If software is transforming *society, politics, and business,* what does that mean about the people who write the software?

Building Composable Abstractions

Eric Normand

PurelyFunctional.tv

Why focus on abstractions?

What is the process?

Can we see an example?

Conclusions

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.

OBJECT TECHNOLOGY SERIES

BOOCH JACOBSON RUMBAUGH

ADDISON-WESLEY

SERIES EDITORS

Refactoring

changing a software system in such a way that it does not alter the external behavior of the code

Newton's Laws of Motion

1. Inertia

$$\sum F = 0 \Leftrightarrow \frac{dv}{dt} = 0$$

2. Acceleration

$$\sum F = ma$$

3. Action-reaction

$$F_a = -F_b$$

Force

Mass

Distance

Time

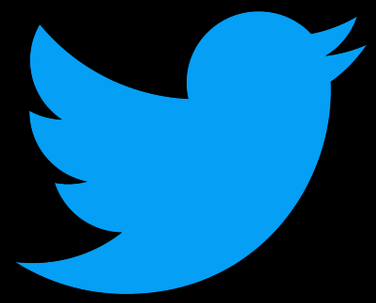
Aristotelian Physics (excerpt)

Ideal speed

Natural place

Natural motion

Unnatural motion



"Choice of abstraction matters. There is no way to *refactor* Aristotle into Newton."

–Eric Normand @ericnormand

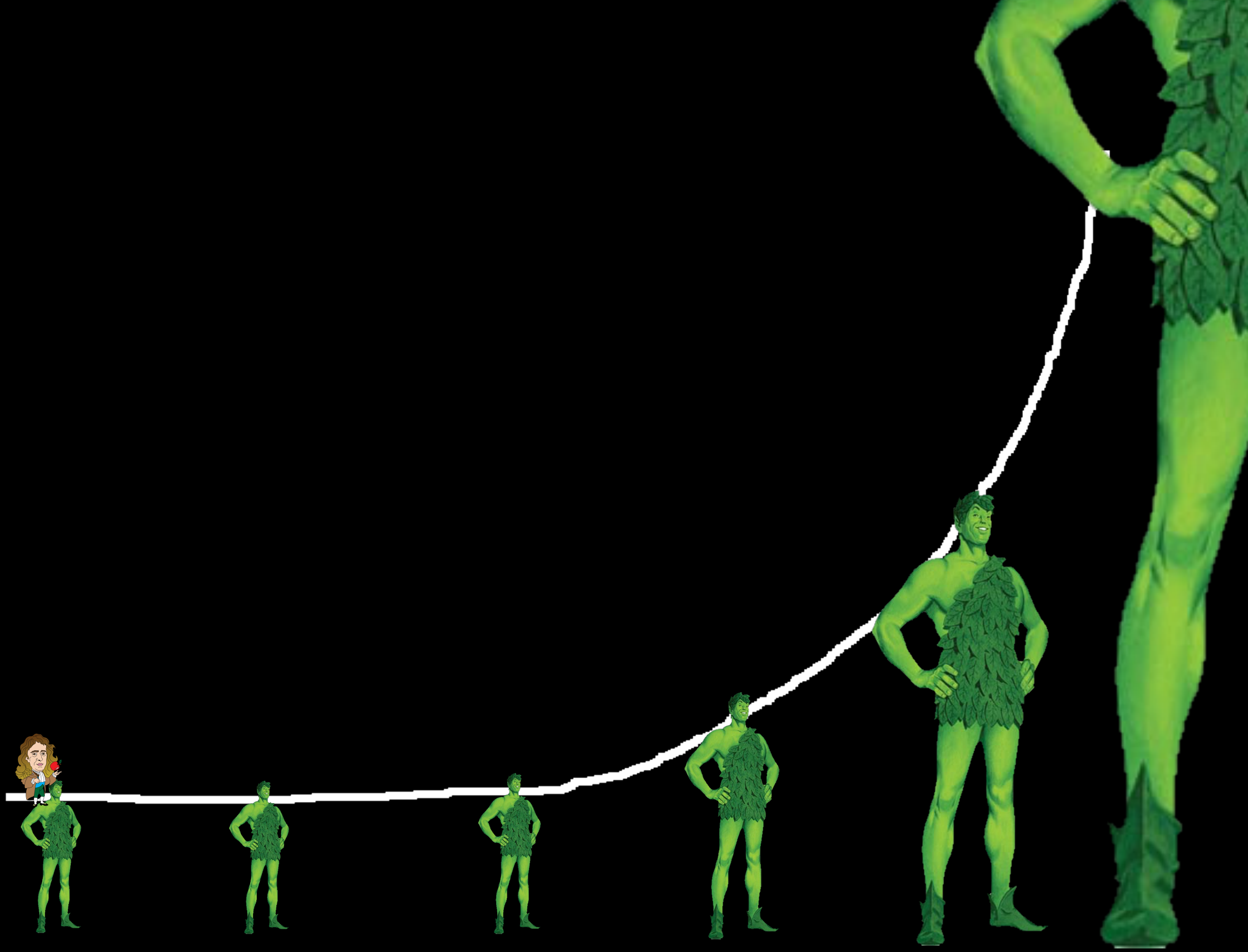
"If I have seen further, it is by standing on the shoulders of giants."

–Isaac Newton

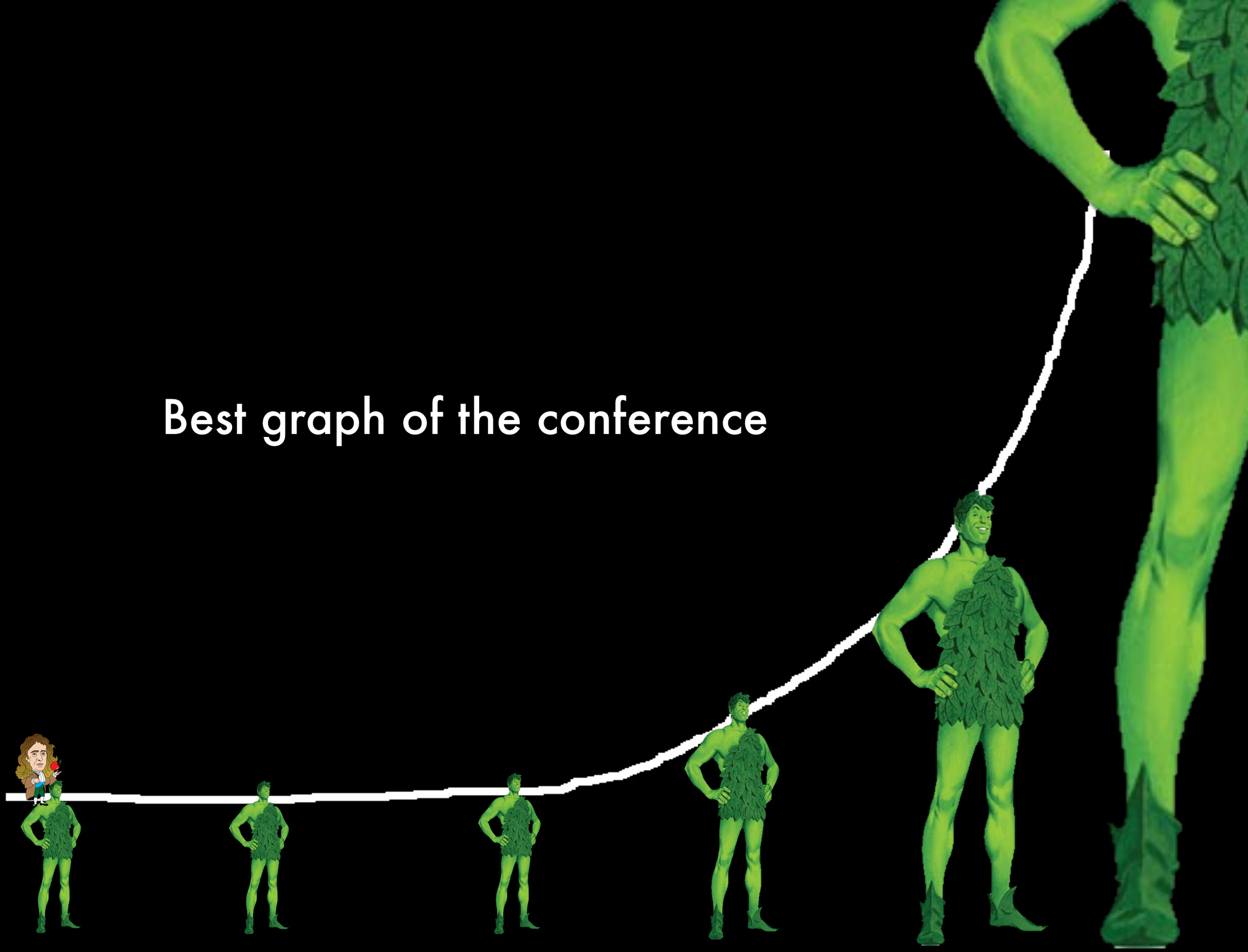
"If I have seen further, it is by standing on the shoulders of giants."

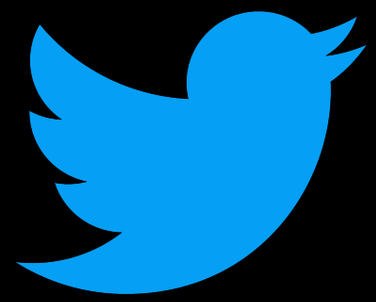
–Isaac Newton

Newton had to invent Calculus to express his abstractions.



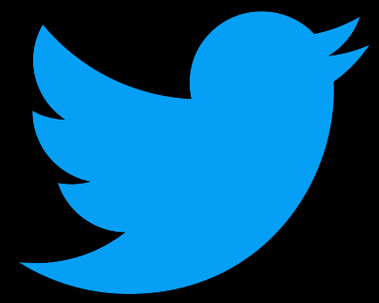
Best graph of the conference





"As programmers, we can create abstractions as powerful as Newton *every day.*"

–Eric Normand @ericnormand



"Computer programming encourages abstractions like Newtonian Mechanics instead of Aristotelian Physics."

–Eric Normand @ericnormand

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

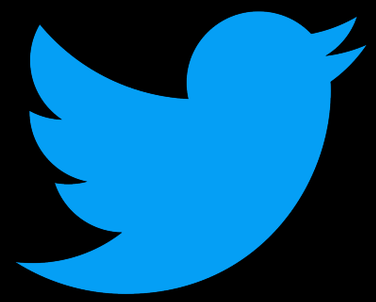
Object Technology International Inc.

OBJECT TECHNOLOGY SERIES

BOOCH JACOBSON RUMBAUGH

ADDISON-WESLEY

SERIES EDITORS



"Cleaning up is important. But we don't talk enough about what to build in the first place."

–Eric Normand @ericnormand

Objectives

Develop a process that

- Consistently produces good abstractions
- Anyone can do
- Fosters collaboration

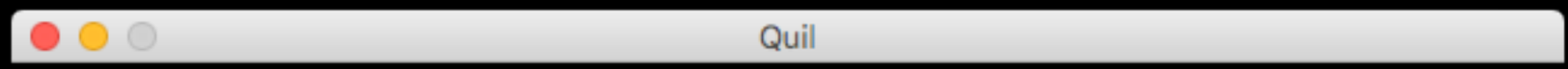
Inspired by Conal Elliott's *Denotational Design*

Example:

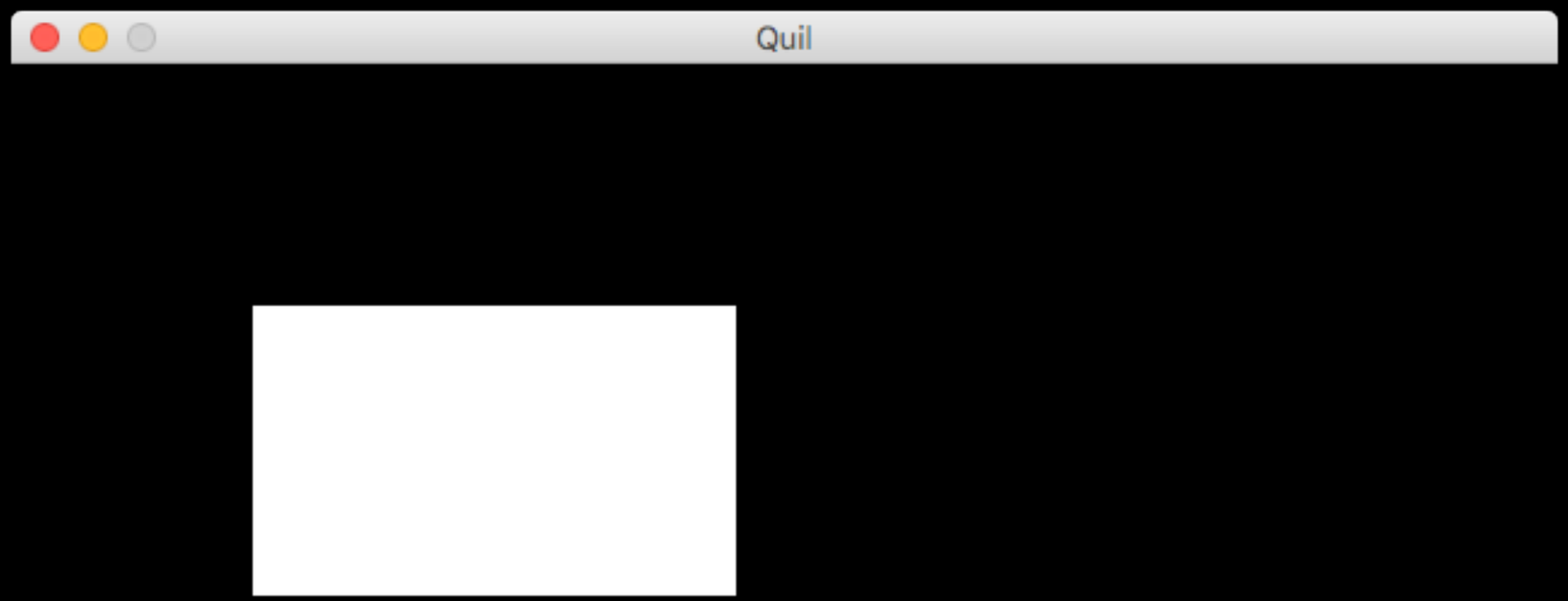
Vector Graphics System

Quil

```clojure
(defn draw []
  (q/background 0 0 0))
```

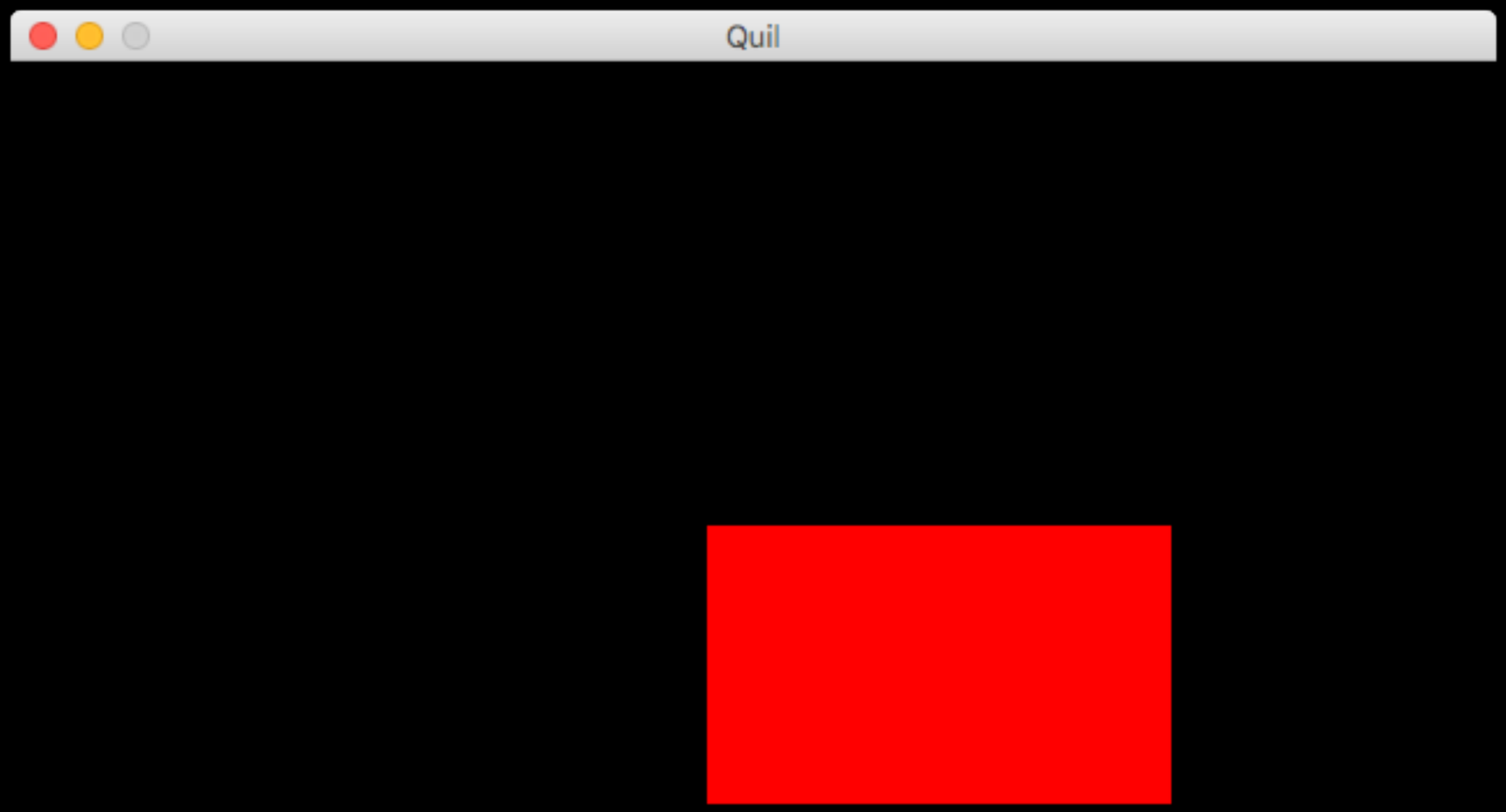


```clojure
(defn draw []
  (q/background 0 0 0)

  (q/rect 100 100 200 120))
```



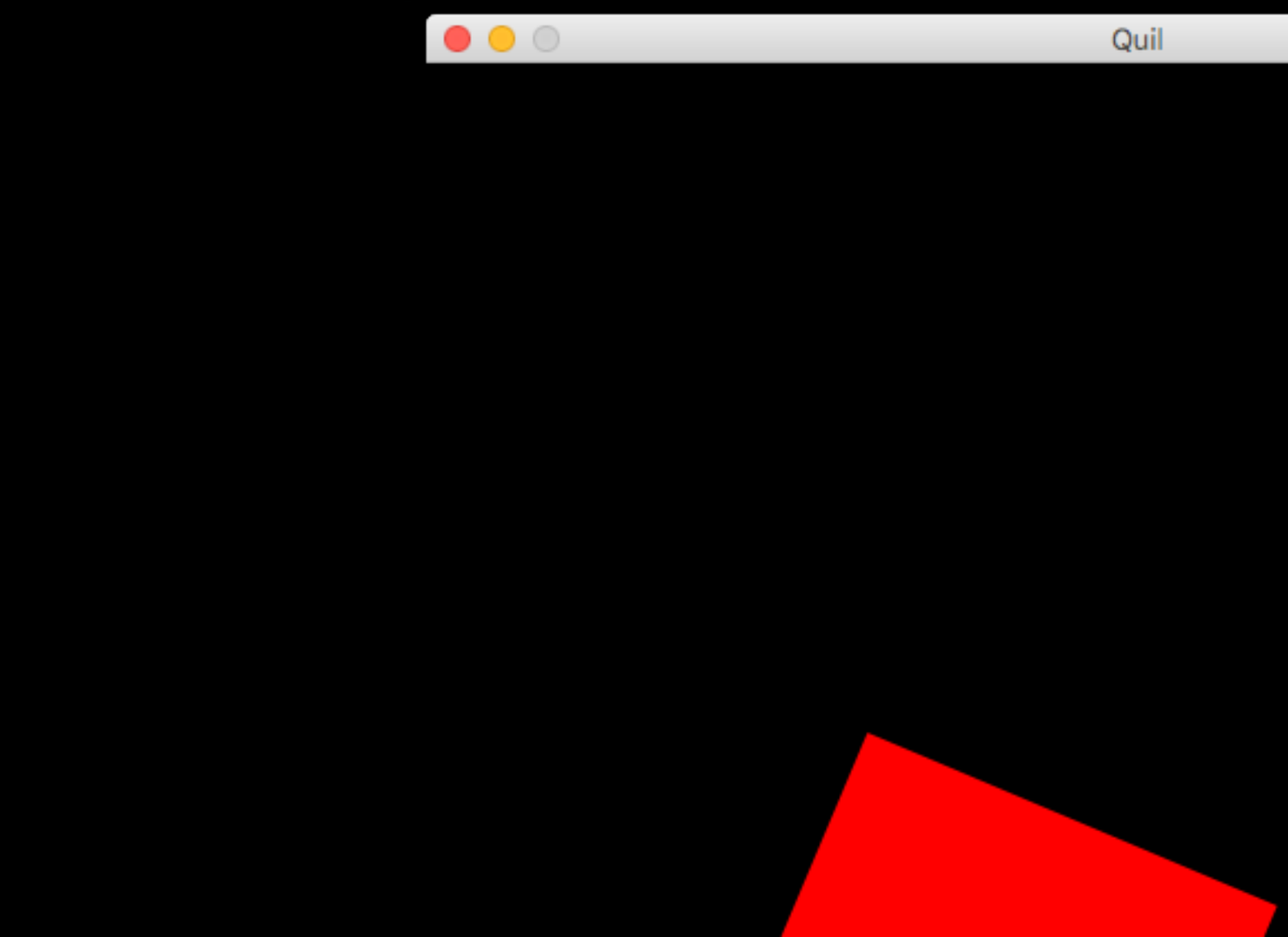
```clojure
(defn draw []
  (q/background 0 0 0)

  (q/fill 255 0 0)
  (q/rect 100 100 200 120))
```

```clojure
(defn draw []
  (q/background 0 0 0)

  (q/translate 200 100)
  (q/fill 255 0 0)
  (q/rect 100 100 200 120))
```



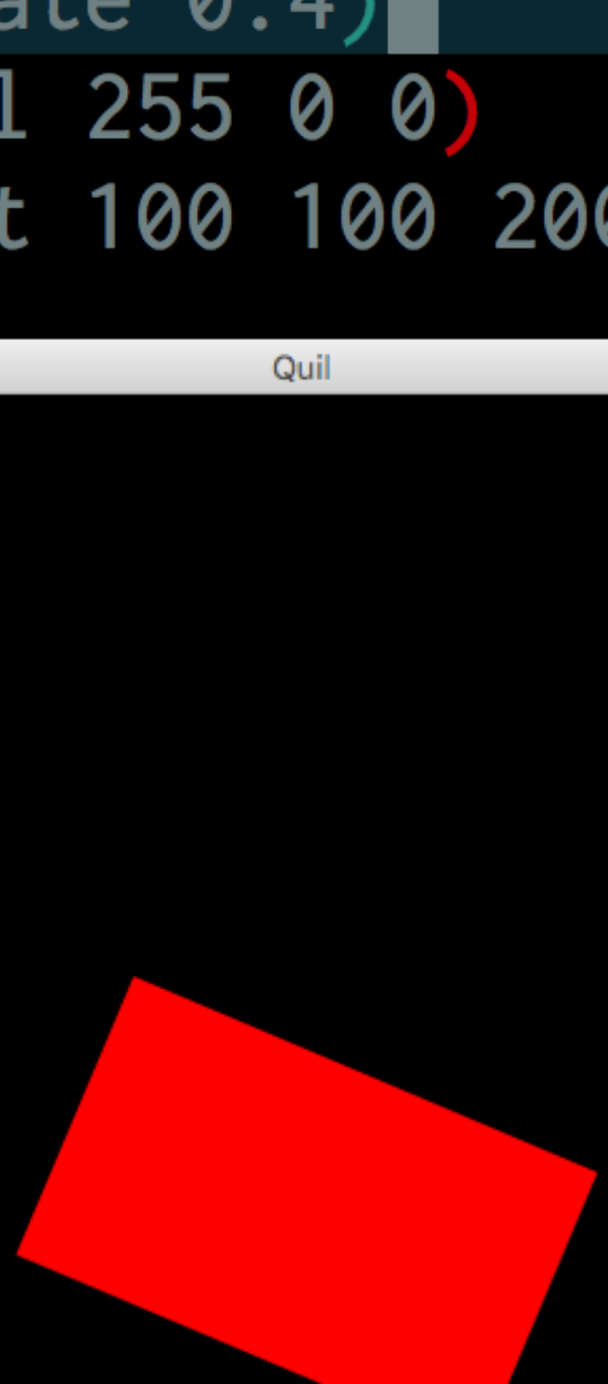
```clojure
(defn draw []
  (q/background 0 0 0)

  (q/rotate 0.4)
  (q/translate 200 100)
  (q/fill 255 0 0)
  (q/rect 100 100 200 120))
```

Quil



```clojure
(defn draw []
  (q/background 0 0 0)

  (q/translate 200 100)
  (q/rotate 0.4)
  (q/fill 255 0 0)
  (q/rect 100 100 200 120))
```

Quil



The Steps

1. Physical metaphor.
2. Meaning construction.
3. Implementation.

"Don't just start writing code.
Think about the problem first."

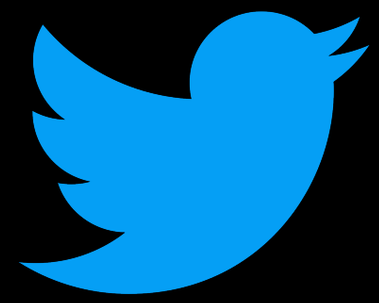
–Every experienced programmer, ever

1. Physical metaphor

Properties of a Good Metaphor

- Answers most questions
- Shared experience

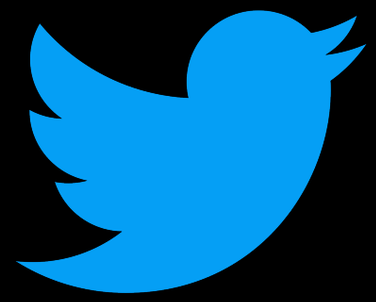
- Painting
- Stencils
- Clay
- Projected light



"Good metaphors contain answers to important questions. Different metaphors might have different answers to the same questions."

–Eric Normand @ericnormand

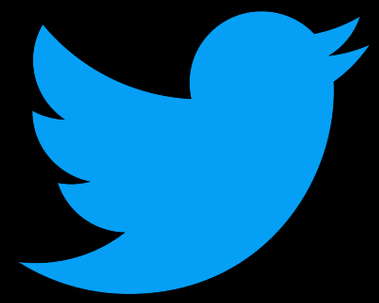
Shapes in construction paper



"A good metaphor gives you common ground for discussion. You might disagree, but at least you're disagreeing about the same thing."

–Eric Normand @ericnormand

How would you do it without computers?



"I've never met a good abstraction I couldn't turn into a good metaphor."

–Eric Normand @ericnormand

Physical metaphor summary

- Our physical intuition is rich.
- Metaphors contain answers to questions.
- Physical metaphors keep you grounded while abstracting.
- Physical metaphors are discussable.

2. Construction of meaning

"Focus first on the interface, not the implementation."

–Every programming teacher ever

What is part of the interface and what is an implementation detail?

Part of the Interface

- Distinguish between shapes
- Construct shapes
- Preservation of shape
- Preservation of color
- Overlay order
- Rotation and translation are independent
- Rotation is additive
- Translation is additive

Distinguish between shapes

```
(s/def ::CutOut some?)
(s/def ::Shape  some?)

(defn shape-of [cut-out])

(s/fdef shape-of
        :args (s/cat :cut-out ::CutOut)
        :ret  ::Shape)
```

Construct shapes

```
(s/def ::Color some?)

(defn rect [color width height])

(s/fdef rect
        :args (s/cat :color ::Color
                     :width number?
                     :height number?)
        :ret  ::CutOut)

(defn ellipse [color width height])

(s/fdef ellipse
        :args (s/cat :color ::Color
                     :width number?
                     :height number?)
        :ret  ::CutOut)
```

Preservation of shape

```clojure
(defn translate [cut-out tx ty])

(s/fdef translate
        :args (s/cat :cut-out ::CutOut :tx number? :ty number?)
        :ret  ::CutOut
        :fn   #(= (shape-of (-> % :ret))
                  (shape-of (-> % :args :cut-out))))

(defn rotate [cut-out r])

(s/fdef rotate
        :args (s/cat :cut-out ::CutOut :r number?)
        :ret  ::CutOut
        :fn   #(= (shape-of (-> % :ret))
                  (shape-of (-> % :args :cut-out))))
```

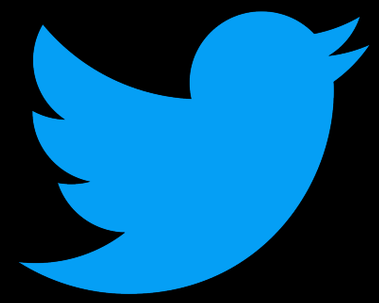
Preservation of color

```
(defn color-of [cut-out])

(s/fdef color-of
        :args (s/cat :cut-out ::CutOut)
        :ret  ::Color)

(s/fdef translate
        :args (s/cat :cut-out ::CutOut :tx number? :ty number?)
        :ret  ::CutOut
        :fn   (s/and #(= (shape-of (-> % :ret))
                         (shape-of (-> % :args :cut-out)))
                     #(= (color-of (-> % :ret))
                         (color-of (-> % :args :cut-out)))))
```

Overlay order

```
(defn overlay [cut-out-a cut-out-b])
```

```
(s/fdef overlay
        :args (s/cat :cut-out-a ::CutOut
                     :cut-out-b ::CutOut)
        :ret ::CutOut)
```



"Avoid corner cases while you can. Corner cases are multiplicative when you compose them."

–Eric Normand @ericnormand

Overlay order (retry)

```
(s/def ::Collage some?)

(defn overlay [collage cut-out])

(s/fdef overlay
        :args (s/cat :collage ::Collage
                     :cut-out ::CutOut)
        :ret ::Collage)
```

Overlay order

```
(def overlay-order
  (prop/for-all [collage (s/gen ::Collage)
                 cut-out-a (s/gen ::CutOut)
                 cut-out-b (s/gen ::CutOut)]
    (if (not= cut-out-a cut-out-b)
      (not= (-> collage
                (overlay cut-out-a)
                (overlay cut-out-b))
            (-> collage
                (overlay cut-out-b)
                (overlay cut-out-a)))
      true)))
```

Rotation and translation independence

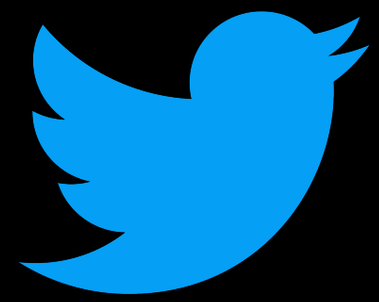
```
(def rotation-translate-independence
  (prop/for-all [cut-out (s/gen ::CutOut)
                 tx gen/int
                 ty gen/int
                 r  gen/double]
    (= (-> cut-out (translate tx ty) (rotate r))
       (-> cut-out (rotate r)        (translate tx ty)))))
```

Rotation is additive

```
(def rotation-additive
  (prop/for-all [cut-out (s/gen ::CutOut)
                 ra gen/double
                 rb gen/double]
    (= (-> cut-out (rotate ra) (rotate rb))
       (-> cut-out (rotate rb) (rotate ra)))))
```

Translation is additive

```clojure
(def translation-additive
  (prop/for-all [cut-out (s/gen ::CutOut)
                 txa gen/int
                 tya gen/int
                 txb gen/int
                 tyb gen/int]
    (= (-> cut-out (translate txa tya) (translate txb tyb))
       (-> cut-out (translate txb tyb) (translate txa tya)))))
```



"We make an abstraction composable by carefully defining the meaning of composition."

–Eric Normand @ericnormand

We forgot to draw

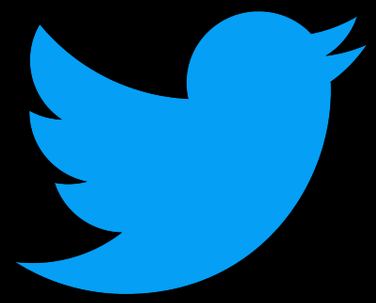
```
(defn draw! [thing])
```

```
(s/fdef draw!
        :args (s/cat :thing (s/alt :collage ::Collage
                                   :cut-out ::CutOut)))
```

We only need two types

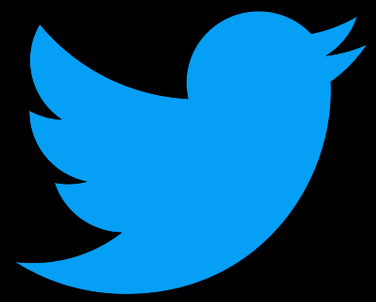
```
(s/def ::Color  (s/cat :r int? :g int? :b int?))
```

```
(s/def ::CutOut (s/fspec :args (s/cat :tx number?
                                      :ty number?
                                      :r  number?)))
```



"Meanings you define have to bottom out somewhere."

–Eric Normand @ericnormand



"Choose meanings that have the structure you're looking for."

–Eric Normand @ericnormand

Rectangles

```
(defn rect [color width height]
  (fn [tx ty r]
    (q/push-matrix)
    (q/translate tx ty)
    (q/rotate r)
    (apply q/fill color)
    (q/rect 0 0 width height)
    (q/pop-matrix)))
```

Translation

```
(defn translate [cut-out tx ty]
  (fn [tx' ty' r]
    (cut-out (+ tx tx') (+ ty ty') r)))
```

Rotation

```
(defn rotate [cut-out r]
  (fn [tx ty r']
    (cut-out tx ty (+ r r'))))
```

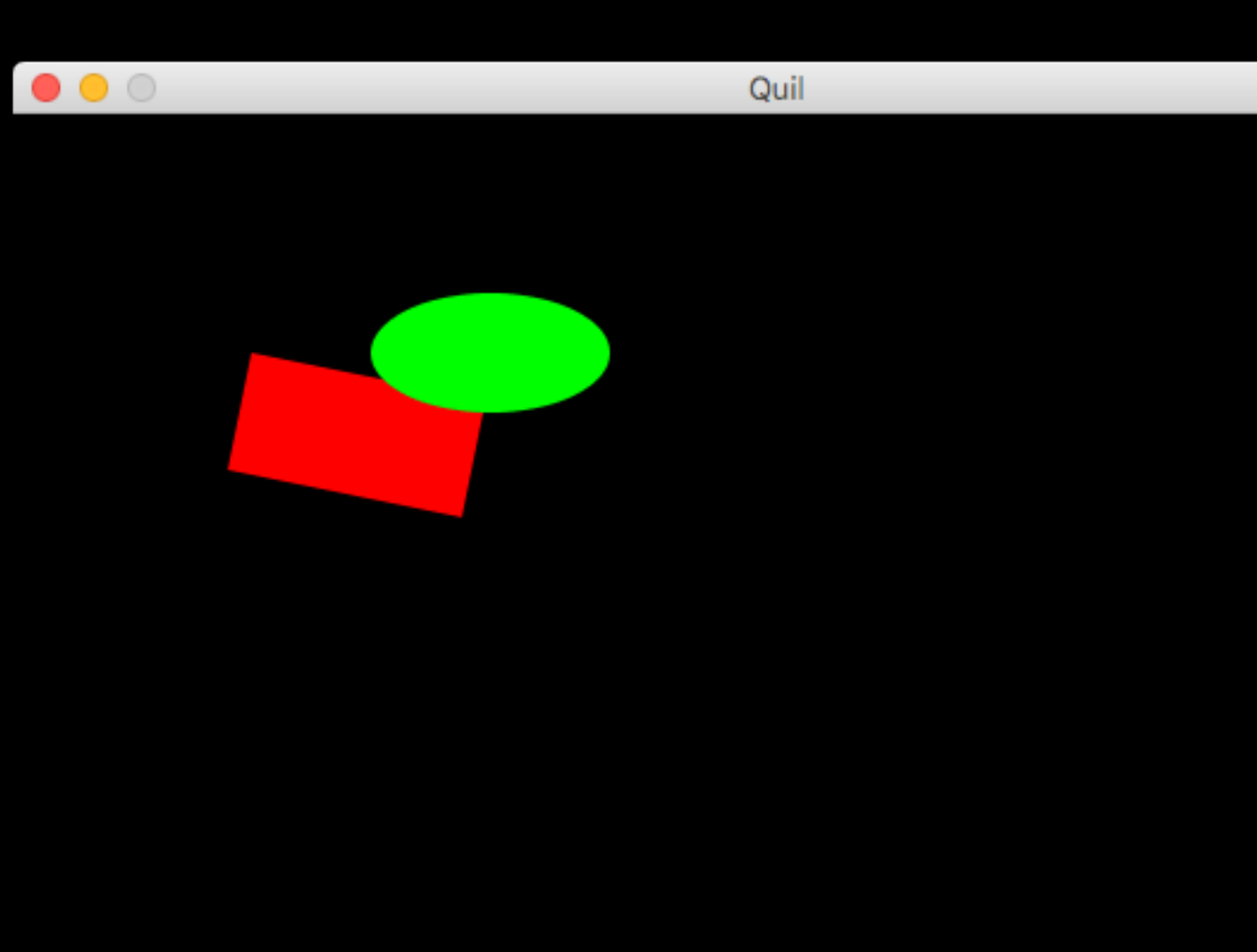
Overlay

```
(defn overlay [cut-out-a cut-out-b]
  (fn [tx ty r]
    (cut-out-a tx ty r)
    (cut-out-b tx ty r)))
```

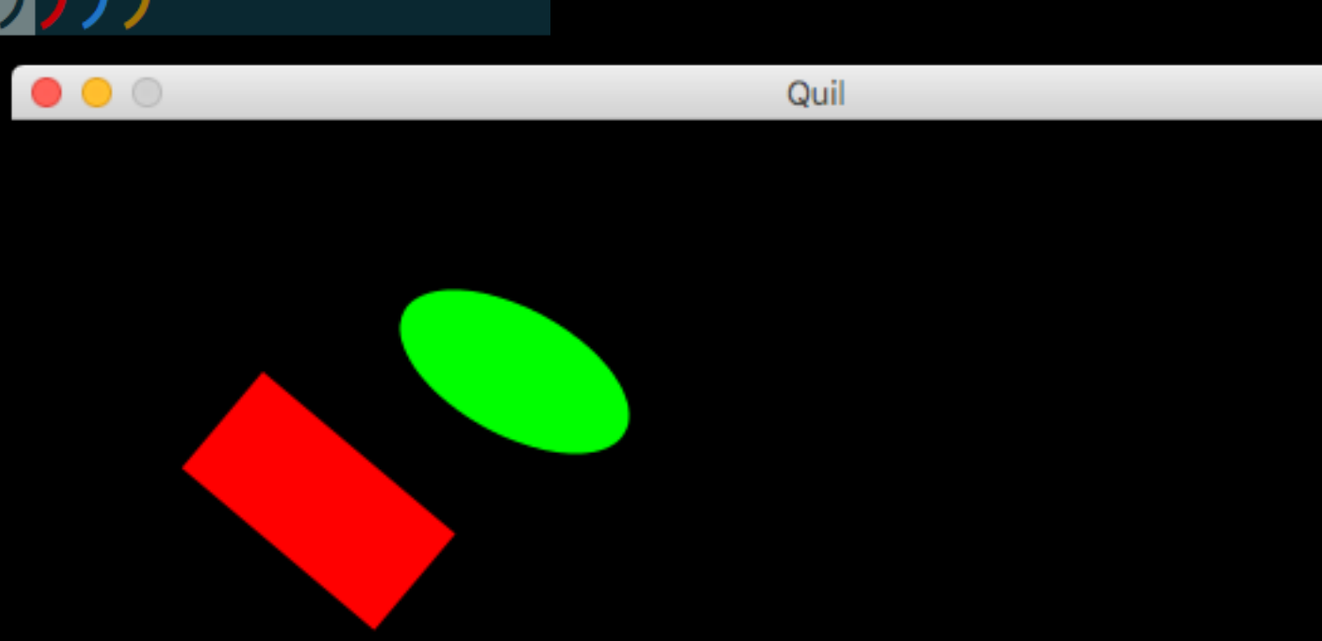
draw!

```
(defn draw! [cut-out]
  (cut-out 0 0 0))
```

```clojure
(defn draw []
  (q/background 0 0 0)

  (let [r (-> (rect [255 0 0] 100 50)
              (rotate 0.2)
              (translate 100 100))
        e (-> (ellipse [0 255 0] 100 50)
              (translate 200 100))]
    (draw! (overlay r e))))
```

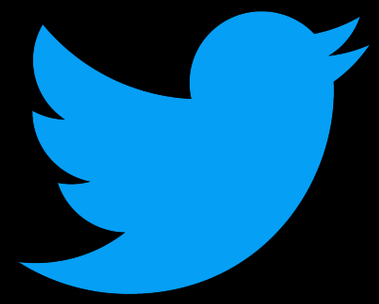


```clojure
(defn draw []
  (q/background 0 0 0)

  (let [r (-> (rect [255 0 0] 100 50)
              (rotate 0.2)
              (translate 100 100))
        e (-> (ellipse [0 255 0] 100 50)
              (translate 200 100))]
    (draw! (-> (overlay r e)
               (rotate 0.5)))))
```



Overlay

```
(defn overlay [cut-out-a cut-out-b]
  (fn [tx ty r]
    (cut-out-a tx ty r)
    (cut-out-b tx ty r)))
```



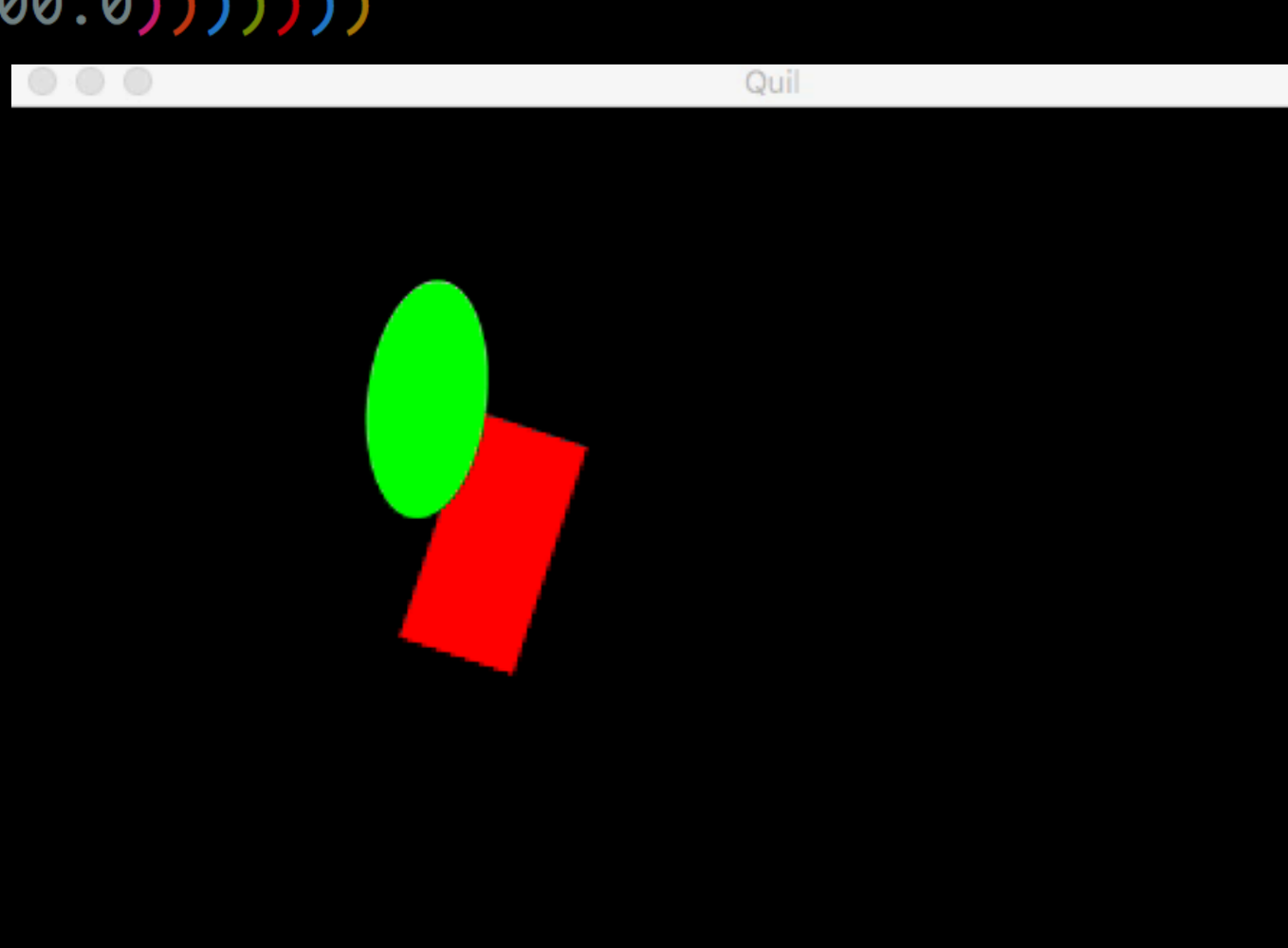
"Revisit your physical metaphor.
It contains the answers."

–Eric Normand @ericnormand

Overlay with center

```
(defn overlay [cut-out-a cut-out-b cx cy]
  (fn [tx ty r]
    (q/push-matrix)
    (q/translate tx ty)
    (q/translate cx cy)
    (q/rotate r)
    (q/translate (- cx) (- cy))
    (cut-out-a 0 0 0)
    (cut-out-b 0 0 0)
    (q/pop-matrix)))
```

```clojure
(defn draw []
  (q/background 0 0 0)

  (let [r (-> (rect [255 0 0] 100 50)
              (rotate 0.2)
              (translate 100 100))
        e (-> (ellipse [0 255 0] 100 50)
              (translate 200 100))]
    (draw! (-> (overlay r e 200 125)
               (rotate (-> (System/currentTimeMillis)
                           (mod 6283)
                           (/ 1000.0))))))))
```



Step 2 Summary

- Preserve features you want to keep.
- Eliminate features you don't need.
- No corner cases.
- Choose existing constructs that share structure.
- Choose existing constructs that are well-defined.
- Focus on composition.

3. Implementation

"Just use a map."

–Well-meaning Clojure programmers everywhere

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.

OBJECT TECHNOLOGY SERIES

BOOCH JACOBSON RUMBAUGH

ADDISON-WESLEY

SERIES EDITORS

Refactoring

changing a software system in such a way that it does not alter the external behavior of the code

Refactoring

changing a software system in such a way that it does not alter the meaning of the code

Objectives

Develop a process that

- Consistently produces good abstractions
- Anyone can do
- Fosters collaboration

The Process

1. Physical metaphor
    - Guidance and grounding
2. Construction of meaning
    - Define the parts and their relationships
    - Precise mathematical language
3. Implementation
    - Refactoring to achieve meta-properties

Corollaries

- Know your domain (metaphor)
- Know your constructs (meaning)
- Know your refactorings (implementation)

Take it further

- Visit bit.ly/ComposableAbstractions
- Enter your email address
- I'll send you
  - Slides
  - Links to the inspirations for this talk
  - Other resources