evident

late 14c., from O.Fr. evident and directly from L. evidentem (nom. evidens) "perceptible, clear, obvious, apparent" from ex- "fully, out of" (see ex-) + videntem (nom. videns), prp. of videre "to see" (see vision).

http://www.etymonline.com/index.php?term=evident

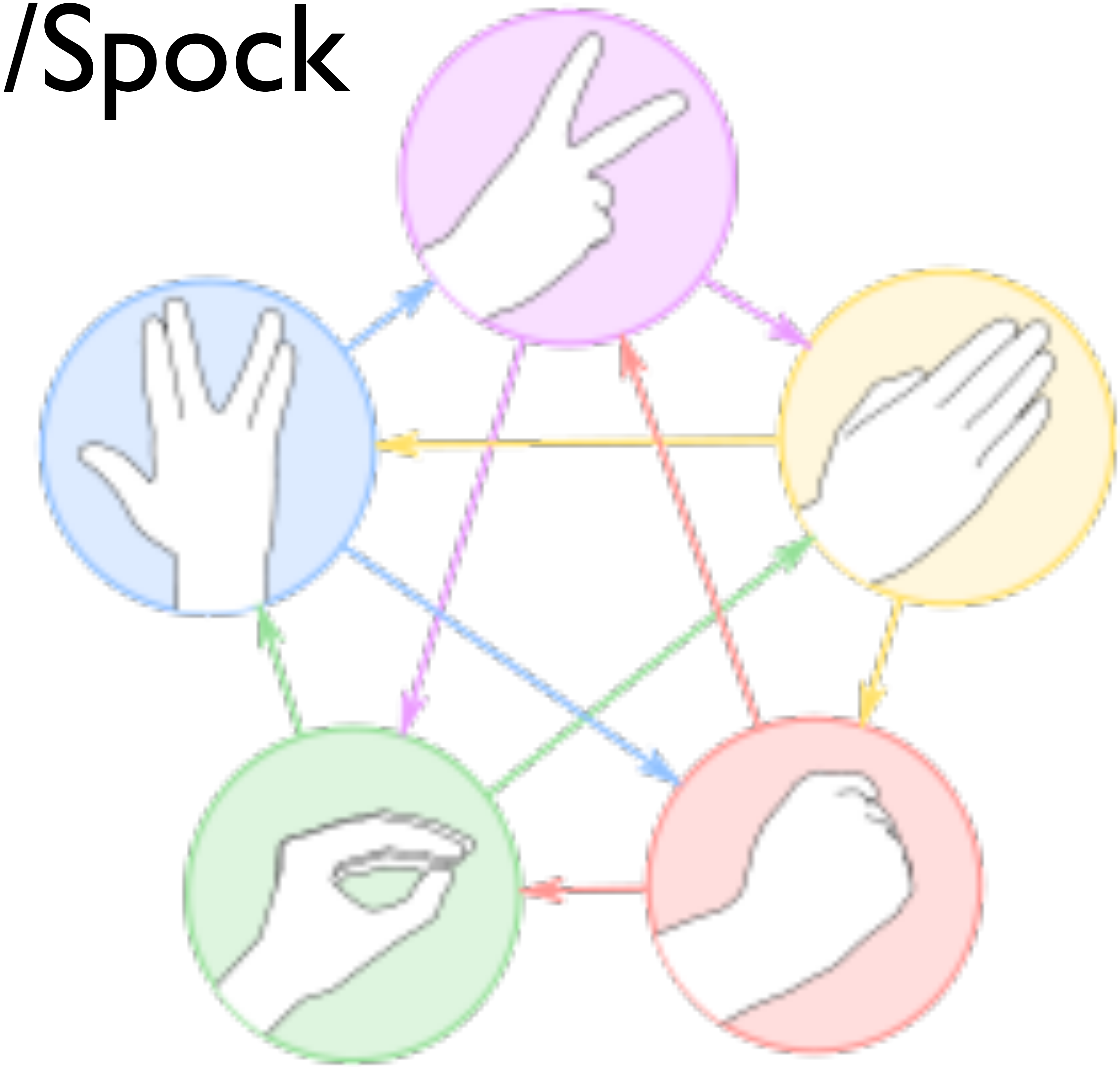
Datomic

declarative programming

In computer science, **declarative programming** is a programming paradigm that expresses the logic of a computation without describing its control flow.[1] Many languages applying this style attempt to minimize or eliminate side effects by describing *what* the program should accomplish, rather than describing *how* to go about accomplishing it.

http://en.wikipedia.org/wiki/Declarative_programming

rock/paper/scissors/ lizard/Spock



http://en.wikipedia.org/wiki/Rock-paper-scissors-lizard-Spock

declarative

```
(def dominates
  {:paper :rock
   :rock :scissors
   :scissors :paper})
```

declarative, functional

```
(def dominates
  {:paper :rock
   :rock :scissors
   :scissors :paper})
```

```
(defn winner [play-1 play-2]
  (cond
   (= play-1 play-2) nil
   (= (dominates play-1) play-2) play-1
   :else play-2))
```

so everything should be declarative, right?

some reasons not to be declarative

- functional requirement
- non-functional requirement
- constraint

abstraction

ideal number of methods

0

too big

```
public interface Map<K, V>   {
    int size();
    boolean isEmpty();
    boolean containsKey(java.lang.Object o);
    boolean containsValue(java.lang.Object o);
    V get(java.lang.Object o);
    V put(K k, V v);
    V remove(java.lang.Object o);
    void putAll(java.util.Map<? extends K,? extends V> map);
    void clear();
    java.util.Set<K> keySet();
    java.util.Collection<V> values();
    java.util.Set<java.util.Map.Entry<K,V>> entrySet();
    boolean equals(java.lang.Object o);
    int hashCode();
}
```

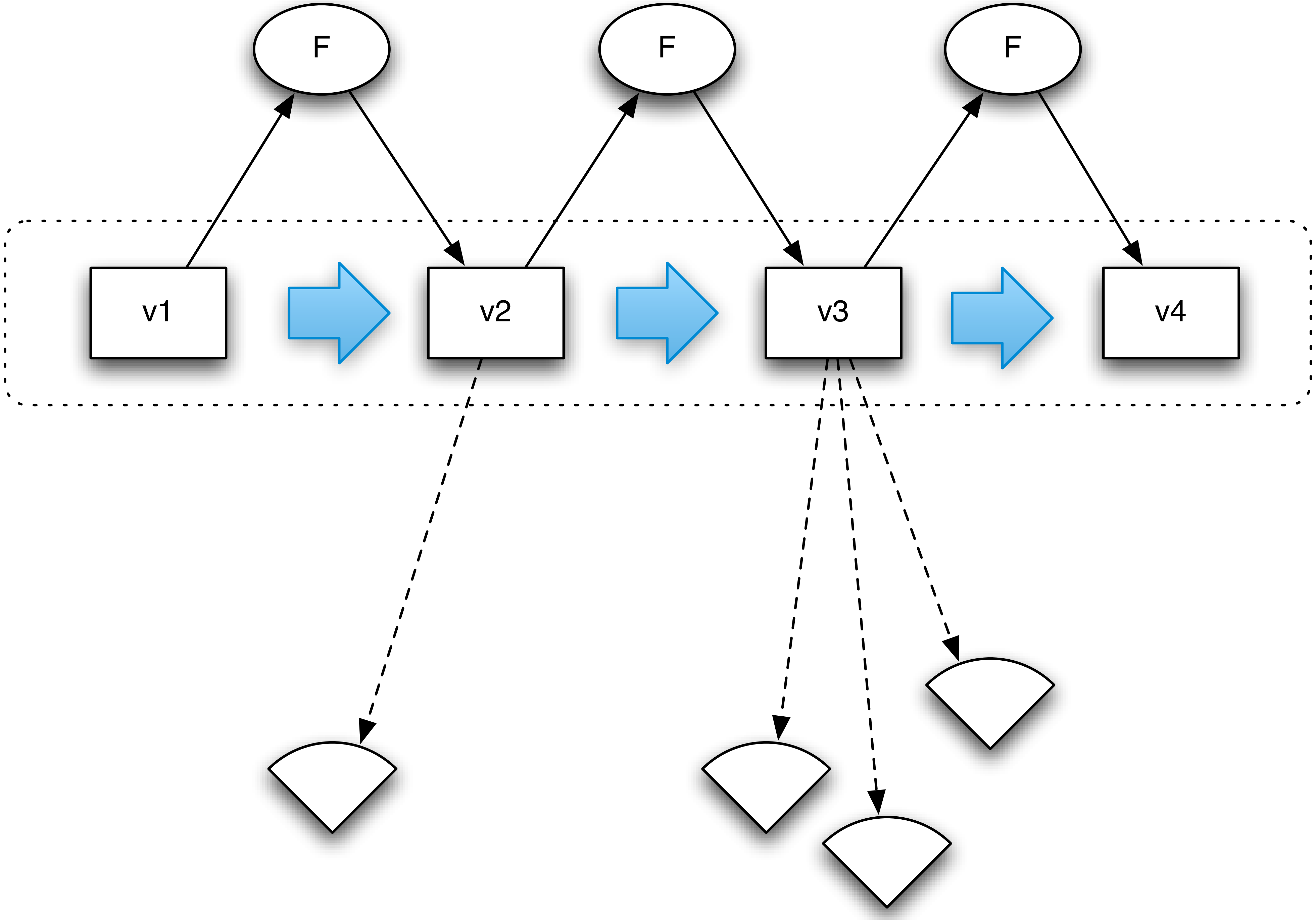
concretion

```java
public class ListTablesRequest {
    public ListTablesRequest();
    public java.lang.String getExclusiveStartTableName();
    public void setExclusiveStartTableName
           (java.lang.String exclusiveStartTableName);
    public ListTablesRequest withExclusiveStartTableName
           (String exclusiveStartTableName);
    public java.lang.Integer getLimit();
    public void setLimit(java.lang.Integer limit);
    public ListTablesRequest withLimit(Integer limit);
    public java.lang.String toString();
    public int hashCode();
    public boolean equals(Object obj);
}
```

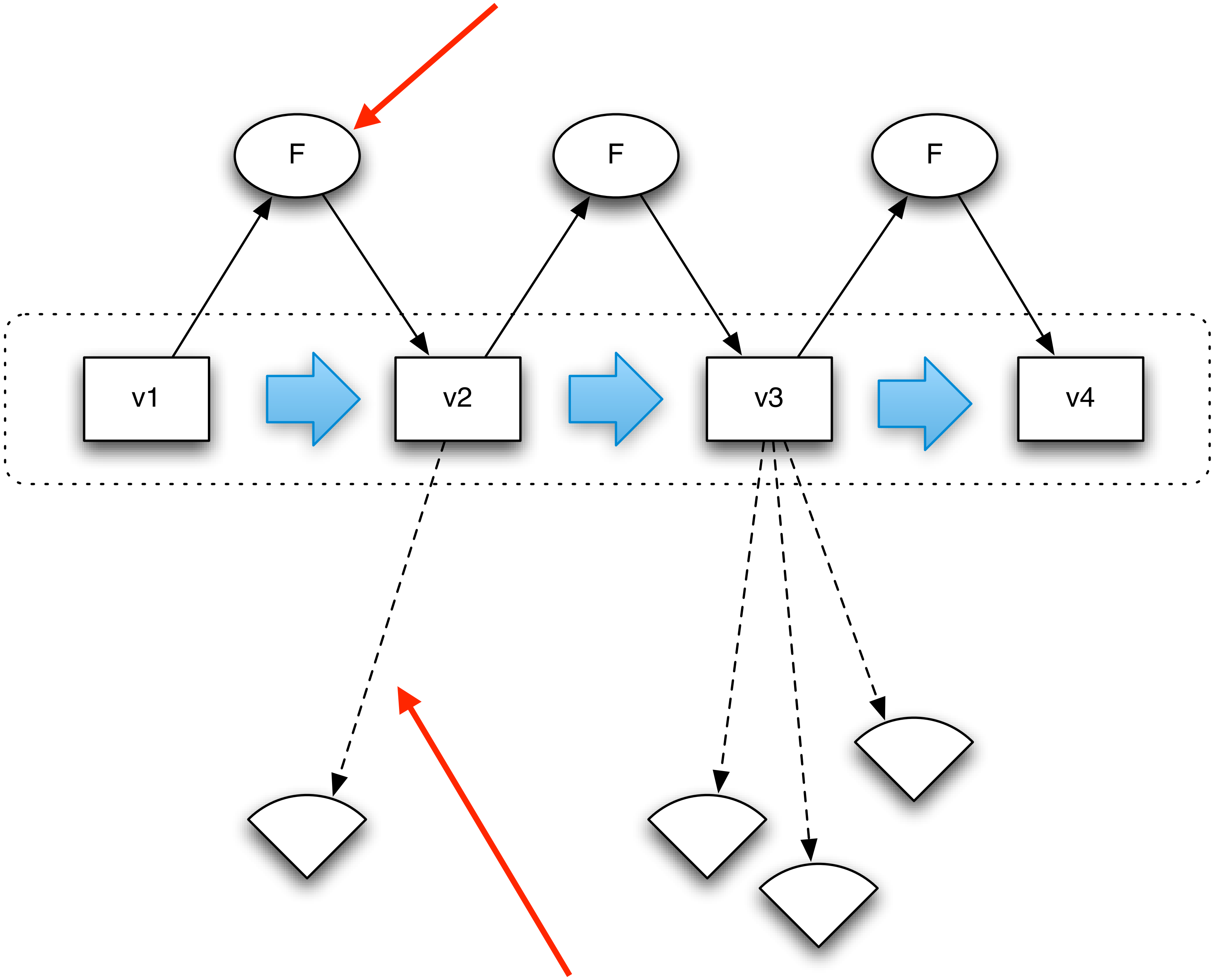
Abstractions may be formed by *reducing* the information content of a concept or an observable phenomenon...

http://en.wikipedia.org/wiki/Abstraction

epochal time model



abstraction here reduces
the information I must provide



abstraction here reduces
the information I can get

abstractions in Datomic

- about 50 protocols / interfaces
- (inter)action
  - between services
  - between components
- perception
  - some new *(generic!)* data structures
  - no concretions

perception example

```
public interface Entity {
    Object get(Object key);
    Set keySet();
}
```

action example

```
public Future<Boolean>
transact(List txData);

public Future<Boolean>
transactAsync(List txData);
```

number of db update methods in Datomic

2

abstraction guidelines

- fine grained
- separate perception and action
- perception is generic

values

queries are data

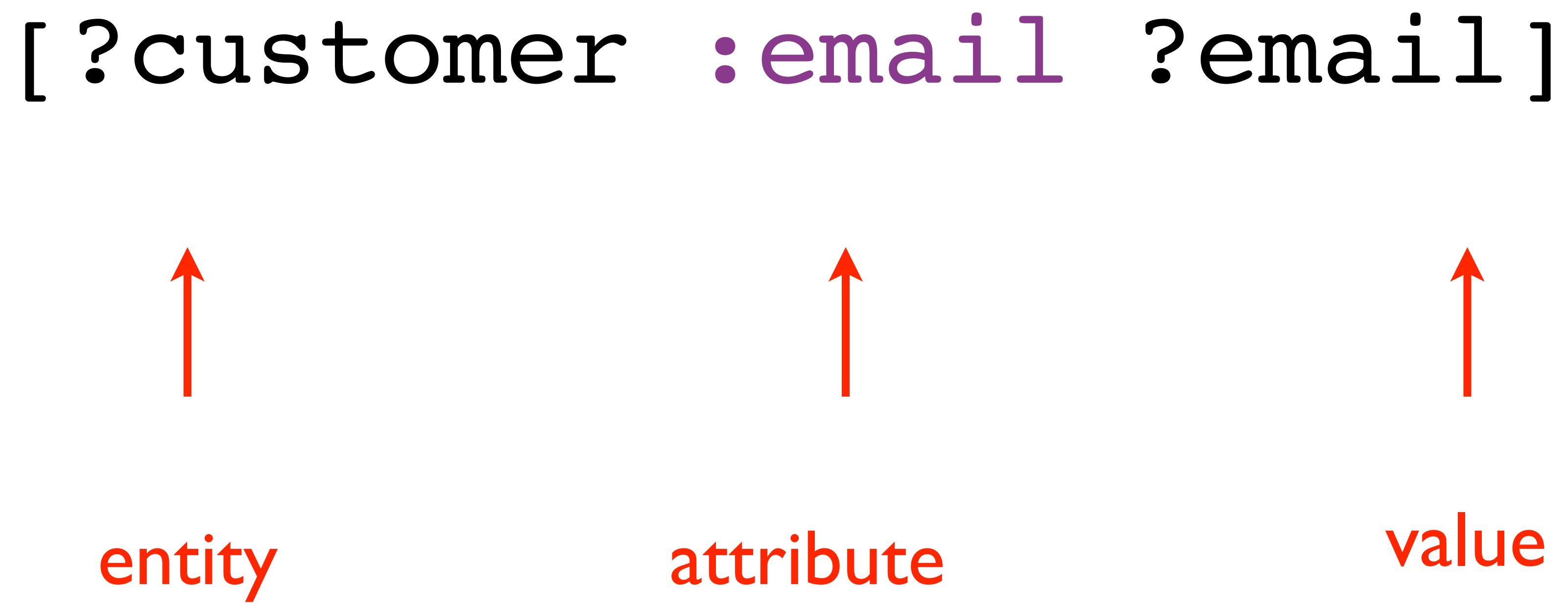
data pattern

Constrains the results returned,
binds variables

```
[?customer :email ?email]
```

data pattern

Constrains the results returned,
binds variables



```
[?customer :email ?email]
```

entity

attribute

value

data pattern

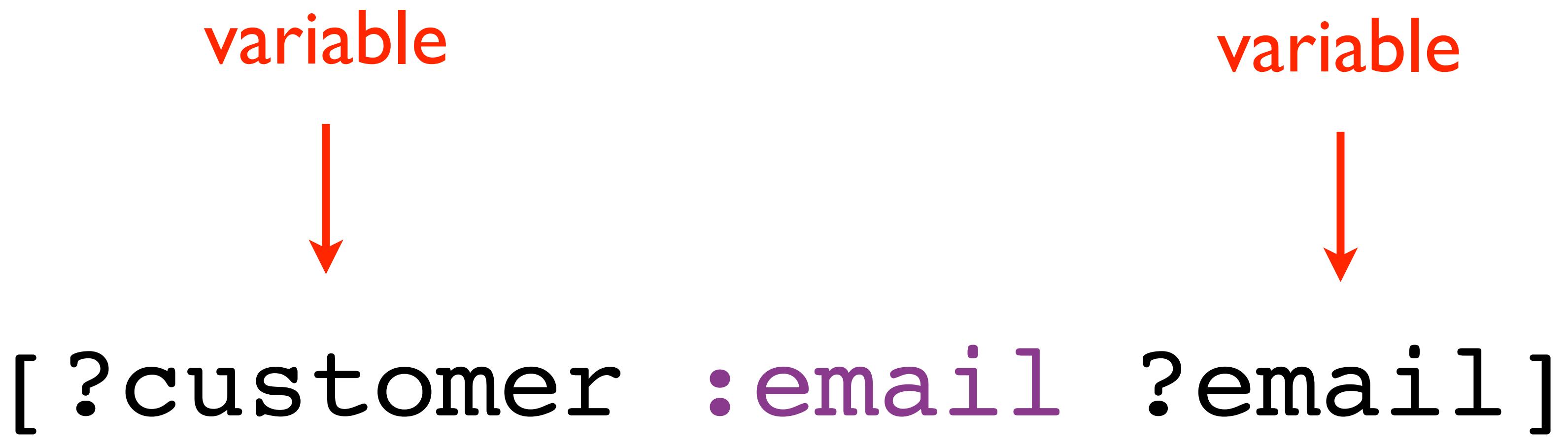
Constrains the results returned,
binds variables

constant

```
[?customer :email ?email]
```

data pattern

Constrains the results returned,
binds variables



variable

variable

```
[?customer :email ?email]
```

constants anywhere

“Find a particular customer’s email”

```
[42 :email ?email]
```

variables anywhere

“What other attributes does customer 42 have?

```
[42 ?attribute]
```

variables anywhere

“What other attributes and values does customer 42 have?

```
[42 ?attribute ?value]
```

find clause

variable to
return

```
[:find ?customer
 :where [?customer :email]]
```

implicit join

“Find all the customers who have placed orders.”

```
[:find ?customer
 :where [?customer :email]
        [?customer :orders]]
```

declarative < evident

declarative	**evident**
tables, documents	datoms
ORM	maps, entities
ambient db	explicit db

API

```
(use '[datomic.api :only (q) :as d])


(q '[:find ?customer
     :where [?customer :id]
            [?customer :orders]]
     db)
```

q

```
(use '[datomic.api :only (q) :as d])
```

```
(q '[:find ?customer
     :where [?customer :id]
            [?customer :orders]]
     db)
```

query

```
(q '[:find ?customer
     :where [?customer :id]
            [?customer :orders]]
     db)
```

input(s)

```
(q '[:find ?customer
     :where [?customer :id]
            [?customer :orders]]
     db)
```

in clause

Names inputs so you can refer to them elsewhere in the query

```
:in $database ?email
```

parameterized query

“Find a customer by email.”

```
(q '[:find ?customer
     :in $database ?email
     :where [$database ?customer :email ?email]]
     db
     "jdoe@example.com")
```

first input

“Find a customer by email.”

```
(q '[:find ?customer
     :in $database ?email
     :where [$database ?customer :email ?email]]
     db
     "jdoe@example.com")
```

second input

“Find a customer by email.”

```
(q '[:find ?customer
     :in $database ?email
     :where [$database ?customer :email ?email]]
     db
     "jdoe@example.com")
```

verbose?

“Find a customer by email.”

```
(q '[:find ?customer
     :in $database ?email
     :where [$database ?customer :email ?email]]
     db
     "jdoe@example.com")
```

shortest name possible

“Find a customer by email.”

```
(q '[:find ?customer
     :in $ ?email
     :where [$ ?customer :email ?email]]
     db
     "jdoe@example.com")
```

elide $ in where

“Find a customer by email.”

```
(q '[:find ?customer
     :in $ ?email
     :where [ ?customer :email ?email]]
     db
     "jdoe@example.com")
```

no need to
specify $

predicates

Functional constraints that can appear in a :where clause

```
[(< 50 ?price)]
```

adding a predicate

“Find the expensive items”

```
[:find ?item
 :where [?item :item/price ?price]
        [(< 50 ?price)]]
```

functions

Take bound variables as inputs
and bind variables with output

```
[(shipping ?zip ?weight) ?cost]
```

function args

```
[(shipping ?zip ?weight) ?cost]
```

function arguments

function returns

```
[(shipping ?zip ?weight) ?cost]
```

bind return values

calling a function

“Find me the customer/product combinations where the shipping cost dominates the product cost.”

```
[:find ?customer ?product
 :where [?customer :shipAddress ?addr]
        [?addr :zip ?zip]
        [?product :product/weight ?weight]
        [?product :product/price ?price]
        [(Shipping/estimate ?zip ?weight) ?shipCost]
        [(<= ?price ?shipCost)]]
```

arbitrary functions

*Functions can be plain
JVM code.*

```
public class Shipping {
  public static BigDecimal
  estimate(String zip1, int pounds);
}
```

find people with interests

```
(q '[:find ?op ?e ?a ?v
     :in $ ?renamings
     :where
     [?person :customer/interests]
     [(.entity $ ?person) ?entity]
     [(datomize ?entity ?renamings)
      [[?op ?e ?a ?v]]]]
   (db customer-conn)
   renamings)
```

as entities

```
(q '[:find ?op ?e ?a ?v
     :in $ ?renamings
     :where
     [?person :customer/interests]
     [(.entity $ ?person) ?entity]
     [(datomize ?entity ?renamings)
      [[?op ?e ?a ?v]]]]
   (db customer-conn)
   renamings)
```

and make ... datoms?

```
(q '[:find ?op ?e ?a ?v
     :in $ ?renamings
     :where
     [?person :customer/interests]
     [(.entity $ ?person) ?entity]
     [(datomize ?entity ?renamings)
      [[?op ?e ?a ?v]]]]
   (db customer-conn)
   renamings)
```

ETL job

```
(->> (q '[:find ?op ?e ?a ?v
          :in $ ?renamings
          :where
          [?person :customer/interests]
          [(.entity $ ?person) ?entity]
          [(datomize ?entity ?renamings)
           [[?op ?e ?a ?v]]]]
      (db customer-conn)
      renamings)
     seq
     (d/transact hobbies-conn))
```

->>

the ETL macro

schema is data

attribute schema

```
{:db/id #db/id[:db.part/db]
 :db/ident :part/name
 :db/valueType :db.type/string
 :db/cardinality :db.cardinality/one
 :db/fulltext true
 :db/doc "The name of the part"
 :db.install/_attribute :db.part/db}
```

why not this?

```
(create-eav :part/business-key
            :db.type/string
            :db.cardinality/one
            "The name of the part"
            :db.part/db)
```

value propositions

- powerful extant API
- make from any language
- read without evaluation
- names, not positions
- extend easily
- build from programs

navigation is data

Entity revisited

```
public interface Entity {
    Object get(Object key);
    Set keySet();
    // evil enhancement
    Object reverseGet(Object key);
}
```

Jack’s town

```
(get-in jack [:town])
```

who else is in Jack’s town?

```
(get-in jack [:town :_town])
```

number of methods in Datomic’s “graph navigation API”

0

evident code is not enough

- specifications
- doc strings
- drawings (graffle)
- outlines (org, Confluence)
- tests (???)

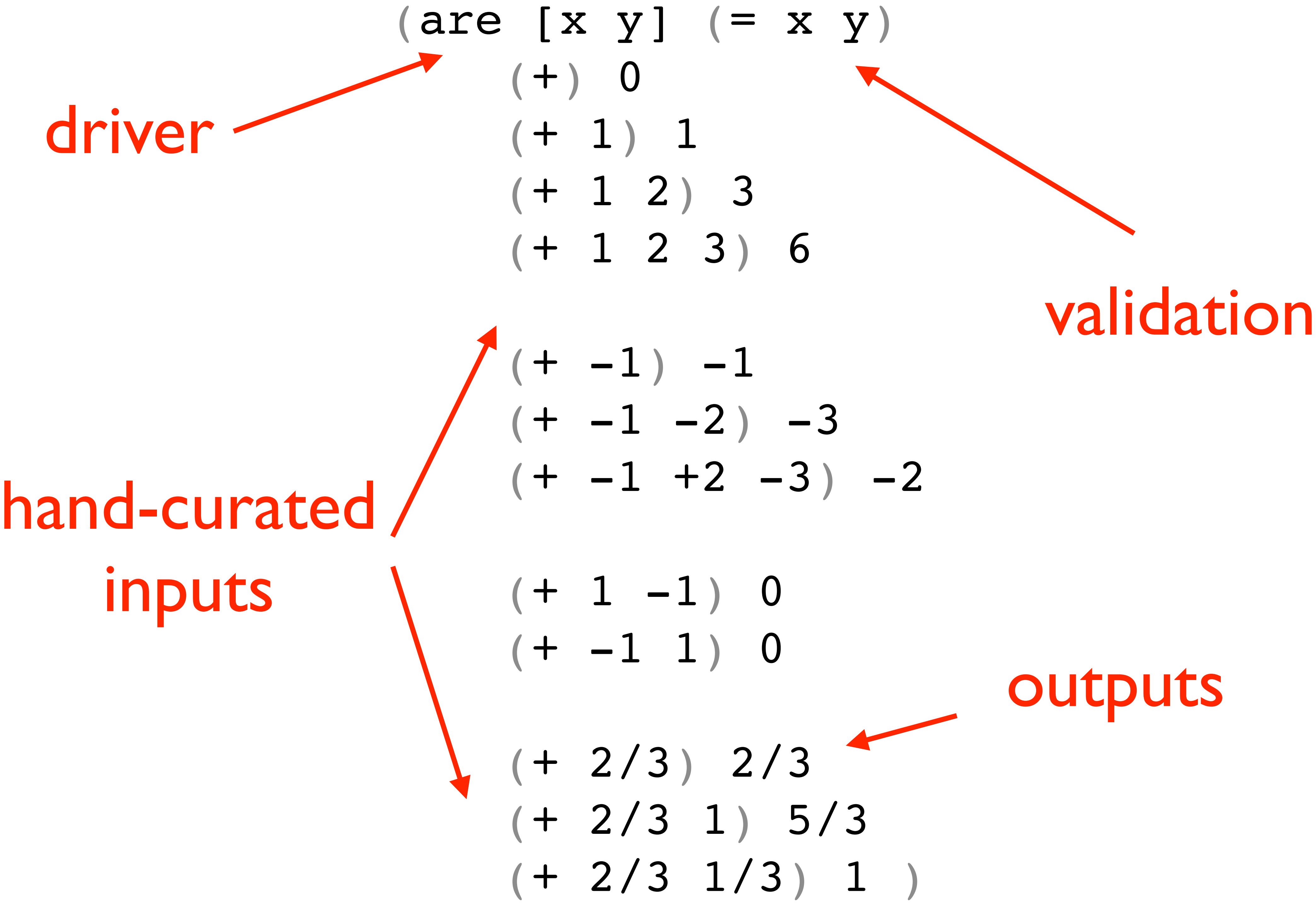
tests are executable documentation (?)

what is a test?

- create input
- driver
- capture output

- validate
- communicate
- record

test, complected



```
(are [x y] (= x y)
  (+) 0
  (+ 1) 1
  (+ 1 2) 3
  (+ 1 2 3) 6

  (+ -1) -1
  (+ -1 -2) -3
  (+ -1 +2 -3) -2

  (+ 1 -1) 0
  (+ -1 1) 0

  (+ 2/3) 2/3
  (+ 2/3 1) 5/3
  (+ 2/3 1/3) 1  )
```

driver

validation

hand-curated inputs

outputs

generative testing

http://en.wikipedia.org/wiki/Mastermind_(board_game)

input generation

```
(defn random-secret
  []
  (gen/vec #(gen/one-of :r :g :b :y) 4))
```

fn under test

```
(defn score
  [c1 c2]
  (let [exact (exact-matches c1 c2)
        ums (unordered-matches c1 c2)
        unordered (apply + (vals ums))]
    {:exact exact
     :unordered (- unordered exact)}))
```

validation

```
(defn scoring-is-symmetric
  [secret guess score]
  (= score (game/score guess secret)))

(defn scoring-is-bounded-by-number-of-pegs
  [secret guess score]
  (< 0 (matches score) (count secret)))
```

driver

fn under test

inputs

```
(defspec score-invariants
  game/score
  [^{:tag `random-secret} secret
   ^{:tag `random-secret} guess]
  (assert (scoring-is-symmetric secret guess %))
  (assert (scoring-is-bounded-by-number-of-pegs
           secret guess %)))
```

validations

...or just enumerate all cases

```
========================================================
:secret       | :guess        | :score
========================================================
(:r :r :r :r) | (:r :r :r :r) | {:exact 4, :unordered 0}
(:r :r :r :r) | (:r :r :r :g) | {:exact 3, :unordered 0}
(:r :r :r :r) | (:r :r :r :b) | {:exact 3, :unordered 0}
(:r :r :r :r) | (:r :r :r :y) | {:exact 3, :unordered 0}
                      ...
```

test inputs and outputs are data

visual inputs

```
:dying  "...
         .O.
         ..."

  :off  "O.O
         ...
         O.O"

  :on   "|||
         O.O
         |||"
```

http://thinkrelevance.com/blog/2009/10/07/brians-functional-brain-take-1-5

tests write docs

```
==============================================================================
:input             | :bind-form   | :bind-type | :count | ?a                 |
==============================================================================
42                 | ?a           | scalar     | 1      | 42                 |
[2 3 5 7]          | ?a           | scalar     | 1      | [2 3 5 7]          |
42                 | [?a ?b]      | tuple      |        |                    |
[2 3 5 7]          | [?a ?b]      | tuple      | 1      | 2                  |
42                 | [?a ...]     | collection |        |                    |
[2 3 5 7]          | [?a ...]     | collection | 4      | 3 / 2 / 7 / 5      |
42                 | [[?a ?b ?c]] | relation   |        |                    |
[2 3 5 7]          | [[?a ?b ?c]] | relation   |        |                    |
==============================================================================
```

once you add time, stateful interactions are

once you add time,
stateful interactions are

data!

time series table

```
===============================
:call                 | :result
===============================
(put q :one)          | true
(offer q :two)        | true
(offer q :three)      | false
(offer q :three 10)   | false
(take q)              | :one
(poll q :missing)     | :two
(poll q :missing)     | :missing
(poll q :missing 10)  | :missing
===============================
```

interleaving

```
-----------------------------------------------------------------
Legend:     incrementing value on reconnect, or [T]ime[O]ut
reconnects: [A]ttempt [S]ucceed [F]ail [B]ackoff [Interrupt]
            [T]throw [Z]sleep [C]leanup c[L]eanupFailed
-----------------------------------------------------------------
```

```
0 A L Z B A I S
1 C A S
```

simulating failure

```
...........................................
...........................................
..............

{:retries 9,
 :blocks-planned 100,
 :blocks-completed #<Atom@6a92d5c0: 100>,
 :storage-fail-rate 0.05}
```

simulating failure

```
...........................................
...........................................
..............

{:retries 9,
 :blocks-planned 100,
 :blocks-completed #<Atom@6a92d5c0: 100>,
 :storage-fail-rate 0.05}
```

possible sources:

timing

logs

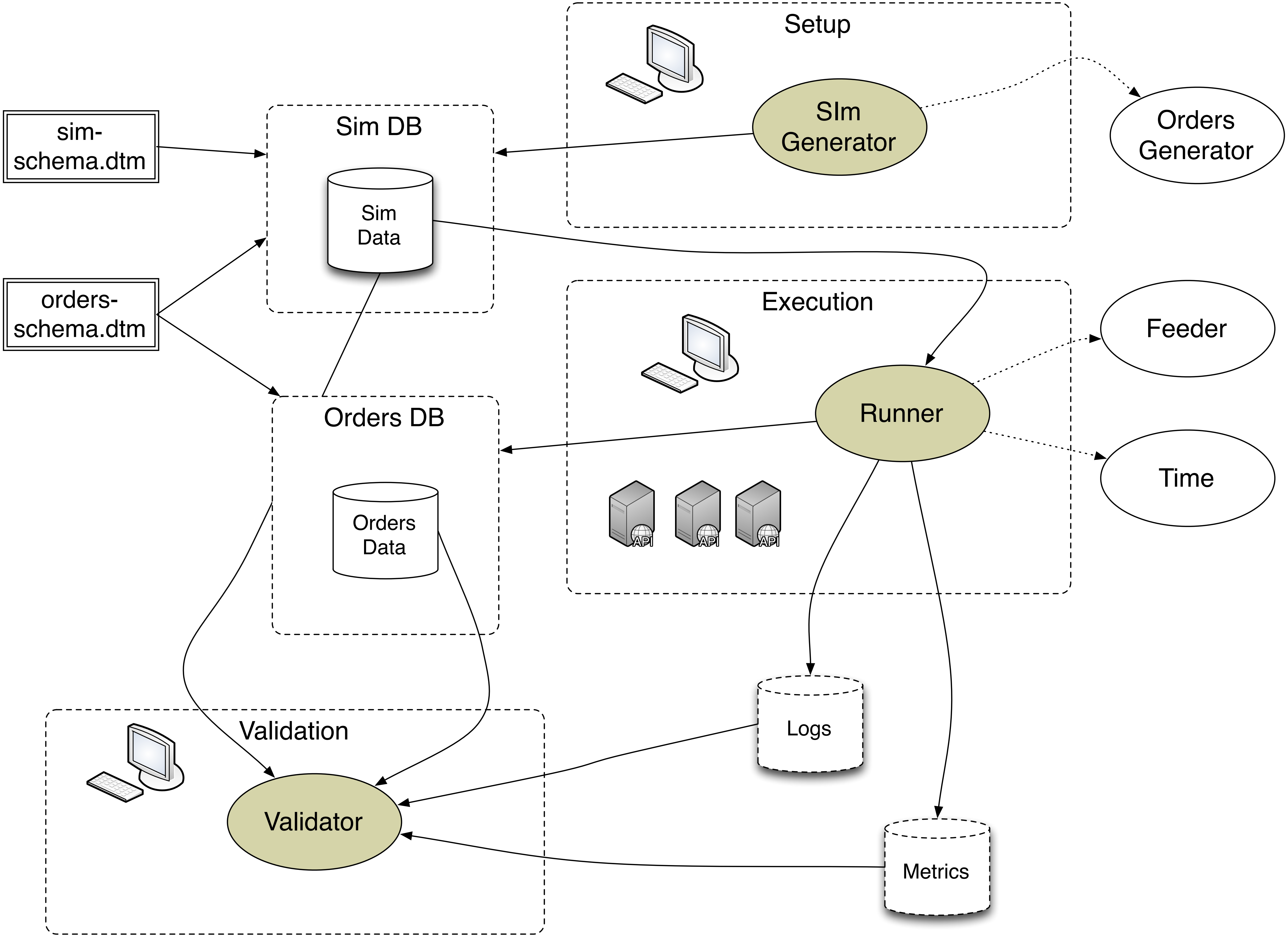
metrics

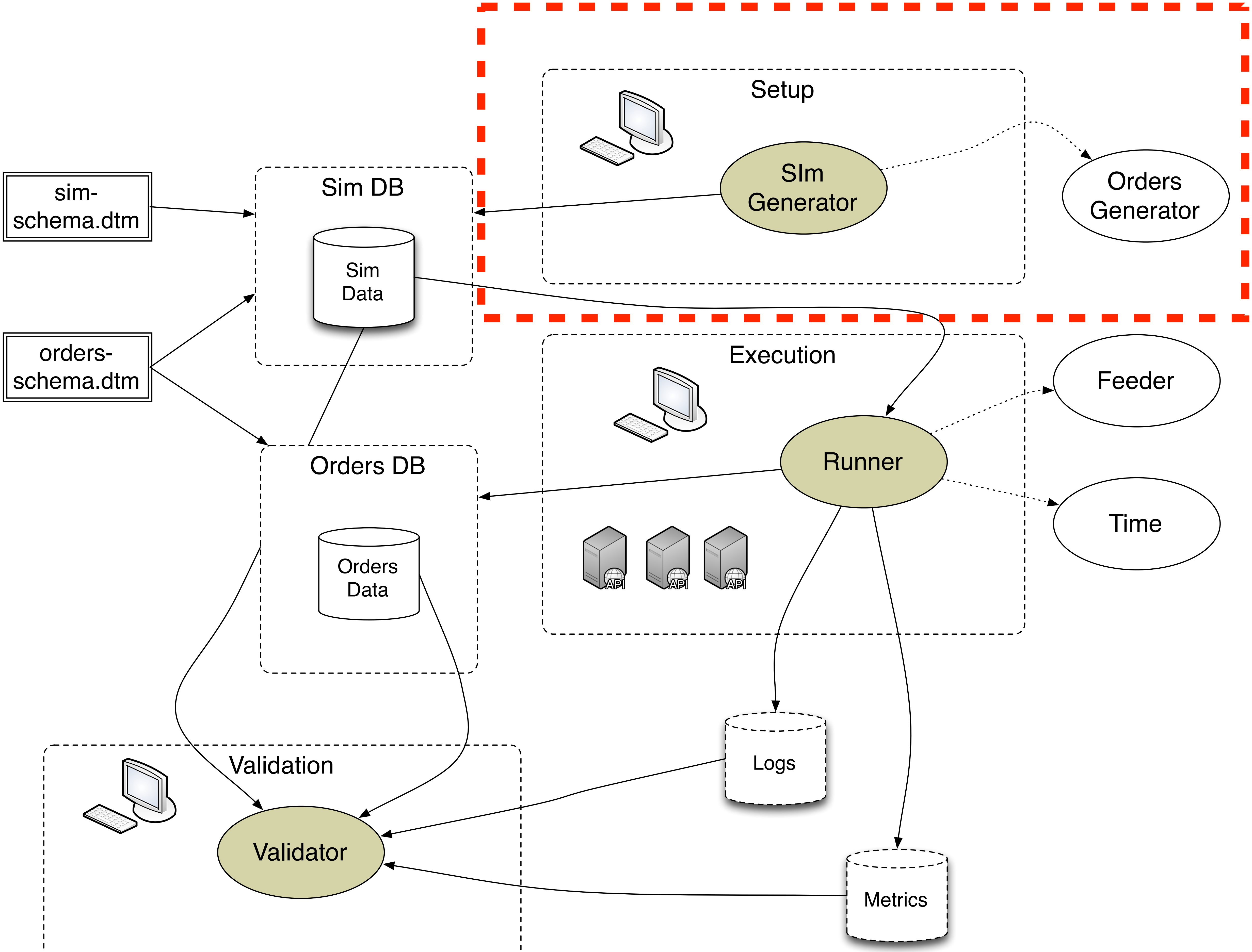
tests are programs

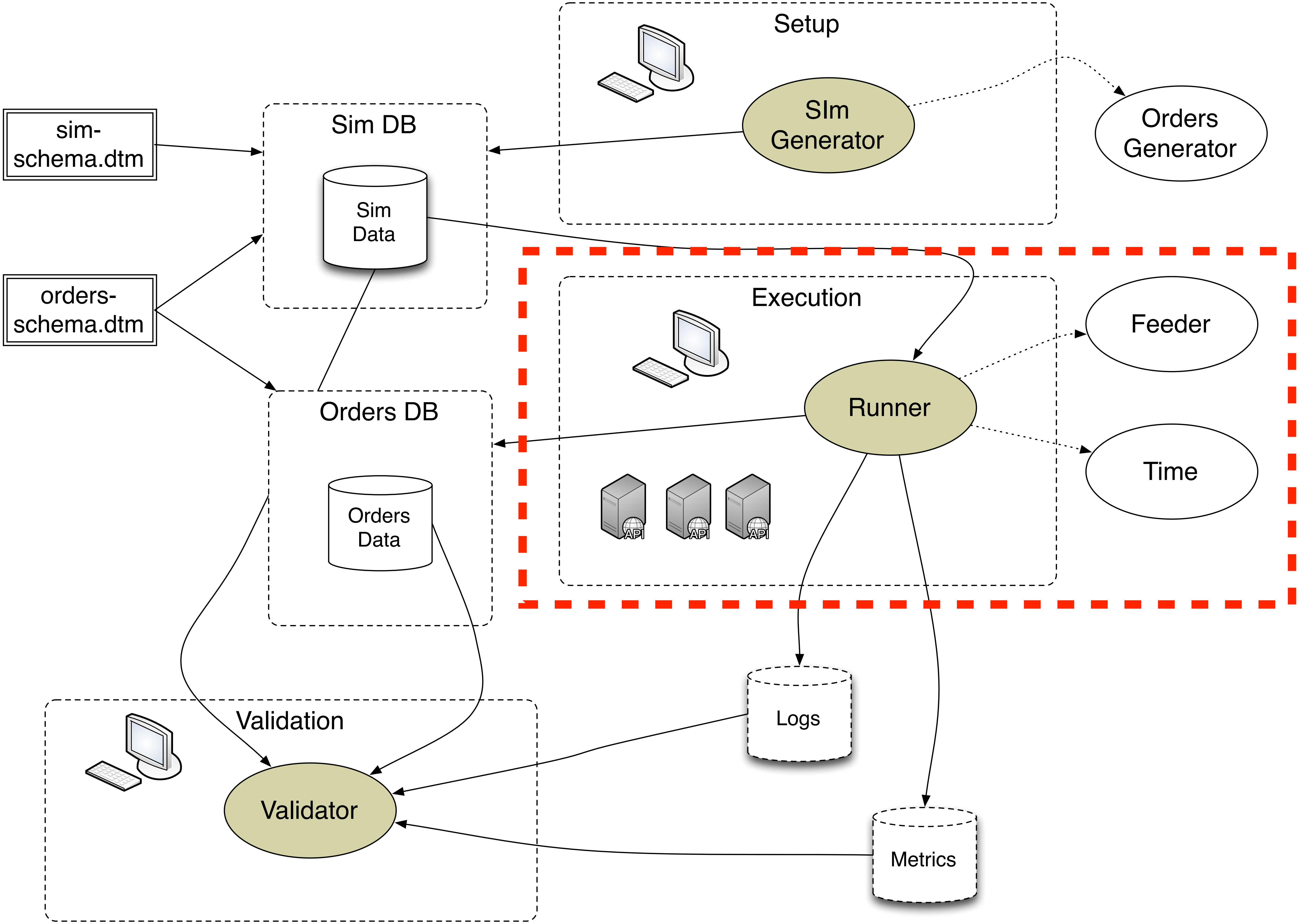
from REPL to regression

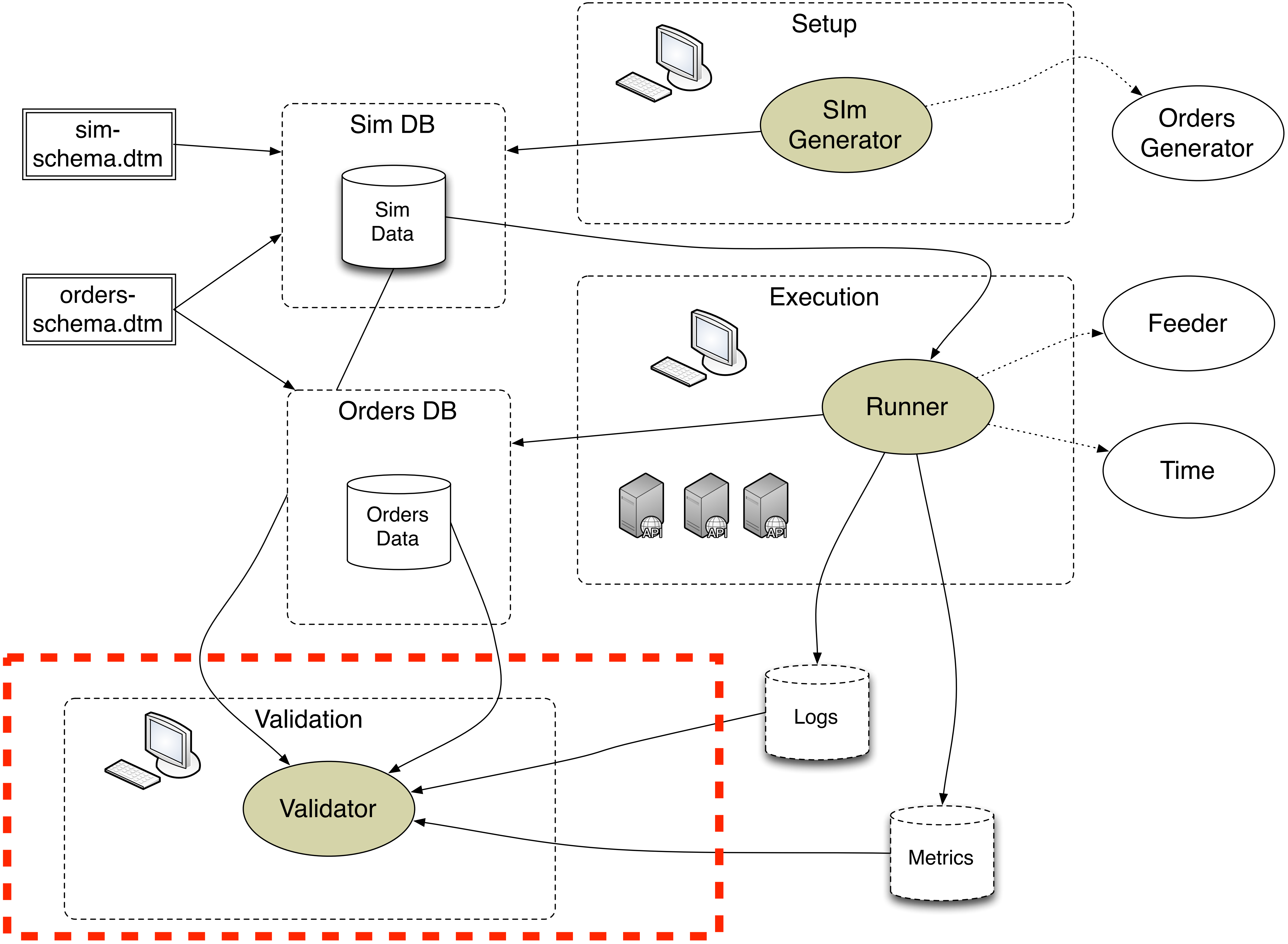
```
(defn run-with-transcript
  [forms]
  (binding [*ns* *ns*]
    (let [temp (gensym)]
      (in-ns temp)
      (clojure.core/use 'clojure.core)
      (doseq [f forms]
        (pprint f)
        (print "=> ")
        (pprint (eval f))
        (println))
      (remove-ns temp)
      :done)))
```

the sim


Setup
SIm Generator
Orders Generator
sim-schema.dtm
Sim DB
Sim Data
orders-schema.dtm
Execution
Feeder
Runner
Time
Orders DB
Orders Data
API
API
API
Logs
Validation
Validator
Metrics


Setup
SIm Generator
Orders Generator
sim-schema.dtm
Sim DB
Sim Data
orders-schema.dtm
Execution
Feeder
Runner
Time
Orders DB
Orders Data
API
API
API
Logs
Validation
Validator
Metrics


Setup
SIm Generator
Orders Generator
sim-schema.dtm
Sim DB
Sim Data
orders-schema.dtm
Execution
Feeder
Runner
Time
API
API
API
Orders DB
Orders Data
Logs
Validation
Validator
Metrics


Setup
SIm Generator
Orders Generator
sim-schema.dtm
Sim DB
Sim Data
orders-schema.dtm
Execution
Feeder
Runner
Orders DB
Time
Orders Data
API
API
API
Validation
Logs
Validator
Metrics

simulation

- not a precious curated path
- uses real input and output points
- models time
- separates testing activities in time and space
- validates via logic programs

- some effort to set up

how Clojure enabled Datomic

Clojure features

- data
- functions
- references
- protocols
- deftype

Clojure philosophy

- architecture

- simplicity
- power
- focus