

CLOJURE MEETUP - 17/05/2017

CLOJURE.SPEC THOUGHTS

GOALS FOR TODAY

- ▶ Intro to Clojure Spec
- ▶ On a real world example
- ▶ Return on experience

RETURN ON EXPERIENCE

- ▶ Contracts vs Types
- ▶ Entities vs Associations
- ▶ The Binary Tree challenge

CLOSURE SPEC

INTRODUCTION & MOTIVATION

SPECIFY SHAPE OF DATA

GAME SCORE

- ▶ Three players: blue, red, green
- ▶ Each one has an positive integer score

Blue - 11

Red - 14

Green - 14

GAME SCORE

```
{ :player/blue    24  
  :player/red     17  
  :player/green  19}
```

SPEC FOR PLAYERS

```
(s/def :player/players  
  #{ :player/blue  
    :player/red  
    :player/green} )
```


SPEC FOR SCORES

```
(s/def :game/scores  
  (s/map-of  
    :player/players  
    pos-int?))
```

DYNAMIC: RUN-TIME

DYNAMIC

★ ? Blue - 11 Red - 14 Green - 14 ↻ ←

The image shows a 15x15 grid game board. The board is populated with colored pieces and empty cells. The pieces are distributed as follows:

Row	Col 1	Col 2	Col 3	Col 4	Col 5	Col 6	Col 7	Col 8	Col 9	Col 10	Col 11	Col 12	Col 13	Col 14	Col 15
1			Blue		Red		Green	Green	Green			Blue			Blue
2					Red				Grey		Grey		Grey		
3	Green	Blue			Red						Grey		Grey		Red
4			Grey		Red		Grey					Grey		Green	
5				Blue	Red								Blue		
6			Red	Blue	Green										
7		Green	Green							Red	Blue			Green	Red
8			Red				Grey		Grey	Red			Grey	Blue	Blue
9												Green			
10									Green	Red			Grey		Green
11			Blue		Green		Red	Green		Red					

<https://github.com/QuentinDuval/triboard>

DYNAMIC

```
(s/def :game/board  
  (s/every  
    (s/every  
      :player/players  
      :count height)  
      :count width))
```

USEFUL: OFFER NICE GOODIES

USEFUL: OFFER NICE GOODIES

```
(s/valid? :game/turn turn)
```

```
(s/conform :game/turn turn)
```

```
(gen/sample (s/gen :game/turn) 1)
```

USEFUL: OFFER NICE GOODIES

- ▶ Validation, Instrumentation
- ▶ Parsing, Deconstructing
- ▶ Generate samples / tests

FLEXIBLE: INFORM, NOT CONSTRAIN

EXAMPLE: A GAME TURN

```
{ : game/board      board  
  : game/scores     scores  
  : game/transitions transitions}
```

NO NEED TO SPEC EVERYTHING

```
(s/def :game/turn  
  (s/keys :req  
    [:game/board  
     :game/scores])
```

NO NEED TO SPEC EVERYTHING

{ :game/board

board

:game/scores

scores}

PRESENCE VS CONFORMANCE

```
(s/def :game/transitions  
  (s/map-of  
    :game/coordinate  
    :game/transitions))
```

PRESENCE VS CONFORMANCE

```
{ :game/board      board  
  :game/scores     scores  
  :game/transitions []} ;;BOOM!
```

PRESENCE VS CONFORMANCE

```
{ : game/board      board
  : game/scores      scores
  : game/transitions {} } ; ; PASS
```

EXPRESS COMMITMENT WITH CLIENT

EXPRESSING COMMITMENT

- ▶ Define a contract with the client
 - ▶ **Preconditions** (never ask for more)
 - ▶ **Postconditions** (never provide less)

EXPRESSING COMMITMENT

- ▶ Avoid **strong coupling** of types
 - ▶ Exact same fields
 - ▶ Ordering of fields
- ▶ Type-Safety vs Easy Evolution

RETURN ON EXPERIENCE

PROBLEMS WITH SYMMETRY

BACK TO GAME/TURN SPEC

```
(s/def :game/turn  
  (s/keys :req  
    [:game/board  
     :game/scores])
```

NOT COMMITTING ON ALL INPUTS

```
(s/fdef new-first-turn  
  :args  (s/cat ...)  
  :ret    :game/turn)
```

SYMMETRY IS THE PROBLEM

```
(s/fdef next-turn
  :args
    (s/cat :turn :game/turn
           :move :game/transition)
  :ret    :game/turn)
```

NEXT-TURN SYMMETRY

- ▶ Need `:game/transitions` as input
 - ▶ Input spec is `incomplete`
 - ▶ Post conditions might not be fulfilled
- ▶ No wish to commit keyword presence

SOLUTION 1: COMMIT EVERYTHING

```
(s/def :game/turn
  (s/keys :req
    [:game/board
     :game/scores
     :game/transitions])
```

SOLUTION 2: ALTERNATE DEFINITION OF TURN

- ▶ A valid turn is
 - ▶ An initial **new-first-turn**
 - ▶ Followed by N **next-turn**
- ▶ Generate turn by successive next-turn

IMPLEMENTATION VS INFORMATION

- ▶ Spec is meant for information
- ▶ Make turn an abstraction
- ▶ Add function to extract turn data

RETURN ON EXPERIENCE

ENTITIES VS ASSOCIATIONS

A DIFFERENT SPEC FOR SCORES

```
{ :player/blue    24  
  :player/red     17  
  :player/green   19}
```

A DIFFERENT SPEC FOR SCORES

```
(s/def :player/scores  
  (s/map-of  
    :player/players  
    pos-int?))
```

A DIFFERENT SPEC FOR SCORES

```
(s/def :player/red pos-int?)  
(s/def :player/green pos-int?)  
(s/def :player/blue pos-int?)  
  
(s/def :game/scores  
  (s/keys :req  
    [:player/red  
     :player/green  
     :player/blue]))
```

GAME OF THE DIFFERENCES

- ▶ **map-of** cannot contain other keys
- ▶ **key sets** are only about membership
- ▶ **key sets** ensures the presence of keys

ENTITY VS ASSOCIATIONS

- ▶ Assign specs to **entities** (intrinsic meaning)
- ▶ Use **key sets** for entities membership
- ▶ Use **map-of** for associations

THE CHALLENGE

MODELLING A BINARY TREE

THE BINARY TREE CHALLENGE

```
class BinaryTree<A>
{
    A value;
    BinaryTree<A> lhs;
    BinaryTree<A> rhs;
};
```

INTEGER BINARY TREE (REPRESENTATION)

[1

{ :lhs [5 {}]

:rhs [2

{ :lhs [3 {}]

:rhs [4 {}]

}]

ENTITIES, NOT A GOOD FIT

```
(s/def :tree/lhs :tree/b-tree)
```

```
(s/def :tree/rhs :tree/b-tree)
```

```
(s/def :tree/b-tree  
  (s/cat :value any?  
         :children  
         (s/keys :opt  
                  [:tree/lhs :tree/rhs])))
```

ASSOCIATIONS

```
(s/def :int-tree  
  (s/cat  
    :value int?  
    :children  
    (s/map-of #{:lhs :rhs}  
              :int-tree)))
```

USING MACROS FOR GENERICS

```
(def-btree-of  
  :int-tree int?)
```

```
(def-btree-of  
  :string-tree string?)
```

CLOSURE SPEC

CONCLUSION & LINKS

CONCLUSION

- ▶ Trade-off between **safety** and **coupling**
- ▶ Oriented toward **information**
- ▶ Requires different thinking than types

RESOURCES

- ▶ <https://github.com/QuentinDuval/ClojureMeetup-2017-05-17>
- ▶ <https://clojure.org/about/spec>
- ▶ <https://clojure.org/guides/spec>
- ▶ <https://www.youtube.com/watch?v=oyLBGkS5ICk>