es!

Tests!

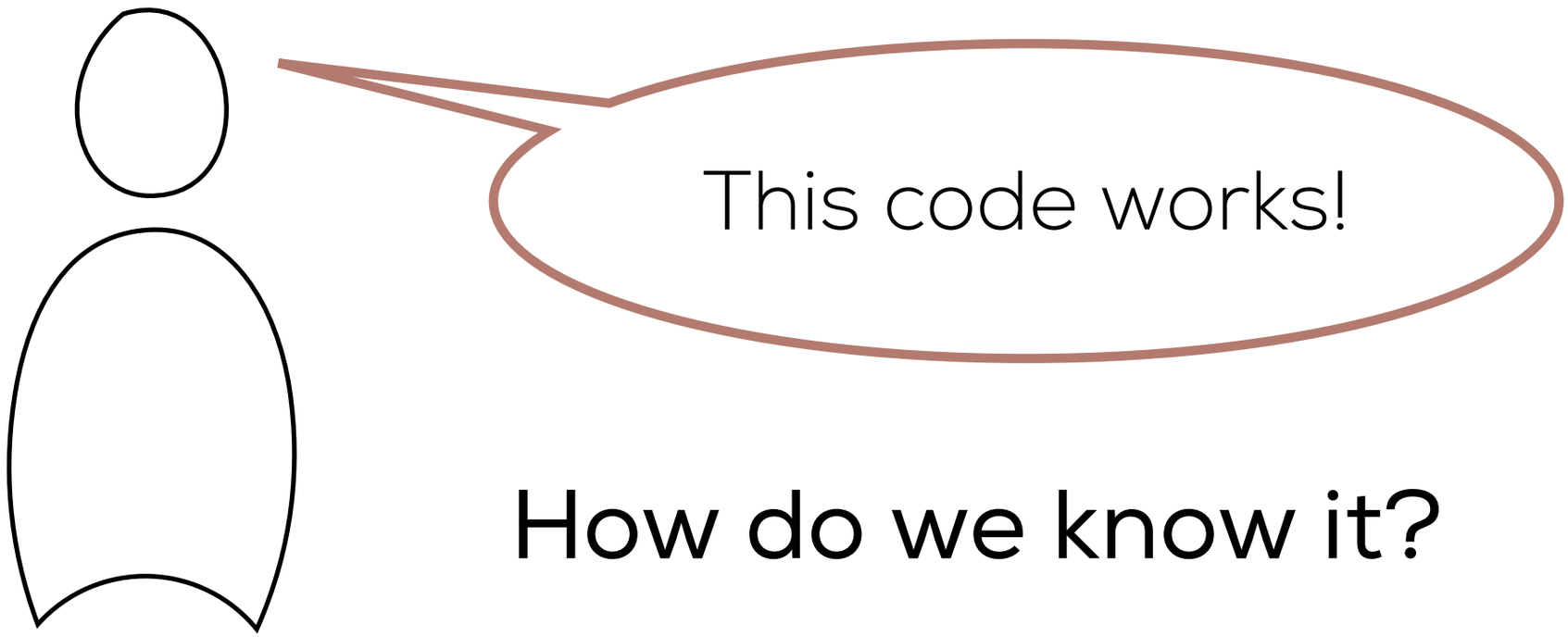
Contracts
in
Clojure!

@jessitron

Contracts and Clojure:
the Best-Yet Compromise Between Types and Tests

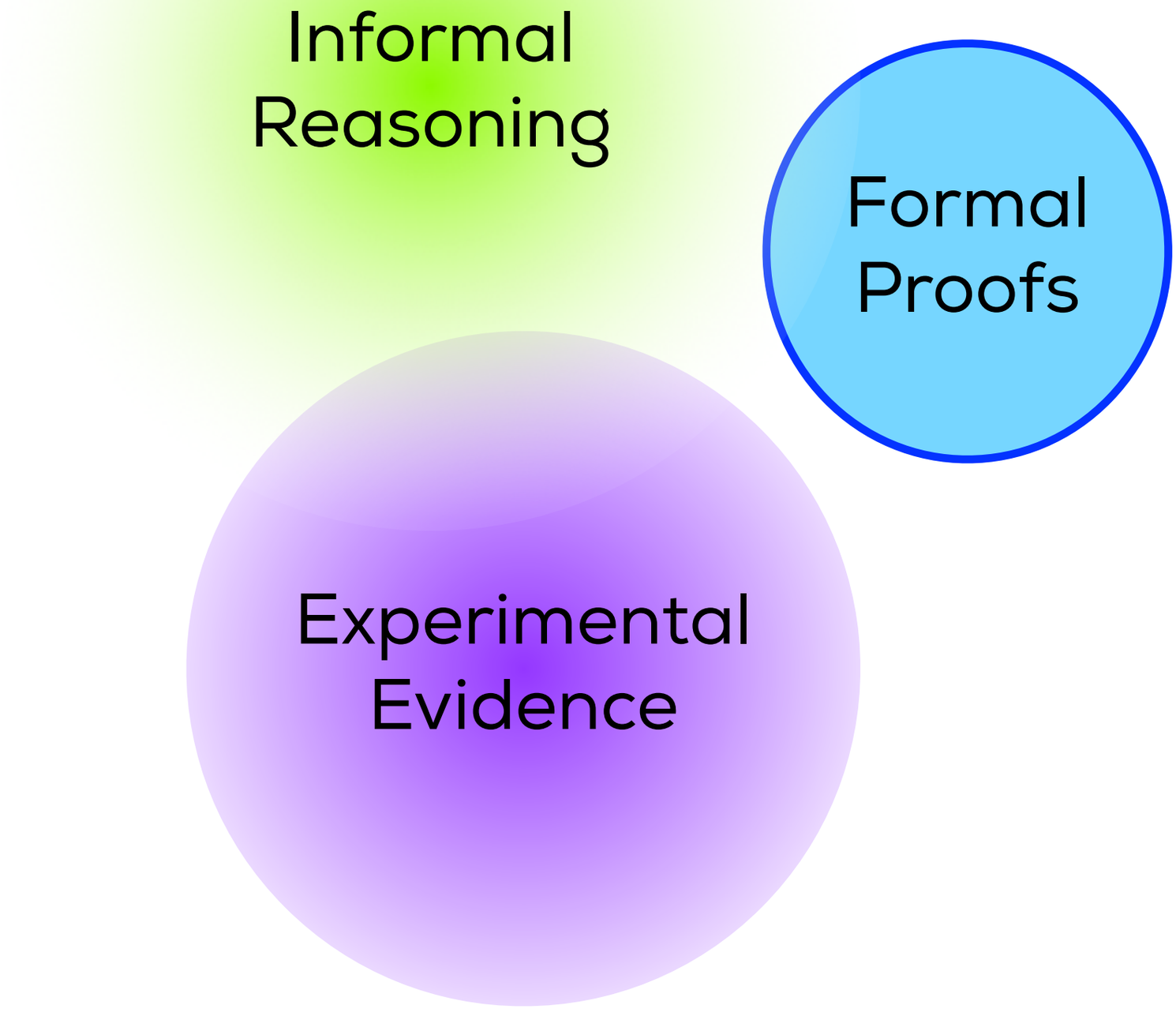
Philly ETE, 7 April 2015

What do we know?

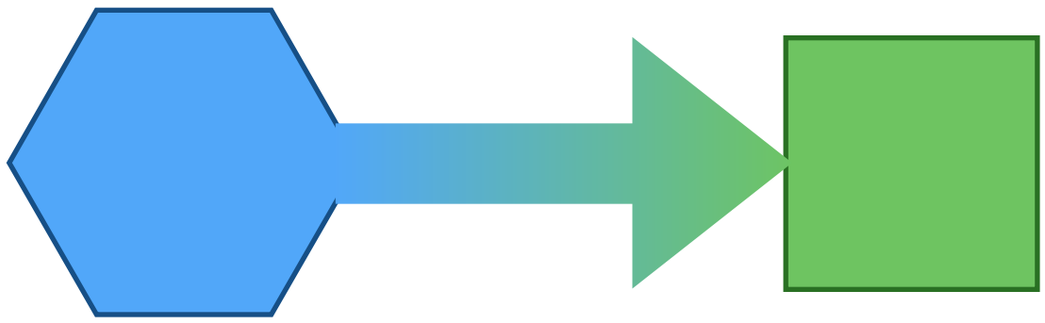


How do we know it?

...


Informal
Reasoning
Formal
Proofs
Experimental
Evidence

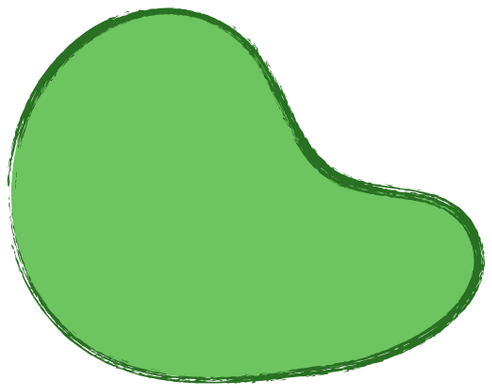
```scala
// Scala

def formatReport(data: ReportData): ExcelSheet
```



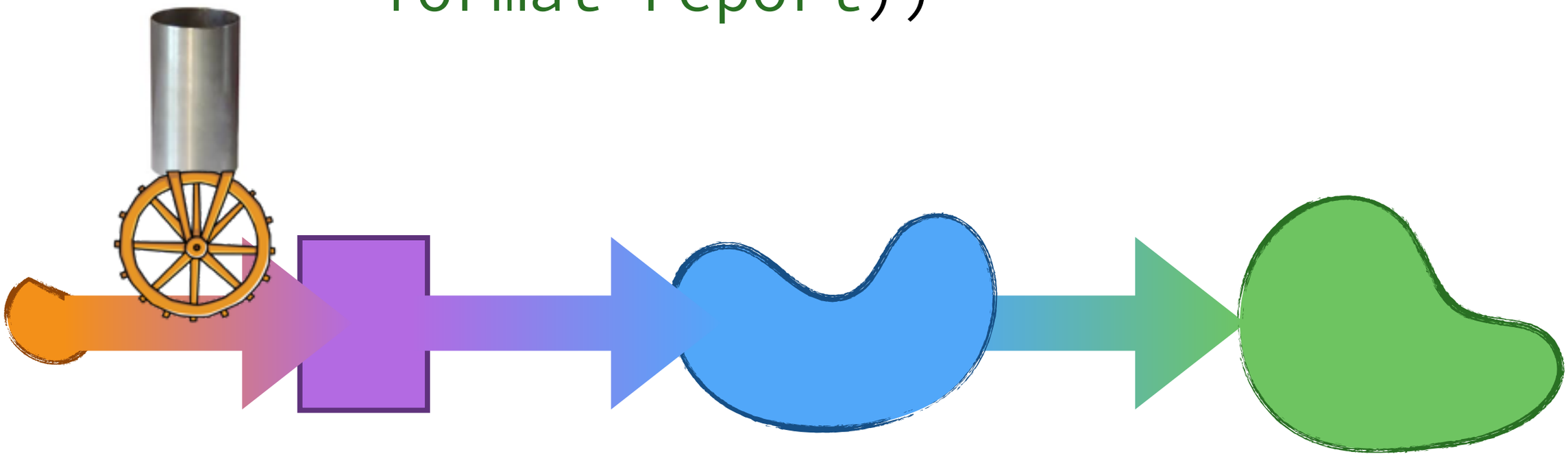

T.GERRARD & CO
SUSSEX & SURREY

```clojure
(defn format-report [report-data]
  ...)
```

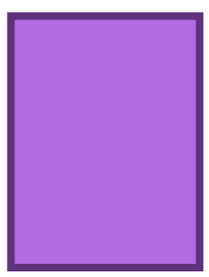
```clojure
(defn ad-performance-report [params]
  (-> (fetch-events params)
      (analyze-ad-performance params)
      format-report))
```

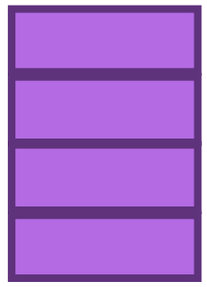


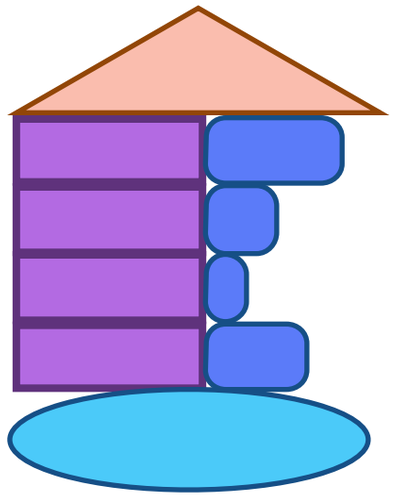
```clojure
(defn ad-performance-report [params]
  (-> (fetch-events params)
      (analyze-ad-performance params)
      format-report))
```

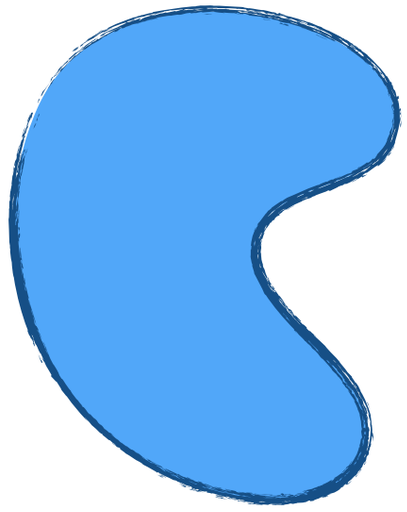


```clojure
(defn analyze-ad-performance [events params]
  (-> events
      (group-up params)
      summarize
      add-total-row
      (add-headers params)))
```

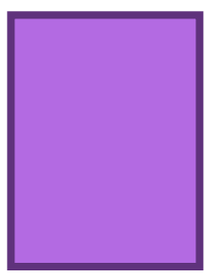


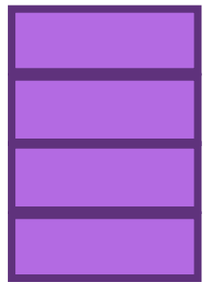


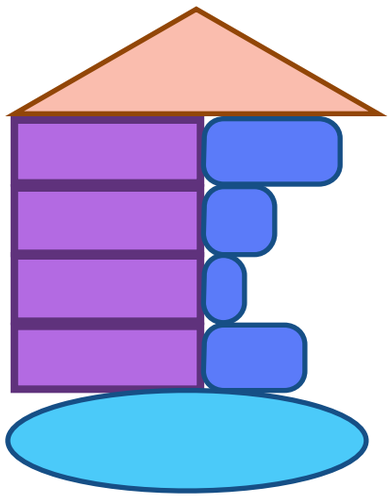


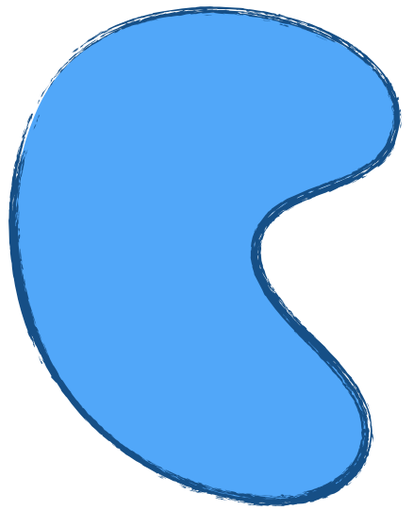


```
{:when 12:34:56 7/8/90
 :what "show"
 :who  "abc123"}
```

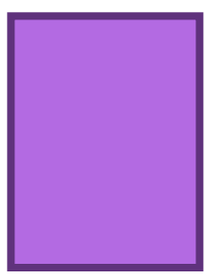


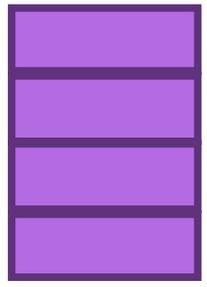


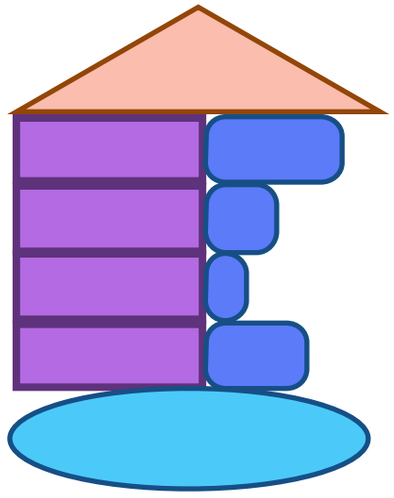


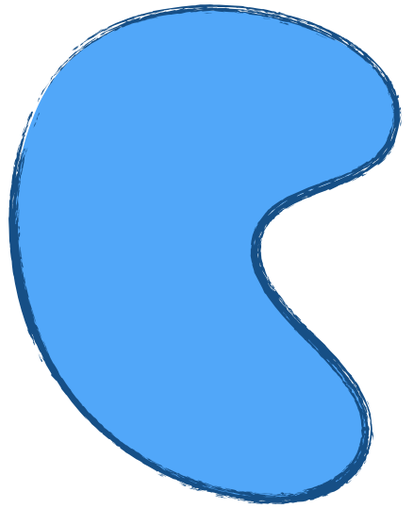


```
(def Event {:when org.joda.time.DateTime
            :what java.lang.String
            :who  java.lang.String})
```

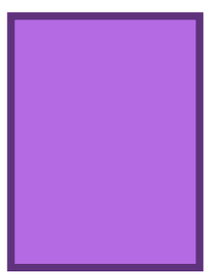


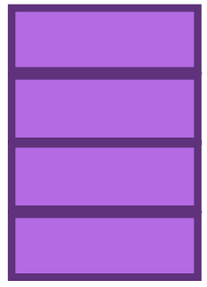


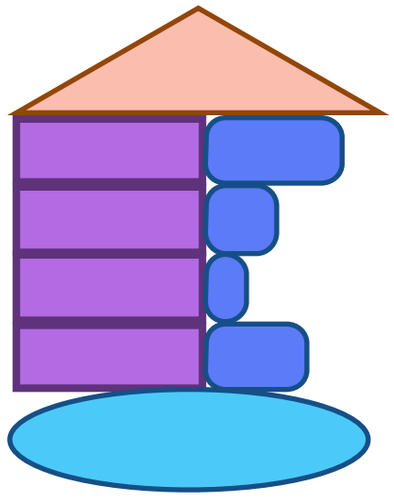


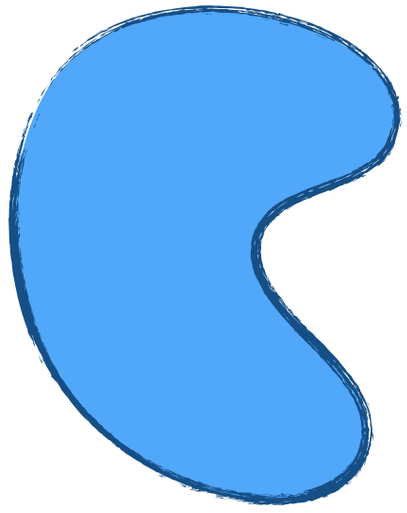


```
(:require [schema.core :as s])

(def Event {:when DateTime
            :what s/Str
            :who  s/Str})
```

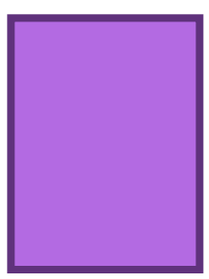


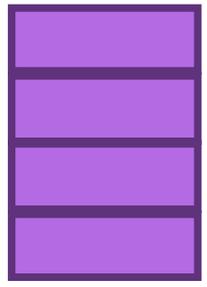


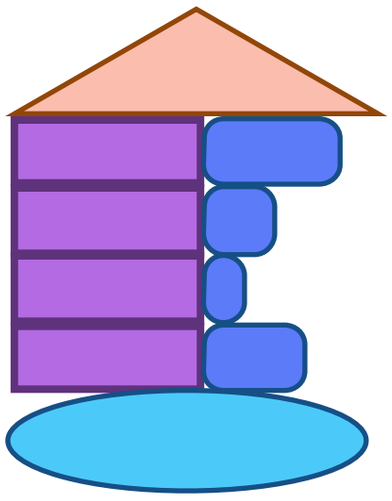


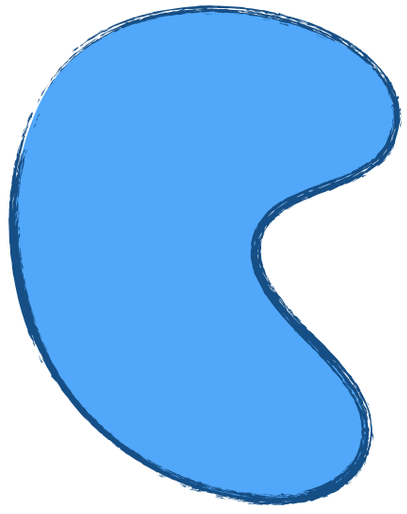


```
(:require [schema.core :as s])

(def Event {:when DateTime
            :what Incident
            :who  Customer})
```



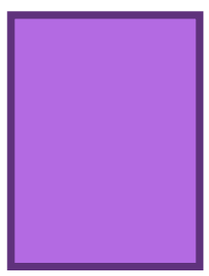


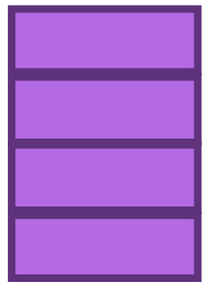


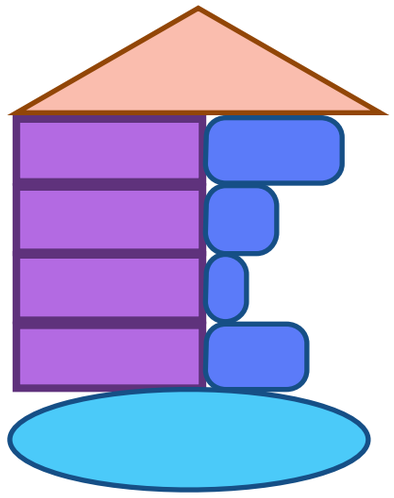


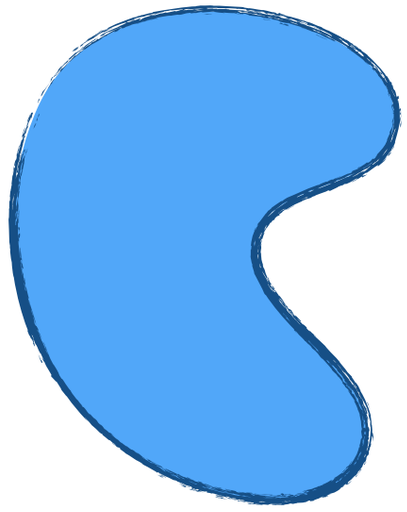
```
(:require [schema.core :as s])

(def Event {:when DateTime
            :what Incident
            :who  Customer})
```

[Event]



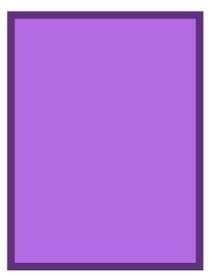


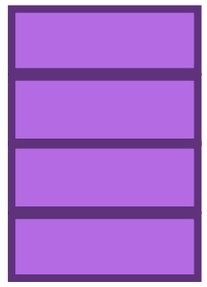


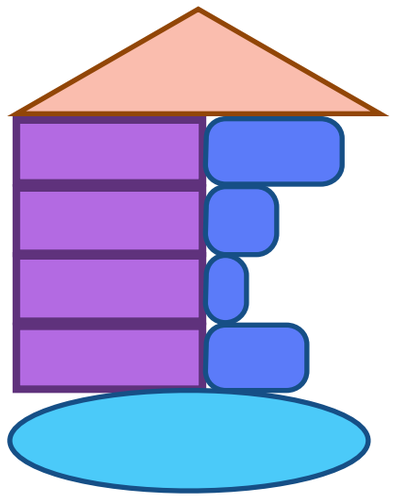


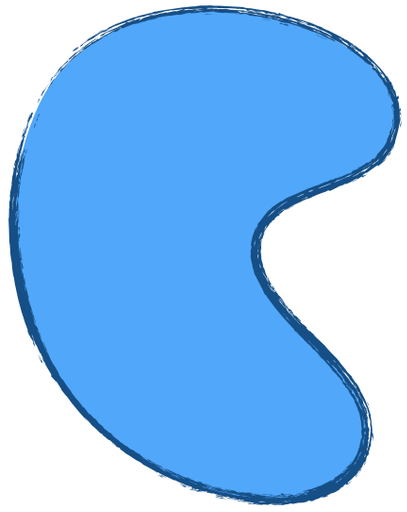
```
(defn ad-performance-report [params]
  (-> (fetch-events params)
      (analyze-ad-performance params)
      format-report))
```

[Event]



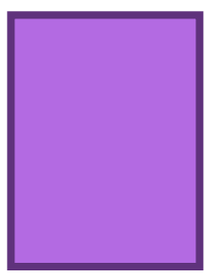


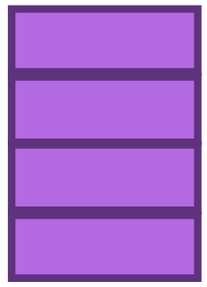


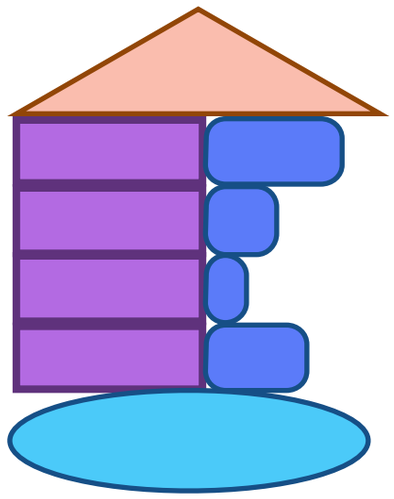


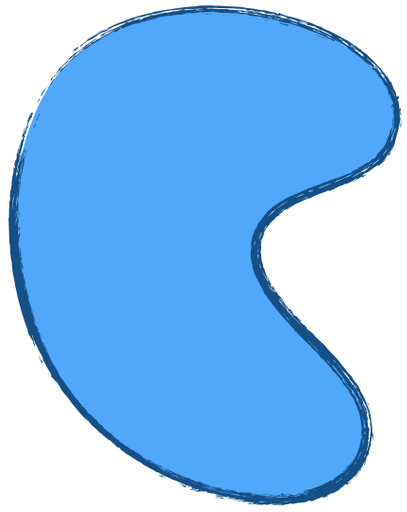
```
(s/defn fetch-events [params]
  …)
```

[Event]

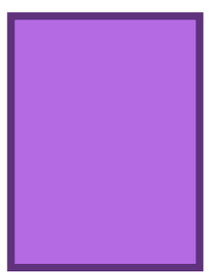


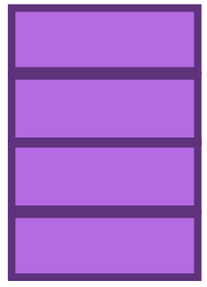


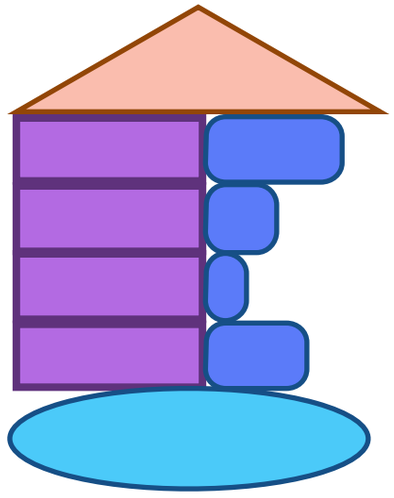


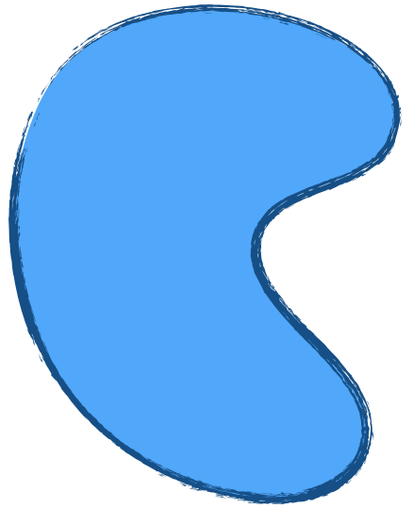


```
(:require [schema.core :as s])

(s/defn fetch-events :- [Event]
  [params]
  …)
```







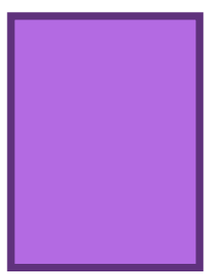


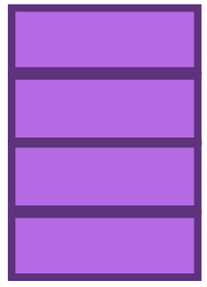
```
(deftest fetch-events-test
  …
   (= (expected (fetch-events input))))
```

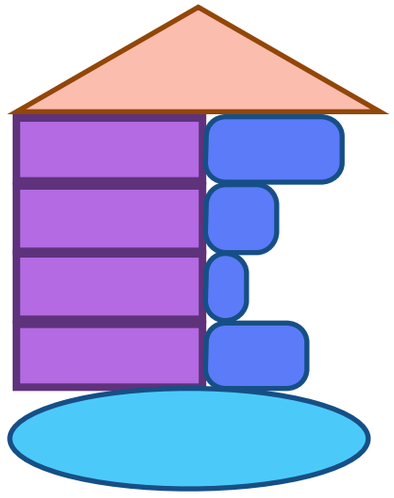
```
(use-fixtures schema.test/validate-schemas)


(deftest fetch-events-test
  …
  (= (expected (fetch-events input))))
```

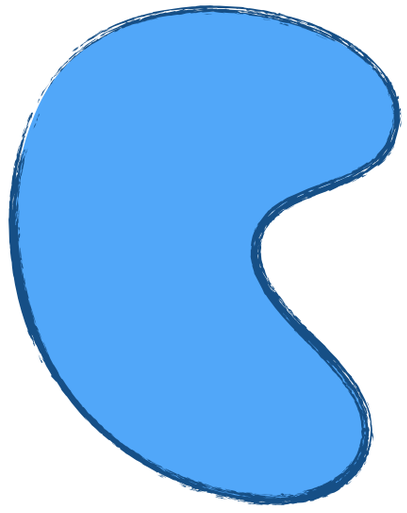
```
(:require [schema.core :as s])


(def Event {:when DateTime
            :what Incident
            :who  Customer})
```

[Event]



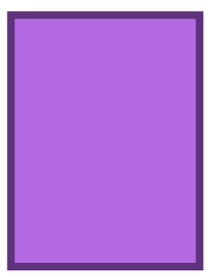


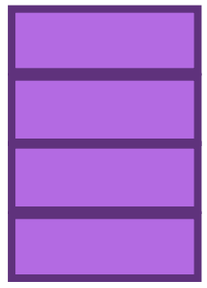


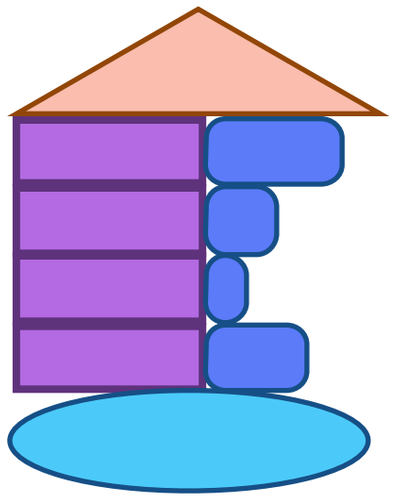


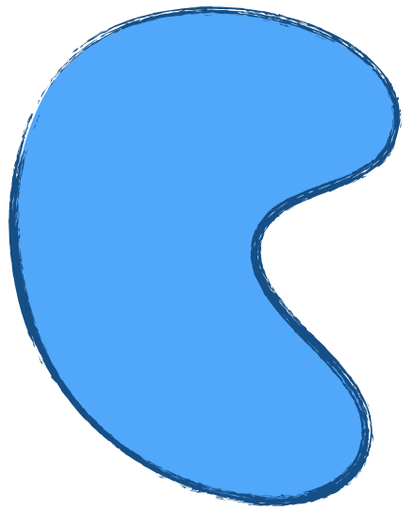
```
(:require [schema.core :as s])

(def Event {:when DateTime
            :what Incident
            :who  Customer})
```

```
[Event]  [[Event]]
```

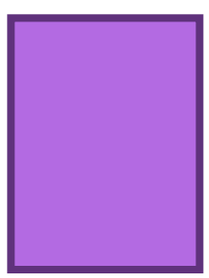


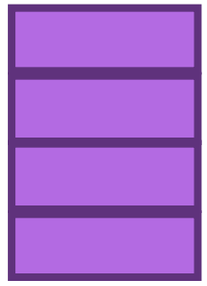


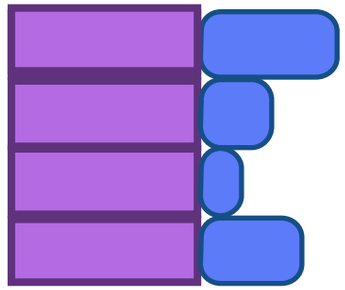


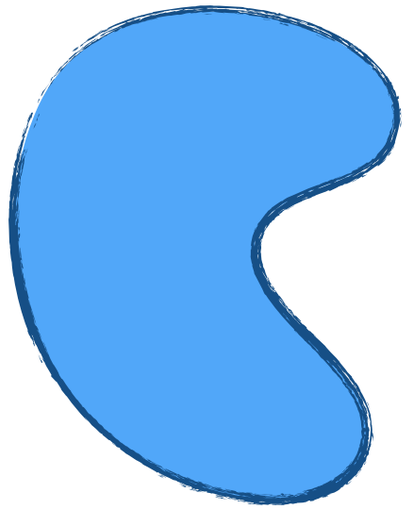


[Event] [[Event]]



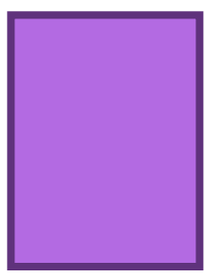


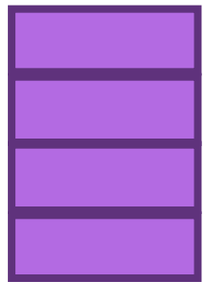


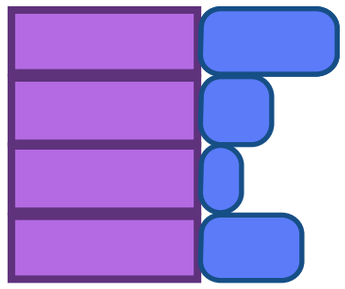


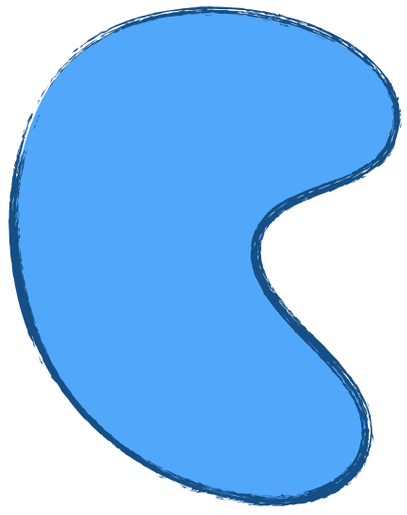
[Event]

[Event] [[Event]]



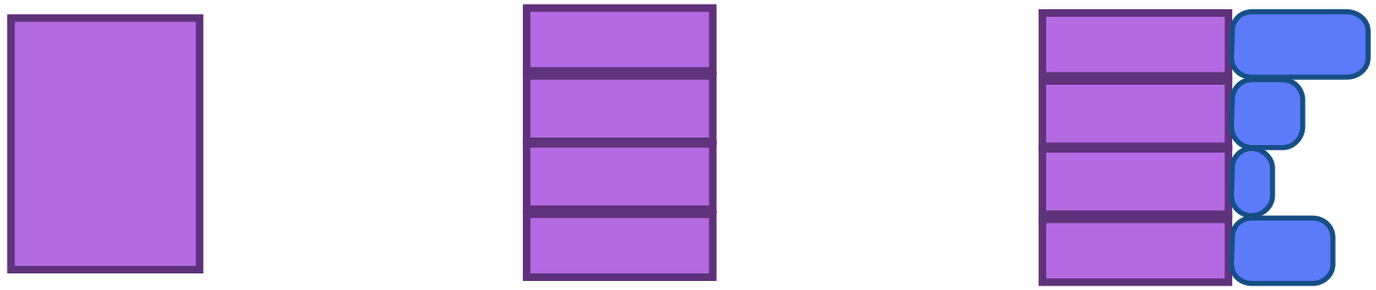






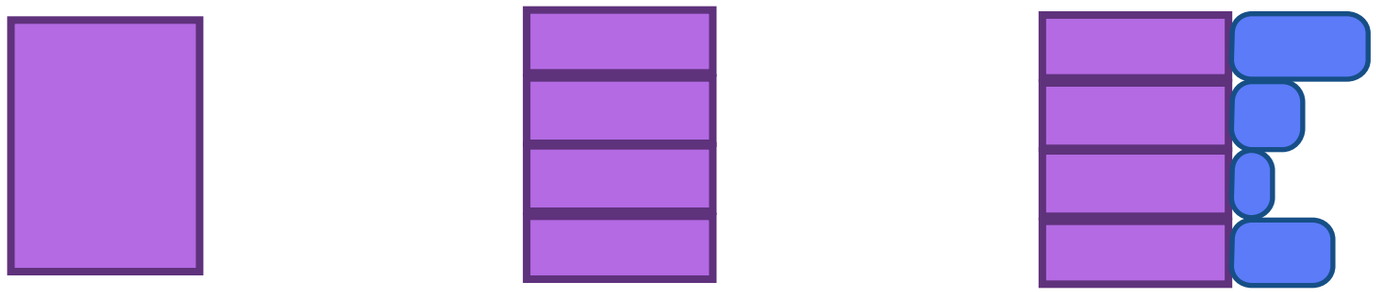
[[Event] Summation]

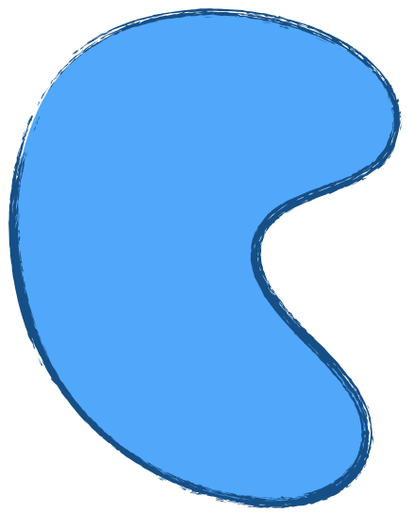
[Event] [[Event]]



```
[(   one [Event]     )
 (   one Summation   )]
```

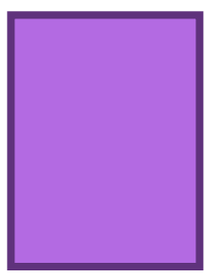
```
[Event]  [[Event]]
```



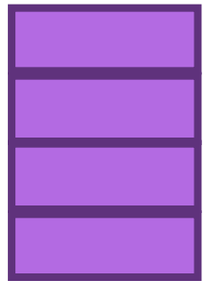


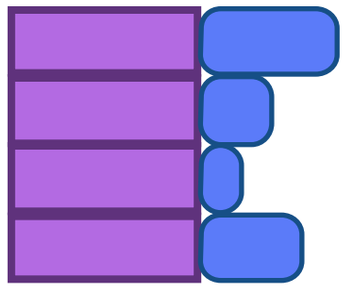
```
[(s/one [Event]   "event list")
 (s/one Summation "group sum")]
```

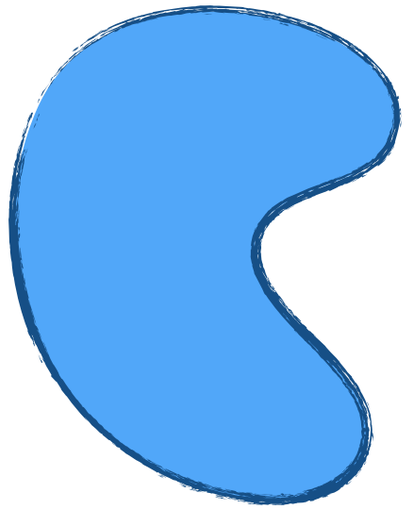
`[Event]`



`[[Event]]`

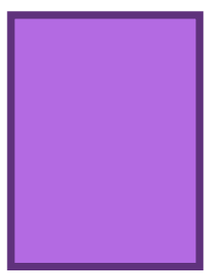




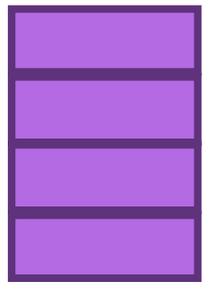


```
(def Group [(s/one [Event]   "event list")
            (s/one Summation "group sum")])
```

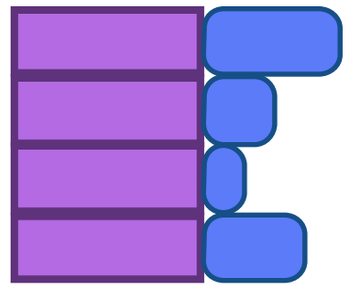
`[Event]`

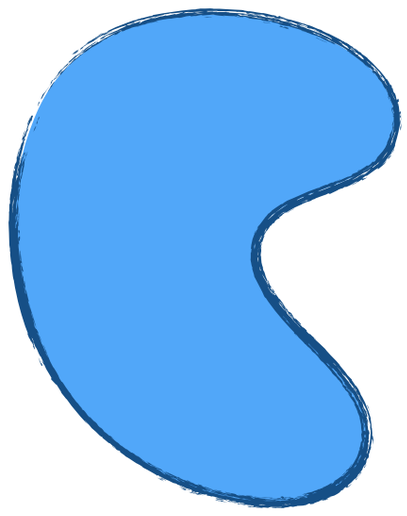


`[[Event]]`



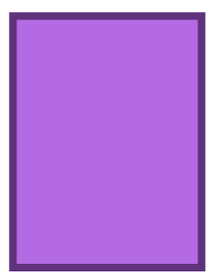
`[Group]`

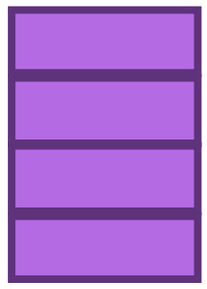


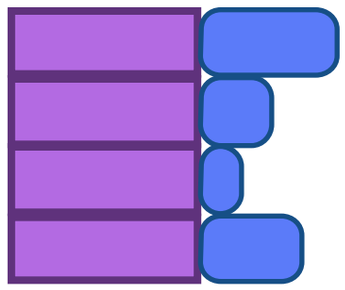


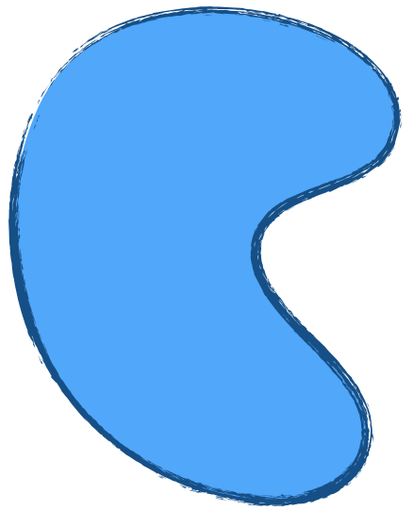
```
{:groups [Group]}
```

```
[Event]  [[Event]]
```



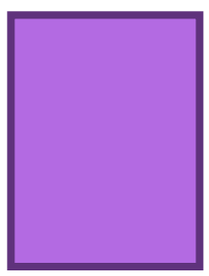


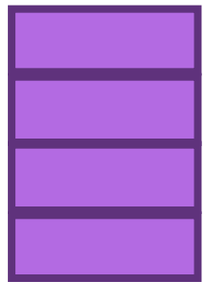


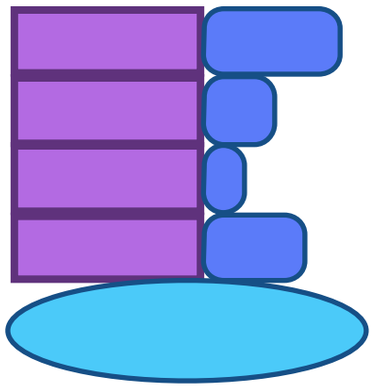


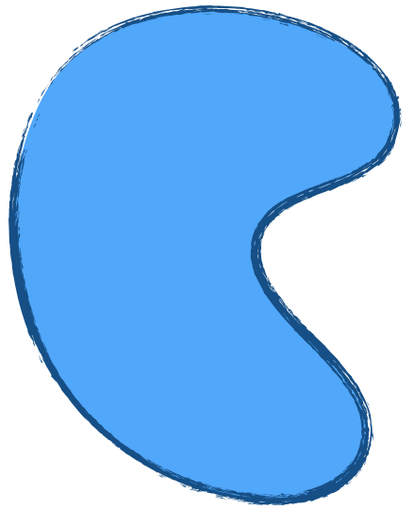
```
{:groups [Group]
 :totals Totals}
```

`[Event]` `[[Event]]`



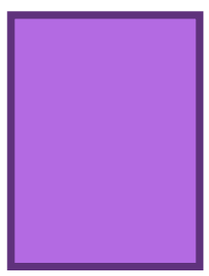




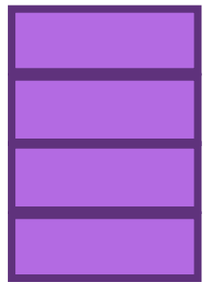


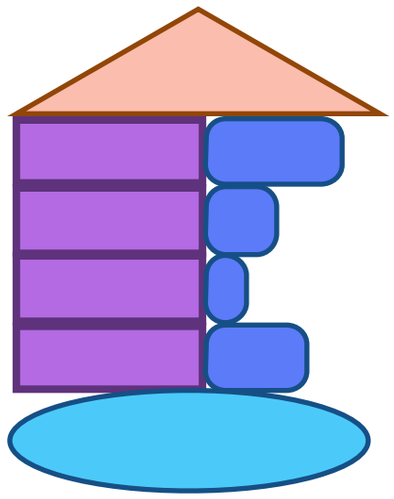
```
{:header Headers
 :groups [Group]
 :totals Totals}
```

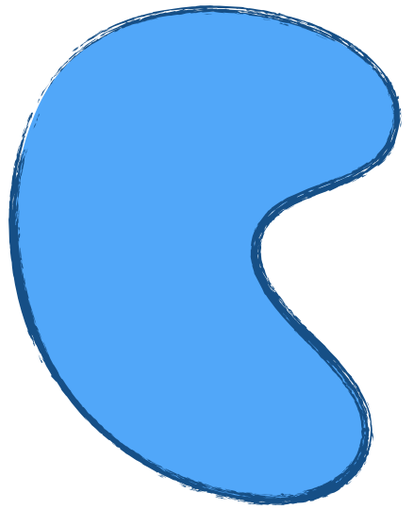
```
[Event]
```



```
[[Event]]
```

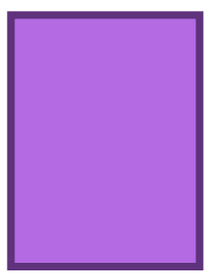


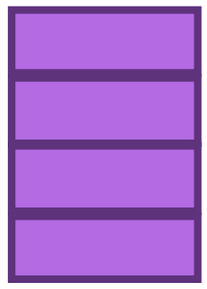


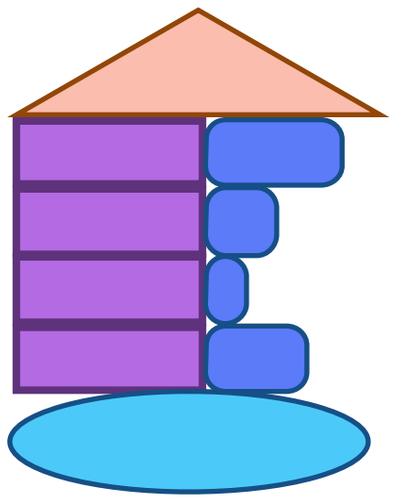


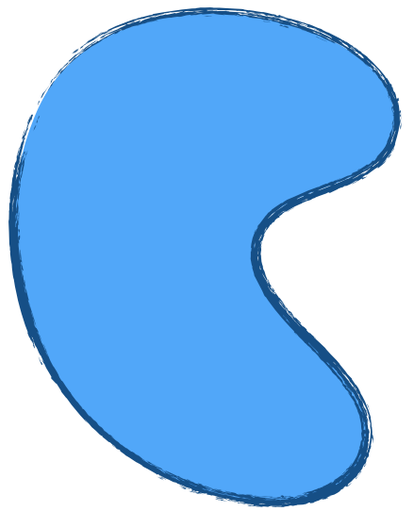
```
(def ReportData {:header Headers
                 :groups [Group]
                 :totals Totals})
```

```
[Event]   [[Event]]
```

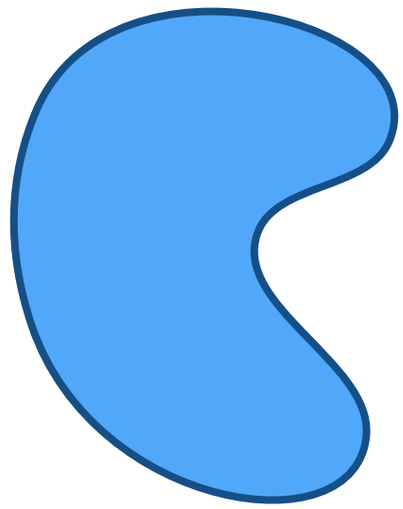








```
(def ReportData {:header Headers
                 :groups [Group]
                 :totals Totals})
```



```
(s/defn analyze-ad-performance :- ReportData
  [events :- [Event]
   params :- Params]
  (-> events
      (group-up params)
      summarize
      add-total-row
      (add-headers params)))
```

```clojure
(deftest analyze-ad-performance-test
  (testing "grouping of rows"
    (let […
          result (analyze-ad-performance
                   events
                   {})
          (is (= expected (:groups result)))))))
```

```
(use-fixtures schema.test/validate-schemas)


(deftest analyze-ad-performance-test
(testing "grouping of rows"
  (let […
        result (analyze-ad-performance
                events
                {})
    (is (= expected (:groups result))))))
```

```
      result (analyze-ad-performance
               events
               {})
  (is (= expected (:groups result)))))))
```

Input does not match schema Params

Missing required key :title

Missing required key :start

Missing required key :end

Params

:title string

:start date

:end date

param-gen

:title string

:start date

:end date

param-gen

:title string
:start date
:end date

:title string
:start date
:end date

param-gen

:title string that isn't empty
:start date before end date
:end date before now

```
(deftest analyze-ad-performance-test
  (testing "grouping of rows"
    (let […
          result (analyze-ad-performance
                   [events]
                   (sample-one param-gen))
(is (= expected (:groups result)))))))
```

```clojure
(defspec analyze-ad-performance-spec 100
  (for-all [events events-gen
            params param-gen]
    (analyze-ad-performance events params))))
```

Q: What do we know?

A: Schemas

Q: How do we know it?

A: Generative Tests

Q: What do we know?

data shape

We live in this weird time where a rose by any other name throws a compile/runtime error. @deech

Q: What do we know?

data shape

data value boundaries

Headers

:title
  - string
  - not empty
  - capitalized

```
(def Headers
{:title
  (s/both
    s/Str
    (s/pred (complement empty?) "nonempty")
    (s/pred capitalized? "Title Caps"))
 …})
```

Q: What do we know?

data shape

value boundaries

relationships within values

Headers

{:title ...
 :start DateTime
 :end   DateTime}
  - start is before end

Q: What could we know?

produced types


```
(s/def params :- (generator-of Params)
                  (gen/hash-map
                    :title …
                    :start …
                    :end …))
```


```
(defgen params Params (gen/hash-map
                       :title …
                       :start …
                       :end …))
```


```
(defgen params Params (gen/hash-map
                       :title …
                       :start …
                       :end …))
```

```
Generator: my-project.generators/params
Value: {:end #<DateTime -58693684-08-30T03:25:35.104Z>,
Error: (not (keyword? a-clojure.lang.PersistentArrayMap)
```

```
(s/defn fetch-events :- [Event]
  [params]
  ...)
```


```
(s/defn fetch-events :- (lazy-seq-of Event)
  [params]
  ...)
```

Generator of A

Lazy sequence of A

Function from A to B

Q: What could we know?

produced types

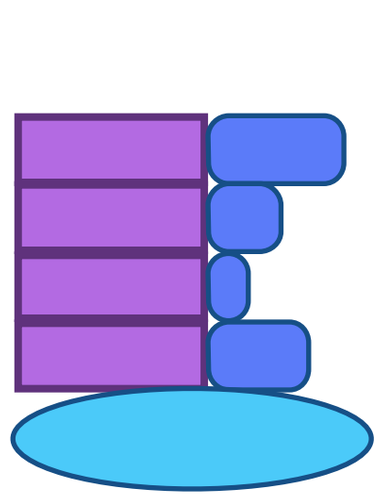
relationships between types

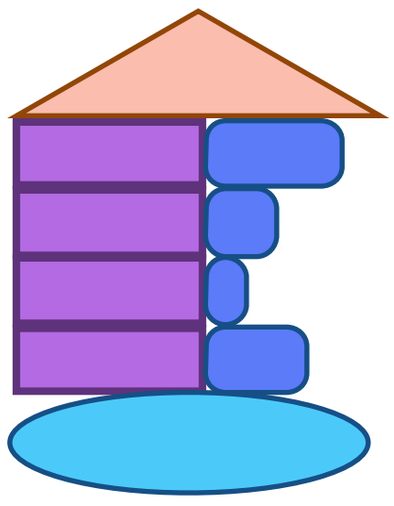
```
(s/defn sample-one [A :- Schema]
  (s/fn :- A [g :- (generator-of A)]
    (last (gen/sample g))))
```

```
((sample-one Params) params-gen)
```


```
(tdefn sample-one :- A
                  [A :- Schema
                   g :- (generator-of A)]
                  (last (gen/sample g)))

(sample-one Params param-gen)

(sample-one param-gen)
```


```
(tdefn add-headers :- (merge A
                             {:header Headers})
       [data-so-far :- [A :< AnyMap]
        params])
```






```
(tdefn add-headers :- (merge A
                             {:header Headers})
      [data-so-far :- [A :< AnyMap]
       params])
```

```
(add-headers data params)
```

```
(add-headers GroupsWithTotal data params)
```

Q: What could we know?

produced types

relationships between types

relationships between values

```clojure
(defn group-up [events params]
  {:post [(as-lazy-as events %)]
   ...))
```

data shape

data value boundaries

relationships within values

produced types

relationships between types

relationships between values


T.GERRARD
SUSSEX & CO
SURREY

```
(def Event {:when DateTime
            :what Incident
            :who  Customer})
```

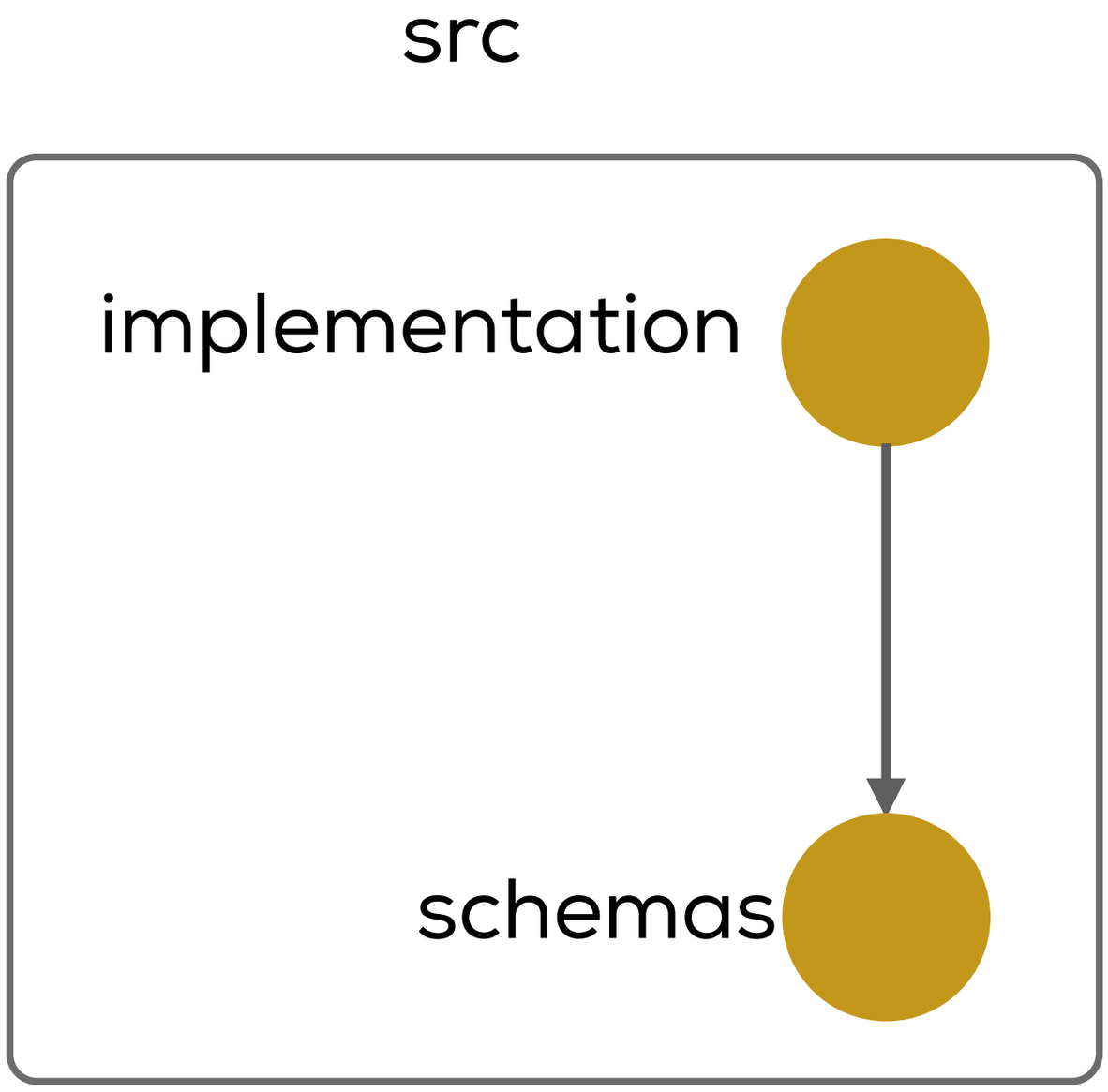
```
(s/defn fetch-events :- [Event]
  [params]
  …)
```

```
my-project.schemas

(def Event {:when DateTime
            :what Incident
            :who  Customer})
```

```
(:require
  [my-project.schemas :as t])

 (s/defn fetch-events :- [t/Event]
   [params]
   …)
```

schemas

implementation

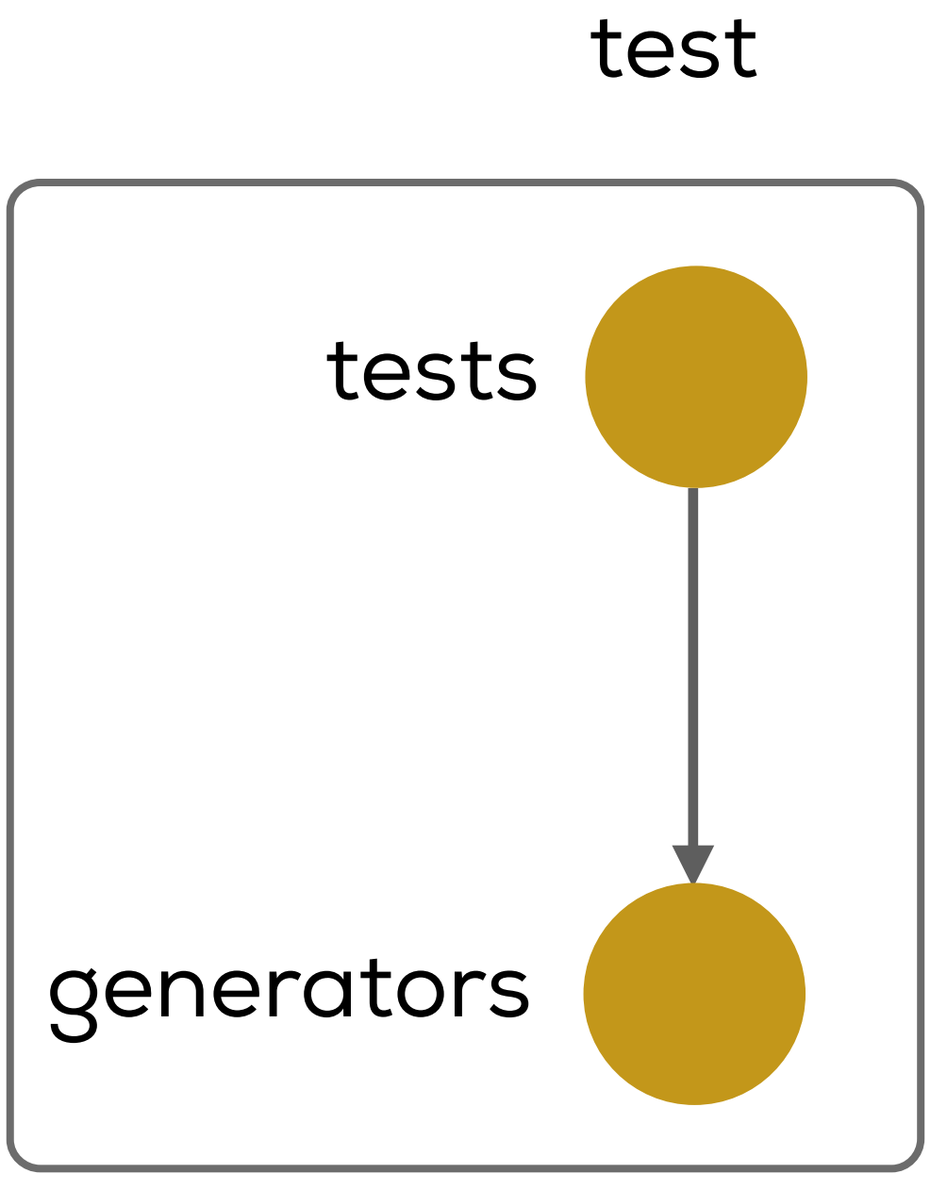

src
implementation
schemas

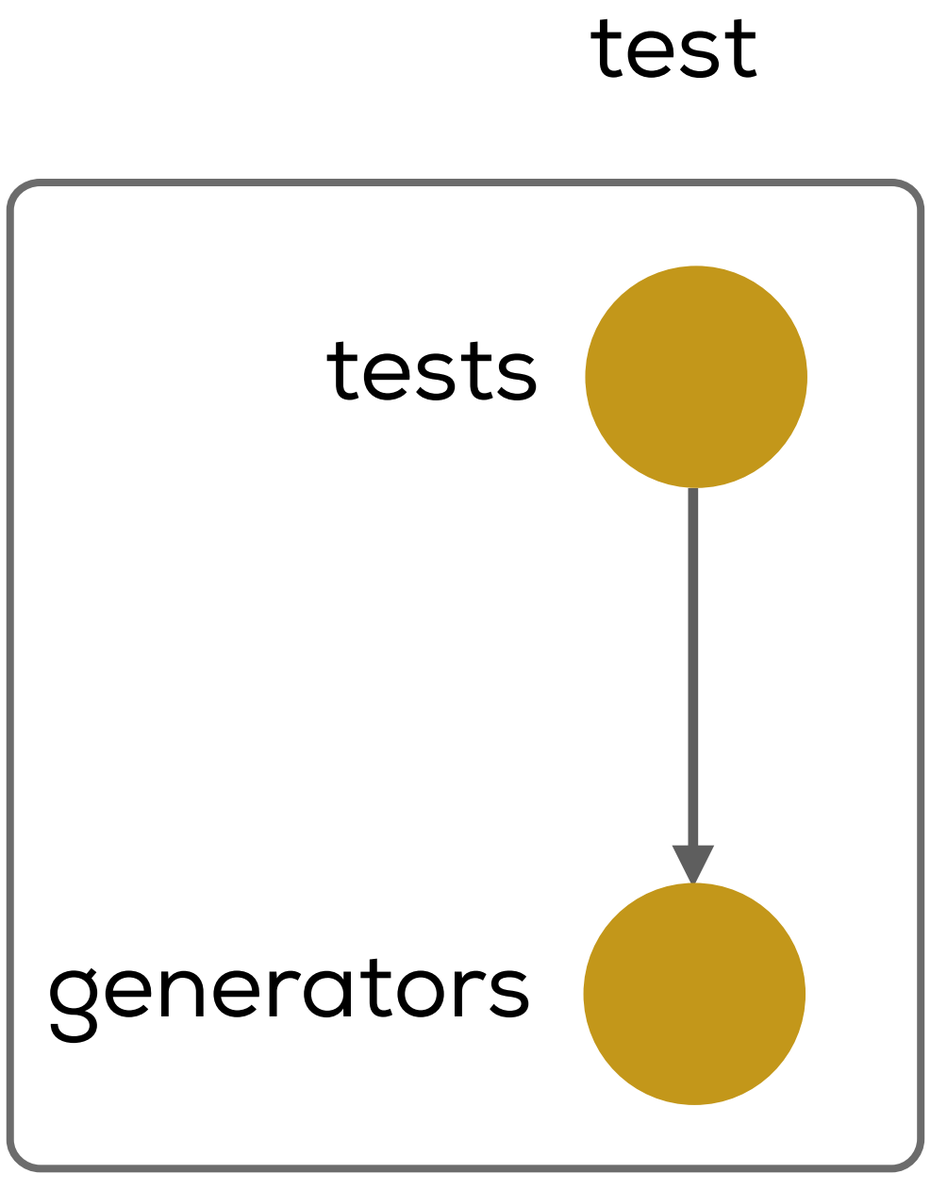
```
my-project.gen


(def params (hash-map :title g
                      :start (
                      :end (da
```

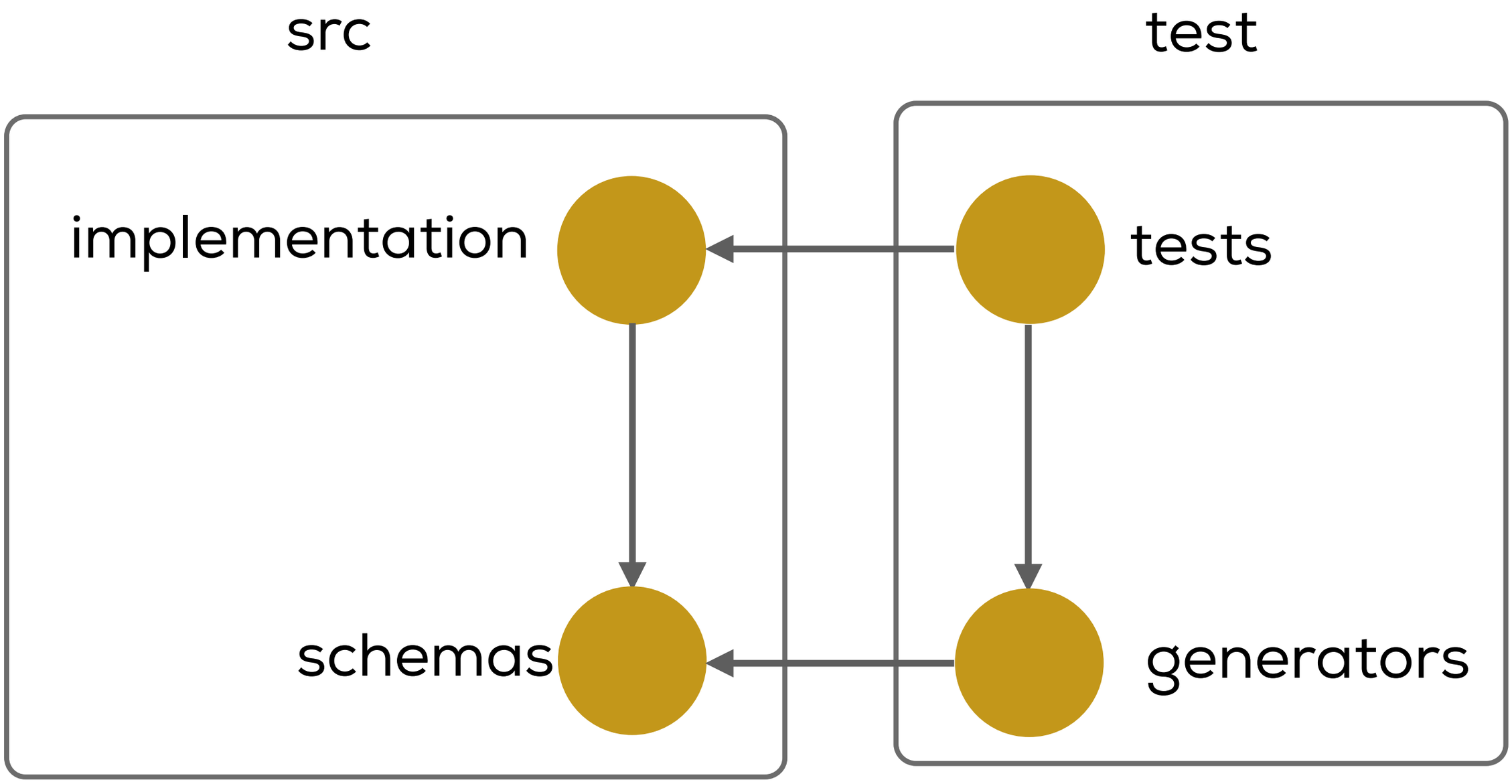
```
(:require
  [my-project.gen :as mygen])

(defspec analyze-ad-performance-sp
  (for-all [events mygen/events
                   params mygen/params]
    (analyze-ad-performance events
```

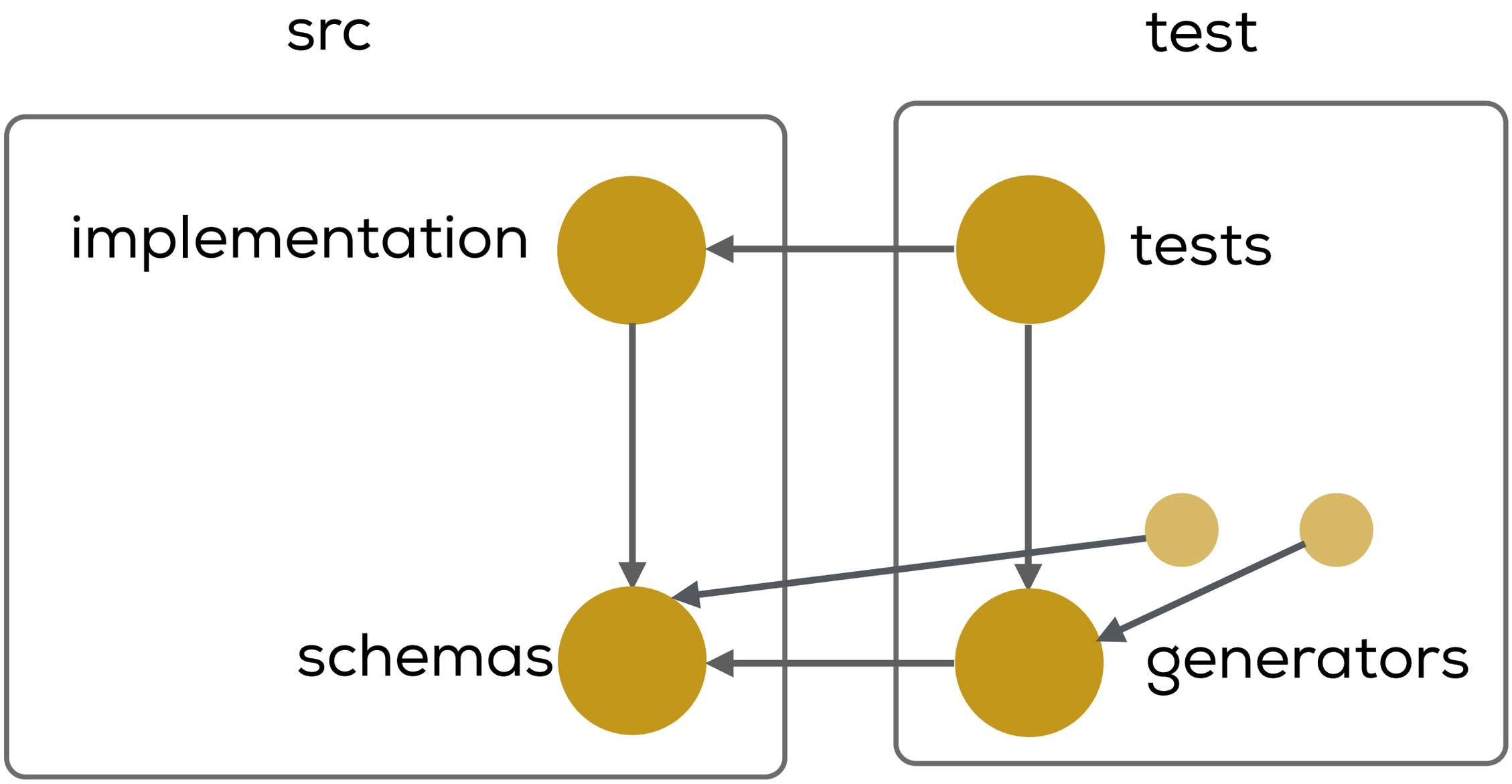
generators

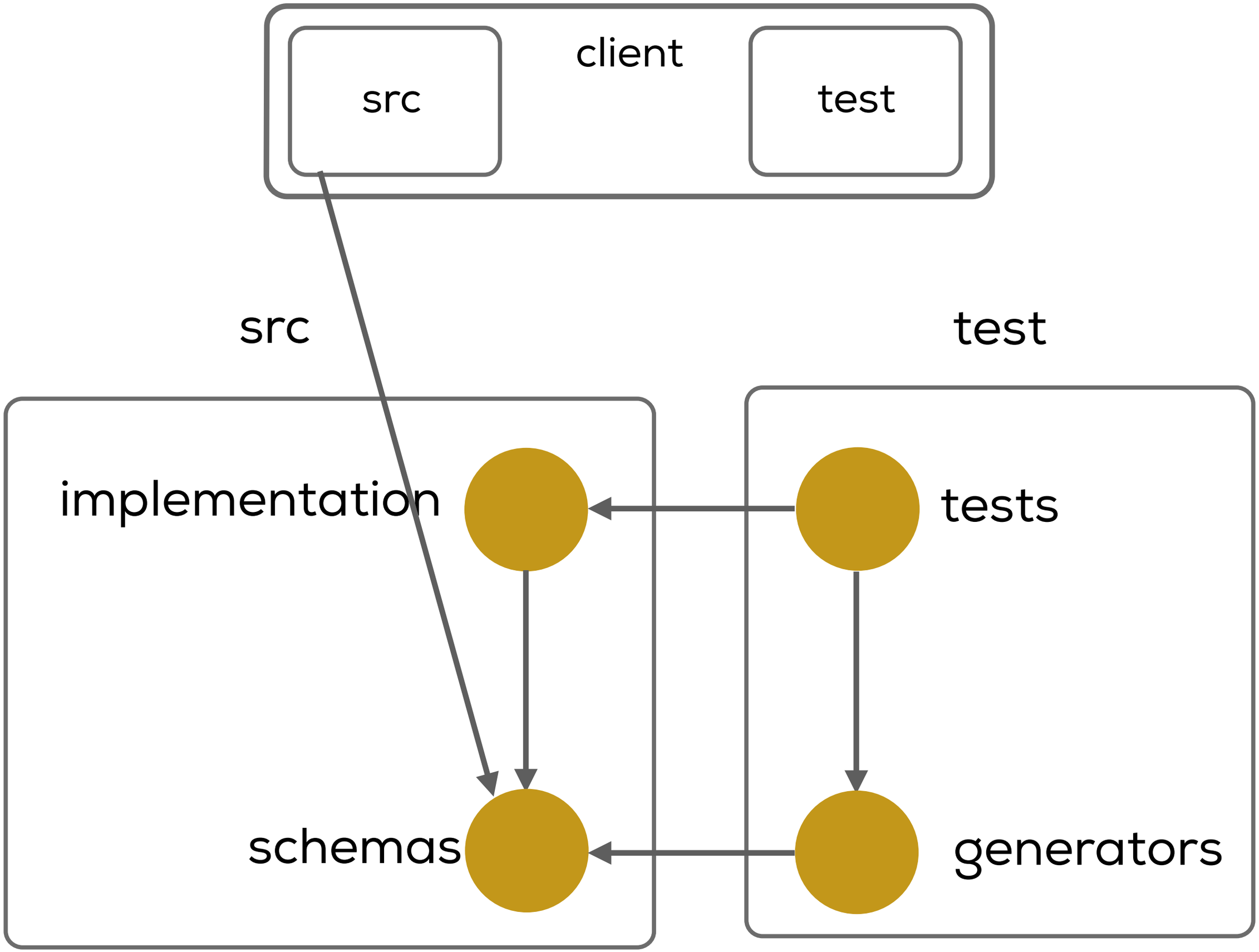
tests

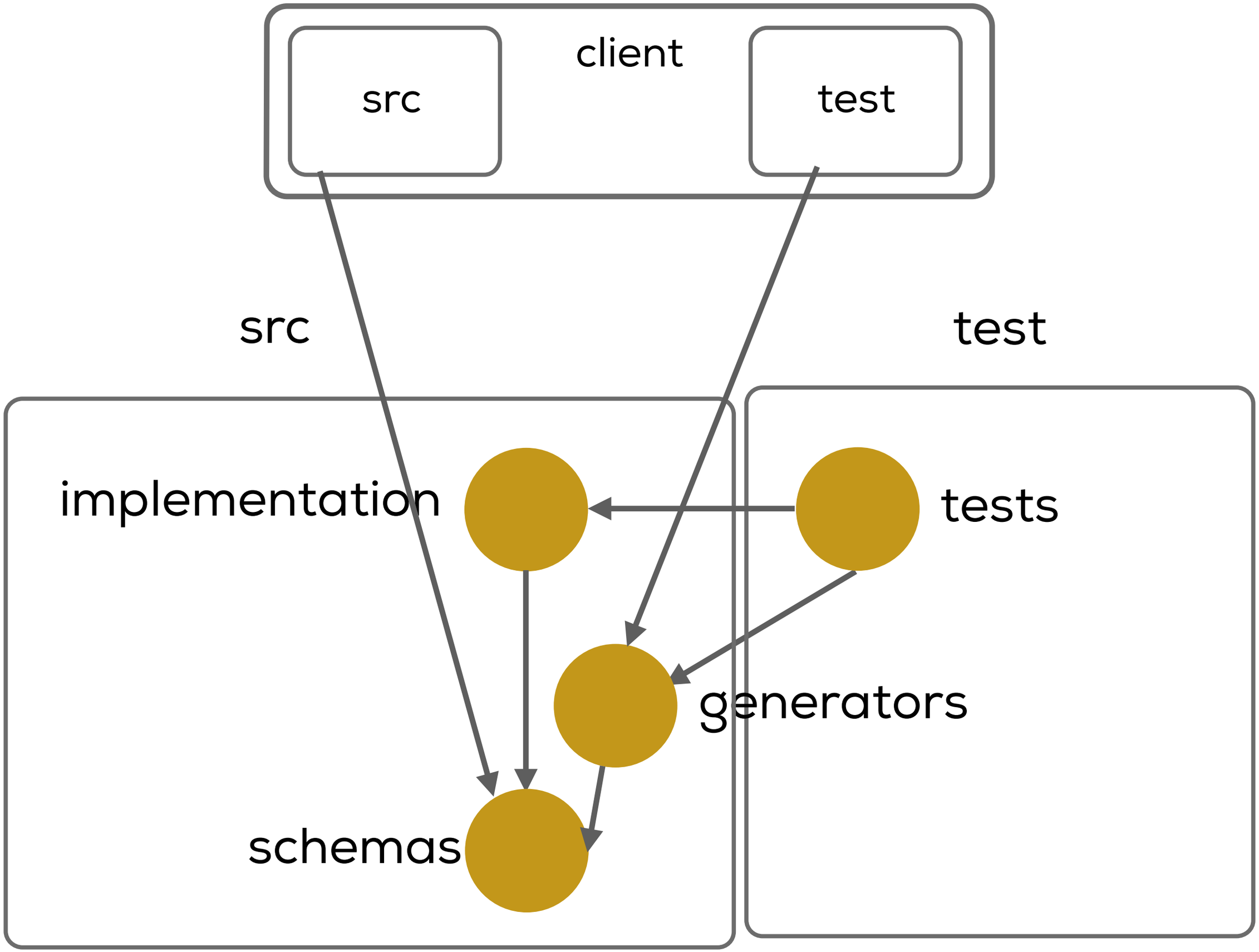

test
tests
generators

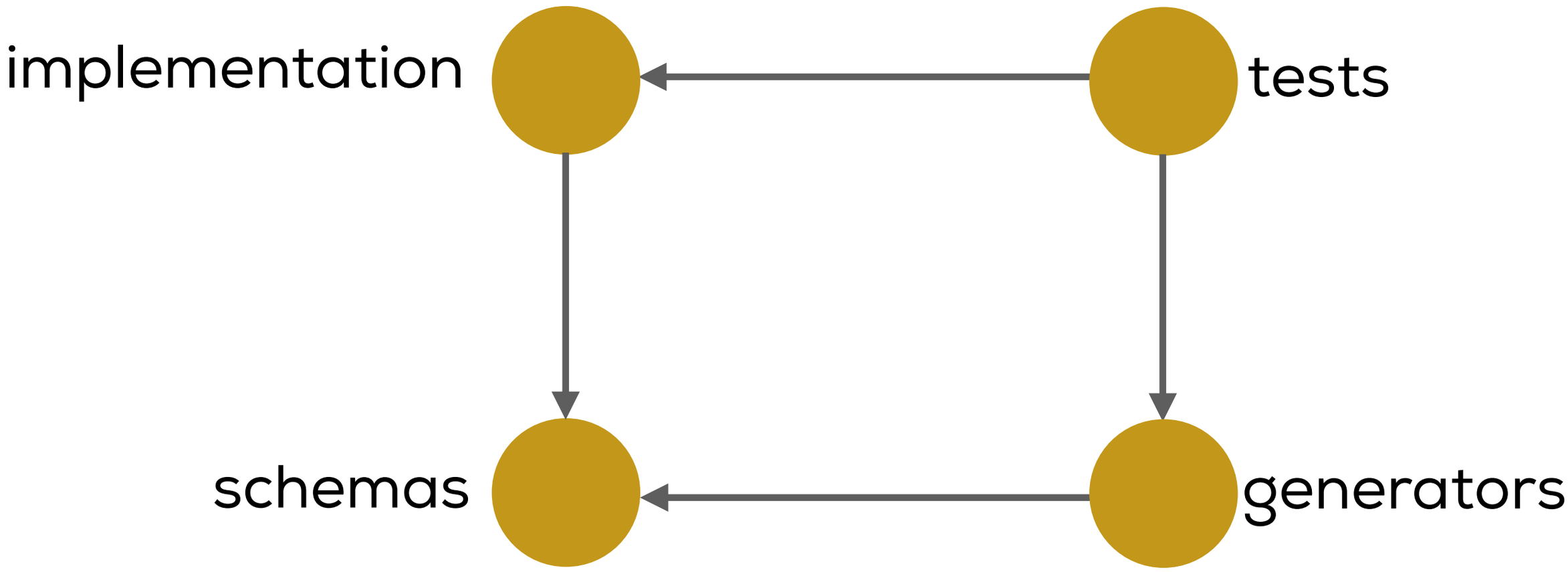

test
tests
generators

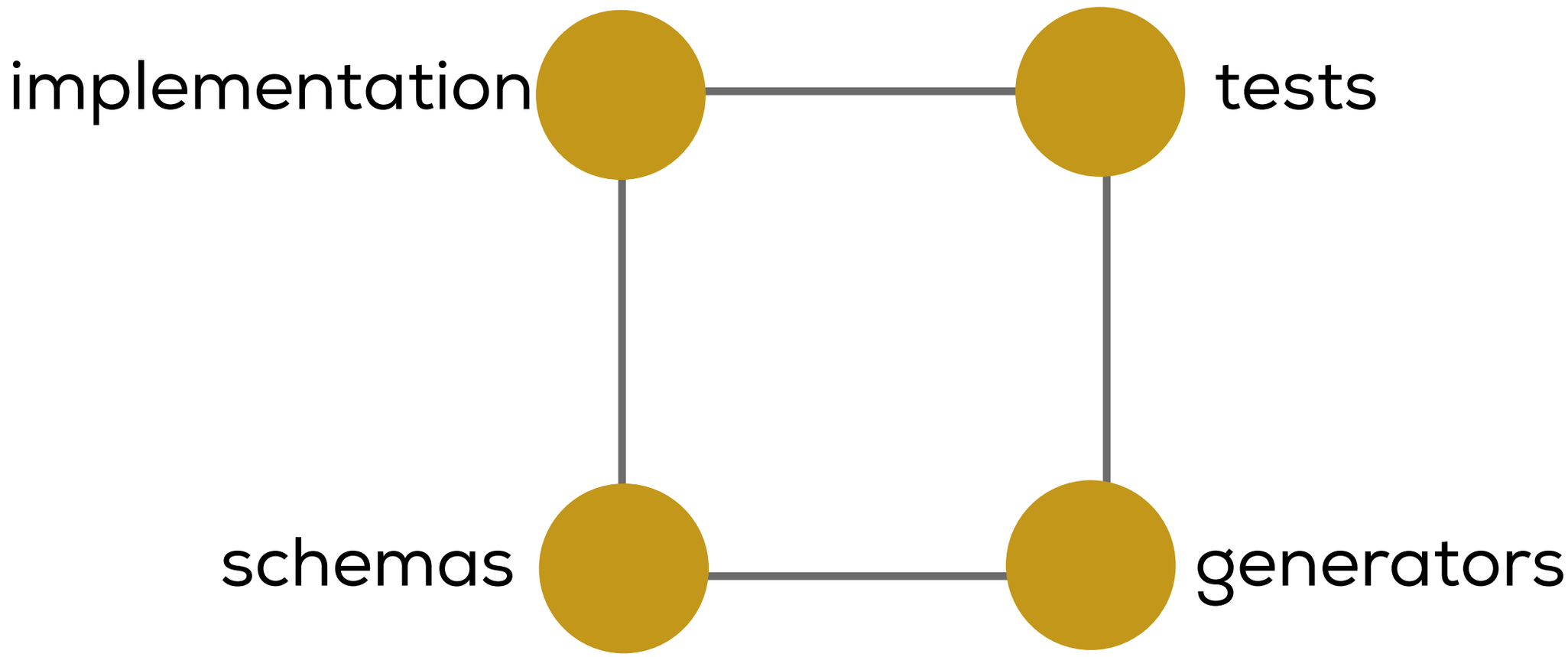

src
test
implementation
tests
schemas
generators

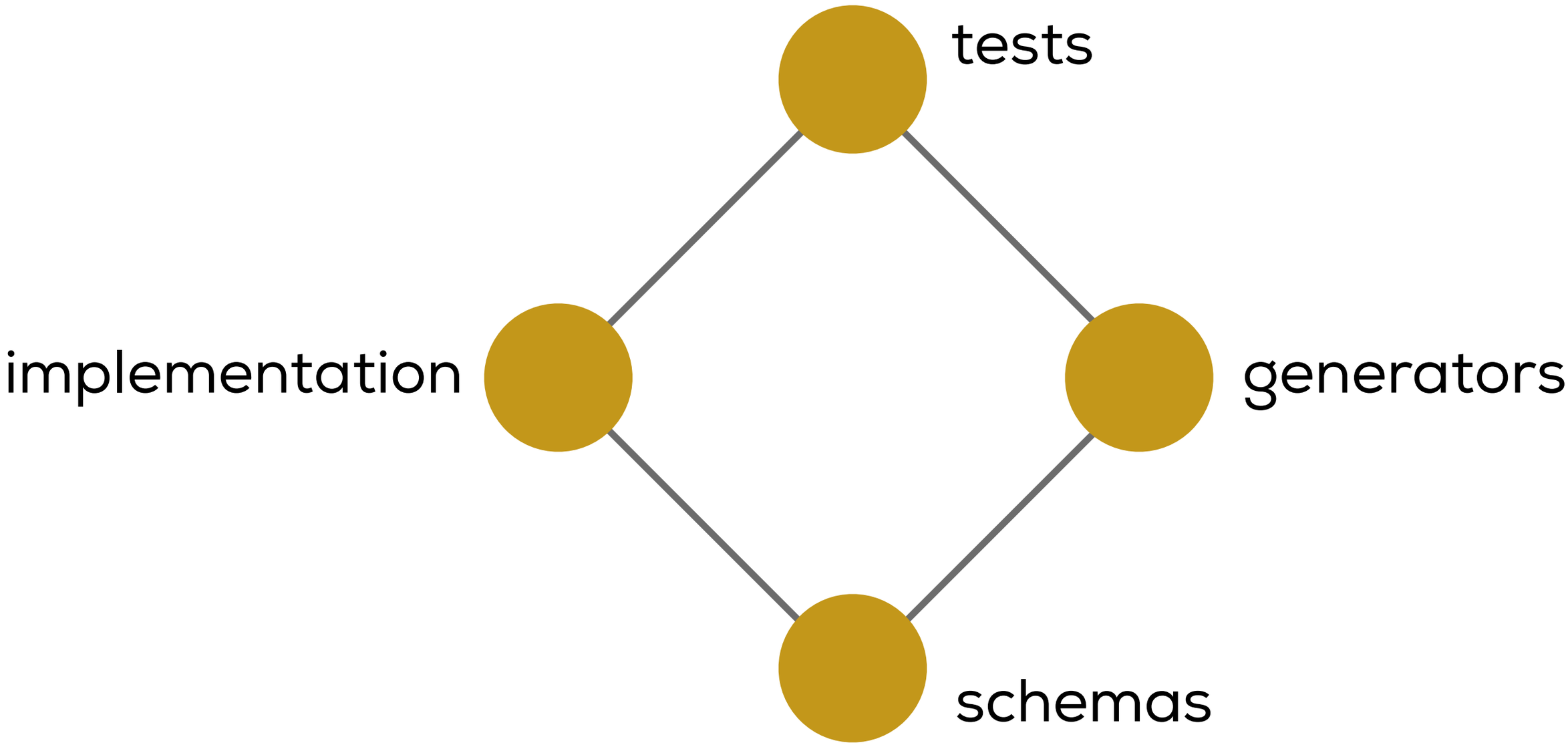

src
test
implementation
tests
schemas
generators

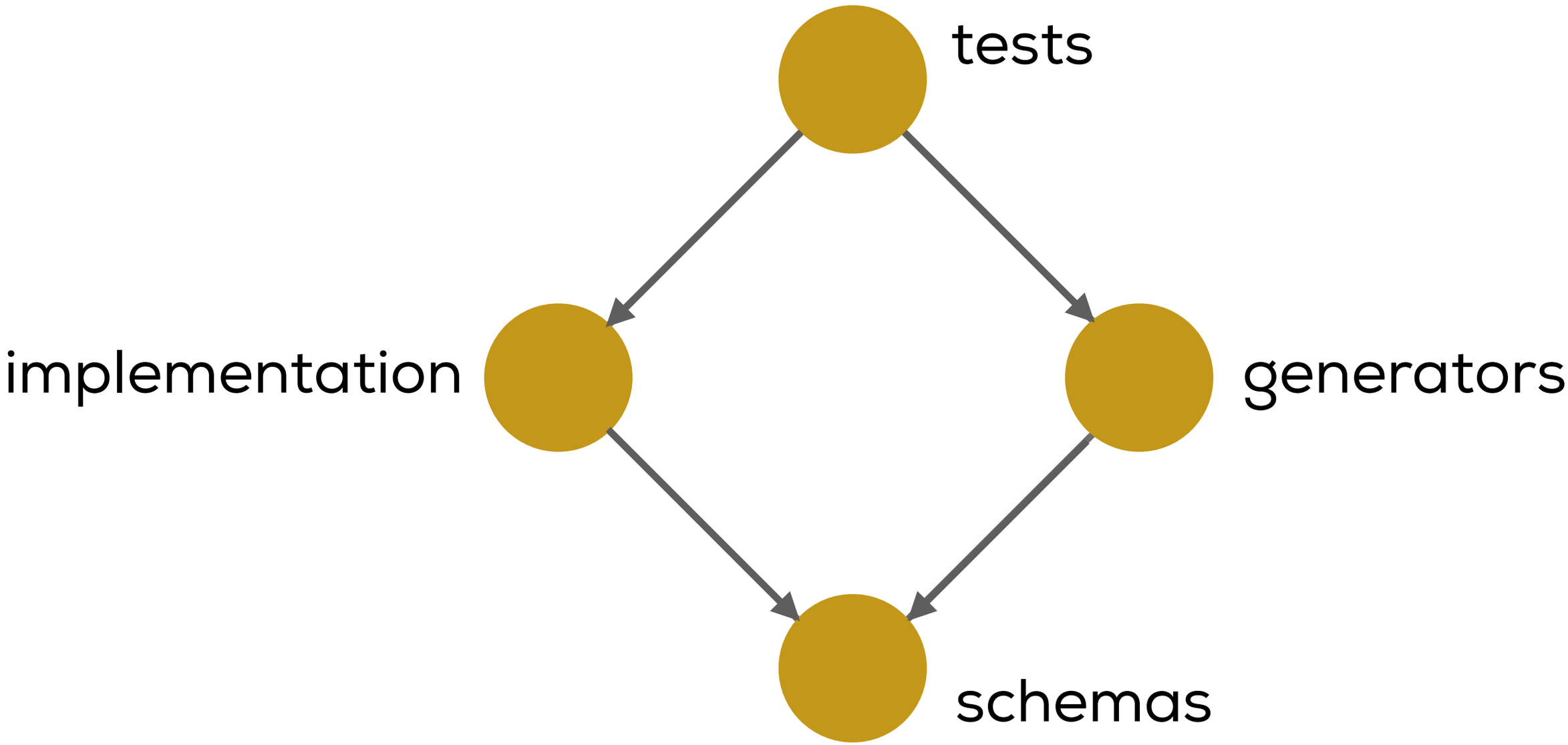

client
src
test
src
test
implementation
tests
schemas
generators

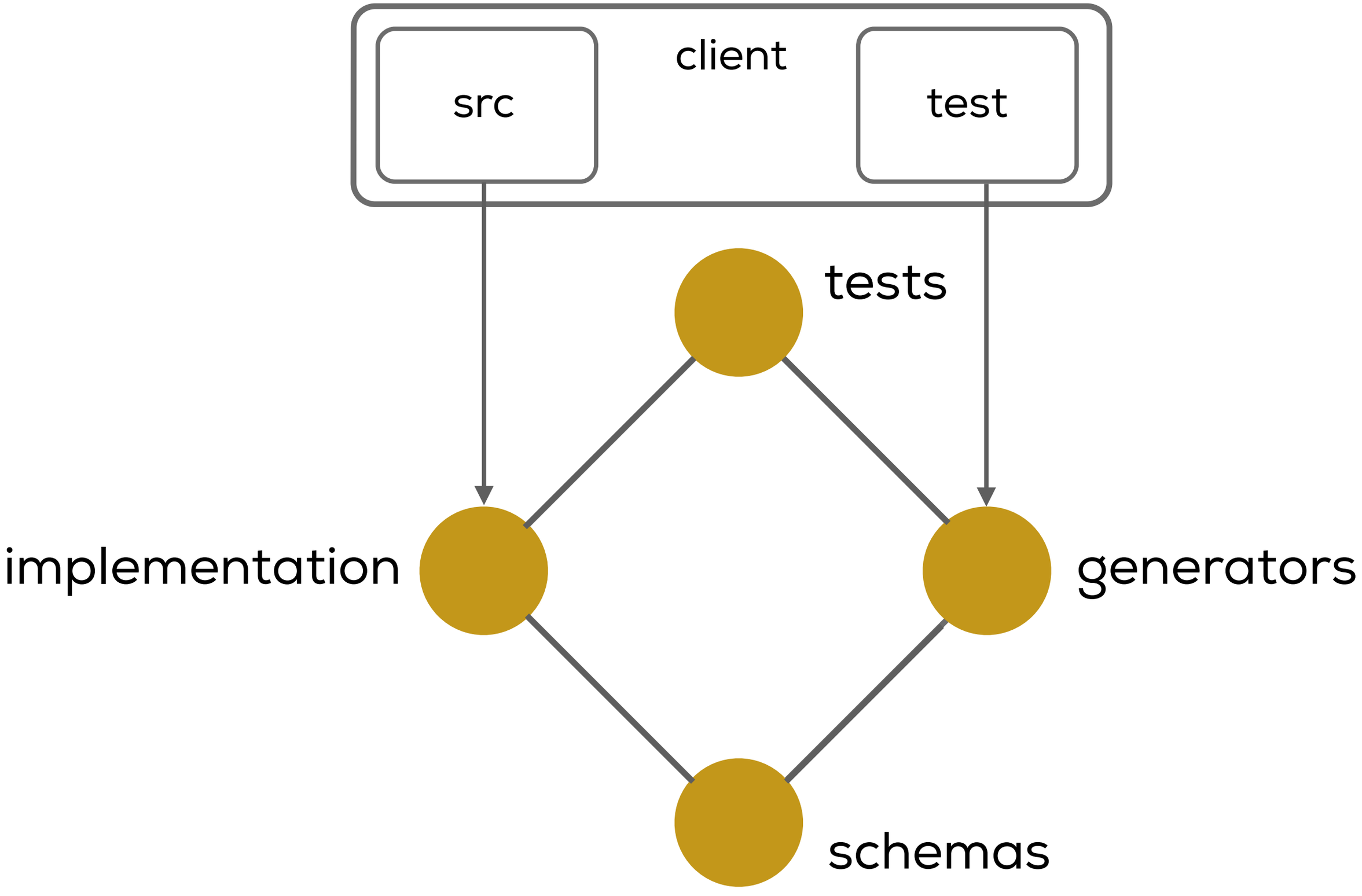

client
src
test
src
test
implementation
tests
generators
schemas

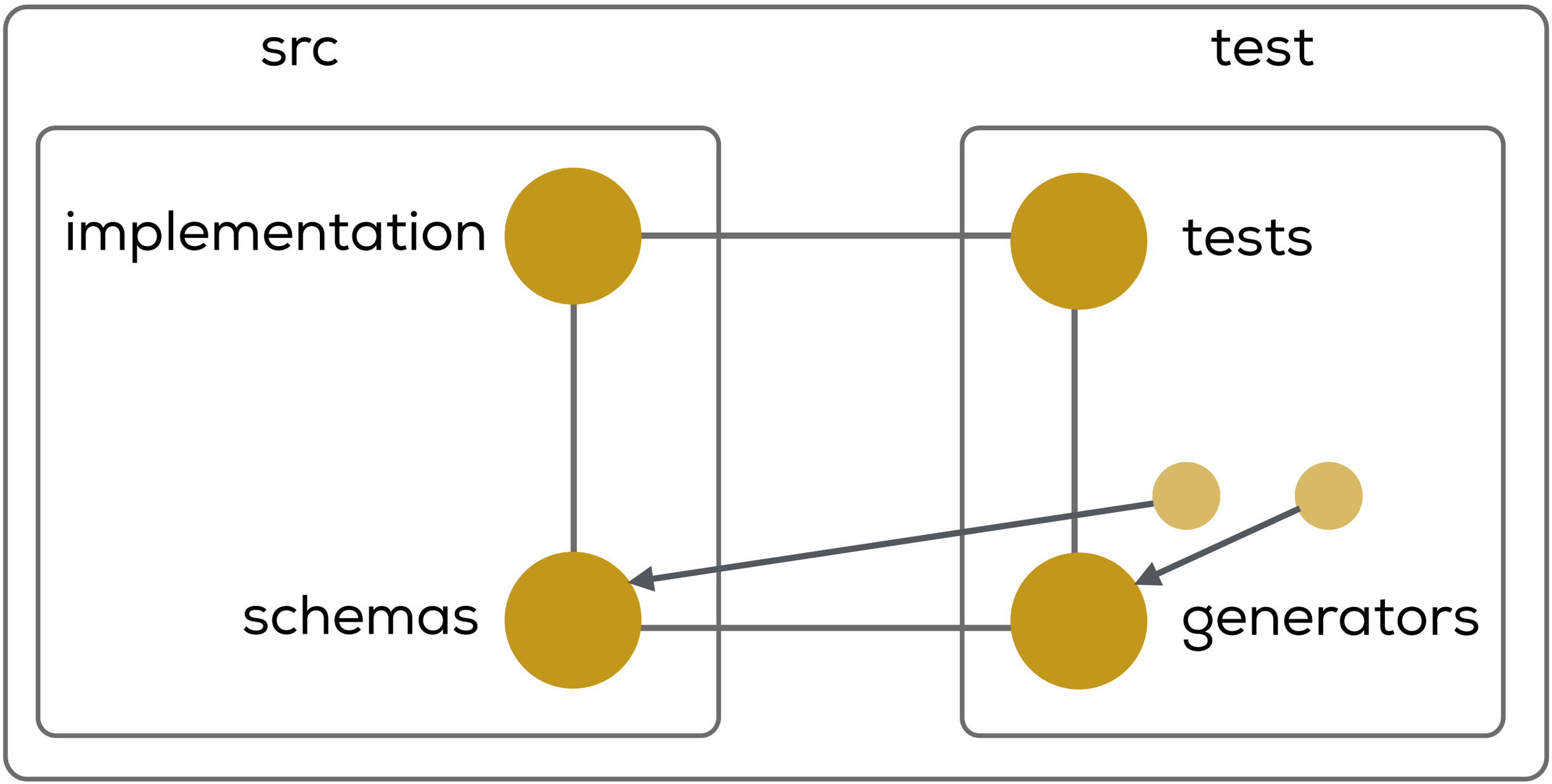

implementation
tests
schemas
generators

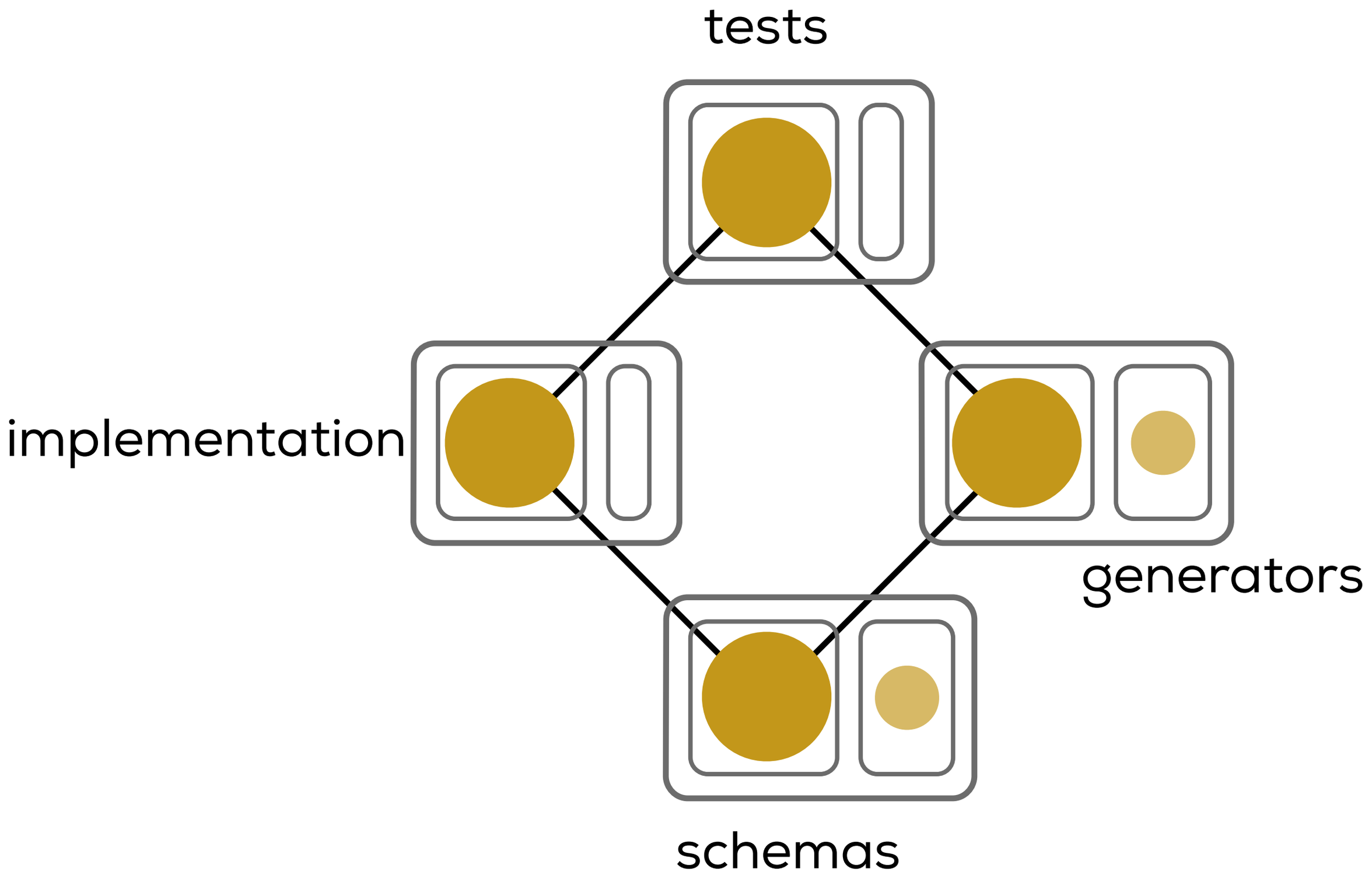

implementation
tests
schemas
generators

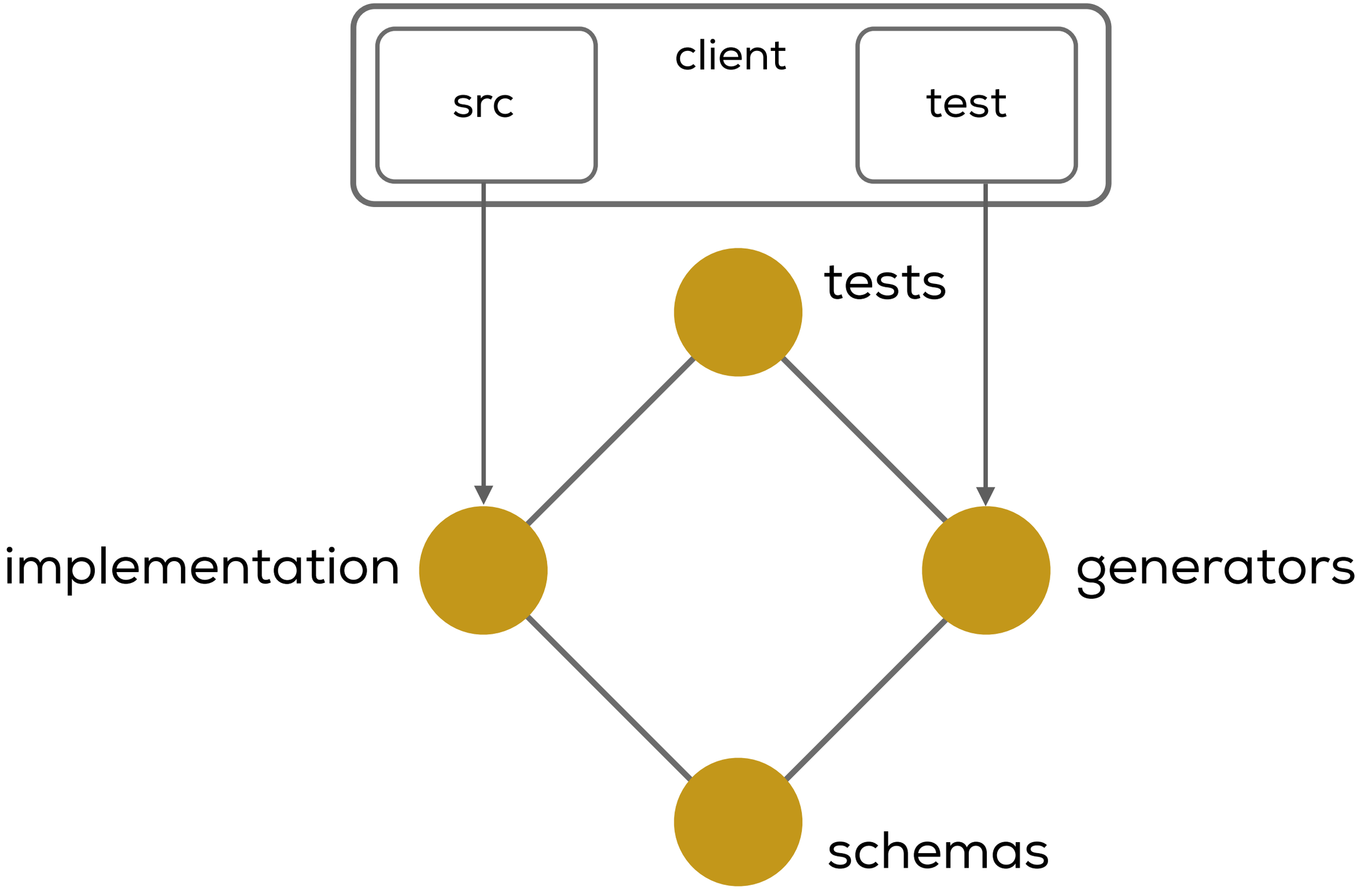

tests
implementation
generators
schemas

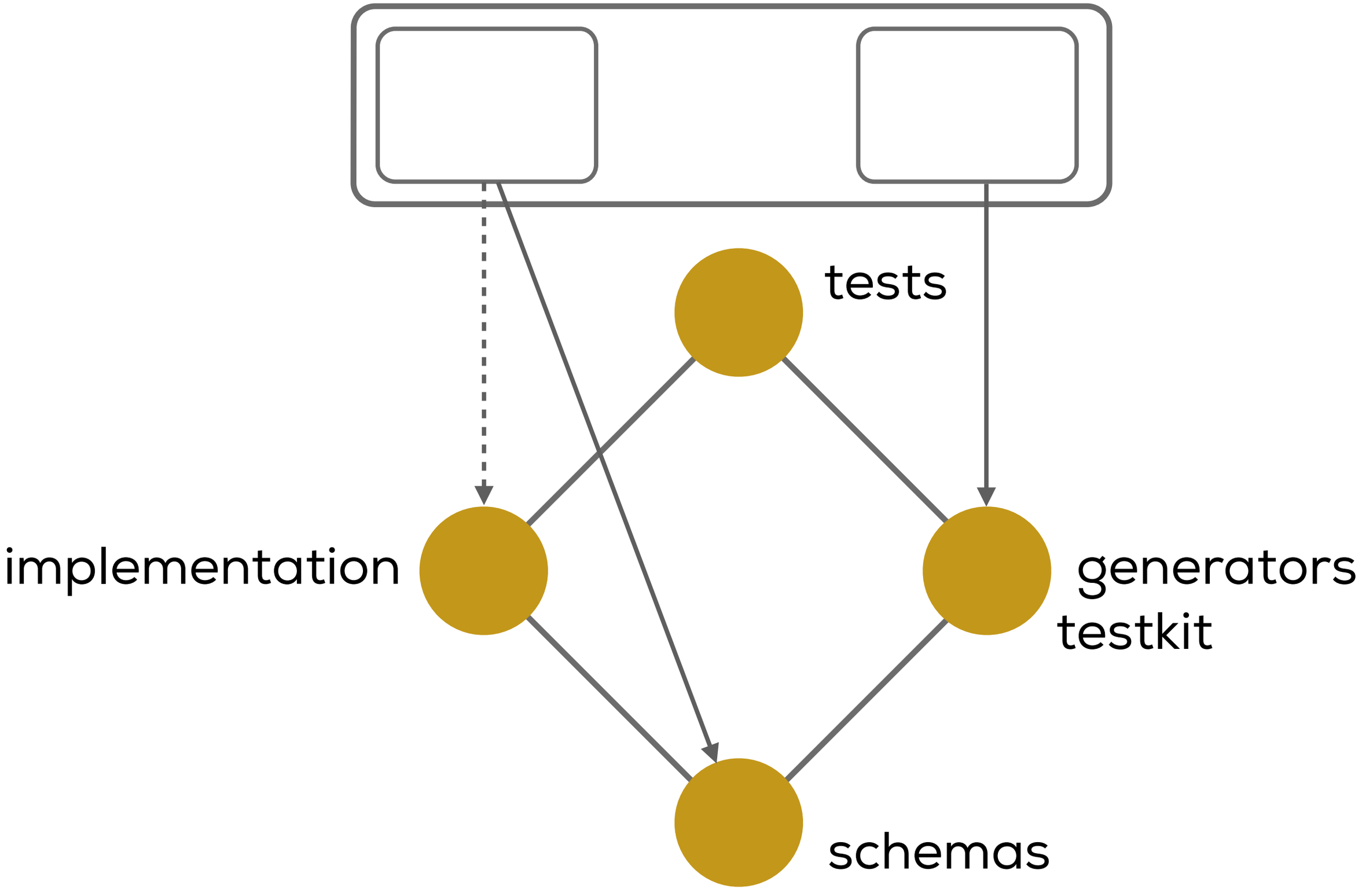

tests
implementation
generators
schemas

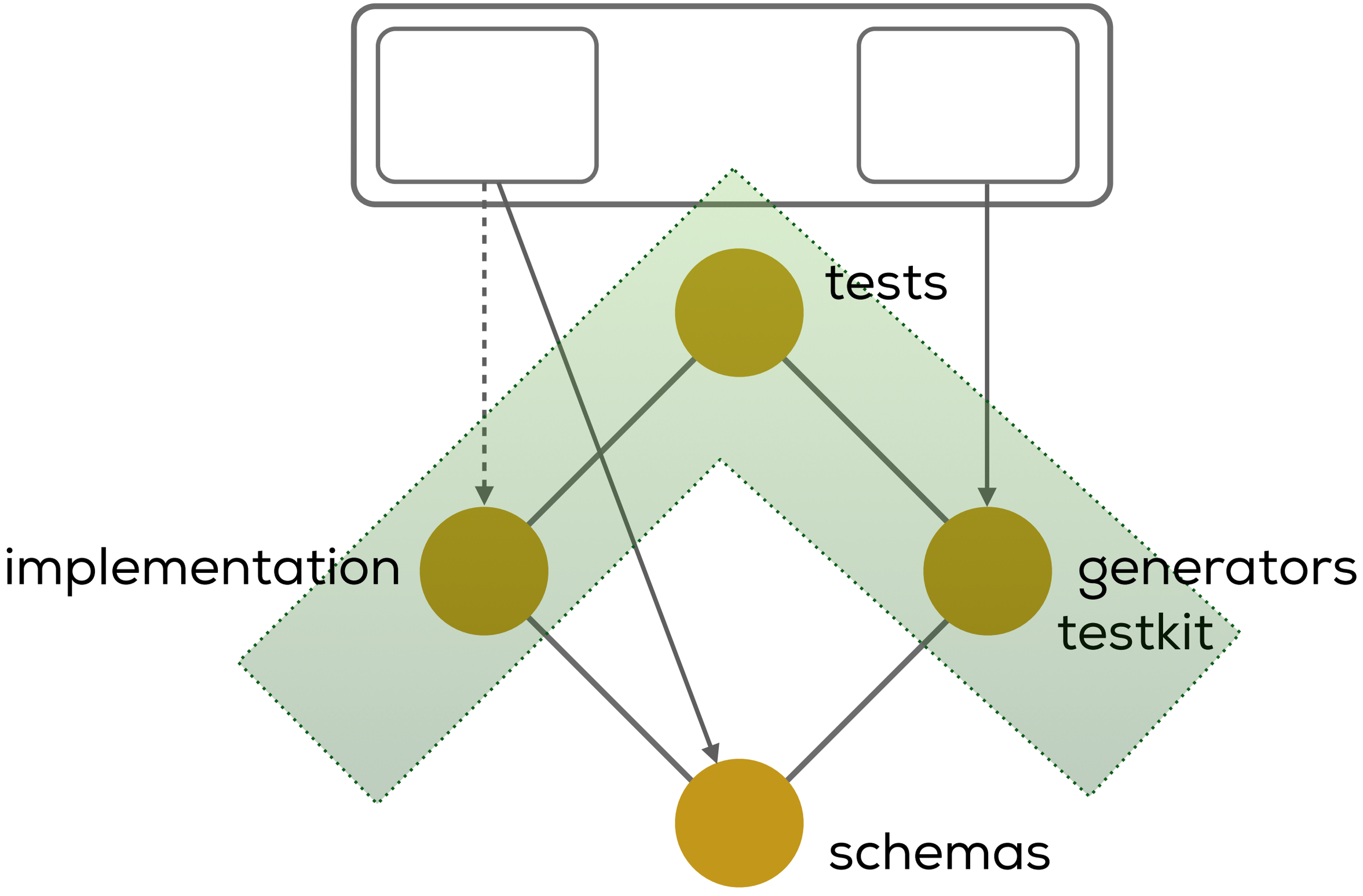

client
src
test
tests
implementation
generators
schemas

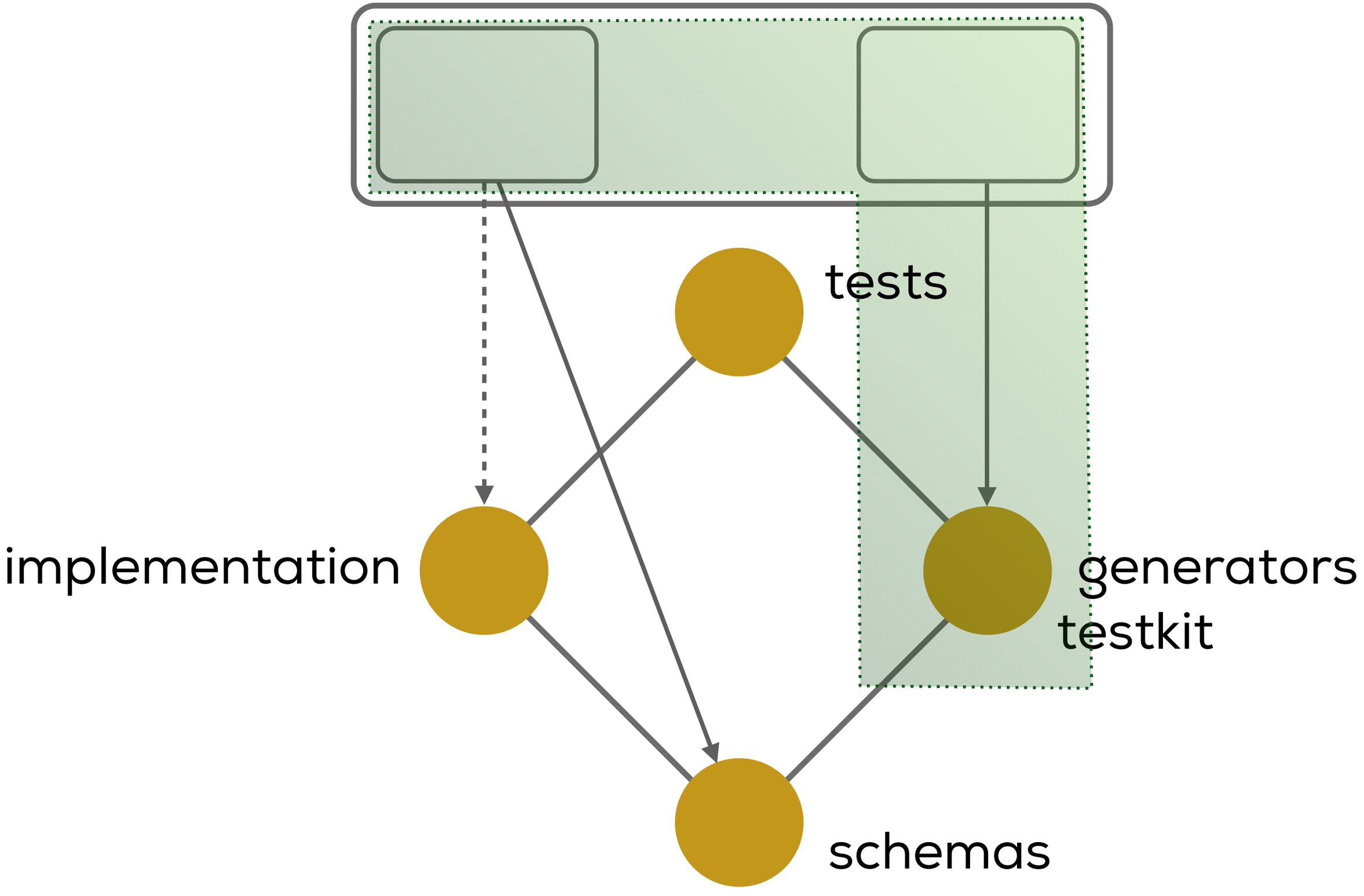

src
test
implementation
tests
schemas
generators

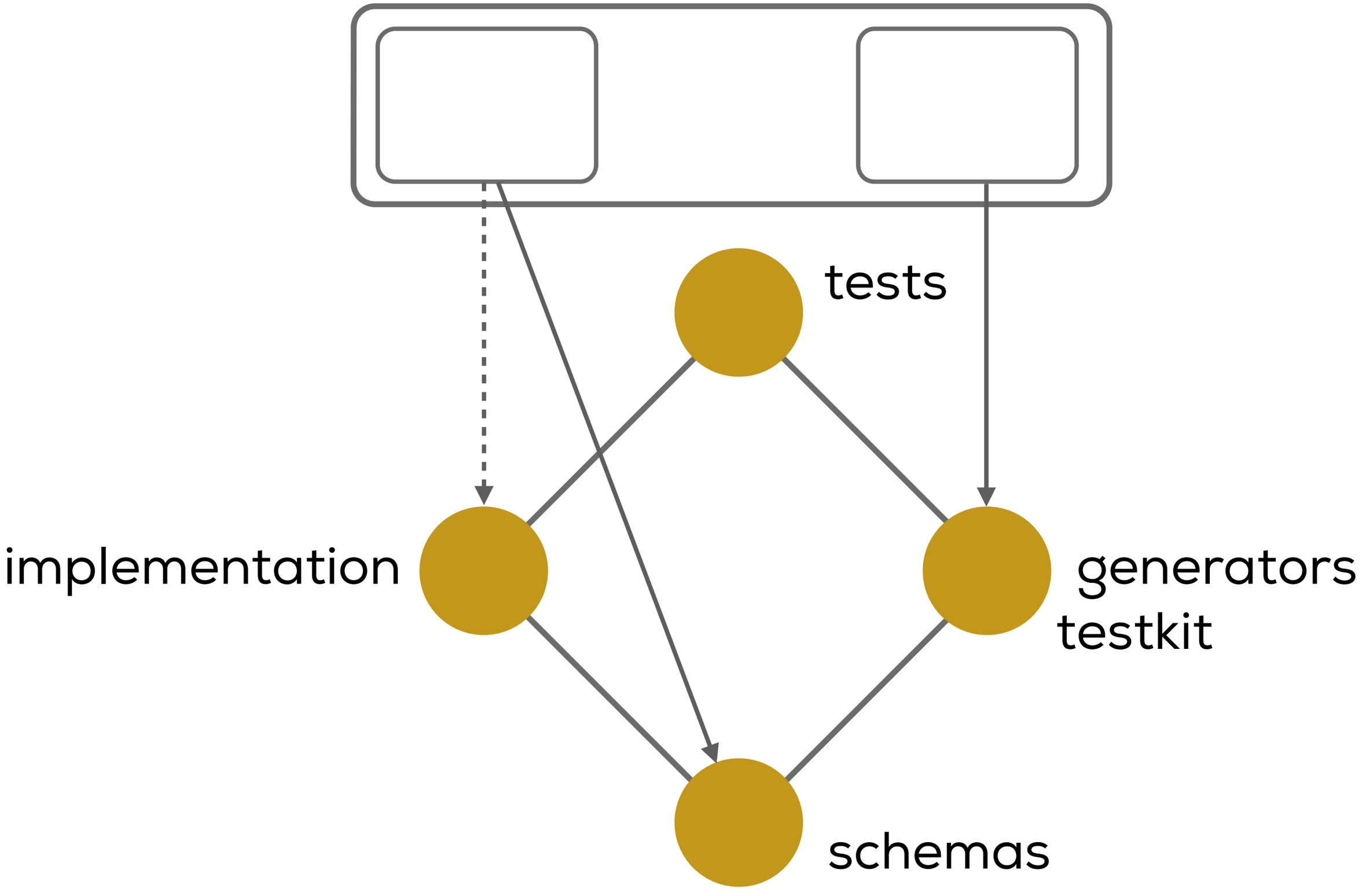

tests
implementation
generators
schemas

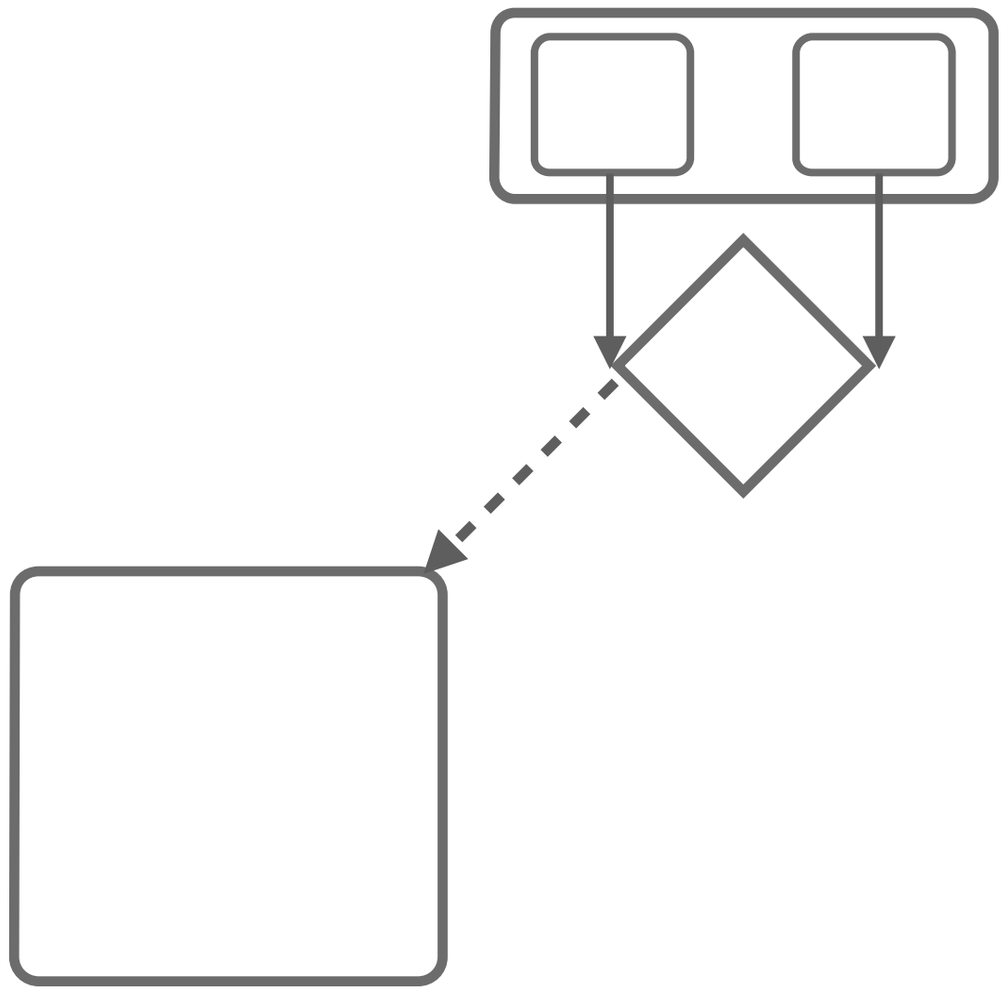

client
src
test
tests
implementation
generators
schemas

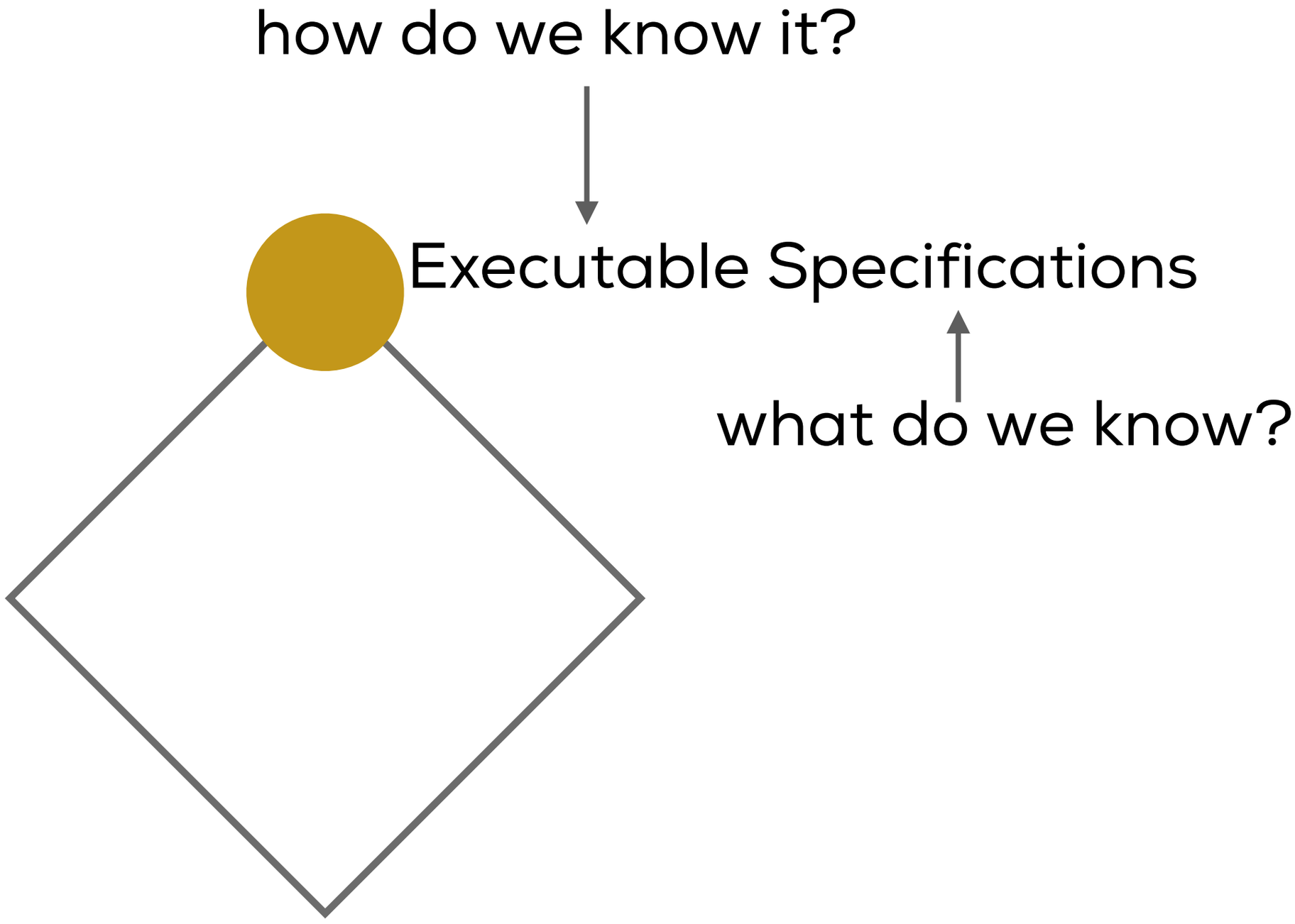

tests
implementation
generators
testkit
schemas


tests
implementation
generators
testkit
schemas


tests
implementation
generators
testkit
schemas


tests
implementation
generators
testkit
schemas




how do we know it?
Executable Specifications
what do we know?

Clojure

prismatic/schema

test.check

Science!

... your language ...

types and contracts

generative tests

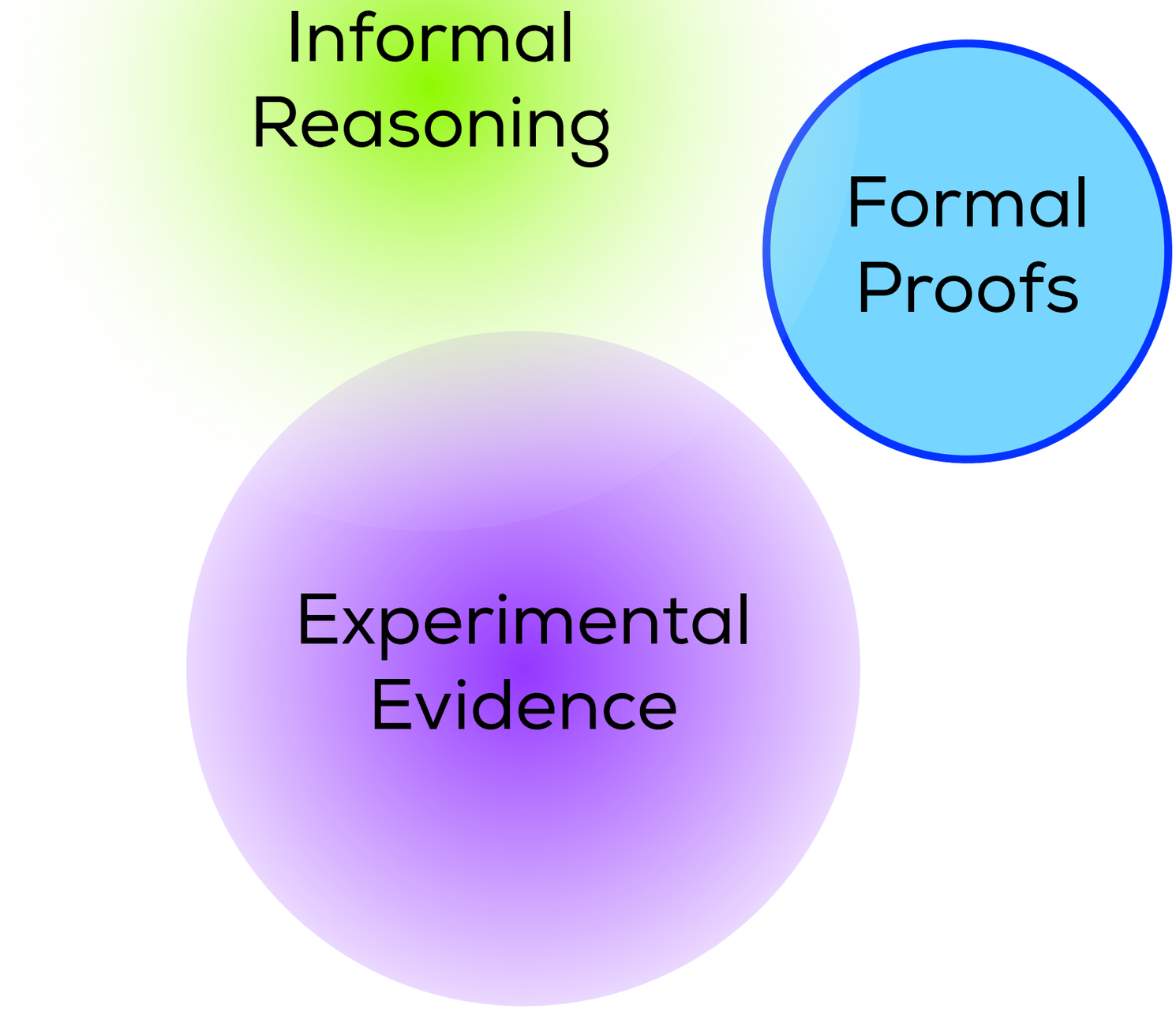
Science!

... your language ...

native API definitions

in-memory test tools

Science! for everyone!


Informal
Reasoning
Formal
Proofs
Experimental
Evidence

examples

https://github.com/jessitron/contracts-as-types-examples

https://github.com/jessitron/slack-client

https://github.com/jessitron/schematron

resources

https://github.com/Prismatic/schema

https://github.com/miner/herbert (value relationships)

http://david-mcneil.com/post/114783282473/extending-prismatic-schema-to-higher-order

Static typing and productivity: Stefik & Hanenberg 2014

http://dl.acm.org/citation.cfm?id=2661156

@jessitron

blog.jessitron.com