Exploring melody space with Clojure, Overtone, core.async and core.logic

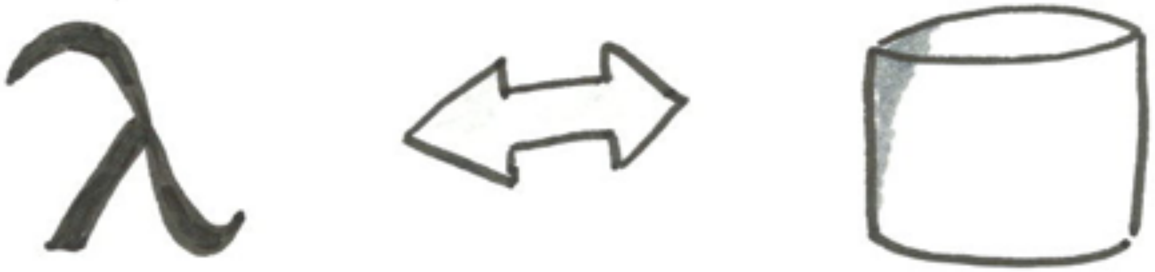
Thomas G. Kristensen
@tgkristensen

FP Days 2014

Traditional

LOOSE COUPLING

Functions from Data



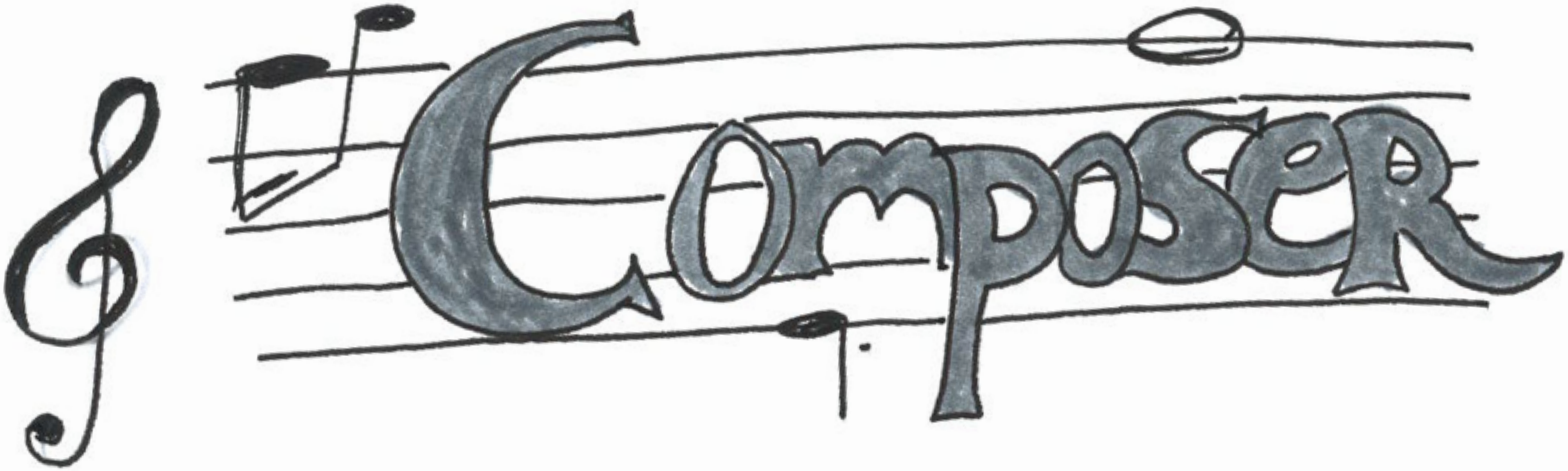
(in functional programming)

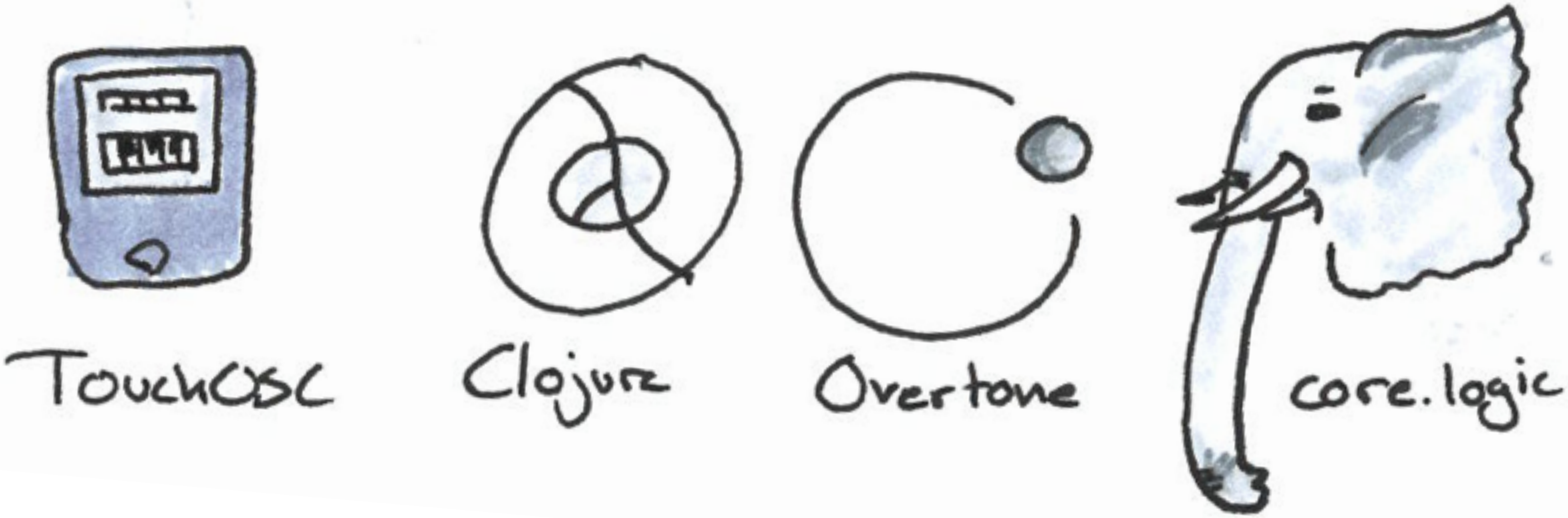
The Clojure ecosystem has tools for writing

truly decoupled programs

A system for exploring rules in Western music

Composer

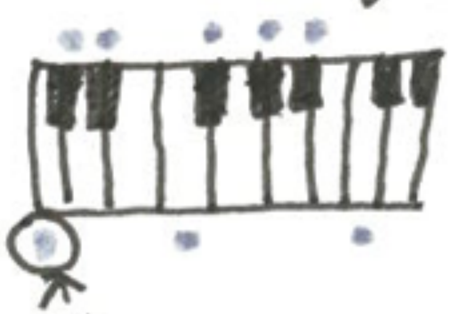




RULES

melodies are permutations of modes

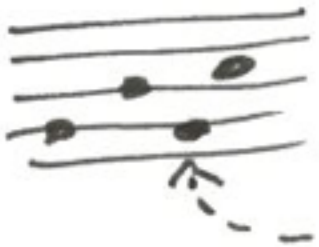
1 Mode



2 Tonic note

"where the mode starts"

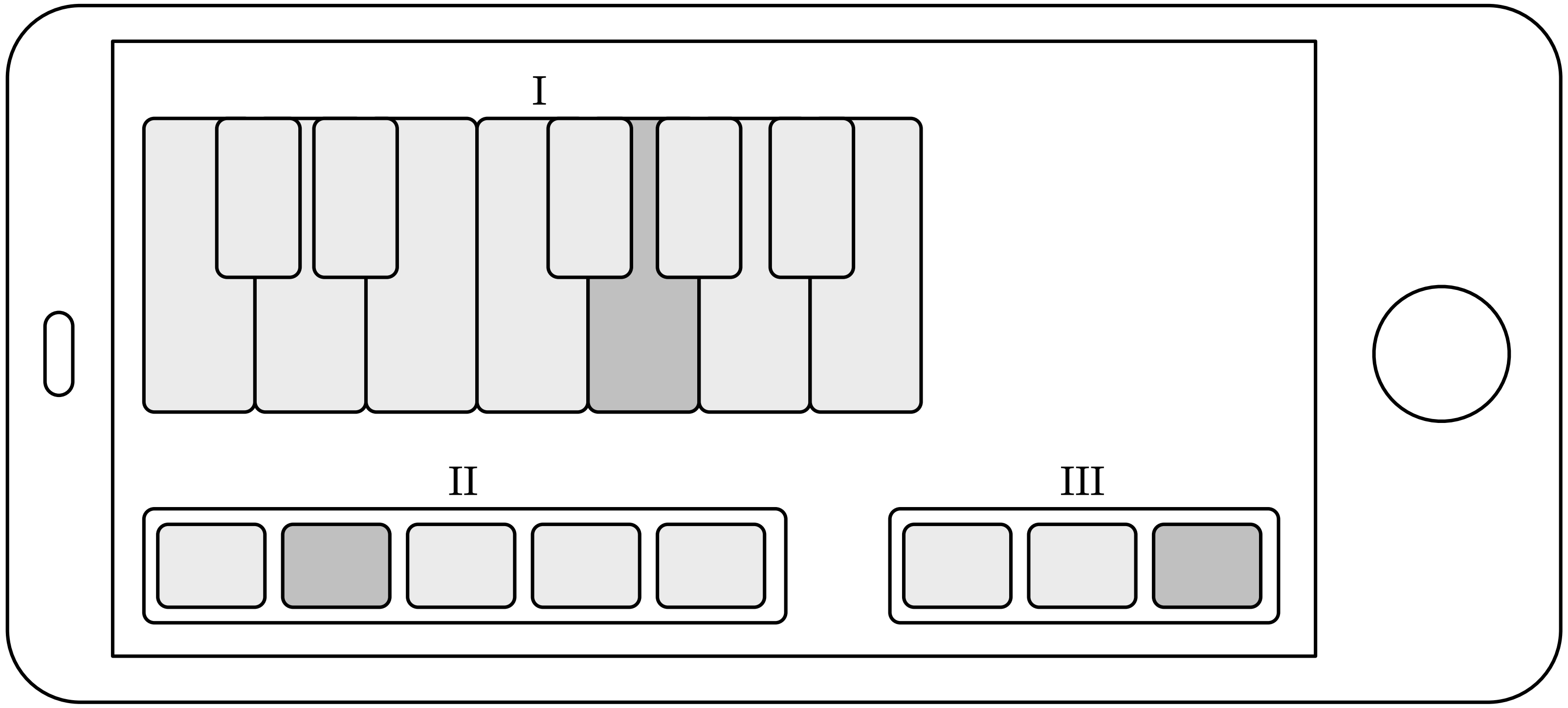
3 Cadence

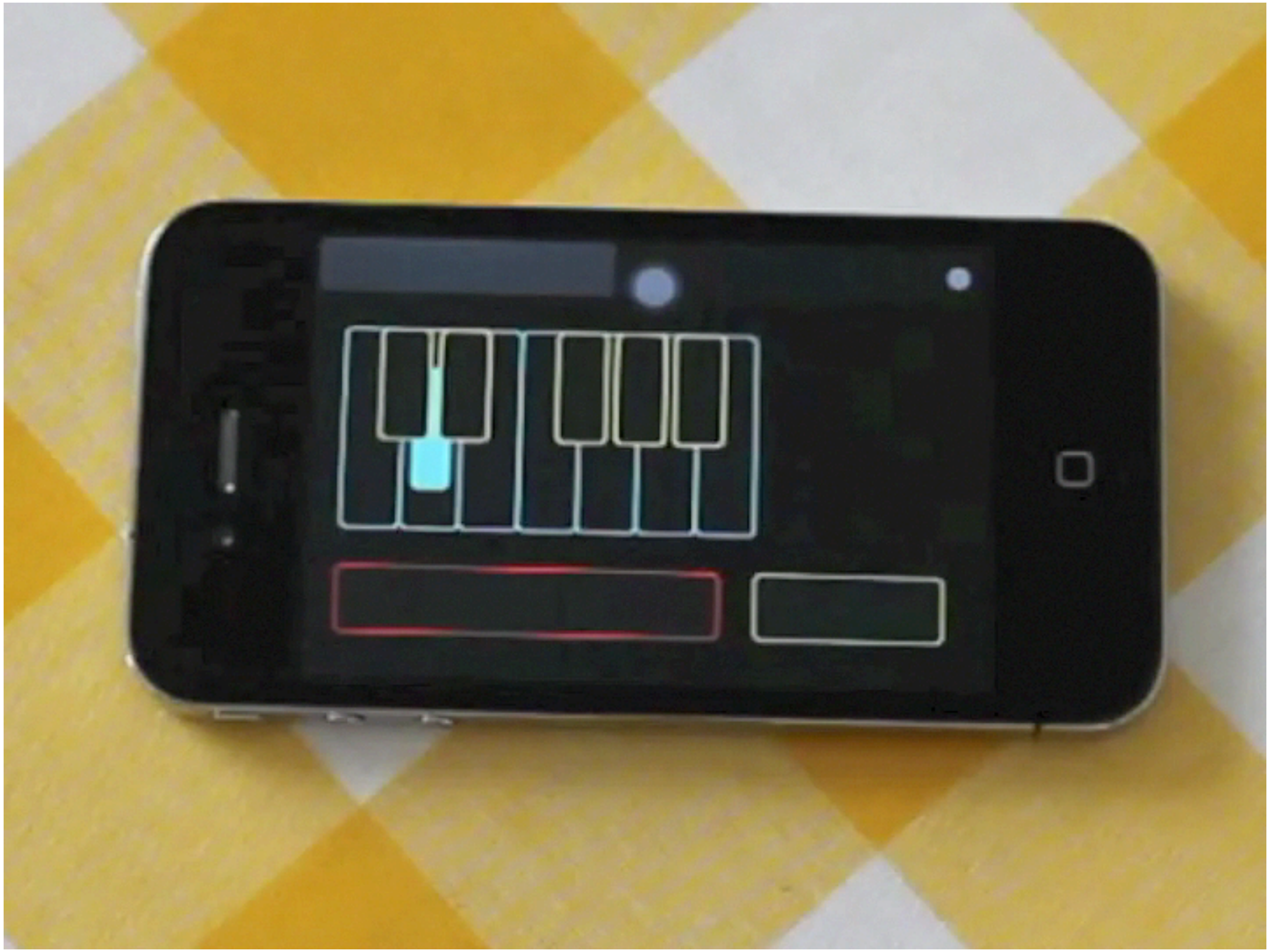


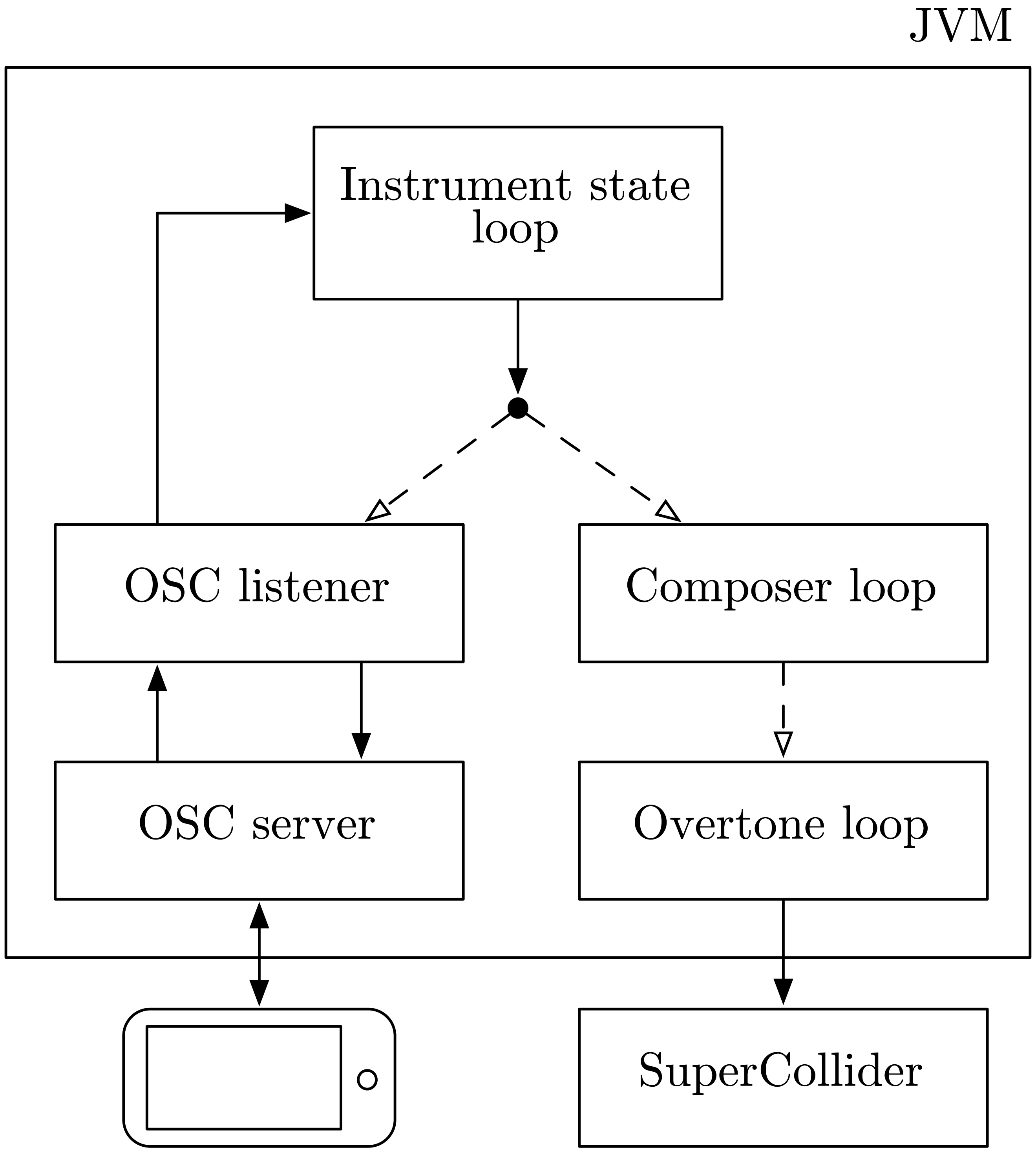
e.g.: second to last note in melody is fifth note of mode

osc
messages
OSC
Open
Sound
Control

```
{:path "/1/toggle8", :host "127.0.0.1"}
```



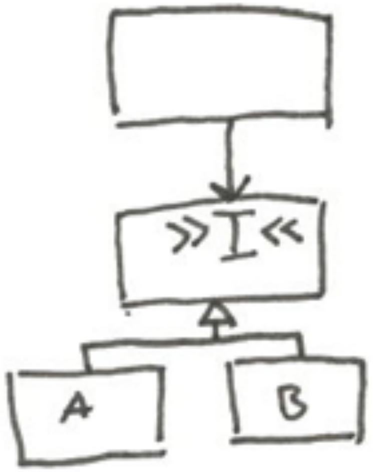



JVM
Instrument state loop
OSC listener
Composer loop
OSC server
Overtone loop
SuperCollider

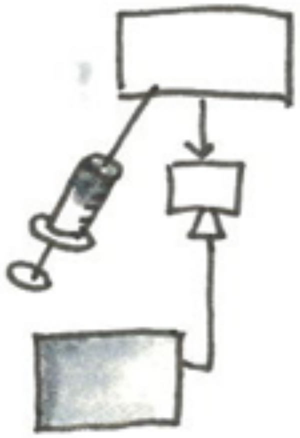
Coupling

coupling refers to the degree of direct knowledge that one component has of another.

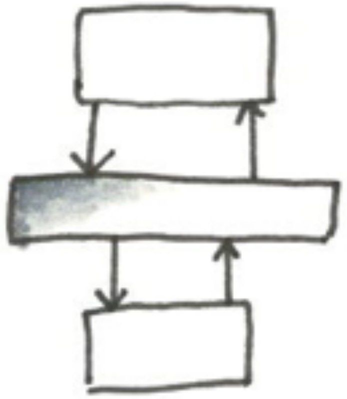
/Wikipedia



Program to interfaces

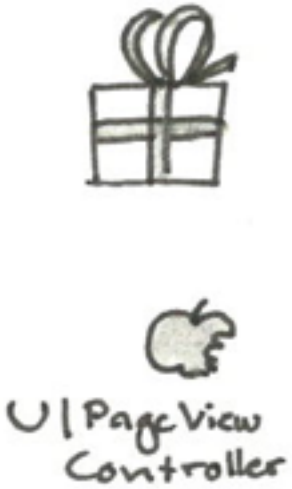


Dependency injection

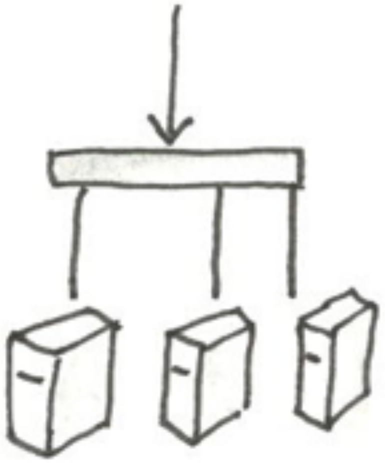


Message bus

Closure and blocks



Namespaces



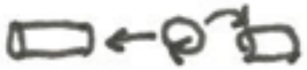
Load balancers

Communicating Sequential Processes

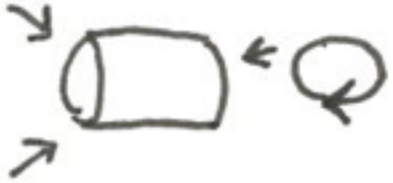
Motivation

I. Decouple what from who

II. Control capacity

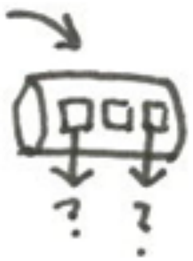
III. Avoid callbacks 

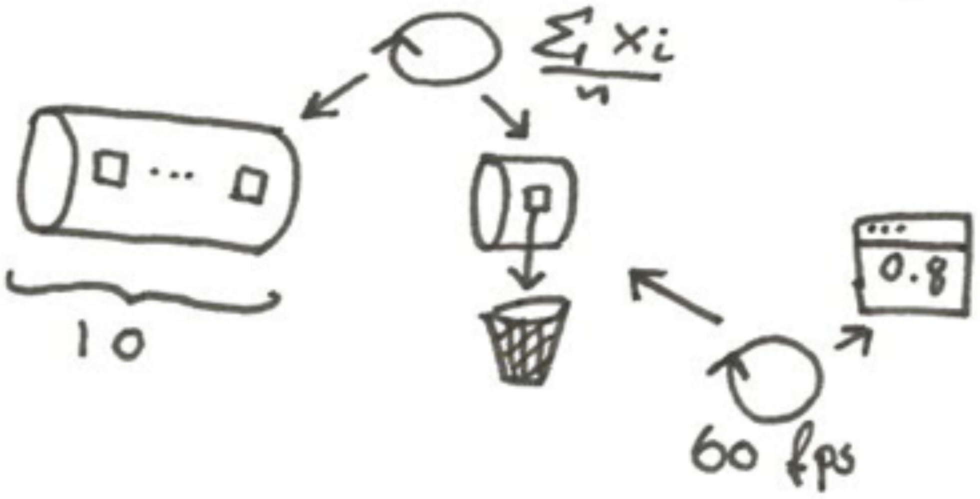
Basic idea



Start processes that consume from channels

Control size of channels and decide on overflow strategy

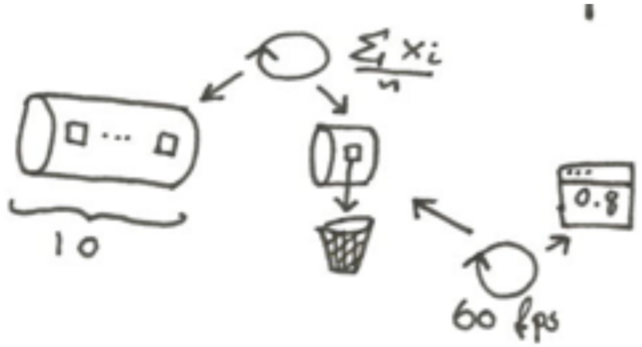


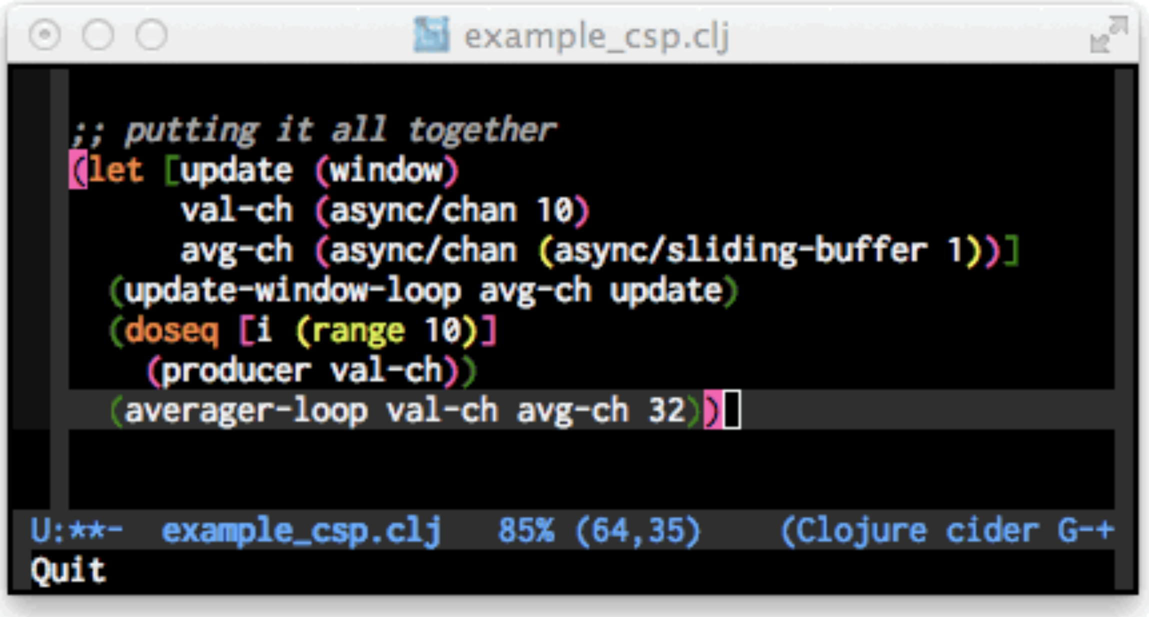


an example

```
(defn window
  "Creates Swing window and returns function for updating quoted
  average."
  []
  (let [frame (JFrame. "Average")
        label (JLabel. "-.-" SwingConstants/CENTER)]
    (.setFont label (Font. (.. label getFont getName) Font/PLAIN 24))
    (doto frame
      (.add label)
      (.setSize 100 100)
      (.setVisible true))
    (fn update [average]
      (.setText label (format "%.2f" (float average))))))

(defn update-window-loop
  "Consumes a new average from ch 60 times a second and calls
update."
  [ch update]
  (async/go-loop
   []
   (update (async/<! ch))
   (async/<! (async/timeout (/ 1000.0 60.0)))
   (recur)))

(def rand (java.util.Random.))

(defn producer
  "Puts pseudo random numbers on channel"
  [ch]
  (async/go-loop
   []
   (async/>! ch (+ (.nextFloat rand) (.nextFloat rand)))
   (async/<! (async/timeout 100))
   (recur)))
```

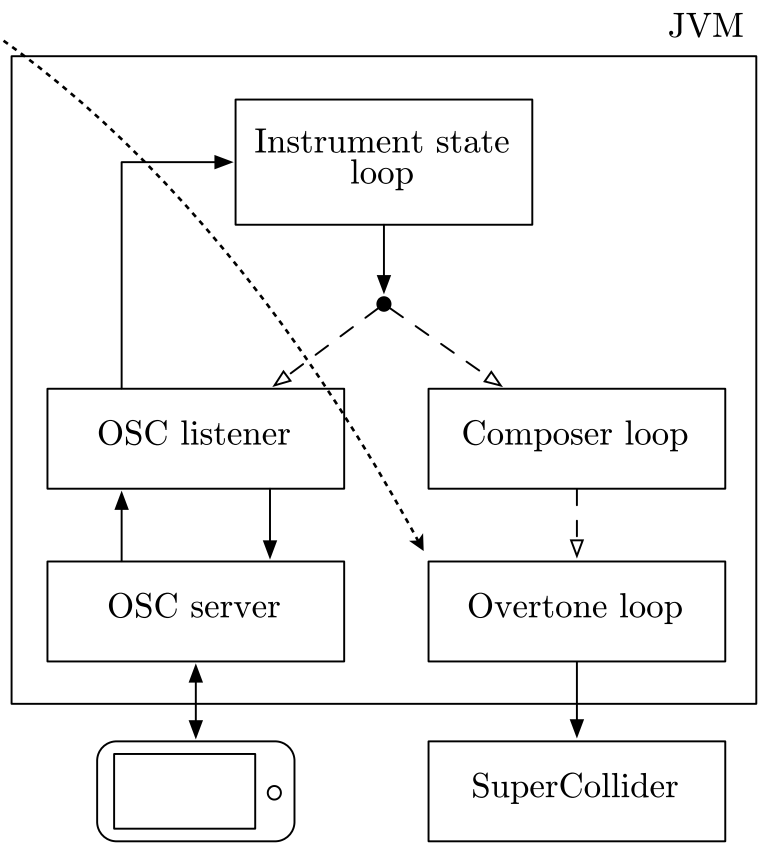
```
(defn update-window
  [val window n]
  (if (> (count window) n)
    (cons val (butlast window))
    (cons val window)))

(defn averager-loop
  [val-ch avg-ch n]
  (async/go-loop
   [window ()]
   (let [val (async/<! val-ch)
         new-window (update-window val window n)
         avg (/ (apply + new-window) (count new-window))]
     (async/>! avg-ch avg)
     (recur new-window))))
```





```
;; putting it all together
(let [update (window)
      val-ch (async/chan 10)
      avg-ch (async/chan (async/sliding-buffer 1))]
  (update-window-loop avg-ch update)
  (doseq [i (range 10)]
    (producer val-ch))
  (averager-loop val-ch avg-ch 32))

U:**-  example_csp.clj    85% (64,35)    (Clojure cider G-+
Quit
```

```
(defn overtone-loop
  "Starts an overtone server and listens for melodies on
  melody-ch."
  [melody-ch]
  (let [melody-atom (atom [])
        metro (metronome 100)]
    (chord-progression-atom metro (metro) melody-atom)
    (go
     (loop []
       (let [melody (<! melody-ch)]
         (if melody
           (do
             (metro :bpm (or (speed->bpm (:speed melody)) 100))
             (reset! melody-atom melody)
             (recur))
           (overtone/stop)))))))
```



Wiring as a Datastructure

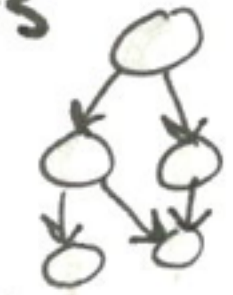
Motivation

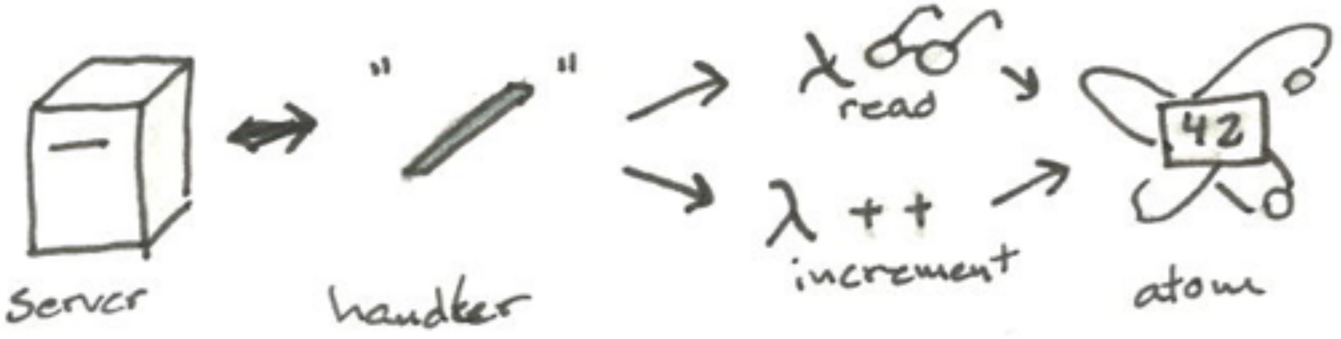
I. Centralise wiring

II. Make wiring declarative

Basic idea

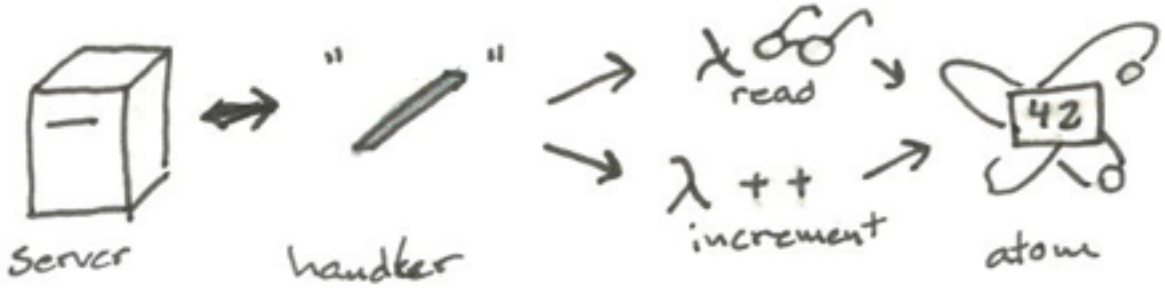
- Define relationships btw components as a graph

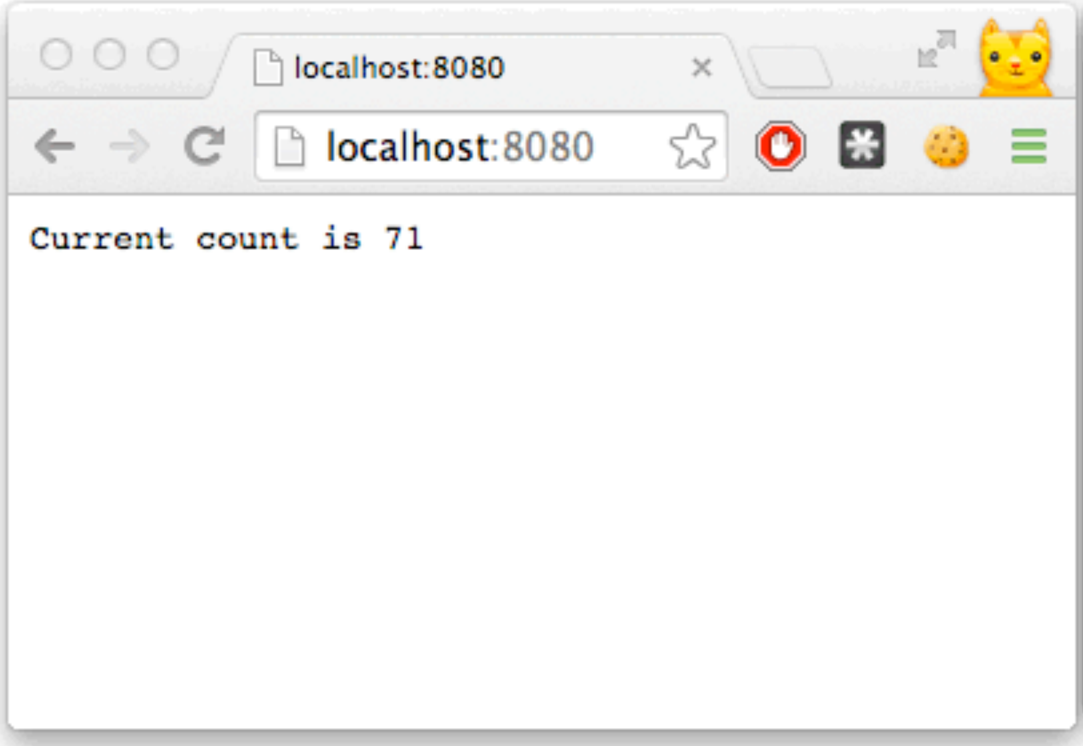




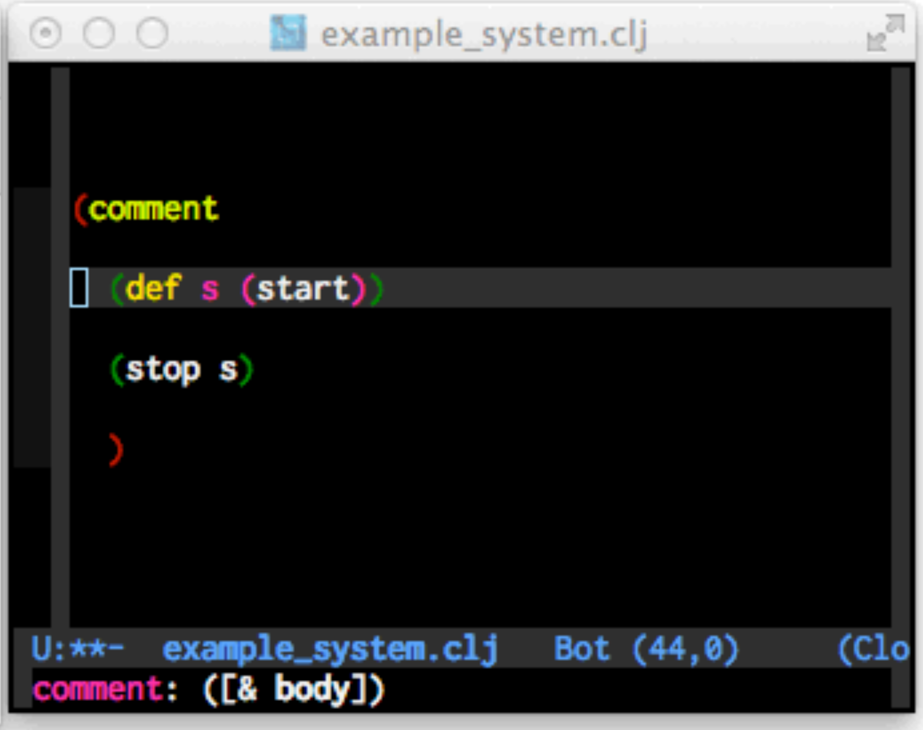
```
(defn build-handler
  [read increment]
  (fn [req]
    (when (= (:uri req) "/")
      (increment)
      {:status 200
       :body   (format "Current count is %d" (read))})))

(def system-flow
  (flow/flow

   :server    ([handler port]
                 (server/run-server handler {:port port}))

   :read      ([counter]
                 (fn [] (deref counter)))
   :increment ([counter]
                 (fn [] (swap! counter inc)))
   :counter   ([]
                 (atom 0))

   :handler   ([read increment]
                 (build-handler read increment))))
```

```
(defn start
  []
  (flow/run system-flow {:port 8080}))

(defn stop
  [system]
  (when-let [stop (:server system)]
    (stop)))
```





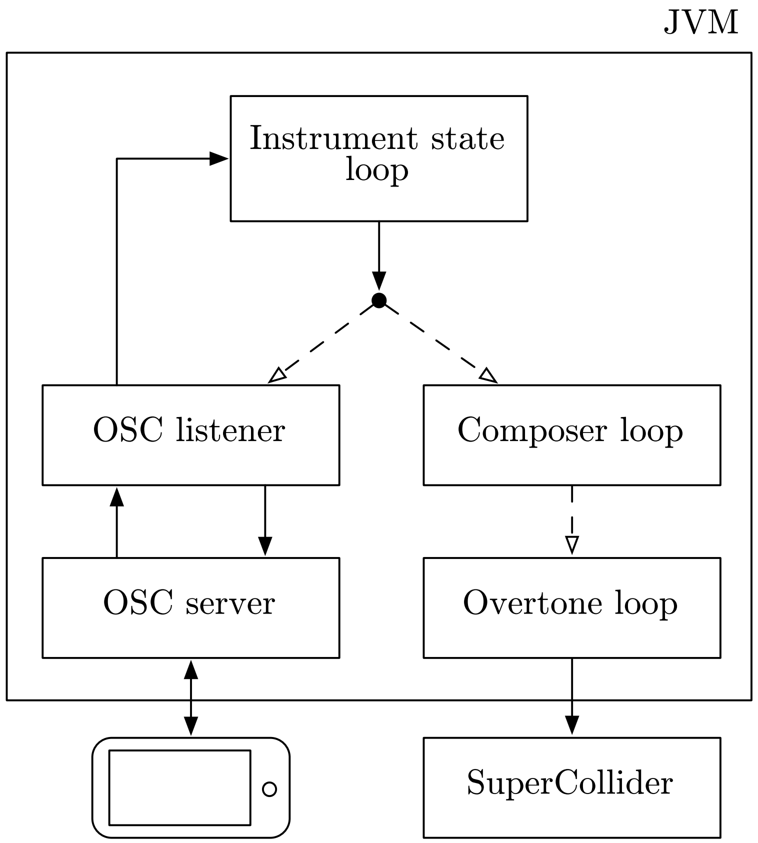
```
Current count is 71
```



```
(comment
  (def s (start))
  (stop s)
  )
```

```
U:**-  example_system.clj   Bot (44,0)     (Clo
comment: ([& body])
```

```
(def system
  (flow/flow

   :osc-listener ([osc-port osc-alias osc-out-ch osc-instrument-state-ch]
                    (osc/start osc-port osc-alias
                               osc-out-ch
                               osc-instrument-state-ch))

   :instrument-state-ch ([osc-instrument-state-ch composer-instrument-state-ch]
                           (broadcast osc-instrument-state-ch
                                      composer-instrument-state-ch))

   :instrument-state-loop ([osc-out-ch instrument-state-ch]
                             (instrument-state/instrument-state-loop
                              osc-out-ch instrument-state-ch))

   :composer-loop ([composer-instrument-state-ch melody-ch]
                     (composer/composer-loop
                      composer-instrument-state-ch
                      melody-ch))

   :overtone-loop ([melody-ch]
                     (overtone/overtone-loop melody-ch))))

(defn start
  [& [options]]
  (flow/run system
            (merge
             {:osc-port                      44100
              :osc-alias                     (str "composer" - (time-string))
              :osc-out-ch                    (chan 64)
              :osc-instrument-state-ch       (chan (sliding-buffer 1))
              :composer-instrument-state-ch  (chan (sliding-buffer 1))
              :melody-ch                     (chan (sliding-buffer 1))}
             options)))
```



Logic Programming

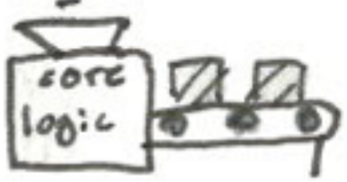
Motivation

I. The act of searching is different from defining what to search for.

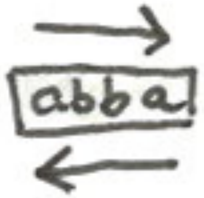
II. Decouple what from how.

Basic idea

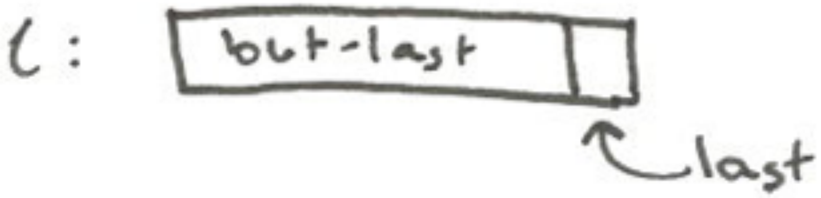
- Describe complex constraints using simple constraints.
- Combine constraints to form logic program.
- Feed logic program through logic engine.



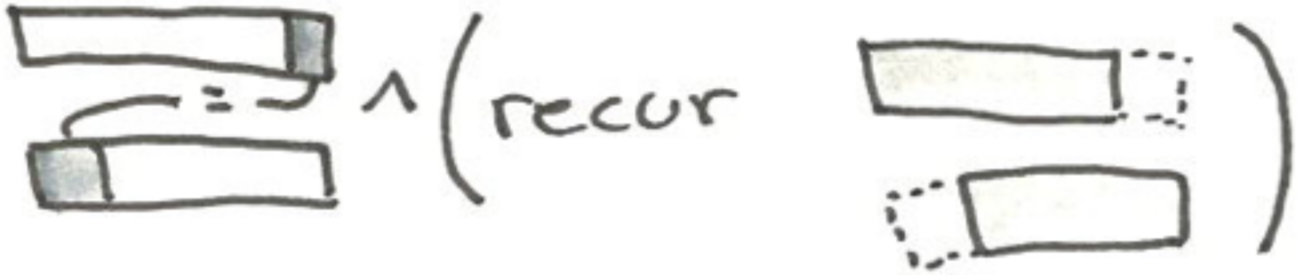
Our example? Palindromes!



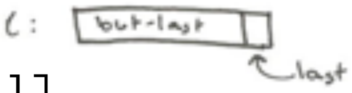
"but-lasto"



"reverso"



```
(defn but-lasto
  [but-last last l]
  (all
   (appendo but-last [last] l)))
```



```
(defne reverseo
  [s1 s2]
  ([() ()])
  ([[e] [e]])
  ([s1 s2]
     (fresh [a b c d]
            (conso a b s1)
            (but-lasto c d s2)
            (== a d)
            (reverseo b c))))
```

```
(defn palindromo
  [s]
  (all (reverseo s s)))
```

```
(run* [q]
      (palindromo [1 2 1]))
;; => (_0 _0)

(run* [q]
      (palindromo [1 2]))
;; => ()

(run 10 [q]
     (palindromo q))
;; => (()
;;     [_0] (_0)
;;     (_0 _0) (_0 _0)
;;     (_0 _1 _0) (_0 _1 _0)
;;     (_0 _1 _1 _0) (_0 _1 _1 _0)
;;     (_0 _1 _2 _1 _0))

(run 10 [q]
     (membero 'a q)
     (membero 'b q)
     (palindromo q))
;; => ((a b a) (a b a)
;;     (a b b a) (a b b a) (a b b a)
;;     (b a b) (a b _0 b a)
;;     (a b b a) (a b _0 b a) (b a b))

(run 10 [q]
     (permuteo q ['a 'b 'a 'b 'c 'c 'a])
     (palindromo q))
;; => ((a c b a b c a) (a c b a b c a) (a b c a c b a) (a b c a c b a)
;;     (c b a a a b c) (c b a a a b c) (c a b a b a c) (c a b a b a c)
;;     (b a c a c a b) (b a c a c a b))
```

```
(defn scale-from-tones [tone-types]
  (take 25
        (->> tone-types
             (map {:semitone [1]
                   :tone [0 1]
                   :minor-third [0 0 1]})
             flatten
             butlast
             (cons 1)
             cycle)))

(def major-scale
  (scale-from-tones
   [:tone :tone :semitone :tone :tone :tone :semitone]))
(def harmonic-minor-scale
  (scale-from-tones
   [:tone :semitone :tone :tone :semitone :minor-third :semitone]))
(def natural-minor-scale
  (scale-from-tones
   [:tone :semitone :tone :tone :semitone :tone :tone]))
(def locrian-mode
  (scale-from-tones
   [:semitone :tone :tone :semitone :tone :tone :tone]))
(def mixolydian-mode
  (scale-from-tones
   [:tone :tone :semitone :tone :tone :semitone :tone]))

(def scale-modes
  [[:major-scale          major-scale]
   [:harmonic-minor-scale harmonic-minor-scale]
   [:natural-minor-scale  natural-minor-scale]
   [:locrian-mode         locrian-mode]
   [:mixolydian-mode      mixolydian-mode]])
```

```
(defn key-restriction
  [instrument-state s1]
  (if-let [key (:key instrument-state)]
    (all (== key s1))
    succeed))
```

```
(db-rel semitone note-1 note-2)

(def keys-from-c
  [:C3 :C#3 :D3 :D#3 :E3 :F3 :F#3 :G3 :G#3 :A3 :A#3 :B3
   :C4 :C#4 :D4 :D#4 :E4 :F4 :F#4 :G4 :G#4 :A4 :A#4 :B4
   :C5])

(def semitone-facts
  (reduce
   (fn [db [note-1 note-2]]
     (db-fact db semitone note-1 note-2))
   empty-db
   (partition 2 1 keys-from-c)))

(defne scaleo [base-note scale notes]
  ([note [1 . scale-rest] [note . ()]])
  ([note [1 . scale-rest] [note . notes-rest]]
     (fresh [next-note]
            (semitone note next-note)
            (scaleo next-note scale-rest notes-rest)))
  ([note [0 . scale-rest] notes]
     (fresh [next-note]
            (semitone note next-note)
            (scaleo next-note scale-rest notes))))

(defn scale-restriction
  [instrument-state scale-type]
  (if (:scale instrument-state)
    (all (membero [(:scale instrument-state) scale-type]
scale-modes))
    succeed))
```

```
(defn cadence-restriction
  [instrument-state m7 s2 s4 s5]
  (case (:cadence instrument-state)
    :perfect   (all (== m7 s5))
    :plagal    (all (== m7 s4))
    :just-nice (all (== m7 s2))
    nil        succeed))
```

```
;; . . .

(defne scaleo [base-note scale notes]
  ([note [1 . scale-rest] [note . ()]])
  ([note [1 . scale-rest] [note . notes-rest]]
     (fresh [next-note]
            (semitone note next-note)
            (scaleo next-note scale-rest notes-rest)))
  ([note [0 . scale-rest] notes]
     (fresh [next-note]
            (semitone note next-note)
            (scaleo next-note scale-rest notes))))

;; . . .
```

```
(run* [notes]
      (scaleo :C3 major-scale notes)
      (counto notes 8))
;; => ([:C3 :D3 :E3 :F3 :G3 :A3 :B3 :C4])

(run 3 [m1 m2 m3 m4 m5 m6 m7 m8]
     (fresh [n1 n2 n3 n4 n5 n6 n7 n8]
            (scaleo :C3 major-scale
                    [n1 n2 n3 n4 n5 n6 n7 n8])
            (permuteo [m1 m2 m3 m4 m5 m6 m7 m8]
                      [n1 n2 n3 n4 n5 n6 n7 n8])
            (== m1 :C3)
            (== m8 :C4)))
;; => ([:C3 :D3 :E3 :F3 :G3 :A3 :B3 :C4]
;;     [:C3 :E3 :D3 :F3 :G3 :A3 :B3 :C4]
;;     [:C3 :F3 :D3 :E3 :G3 :A3 :B3 :C4])

(run* [tonic-note pattern]
      (scaleo tonic-note pattern
              [:C3 :D3 :E3 :F3 :G3 :A3 :B3 :C4]))
;; => ([:C3 (1 0 1 0 1 1 0 1 0 1 0 1 1 . _0)])
```

The Clojure ecosystem has tools for writing *truly* decoupled programs

Composer embodies three of these decoupling strategies, allowing it

- to have a small codebase
- to be easy to understand
- to be responsive

Thank you