n ways to FizzBuzz in Clojure

(Or, How to Stop Worrying and Start Decomplecting.)

Aditya Athalye

2022-03-26 Sat

Demo Clojure concepts & stdlib via FizzBuzz

- The Demo Plan: pray to Demogods and...
  - Do rapid-fire live demo, until timer runs out
  - Where each FizzBuzz has reason to exist
  - And *"decomplect"* is said so often you think it's normal

Demo Clojure concepts & stdlib via FizzBuzz

- The Demo Plan: pray to Demogods and...
  - Do rapid-fire live demo, until timer runs out
  - Where each FizzBuzz has reason to exist
  - And *"decomplect"* is said so often you think it's normal
- The Demofail Plan: Blog post (may also upload video)

Demo Clojure concepts & stdlib via FizzBuzz

- The Demo Plan: pray to Demogods and...
  - Do rapid-fire live demo, until timer runs out
  - Where each FizzBuzz has reason to exist
  - And *"decomplect"* is said so often you think it's normal
- The Demofail Plan: Blog post (may also upload video)
- The Actual Plan: Drop some sizzlin' hot takes. Leik dis...

If you do FP right, you get OOP for free and vice-versa.
#Smalltalk #Clojure #Erlang #OCaml #Haskell
- Yours Truly :)

O Lambda the Ultimate, bless we who are in this demo...

That our core be functional,
and our functions be pure.
That our data be immutable,
so we may know the value of values.
That our systems be composable,
so they may scale with grace.
That their States only mutate
in pleasantly surprising ways.
That the networks and servers stay up.
Well, at least through this demo.

For otherwise, nothing lives, nothing evolves.

In the name of the α and the β and the η...
$(\lambda x.x\ x)\ (\lambda x.x\ x)$; eternally

Definitions

- FizzBuzz

  Fizz buzz is a group word game for children to teach them about division. Players take turns to count incrementally, replacing any number divisible by `THREE` with the word "Fizz", and any number divisible by `FIVE` with the word "Buzz". - Wikipedia

Definitions

- FizzBuzz

  Fizz buzz is a group word game for children to teach them about division. Players take turns to count incrementally, replacing any number divisible by `THREE` with the word "Fizz", and any number divisible by `FIVE` with the word "Buzz". - Wikipedia

- "Numbers": Natural Numbers starting at 1

Definitions

- FizzBuzz

  Fizz buzz is a group word game for children to teach them about division. Players take turns to count incrementally, replacing any number divisible by `THREE` with the word "Fizz", and any number divisible by `FIVE` with the word "Buzz". - Wikipedia

- "Numbers": Natural Numbers starting at 1

- *Clojurish*: postmodern revival of Latin roots of US English
  - *complect* : braided, entwined, hopelessly
  - *complected* : Thing that makes Clojurian frown.
  - *decomplect* : unbraid, disentwine
  - *decomplected* : Thing that makes Clojurian smile.
  - *decomplecting* : Activity that Clojurian enjoys. (coming up!)

Le FizzBuzz Classique: First try

- Accidentally discover *for* in stdlib
- "Ooh, List Comprehension. Nice!" (- The Python gentlenerds in the house :)

Le FizzBuzz Classique: First try

- Accidentally discover *for* in stdlib
- "Ooh, List Comprehension. Nice!" (- The Python gentlenerds in the house :)

```
(ns user)
(defn fizz-buzz-classic
  [num-xs]
  (for [n num-xs]
    (cond
      (zero? (rem n 15)) (println "FizzBuzz")
      (zero? (rem n 3)) (println "Fizz")
      (zero? (rem n 5)) (println "Buzz")
      :else (println n))))
```

Le FizzBuzz Classique: First try

- Accidentally discover *for* in stdlib
- "Ooh, List Comprehension. Nice!" (- The Python gentlenerds in the house :)

```
(ns user)
(defn fizz-buzz-classic
  [num-xs]
  (for [n num-xs]
    (cond
      (zero? (rem n 15)) (println "FizzBuzz")
      (zero? (rem n 3)) (println "Fizz")
      (zero? (rem n 5)) (println "Buzz")
      :else (println n))))
```

- Those pesky nils, tho.. (see REPL console)

Le FizzBuzz Classique est mort à Clojure

Désolé :(

- *for* is Lazy
- REPL is eager
- Here be Dragons
- Unlearn old habits
- Or learn from prod outage

Le FizzBuzz Classique: Remedied

No more *println*, no more nils.

```
(defn lazybuzz
  [num-xs]
  (for [n num-xs]
    (cond
      (zero? (rem n 15)) "FizzBuzz"
      (zero? (rem n 3)) "Fizz"
      (zero? (rem n 5)) "Buzz"
      :else n)))

(lazybuzz [1 3 5 15 16]) ; yes

(fizz-buzz-classic [1 3 5 15 19]) ; bleh
```

Le FizzBuzz Classique: dissected

- "Classic" FizzBuzz considered harmful (in Clojure)
- Examine & avoid its severe defects:
  - Broken behaviour
    - calculations functional
    - `println` non-deterministic
  - Broken API contract
    - "Classic" version returns useless nils
    - `lazybuzz` returns useful values
    - We like useful values
  - Broken time model
    - Effects ("do NOW") + Laziness ("maybe never") = Bad!
    - Define separately, join later in safe ways
  - Broken aesthetic
    - Do one job, do it well. Printing is *second* job.
    - "That's George's problem." - Hal & Gerry
- Bonus: See blog post for ideas to get *fired* with fizzbuzz.

Le FizzBuzz Classique: resurrected, the Clojure way

- Keep your fns pure, like `lazybuzz`
- Laziness becomes friend, as nice bonus! (Recall the children's game definition)

```
(def all-naturals (rest (range)))
(def all-fizz-buzzes (lazybuzz all-naturals))
```

- Let REPL print. Separately.

```
(take 15 all-fizz-buzzes)
```

decomplect sequence-making v/s choice-making

- Lift out logic as its own definition
  - "Do one thing well"

```
(defn basic-buzz [n]
  (cond
    (divisible? n 15) "FizzBuzz"
    (divisible? n 3) "Fizz"
    (divisible? n 5) "Buzz"
    :else n))
```

  - Retain composability with `for`

```
(def all-fizz-buzzes
  (for [n (rest (range))]
    (basic-buzz n)))
```

  - *And* open up design space

```
(def all-fizz-buzzes
  (map basic-buzz (rest (range))))
```

  - `reduce` is homework (lazy v/s eager)

decomplect execution (CPU-bound parallelism)

- Almost too embarrassing to write...

```
(def fizz-buzz map)
(def par-buzz pmap)
```

- Get CPU-bound parallelism trivially...

```
(= (fizz-buzz basic-buzz (range 1 101))
   (par-buzz basic-buzz (range 1 101)))
```

- Not too hard to understand!

```
(clojure.repl/source pmap)
```

decomplect domain: solution side as well as problem side

- "Solution side" => the language of the domain
  - Function names
  - APIs and contracts
  - Domain abstractions and entity relationships
- "Problem side" => the nature of the domain
  - Direct ("declarative") expression of middle-school maths
  - Pry apart the "what" from the "how"

decomplect solution domain (concept of divisibility)

- Name locally or lift to top level?
  - We can let-bind a lambda

```
(defn letbuzz [num-xs]
  (for [n num-xs]
    (let [divisible? (fn [n1 n2] (zero? (rem n1 n2)))]
      (cond
        (divisible? n 15) "FizzBuzz"
        (divisible? n 3) "Fizz"
        (divisible? n 5) "Buzz"
        :else n))))
```

  - But we like tiny fns that add compositional firepower

```
(defn divisible? [n1 n2]
  (zero? (rem n1 n2)))

(def divisible? (comp zero? rem))
```

- All 3 variants are refrentially transparent

decomplect solution domain more (language of fizzbuzz)

- Open up design space more with more domain concepts

```
(defn divisible?
  "Return the-word (truthy) when n divisible,
  nil otherwise (falsey)."
  [divisor the-word n]
  (when (zero? (rem n divisor))
    the-word))

(def fizzes? (partial divisible? 3 "Fizz"))
(def buzzes? (partial divisible? 5 "Buzz"))
(def fizzbuzzes? (partial divisible? 15 "FizzBuzz"))
```

- Note:
  - Truthiness/falseyness
  - args list ordered as more constant -to-> more variable

decomplect solution domain more (language of fizzbuzz)

- Now we can do *or* buzz
  - *or* is short-circuiting

```
(defn or-buzz [n]
  (or (fizzbuzzes? n)
      (buzzes? n)
      (fizzes? n)
      n))
```

- Or, *juxt* express our choice
  - given *((juxt f g h) 42) -> [(f 42) (g 42) (h 42)]*

```
(defn juxt-buzz [n]
  (some identity
        ((juxt fizzbuzzes? buzzes? fizzes? identity)
         n)))
```

  - Here `juxt` is too subtle for production, BUT useful later
- Sadly, order of conditionals still matters in both cases

decomplect problem domain (school maths, 15 is LCM)

- Make order of calculation *not* matter
- A table of remainders of 15, in a hash-map

```
(def rem15->fizz-buzz
  {0  "FizzBuzz"
   3  "Fizz"
   6  "Fizz"
   9  "Fizz"
   12 "Fizz"
   5  "Buzz"
   10 "Buzz"})
```

- Maps are functions too!

```
(rem15->fizz-buzz (rem 3 15))
;; ~nil~ implies "no result found"
(rem15->fizz-buzz (rem 1 15))
```

decomplect problem domain (school maths, 15 is LCM)

- nil-pun with short-circuiting or

```
(defn or-rem15-buzz
  [n]
  (or (rem15->fizz-buzz (rem n 15))
      n))
```

- But get is more right, with fallback for "not found"

```
(defn get-rem15-buzz
  [n]
  (get rem15->fizz-buzz
       (rem n 15)
       n))
```

- And we can do

```
(fizz-buzz get-rem15-n (range 1 16))
```

decomplect problem domain more (modulo math)

- FizzBuzz cyclical in modulo 3, 5, 15

```
(def mod-cycle-buzz ; sequence ordered by definition
  (let [n  identity
        f  (constantly "Fizz")
        b  (constantly "Buzz")
        fb (constantly "FizzBuzz")]
    (cycle [n n f n b f n n f b n f n n fb])))
```

- Map can operate over n collections.

```
(def all-fizz-buzzes
  (map (fn [f n] (f n))
       mod-cycle-buzz ; countless modulo pattern
       (rest (range)))) ; countless natural numbers
```

decomplect ALL the FizzBuzzes (prime factors)

- Think prime factors and modulo cycles
  - e.g. `[nil nil "Fizz"]`, `[nil nil nil nil "Buzz"]`

```
(defn any-mod-cycle-buzz
  [num & words]
  (or (not-empty (reduce str words))
      num))
```

  - Recall *map* is variadic, so bring on all the primes!

```
(map any-mod-cycle-buzz
     (range 1 16)
     (cycle [nil nil "Fizz"])
     (cycle [nil nil nil nil "Buzz"])
     (cycle [nil "Biz"])
     (cycle [nil nil nil nil nil nil "Fuz"]))
```

  - Bonus: get *identity* (`I`) definition too: `I` of `+` is 0, `I` of `*` is 1, `I` of FizzBuzz is all naturals

```
(map any-mod-cycle-buzz (range 1 16))
```

decomplect *mechanism* and *policy* (Say what?)

- Classically, "mechanism" and "policy" hard-wired together

```
<-- ------- MECHANISM -------- -->|<-- POLICY -->

| n divisible? 3 | n divisible? 5 | Final value |
|----------------+----------------+-------------|
| true           | true           | FizzBuzz    |
| true           | false          | Fizz        |
| false          | true           | Buzz        |
| false          | false          | n           |
```

decomplect *mechanism* and *policy* (pry the two apart)

- Mechanism: the way to construct *a* truth table

```
(ns dispatch.buzz)
(defn mechanism
  "Given two fns, presumably of any-to->Boolean,
   return a fn that can construct inputs of a
   2-input truth table."
  [f? g?]
  (juxt f? g?))
```

- Policy: the way to calculate Fizz Buzz

```
(defn divisible? [divisor n]
  (zero? (rem n divisor)))

(def fizzes? (partial divisible? 3))
(def buzzes? (partial divisible? 5))
```

decomplect *mechanism* and *policy* (recompose a-la-carte)

- Mechanism + Policy: Polymorphic dispatch joins truth table mechanism with FizzBuzz policy
  - Key: specialise the truth table mechanism to FizzBuzz

```
(map (mechanism fizzes? buzzes?)
     [15 3 5 1])
```

- Use in dispatch mechanism
  - connect truth table rows to results

```
(def fizz-buzz map)
(def fizz-buzz-mecha (mechanism fizzes? buzzes?))
(defmulti dispatch-buzz
  "Each method yields result record
  for truth table record."
  fizz-buzz-mecha)
```

decomplect *mechanism* and *policy* (recompose a-la-carte)

- Mechanism + Policy: Yes, 'tis a wee FizzBuzz interpreter!

```
(defmethod dispatch-buzz [true true]
  [n]
  "FizzBuzz")
(defmethod dispatch-buzz [true false]
  [n]
  "Fizz")
(defmethod dispatch-buzz [false true]
  [n]
  "Buzz")
(defmethod dispatch-buzz :default
  [n]
  n)
(fizz-buzz dispatch-buzz [1 3 5 15 16])
```

decomplect OOP: What is complected?

Classical OOP complects these things:

- Name (Class name / Java type)
- Structure (Class members, methods etc.)
- Behaviour (effects caused by methods)
- State (contained in the run-time instance of the Class)

decomplect OOP: with Clojure Polymorphism

- Bring back usual suspects

```
(ns oops.fizzbuzz)
(def divisible? (comp zero? rem))
(def fizz-buzz map)
(defn basic-buzz [n]
  (cond
    (divisible? n 15) "FizzBuzz"
    (divisible? n 3) "Fizz"
    (divisible? n 5) "Buzz"
    :else n))
```

- Introduce protocols (like Java Interfaces, but better)

```
(defprotocol IFizzBuzz
  (proto-buzz [this]))
```

decomplect OOP: with Clojure Polymorphism

- Add new behaviour to existing types including *any* Java builtin

```
(extend-protocol IFizzBuzz
  java.lang.Number
  (proto-buzz [this]
    (basic-buzz this)))
```

- Like this: Java type-based Polymorphic dispatch

```
(fizz-buzz proto-buzz [1 3 5 15 16])
(fizz-buzz proto-buzz [1.0 3.0 5.0 15.0 15.9])
```

decomplect OOP: with Clojure Polymorphism

- Clojure protocols cleanly solve the *Expression Problem*
- *Without* breaking Equality or any other existing semantics

```
(= 42 42) ; => true (long and long)
(= 42 42.0) ; => false (long and double)
(= 42.0 42.0) ; => true (double and double)

;; False, as it should be.
(= (fizz-buzz proto-buzz [1 3 5 15 16])
   (fizz-buzz proto-buzz [1.0 3.0 5.0 15.0 15.9]))
```

- Without performance overhead (JVM hotspot optimization)

decomplect information (nondestructive fizzbuzz)

- All fizz-buzzes so far *lose information*
- Can't undo entropy
- Very Very Bad (especially in an age of plentiful memory)
- We can FizzBuzz with "Composite" Data

decomplect information (Peano arithmetic representation)

- Define PeanoBuzz number representation starting at `[0 0]`
- PeanoBuzz is closed under this definition of Successor (S)

```
(def S (comp (juxt identity get-rem15-buzz)
             inc
             first))
(def all-peano-buzzes (iterate S [0 0]))
```

- This is nondestructive (we don't lose our Numbers)

```
(take 16 all-peano-buzzes)
```

- Trivially map PeanoBuzz back to Standard FizzBuzz

```
(= (fizz-buzz basic-buzz (range 1 101))
   (fizz-buzz second
              (take 100 (rest all-peano-buzzes))))
```

decomplect information (Records to represent FizzBuzz)

- Records provide Java Types + all generic hash-map properties

```
(ns boxed.fizz.buzz)
(defrecord Fizz [n])
(defrecord Buzz [n])
(defrecord FizzBuzz [n])
(defrecord Identity [n])
```

decomplect information (Records to represent FizzBuzz)

- Boxed variant of `basic-buzz`

```
(def divisible? (comp zero? rem))
(def fizz-buzz map)

(defn boxed-buzz [n]
  (cond
    (divisible? n 15) (->FizzBuzz n)
    (divisible? n 3) (->Fizz n)
    (divisible? n 5) (->Buzz n)
    :else (->Identity n)))

(def all-boxed-buzzes
  (map boxed-buzz (rest (range))))
```

decomplect information (Records to represent FizzBuzz)

- Composite hash-map-like data!

```
(= (fizz-buzz boxed-buzz [1 3 5 15])
   [#boxed.fizz.buzz.Identity{:n 1}
    #boxed.fizz.buzz.Fizz{:n 3}
    #boxed.fizz.buzz.Buzz{:n 5}
    #boxed.fizz.buzz.FizzBuzz{:n 15}])
```

- Which is nondestructive!!

```
(= [1 3 5 15]
   (fizz-buzz (comp :n boxed-buzz) [1 3 5 15]))
```

- *And* which has real Java types!!!

```
(= (map type (fizz-buzz boxed-buzz [1 3 5 15]))
   [boxed.fizz.buzz.Identity
    boxed.fizz.buzz.Fizz
    boxed.fizz.buzz.Buzz
    boxed.fizz.buzz.FizzBuzz])
```

decomplect context (whence a number FizzBuzzes)

- Context thus far was run-time calculation
  - Meaning embedded in in-line logic
  - Not optional / situational by default
- Some ideas to pull out FizzBuzz interpretation *context*
  - Super handy in *some* situations
  - Utility is is contextual

decomplect context (parse, don't calculate or interpret)

- Off-label use of _Clojure Spec_'s *conform* as parser
- Skirts the "can be a very bad idea" territory. YMMV.

```
(ns conformer.buzz)
(require '[clojure.spec.alpha :as s])

(defn divisible? [divisor n]
  (zero? (rem n divisor)))
(def fizzes? (partial divisible? 3))
(def buzzes? (partial divisible? 5))

(s/def ::number number?)
(s/def ::fizzes (s/and ::number fizzes?))
(s/def ::buzzes (s/and ::number buzzes?))
```

decomplect context (parse, don't calculate or interpret)

- Now we can parse input data...

```
(s/conform ::fizzes 3) ; 3
(s/conform ::buzzes 5) ; 5
(s/conform ::buzzes 3) ; :clojure.spec.alpha/invalid
(s/conform (s/and ::fizzes ::buzzes) 15) ; 15
```

- *And* handle non-conforming data gracefully, instead of panicking and throwing exceptions

```
(s/conform (s/or ::fizzes ::buzzes) "lol")
;; => :clojure.spec.alpha/invalid
```

decomplect context (parse, don't calculate or interpret)

- Relate numbers, parsers, parser results
- Set of FizzBuzz parsers

```
(def fizz-buzz-specs #{::fizzes ::buzzes ::number})
```

- Parser-accumulator

```
(defn spec-parse-buzz [x]
  [x (zipmap fizz-buzz-specs
             (map #(s/conform % x) fizz-buzz-specs))])
```

  - Which describes parse result in a tuple

```
;; e.g. (spec-parse-buzz 1)
[1 #:conformer.buzz{:fizzes :clojure.spec.alpha/invalid,
                    :buzzes :clojure.spec.alpha/invalid,
                    :number 1}]
```

decomplect context (parse, don't calculate or interpret)

- Accumulate parser results like this

```
(into {} (map spec-parse-buzz [3 15 "lol"]))
```

  - A hash-map with number assoc'd with parse result

```
{3
 #:conformer.buzz{:fizzes 3,
                  :buzzes :clojure.spec.alpha/invalid,
                  :number 3},
 15
 #:conformer.buzz{:fizzes 15,
                  :buzzes 15,
                  :number 15},
 "lol"
 #:conformer.buzz{:fizzes :clojure.spec.alpha/invalid,
                  :buzzes :clojure.spec.alpha/invalid,
                  :number :clojure.spec.alpha/invalid}}
```

decomplect context (wicked pprint Buzz)

"Let no number escape fizzbuzzness when showing itself."

- @rdivyanshu

(Truly a genetlenerd and a scholar.)

decomplect context (wicked pprint Buzz)

- Write a plain ol' function to pretty-print custom format

```
(ns pprint.buzz)
(require '[clojure.pprint :as pp])
(defn pprint-buzz [n]
  (let [divisible? (comp zero? rem)
        prettyprint
        (comp prn (partial format "%d doth %s"))]
    (cond
      (divisible? n 15) (prettyprint n "FizzBuzzeth")
      (divisible? n 3) (prettyprint n "Fizzeth")
      (divisible? n 5) (prettyprint n "Buzzeth")
      :else (prettyprint n
                         "not Fizzeth nor Buzzeth"))))
```

decomplect context (wicked pprint Buzz)

- Hotpatch pprint dispatcher to use `pprint-buzz` formatter for all Numbers!

```
(#'pp/use-method pp/simple-dispatch
                 java.lang.Number
                 pprint-buzz)
```

- Enjoy a nondestructive, hilarious FizzBuzz experience!

```
(doseq [n [1 3 5 15]] (pp/pprint n)) ;; see REPL :)
```

- This is a joke implementation of a serious idea. . .
  pretty-printing data is open, fully extensible, and can be done (and undone) in a live runtime.

decomplect what, now? (Suddenly, Transducers.)

You:

> *Uh, what more could we possibly decomplect?*

Clojure:

> *(whatever, input -> whatever) -> (whatever, input -> whatever)*

Rich Hickey:

> *Seems like a good project for the bar, later on.*

decomplect what, now? (Suddenly, Transducers.)

- Data source
  - sequence, stream, channel, socket etc.
- Data sink
  - sequence, stream, channel, socket etc.
- Data transformer
  - function of any value -> any other value
- Data transformation process
  - mapping, filtering, reducing etc.
- Some process control
  - Transduce finite data (of course)
  - Transduce streams
  - With optional early termination in either case

decomplect whatever (some setup)

- Bring back the usual suspects (this is key... reuse logic)

```
(ns transducery.buzz)

(def divisible? (comp zero? rem))

(defn basic-buzz [n]
  (cond
    (divisible? n 15) "FizzBuzz"
    (divisible? n 3) "Fizz"
    (divisible? n 5) "Buzz"
    :else n))
```

decomplect Computation and *Output* format (Demo 1)

- Separately define *only* the transformation "xform"

```
(def fizz-buzz-xform
  (comp (map basic-buzz)
        (take 100))) ;; early termination
```

- Separately define *only* input data

```
(def natural-nums (rest (range)))
```

decomplect Computation and *Output* format (Demo 1)

- Compose in various ways
  - To produce a sequence

```
(transduce fizz-buzz-xform ;; calculator step
           conj ;; output method
           []   ;; output sink
           natural-nums) ;; input source
```

  - To produce a string

```
(transduce fizz-buzz-xform
           str
           ""
           natural-nums)
```

  - To produce a CSV string

```
(transduce (comp fizz-buzz-xform
                 (map #(str s "," %)))
           str
           ""
           natural-nums)
```

decomplect Computation and *Output* format (Demo 1)

- Consider:
  - Parts *not* modified, though *output* and/or *xform* modded
  - Effort needed to reuse fizz-buzzes other than `basic-buzz`?
- Try it!

decomplect Computation and *Input* format (Demo 2)

- Setup

```
(ns transducery.buzz)
(def numbers-file
  "Plaintext file containing numbers in some format."
  "/tmp/deleteme-spat-by-clj-fizz-buzz-demo.txt")
#_(spit numbers-file
        (clojure.string/join "\n" (range 1 10001)))
#_(slurp numbers-file)
```

decomplect Computation and *Input* format (Demo 2)

- Transduce! Now, over file data.

```
(transduce (comp (map #(Integer/parseInt %))
                 fizz-buzz-xform) ;; calculator step
           conj ;; output method
           []   ;; output sink
           (clojure.string/split-lines
            (slurp numbers-file))) ;; input source
```

- Split-lines and file slurpin' still complected!!!
  - decomplect clues in Tim Baldridge's (excellent) tutorials

decomplected xform as a standalone calculator (Demo 3)

- The xform can still calculate just a single item

```
(ns transducery.buzz)
((fizz-buzz-xform conj) [] 3) ;; => ["Fizz"]
((fizz-buzz-xform str) "" 3) ;; => "Fizz"
((fizz-buzz-xform str) "" 1) ;; => "1"
((fizz-buzz-xform (fn [_ out] out)) nil 3) ;; "Fizz"
((fizz-buzz-xform (fn [_ out] out)) nil 1) ;; 1
```

- Meditate:
  - The transducer's mandate of *a la carte* re-composition *demands* that *all* the new pulling apart *must be fully compatible* with *all* the old pulling apart.

So Long And Thanks For All The λs

- Acknowledgements

  To everyone who reviewed drafts, gave feedback, ideas, encouragement, time...
  ... friends, fellow Clojurian Slack-ers, sundry gentlenerds, one's better half, and you dear reader.

  May the Source be with you.

  λ <3

- Blog post: evalapply.org/posts/n-ways-to-fizzbuzz-in-clojure
- Email complaints or fizzbuzzes to
  - fizzbuzz@evalapply.org

Buzz

Ideas on deck, to put self on the hook. . .

- curried fizzbuzz (like Ring libraries),
- concurrent fizzbuzz (with agents)
- advanced transducing fizzbuzz (xform all the fizz-buzzes in all ways)
- Maaaybe re-do Rich's ants sim 4 species of, ah, ConcurrAnts: FizzAnt, BuzzAnt, FizzBuzzAnt, IdentiAnt
- Outside of clojure.core? maaybe core.async if not too contrived