim provides a {g1.gearInches} inch gear"
16 def g2 = gearFromChainring 52 cog 11 rim 24 tire 1.25
17 print "a {g2.chainring}-T chainring and {g2.cog}-T cog on a {g2.rim}-inch rim provides a {g2.gearInches} inch gear"
18 print "g1 is {g1}"
```

Build

```
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
```

Single Responsibility?

- Not really!
  - since when does a gear have a a *tire* and a *rim*?
  - mixed up with other bits of *bicycle*
- Does it matter?
  - *Maybe!*

Arguments to “Leave it be”

- Code is *Transparent* and *Reasonable*
  - Consequence of a change should be *Transparent*
  - Cost of change *proportional* to benefits
- Why? Because there *are no dependents*
- *How* should we improve it?
  - We don't yet know — but the moment that we *acquire some dependents*, we will

➡ Wait until that time

Arguments for Change

- Code is neither (re)usable, nor exemplary
  - multiple responsibilities ⇒ can't be used for other *gear*s
  - *not* a pattern to be emulated

Improve it now *vs.* improve it later

- This tension always exists!
  - designs are *never* perfect
  - the designer has to weigh the costs and benefits of a change

Embracing Change

Some basic rules that will help, regardless of what change happens:

- Depend on Behaviour, not data
  - encapsulate instance variables
    - Grace gives us this one for free
  - encapsulate data
    - *e.g.*, don't expose an array of pairs of numbers

obscuringRef

method knows all about the structure of *d*

Download obscuringReferences.grace

```
1  class obscuringReferences(d) {
2      method diameters {
3        data.map { pair -> pair.first + (pair.second * 2) }
4      }
5      def data is public = d
6  }
7
8  def or = obscuringReferences(
9    list.withAll [
10     list.withAll [622, 20], list.withAll [622, 23],
11     list.withAll [559, 30], list.withAll [559, 40] ] )
12
13
14 print(or.diameters)
15 print(or.data)
16
```

Run ▶

```
(662, 668, 619, 639)
[[622, 20], [622, 23], [559, 30], [559, 40]]
```

Separate Structure from Meaning

- If you need a table of wheel and tire sizes, make it contain *objects*, not lists
- Metz uses a Ruby Struct to create a transparent object.
- In Grace:

```
class wheelWithRim(r) tire(t) {
    // this is equivalent to the Ruby `Struct.new(:rim, :tire)`
    method rim { r }
    method tire { t }
    method asString { "{rim} wheel with {tire} tire" }
}
```

Download revealingReferences.grace Help? Search Delete

```
1  class revealingReferences(d:List⟦List⟦Number⟧⟧) {
2      method diameters {
3        wheels.map { each -> each.rim + (each.tire * 2) }
4      }
5      def wheels is public = wheelify(d)
6      method wheelify(pairs) {
7        pairs.map { pair -> wheelWithRim(pair.first) tire(pair.second) } >> list
8      }
9      class wheelWithRim(r) tire(t) {
10         // this is equivalent to the Ruby `Struct.new(:rim, :tire)`
11         method rim { r }
12         method tire { t }
13         method asString { "{rim} wheel with {tire} tire" }
14     }
15 }
16
17
18
19 def rr = revealingReferences(
20       list.withAll [
21         list.withAll [622, 20], list.withAll [622, 23],
22         list.withAll [559, 30], list.withAll [559, 40] ] )
23
24
25 print(rr.diameters)
26 print(rr.wheels)
27
```

Run ▶

```
(662, 668, 619, 639)
[622 wheel with 20 tire, 622 wheel with 23 tire, 559 wheel with 30 tire, 559 wheel
with 40 tire]
```

Embracing Change

- Enforce Single Responsibility Everywhere
  - Extract extra responsibilities from methods
  - Isolate responsibilities in classes
    - Grace lets you create “local” classes

This method is too long!

```
14    def gear_inches
15        # tire goes around rim twice for diameter
16      ratio * (rim + (tire * 2))
17    end
```

How can it be too long? It's just one line!

```
1    def gear_inches
2      ratio * diameter
3     end
4
5    def diameter
6      rim + (tire * 2)
7    end
```

Do these refactorings even when you do not know the ultimate design. They are needed, not because the design is clear, but because it isn't. You do not have to know where you're going to use good design practices to get there. Good practices reveal design.

The Real Wheel

- The customer tells you that she has need for computing *wheel circumference*.
- This tells you that your “bicycle calculator app” needs to model *wheels*.
- So let’s move *wheel* out of *gear*.