Agility & Robustness: Clojure spec

brought to you by

cognitect

Plan

why spec?

creating specs

leveraging specs

What I Want

correct: “free from error”

agile: “able to move quickly and easily”

robust: “able to withstand or overcome adverse conditions”

How

Goal	Industry Practice
correct	types example tests
agile	encapsulation IDE refactoring local concision
robust	TBD?

Clojure Power

Goal	Industry Practice
correct	types example tests
agile	~~encapsulation~~ IDE refactoring local concision
robust	

+

Clojure Power
pure functions state, flow systemic generality
simplicity systemic generality
immutable data systemic generality

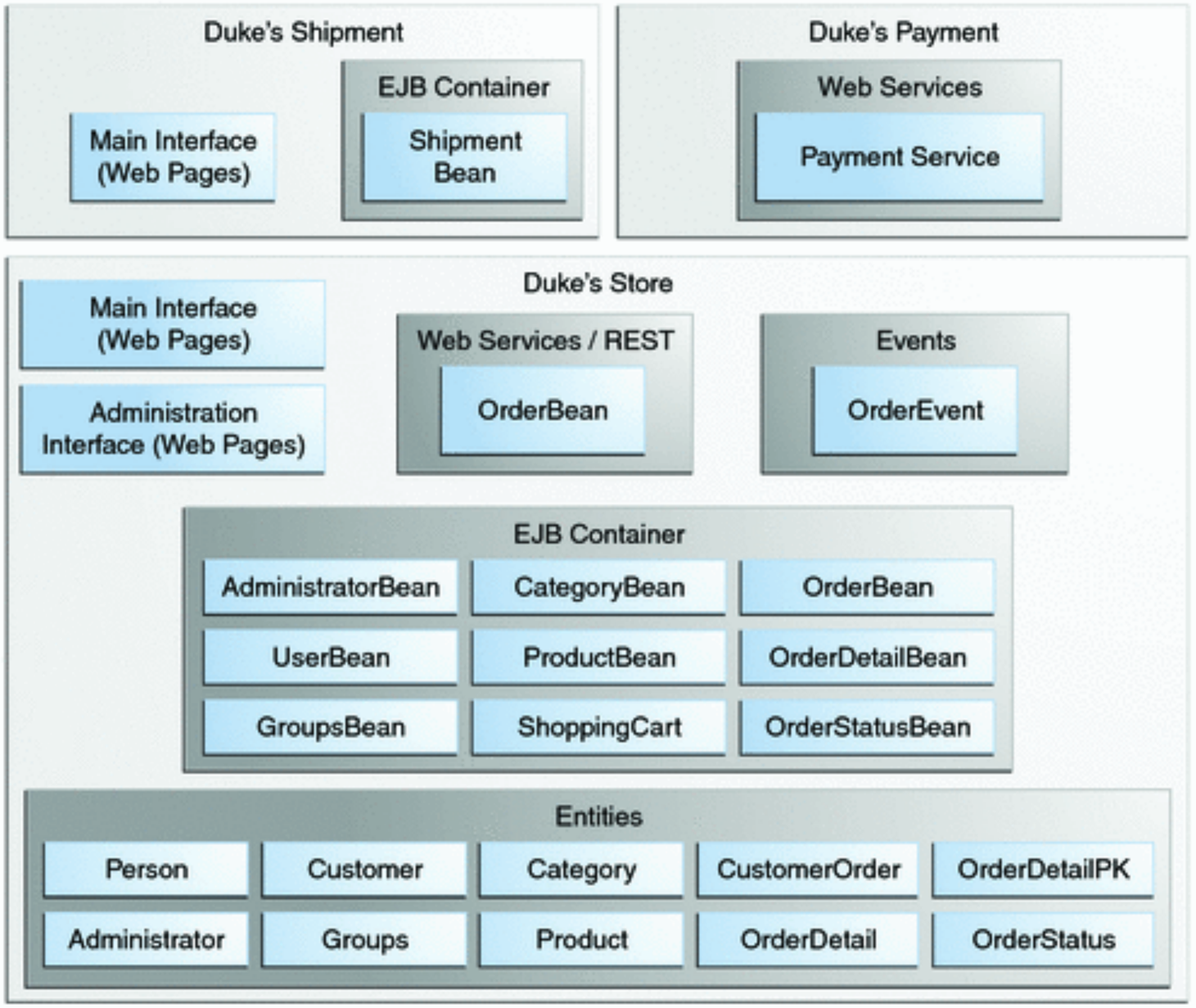
Systemic Generality

generality

all domains use the same general-purpose data structures and functions

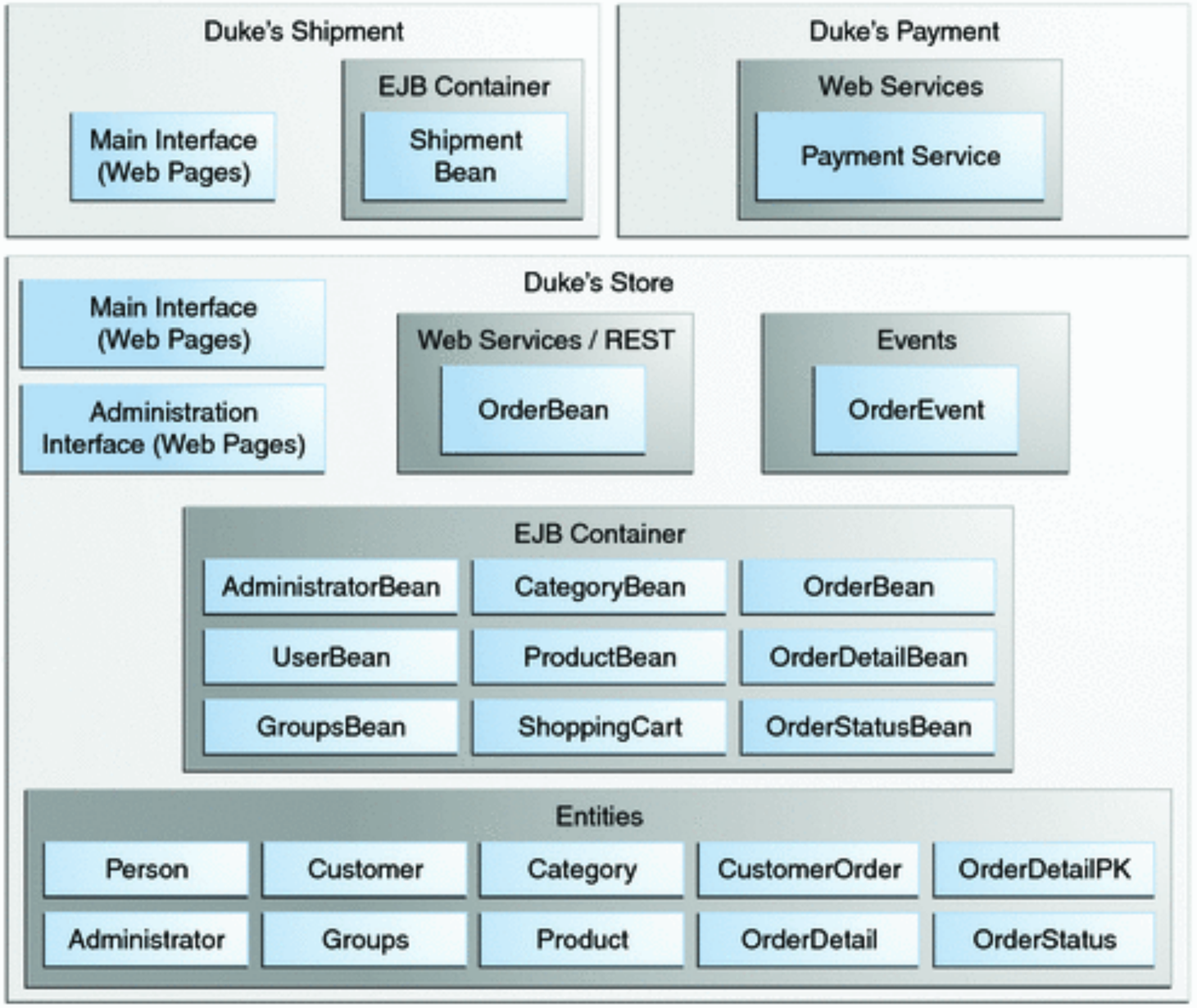
systemic

in libs, in apps, in config, on wires, at rest



Death By Specificity

http://docs.oracle.com/javaee/6/tutorial/doc/figures/dukesforest-architecture.gif

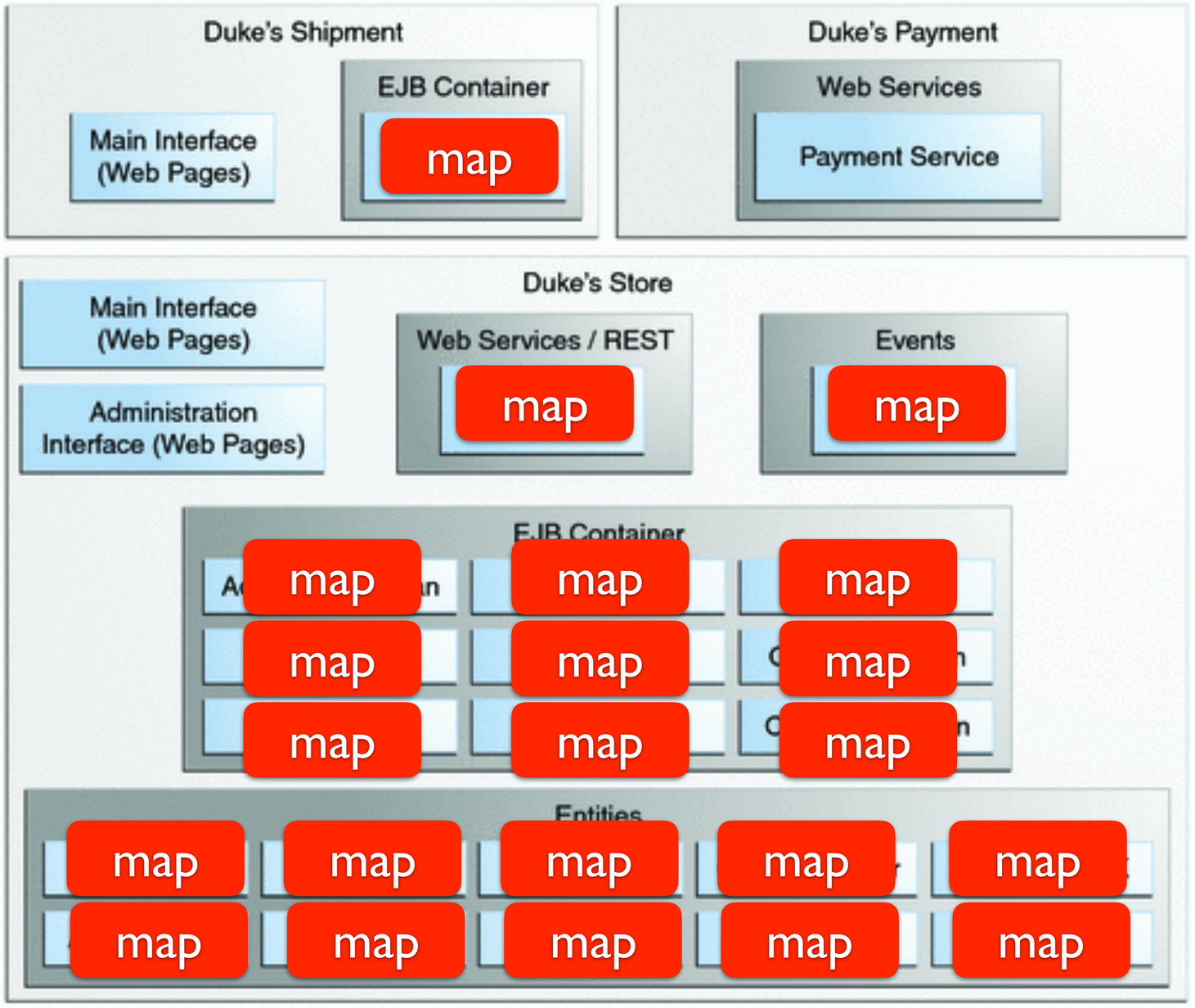


Death By Specificity

property files
JSON
XML config

servlet API entities
data API entities
config API entities

http://docs.oracle.com/javaee/6/tutorial/doc/figures/dukesforest-architecture.gif



maps
maps
maps

maps
maps
maps

http://docs.oracle.com/javaee/6/tutorial/doc/figures/dukesforest-architecture.gif

Information

Paradigm	Examples	Representation	Correctness	Reach
“Enterprise OO”	Java, C#, C++	encapsulated specificity	types	libs, apps
“Agile Scripting”	Ruby, Python, JavaScript	encapsulated specificity	tests	libs, apps
	Clojure	systemic generality	functional	systemic

Pretty Good!

"A lot of the best programmers and the most productive programmers I know are writing everything in Clojure and swearing by it, and then just producing ridiculously sophisticated things in a very short time. And that programmer productivity matters."

http://thenewstack.io/the-new-stack-makers-adrian-cockcroft-on-sun-netflix-clojure-go-docker-and-more/

Challenge

discipline required to deal with specificity

is systemic generality

- scary?
- a winning tradeoff?
- winning?

Clojure spec

a standard, expressive, powerful,
integrated system for
specification and testing

spec Answers the Challenge

integrated language discipline for *a la carte* specificity

- without sacrificing generality

dynamic leverage

- anytime
- anywhere
- up to you

spec Power

Goal	Industry Practice
correct	types example tests
agile	~~encapsulation~~ IDE refactoring local concision
robust	

+

Clojure Power
pure functions state, flow systemic generality
simplicity systemic generality
immutable data systemic generality

+

spec Power
specification, predicates generative testing instrumentation
auto-documentation explanation, conformance example data
assertion validation

Correct / Agile / Robust

	Example Tests	Types	Spec
expressive	very	varying	very
powerful	stakeholder correctness	type correctness	stakeholder correctness
integrated	rare	compile-time, must flow	dynamic
specification	no	static	yes
testing	manual	rare	generative
agility	expensive	fragility	dynamic
reach	expensive	libs, apps	systemic

spec Leverage

What	How
what are the building blocks?	declarative structure
what invariants hold?	arbitrary predicates
how do I check?	validation
what went wrong?	explanation
what went right?	conformance
docs please	autodoc
examples please	sample generator
am I using this right?	instrumentation
is my code correct?	generative testing
can I recombine pieces like this?	assertion

Clojure

Data

```
{ :firstName "John"
  :lastName "Smith"
  :age 25
  :address {
    :streetAddress "21 2nd Street"
    :city "New York"
    :state "NY"
    :postalCode "10021" }
  :phoneNumber
    [ {:type "name" :number "212 555-1234"}
      {:type "fax" :number "646 555-4567" } ] }
```

Data Literals

type	examples
string	`"foo"`
character	`\f`
integer	`42, 42N`
floating point	`3.14, 3.14M`
boolean	`true`
nil	`nil`
symbol	`foo, +`
keyword	`:foo, ::foo`

type	properties	examples
list	sequential	`(1 2 3)`
vector	sequential and random access	`[1 2 3]`
map	associative	`{:a 100 :b 90}`
set	membership	`#{:a :b}`

Programs Are Data

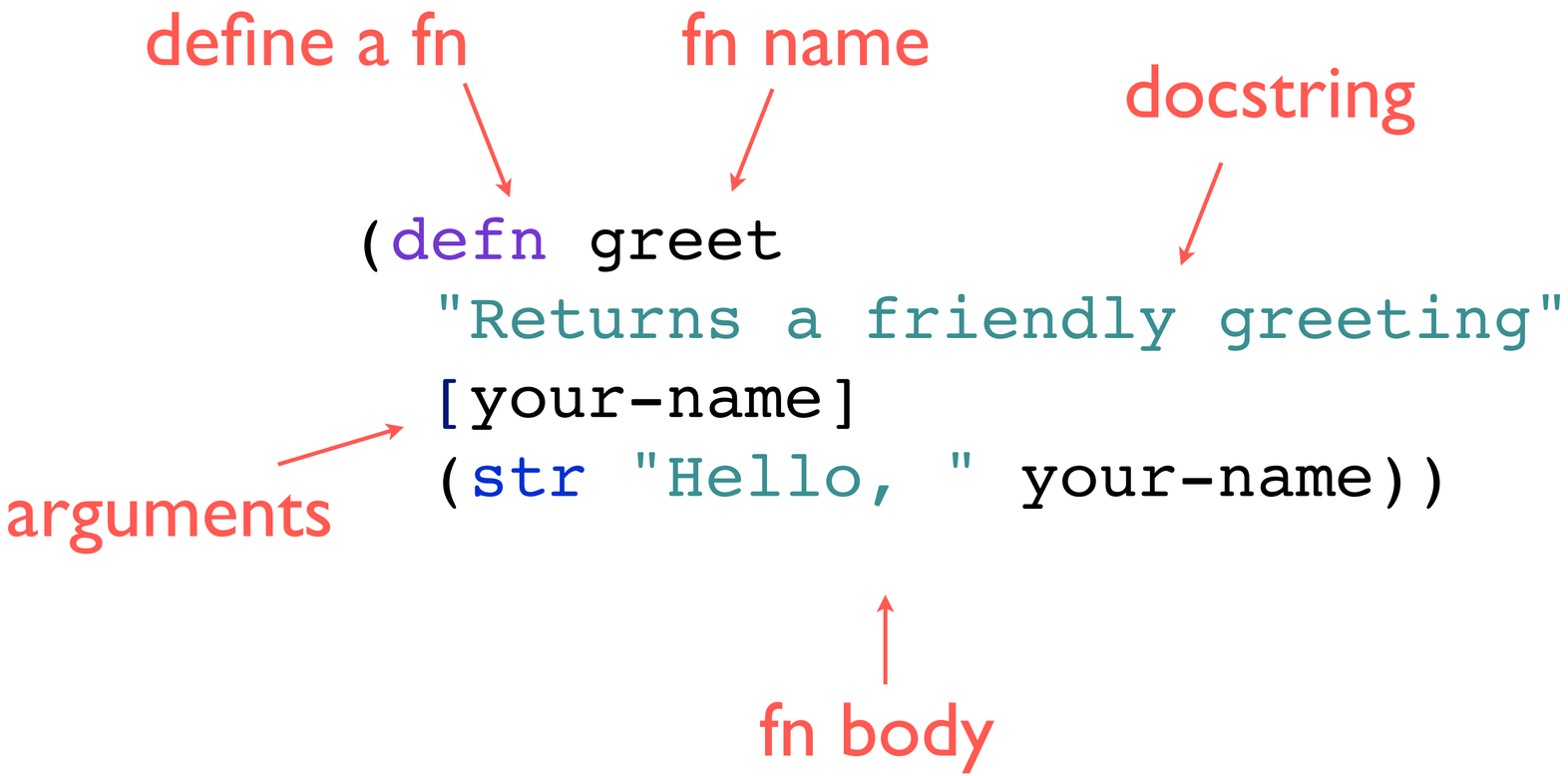
type	examples
string	`"foo"`
character	`\f`
integer	`42, 42N`
floating point	`3.14, 3.14M`
boolean	`true`
nil	`nil`
symbol	`foo, +`
keyword	`:foo, ::foo`

type	properties	examples
list	sequential	`(1 2 3)`
vector	sequential and random access	`[1 2 3]`
map	associative	`{:a 100 :b 90}`
set	membership	`#{:a :b}`

symbols point to stuff
lists are function calls

Function Call

Function Definition



Declaration

What	How
what are the building blocks?	**declarative structure**
what invariants hold?	**arbitrary predicates**
how do I check?	validation
what went wrong?	explanation
what went right?	conformance
docs please	autodoc
examples please	sample generator
am I using this right?	instrumentation
is my code correct?	generative testing
can I recombine pieces like this?	assertion

Structural Predicates

```
boolean?
char?
double?
int?
float?
```

primitives

```
bigdec?
keyword?
ratio?
string?
symbol?
uuid?
```

classes

```
any?
nil?
rational?
some?
ident?
```

crosscutting

Arbitrary Predicates

```
true?
false?
zero?
#{0 1 2}
```

explicit values

```
pos?
int?
pos-int?
neg-int?
nat-int?
```

value ranges

```
qualified-symbol?
simple-symbol?
```

arbitrary runtime facts

Collections

```
(s/coll-of string?)
```

homogeneous

```
(s/coll-of int?
           :kind vector?
           :min-count 5
           :max-count 10
           :distinct true)
```

size and type

```
(s/tuple int? int? keyword?)
```

heterogeneous tuple

```
(s/map-of keyword?
          (s/coll-of int?))
```

composition

Boolean Logic

```
(s/and string?
       #(str/starts-with "SKU-" %))

(s/or :id pos-int?
      :email string?)
```

Named Specs

```
(s/def :my.app/sku
  (s/and string?
         #(str/starts-with "SKU-" %)))

(s/def ::purchaser
  (s/or :account-id pos-int?
        :email string?))

(s/def ::import-line-item
  (s/tuple [::purchaser ::sku pos-int?]))
```

strong global names

by reference

Syntax

syntaxis

“an arranging in order”

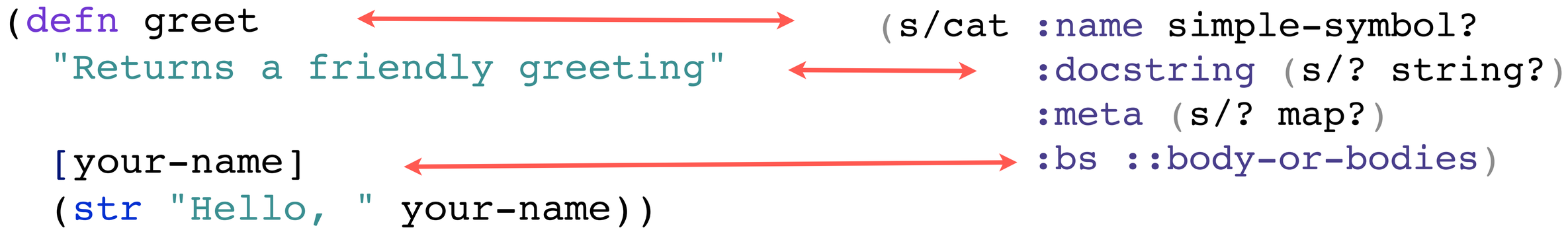
spec Syntax with Regexes

What	How
order	s/cat
choice	s/alt
optionality	s/?
repetition	s/+
optional repetition	s/*

defn Syntax

```
(defn greet
  "Returns a friendly greeting"
  [your-name]
  (str "Hello, " your-name))
```

specing defn



Maps as Information

```
(s/keys :req [::addr/name
              ::addr/street-1
              ::addr/city
              ::addr/state
              ::addr/zip]
        :opt [::addr/street-2])
```

strong, global names

critical decoupling: key *presence* only

Functions

```
(str/index-of "pirate" "rat")
(str/index-of "pirate" \r     10)
```

```
(s/def ::args-for-index-of
  (s/cat :source string?
         :search (s/alt :string string?
                        :char char?)
         :from (s/? nat-int?)))

(s/fdef my-index-of
        :args ::args-for-index-of
        :ret (s/nilable nat-int?))
```

spec fn args

spec fn returns

Function Semantics

```
(s/fdef my-index-of
        :args ::args-for-index-of
        :ret (s/nilable nat-int?)
        :fn (s/or
             :not-found #(nil? (:ret %))
             :found #(<= (:ret %)
                         (-> % :args :source count))))
```

not found: nil

two categoric outcomes

ret is bound by size of input source

When?

What	How	When?
what are the building blocks?	declarative structure	up to you
what invariants hold?	arbitrary predicates	up to you
how do I check?	validation	up to you
what went wrong?	explanation	up to you
what went right?	conformance	up to you
docs please	autodoc	autogen
examples please	sample generator	up to you
am I using this right?	instrumentation	up to you
is my code correct?	generative testing	up to you
can I recombine pieces like this?	assertion	up to you

Execution

What	How
what are the building blocks?	declarative structure
what invariants hold?	arbitrary predicates
how do I check?	**validation**
what went wrong?	**explanation**
what went right?	**conformance**
docs please	autodoc
examples please	sample generator
am I using this right?	instrumentation
is my code correct?	generative testing
can I recombine pieces like this?	assertion

Validation

```
(s/valid?
 (s/coll-of keyword?)
 [:a :b :c "oops"])

=> false
```

Explanation

```
(s/explain (s/coll-of keyword?) [:a :b :c "oops"])
=> In: [3] val: "oops" fails predicate: keyword?
```

```
(s/explain-data (s/coll-of keyword?) [:a :b :c "oops"])
=> #:clojure.spec{:problems
                  ({:path [],
                    :pred keyword?,
                    :val "oops",
                    :via [],
                    :in [3]})}
```

what was wrong?

how was it wrong?

at what position was it wrong?

Conformance

```
(s/conform
 :clojure.core.specs/defn-args
 '(greet
    "Returns a friendly greeting"
    [your-name]
    (str "Hello, " your-name)))
```

how the value matched

```
{:name greet,
 :docstring "Returns a friendly greeting",
 :bs [:arity-1 {:args {:args [[:sym your-name]]},
                :body [(str "Hello, " your-name)]}]}
```

Dev Assistance

What	How
what are the building blocks?	declarative structure
what invariants hold?	arbitrary predicates
how do I check?	validation
what went wrong?	explanation
what went right?	conformance
docs please	**autodoc**
examples please	**sample generator**
am I using this right?	instrumentation
is my code correct?	generative testing
can I recombine pieces like this?	assertion

Autodoc

```
(s/fdef letter-grade
        :args (s/cat :n ::grade)
        :ret #{:A :B :C :D :F})

(doc letter-grade)
-------------------------
user/letter-grade
([n])
Spec
  args: (cat :n :user/grade)
  ret: #{:A :F :D :B :C}
```

Example Data

```
(s/def ::grade (s/int-in 0 100))
(s/exercise ::grade 25)

=> ([0 0] [1 1] [1 1] [1 1] [3 3]
    [9 9] [0 0] [5 5] [3 3] [8 8]
    [1 1] [98 98] [6 6] [6 6] [4 4]
    [50 50] [26 26] [63 63] [1 1] [69 69]
    [61 61] [63 63] [60 60] [71 71] [7 7])
```

value

conformed value

Example Fn Invocations

```
(s/exercise-fn #'letter-grade 25)

=> ([(0) :F] [(0) :F] [(1) :F] [(2) :F] [(0) :F]
    [(0) :F] [(0) :F] [(10) :F] [(1) :F] [(3) :F]
    [(52) :F] [(1) :F] [(0) :F] [(2) :F] [(11) :F]
    [(26) :F] [(60) :D] [(60) :D] [(61) :D] [(68) :D]
    [(94) :A] [(52) :F] [(63) :D] [(7) :F] [(50) :F])
```

Robustness

What	How
what are the building blocks?	declarative structure
what invariants hold?	arbitrary predicates
how do I check?	validation
what went wrong?	explanation
what went right?	conformance
docs please	autodoc
examples please	sample generator
am I using this right?	**instrumentation**
is my code correct?	**generative testing**
can I recombine pieces like this?	**assertion**

Instrumentation

```
(test/instrument `start-server)
(start-server {:host "localhost" :port :default})
```

dev-time switch

```
ExceptionInfo Call to #'user/start-server did not conform to spec:
In: [0 :port] val: :default fails spec: :user/port
              at: [:args :endpoint :port] predicate: pos-int?
:clojure.spec/args  ({:host "localhost", :port :default})
:clojure.spec/failure  :instrument
:clojure.spec.test/caller  {:file "example.clj",
                            :line 160,
                            :var-scope user/eval1552}
```

position of bad value

pinpoint invalid call

what spec failed

Generative Testing

```
(->> (test/check `index-of)
     test/summarize-results)
```

generate a huge number of test cases

```
{:spec ...,
 :sym user/index-of,
 :failure
 {:clojure.spec/problems
  [{:path [:ret], :pred nat-int?,
    :val nil, :via [], :in []}],
  :clojure.spec.test/args ("" "0"),
  :clojure.spec.test/val nil,
  :clojure.spec/failure :check-failed}}
```

find problem

shrink test case to smallest repro

Assertion

```
(s/check-asserts true)
(s/assert
 (s/coll-of (s/int-in 1 7))
 [6 6 9])

ExceptionInfo Spec assertion failed
In: [2] val: 9 fails predicate: (int-in-range? 1 7 %)
:clojure.spec/failure  :assertion-failed
```

turn on spec assertion

validate against spec

precise errors

Power

What	How	When
what are the building blocks?	declarative structure	*up to you*
what invariants hold?	arbitrary predicates	*up to you*
how do I check?	validation	*up to you*
what went wrong?	explanation	*up to you*
what went right?	conformance	*up to you*
docs please	autodoc	*automatic*
examples please	sample generator	*up to you*
am I using this right?	instrumentation	*up to you*
is my code correct?	generative testing	*up to you*
can I recombine pieces like this?	assertion	*up to you*

Experience Report

early days still

I already want this ~~everywhere~~ anywhere

- pro re nata

be thoughtful about

- overspecification
- generation

clojure.org/about/spec

cognitect.com