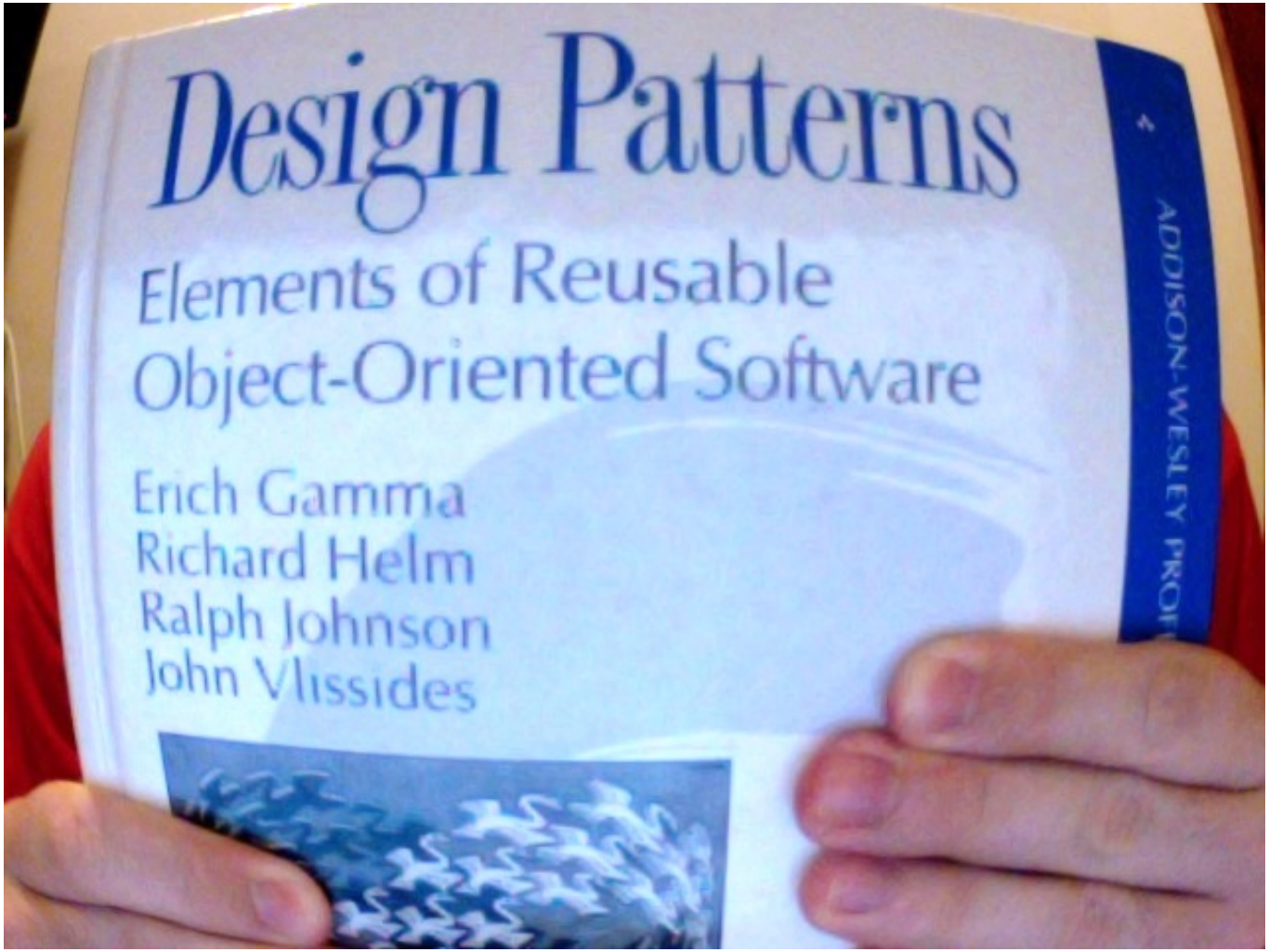
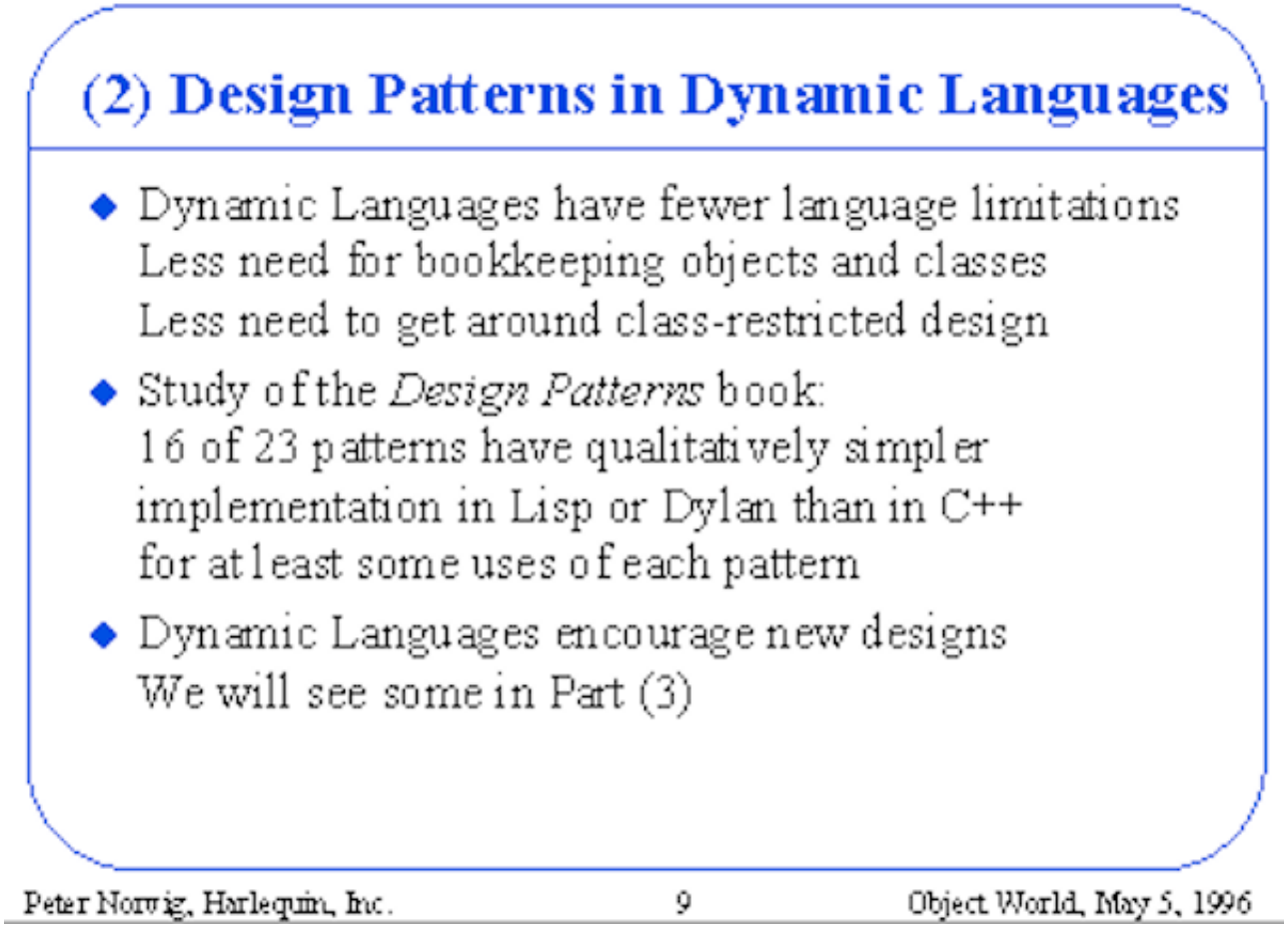
# Functional Design Patterns

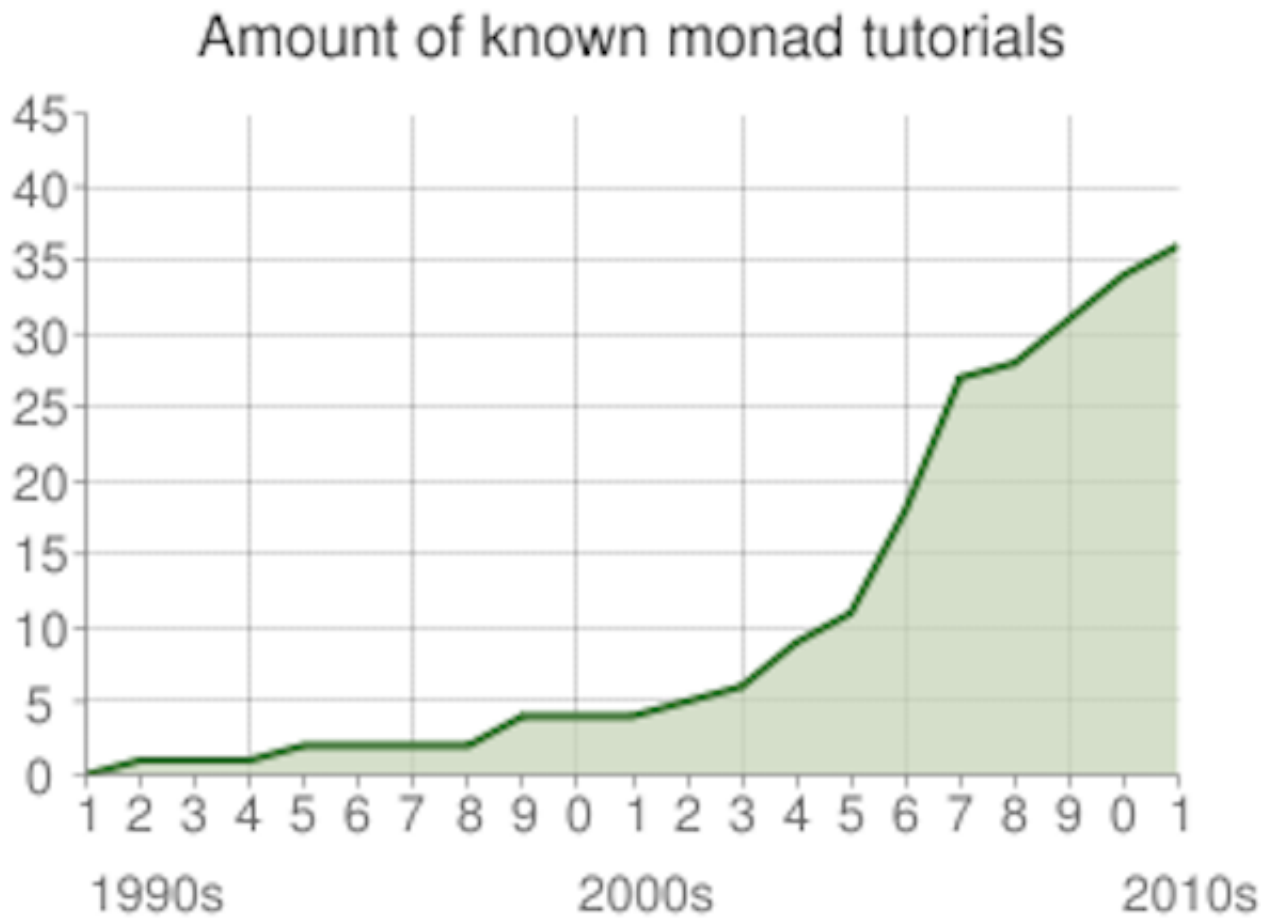
@stuartsierra

clojure.com

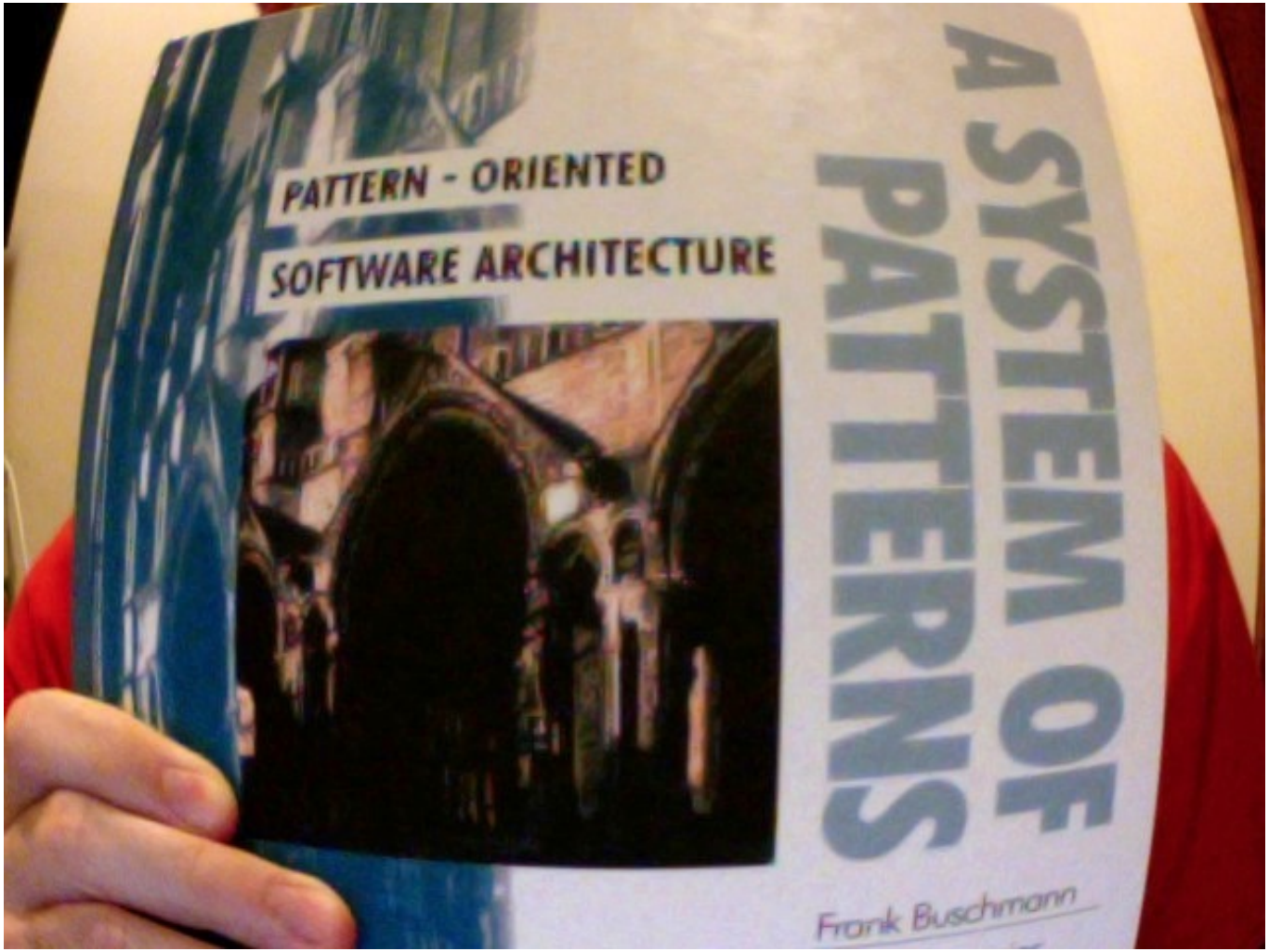

Design Patterns
Elements of Reusable
Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
ADDISON-WESLEY




# Design Patterns in Dynamic Programming
(Peter Norvig, 1996-1998)




Monad Tutorials Timeline (Haskell.org)


PATTERN - ORIENTED
SOFTWARE ARCHITECTURE
A SYSTEM OF PATTERNS
Frank Buschmann

"An *architectural pattern* expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them."

"A *design pattern* provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context."

"An *idiom* is a low-level pattern specific to a programming language. an idiom describes how to implement particular aspects of components or the relationships between them using the features of a given language."

# State Patterns

# State/Event Pattern

- State is derived from previous state + input
- Many diverse inputs
- Need to recover past states
- Need to visualize intermediate states

# State/Event Pattern

```
(defn update-state [current-state event]
  ;; return new state
  )
```

# State/Event Pattern

Recreate any past state by reducing over events

```
(defn end-state [start-state events]
  (reduce update-state start-state events))
```

# Consequences Pattern

- Each event can trigger multiple events
- Generated events cause state changes
- Need to visualize intermediate states

# Consequences Pattern

```
(defn consequences [current-state event]
  ;; return a sequence of new events
  )

(defn apply-consequences [current-state event]
  (reduce update-state current-state
          (consequences current-state event)))
```

# Consequences Pattern

Do you allow consequences to have consequences?

```
(defn recursive-consequences [current-state event]
  (reduce (fn [state event]
            (recursive-consequences
               state (update-state state event)))
          current-state
          (consequences current-state event)))
```

# Consequences Pattern

- Consequence functions do not compose naturally
- Have to update state in between

```
(defn chain-consequences [initial-state consequence-fns]
  (loop [state  initial-state
         fs     consequence-fns
         output []]
    (if (seq fs)
      (let [events    ((first fs) state)
            new-state (reduce update-state state events)]
        (recur new-state (rest fs) (into output events)))
      output)))
```

# Data Building Patterns


ICL
EXCHANGEABLE DISC DRIVE
ED40
ED80
48
4

# Accumulator Pattern

- Large collection of inputs
    - Maybe larger than memory
- Small or scalar result

# Accumulator Pattern

- Lazy sequences
    - `map`, `mapcat`, `filter`, etc.
- `reduce` is the universal accumulator

```
(reduce (fn [result input]
          ;; return updated result
          )
        initial-result
        inputs)
```

# A Digression: MapReduce

- Input is linear, maybe larger than 1 disk
- Disks are slow and local to one machine
- Networks are slow
- Not quite `map` and `reduce`

# A Digression: MapReduce

```
(defn mapper [value]
  ;; return a sequence of [key value] pairs
  )

(defn reducer [[key values]]
  ;; return a sequence of [key value] pairs
  )

(defn shuffle [pairs]
  (reduce (fn [m [k v]]
            (update-in m [k] (fnil conj []) v))
          {} pairs))

(defn mapreduce [inputs]
  (mapcat reducer (shuffle (mapcat mapper inputs))))
```

# Reduce/Combine Pattern

- Input is tree-like
- Divide-and-conquer approach
- Combining intermediate results is *associative*
    - $(a + b) + c = a + (b + c)$

# Reduce/Combine Pattern

```
(defn reduce-fn
  ([] ;; return initial "identity" value
     )
  ([result input] ;; return updated result
     ))

(defn combine-fn
  ([] ;; return initial "identity" value
     )
  ([result-1 result-2]
     ;; return merged or combined results
     ))
```

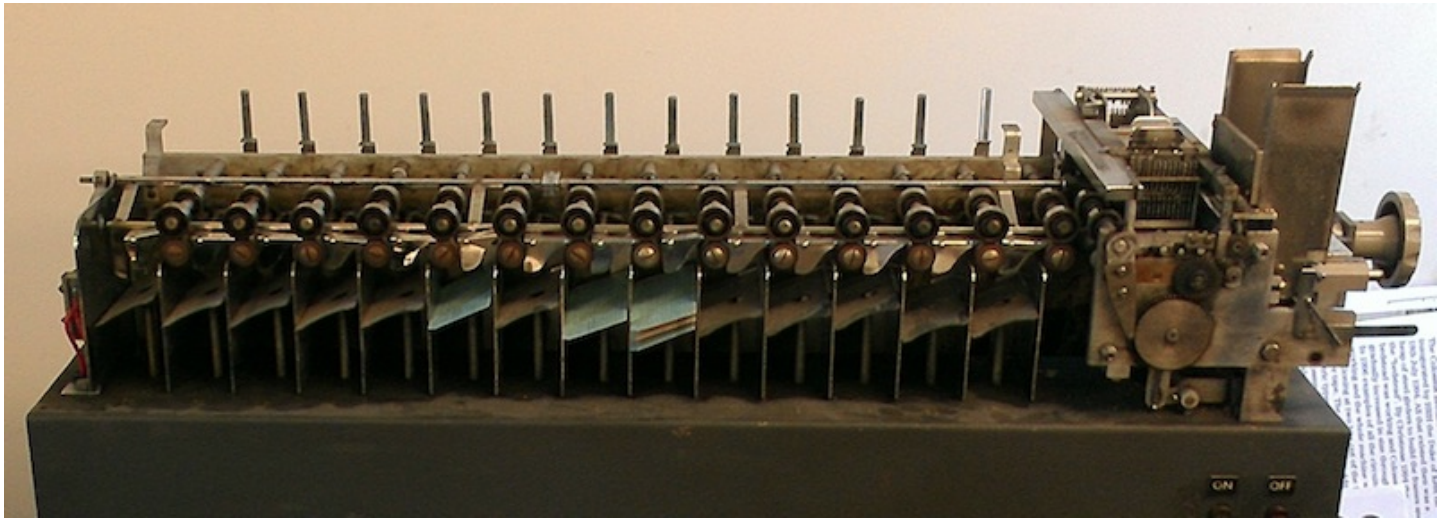
# Recursive Expansion Pattern

- Build up result out of primitives
- Build abstractions in layers
- Recurse until no more work left to do
- e.g. macroexpansion, Datomic transaction fns

# Recursive Expansion Pattern

```
(defn recursive-expansion [expander input]
  (let [output (expander input)]
    (if (= output input)
      input
      (recur expander output))))
```

# Flow Control Patterns



# Pipeline Pattern

- Process with many discrete steps
- Similar "shape" of data at each step
    - Usually a map or record
- Only one execution path

# Pipeline Pattern

```
(defn large-process [input]
  (-> input
      subprocess-a
      subprocess-b
      subprocess-c))

(defn subprocess-a [data]
  (let [{:keys [alpha beta]} data]
    (-> data
        (assoc :epsilon (compute-epsilon alpha))
        (update-in [:gamma] merge (compute-gamma beta)))))
```

# Wrapper Pattern

- Process with many discrete steps
- One main execution path
- Possible branch at each step

# Wrapper Pattern

```
(defn wrapper [f]
  (fn [input]
    ;; ... before ...
    (f input)
    ;; ... after ...
    ))

(def final-function
  (-> original-function wrapper-a wrapper-b wrapper-c))
```

# Token Pattern

- May need to cancel an operation
- Operation itself is not an identity

# Token Pattern

```
(defn begin [target & args]
  ;; ... begin operation or create state in target ...

  ;; Return a function:
  (fn []
    ;; ... cease operation or destroy state ...
    ))
```

# Token Pattern

Variation: caller supplies token value

```
(defn add-watch [reference key function]
  ;; attach a watcher to reference
  )

(defn remove-watch [reference key]
  ;; remove the watcher
  )
```

# Observer Pattern

Register an `observer` function with a stateful `container`

```
;; classic Observer:
(observer container)

(observer new-state)
(observer old-state new-state)

;; watches, plus token:
(observer container old-state new-state)

(observer old-state new-state delta)

;; useful for GUIs:
(observer old-state new-state event)
```

# Strategy Pattern

- Many processes with a similar structure
- Need extension points for future variations

# Strategy Pattern

```
(defprotocol Strategy
  (step-one [this operation])
  (step-two [this operation])
  (step-three [this operation]))

(defn process [strategy]
  (->> (initialize-operation)
       (step-one strategy)
       (step-two strategy)
       (step-three strategy)))
```

# Strategy Pattern

```clojure
(defmulti extensible-step (fn [strategy input] strategy))

(defn process [input strategy]
  ;; ... common behavior ...
  (extensible-step strategy input)
  ;; ... common behavior ...
  )

(defmethod extensible-step :strategy-one [_ input] ...)
(defmethod extensible-step :strategy-two [_ input] ...)
(defmethod extensible-step :strategy-three [_ input] ...)
```

# Strategy Pattern

```clojure
(def default-strategy
  {:step-one (fn [operation] ...)
   :step-two (fn [operation] ...)
   :step-three (fn [operation] ...)})

(def alternate-strategy
  (assoc default-strategy
    :step-two (fn [operation] ...)))

(defn process [strategy]
  (-> (initialize-operation)
      ((:step-one strategy))
      ((:step-two strategy))
      ((:step-three strategy))))
```


BBC Computer 32K
Acorn DFS
BASIC
>10 PRINT "HELLO, WORLD!"
>RUN
HELLO, WORLD!
>

# The End

- Me: @stuartsierra
- Us: clojure.com, thinkrelevance.com
- JavaOne: San Francisco, CA, September 30 - October 4
- The Science of Big Data: Philadelphia, PA, October 30
- Clojure Conj: Raleigh, NC, November 15-17
    - Clojure Training: November 12-14
- ClojureScript: Up and Running early release from O'Reilly