You are in a maze of deeply nested maps, all alike

Eric Normand
IN/Clojure 2019

PurelyFunctional.tv
Symptoms
Symptom:

I can’t remember what keys belong in this map
Symptom:

I don’t even know what kind of entity I have
Symptom:

Working with deeply nested data is awkward
{:date #inst "1984-04-03T13:08:42.020"
:name "Soyuz T-11"
:spacecraft "Soyuz"
:destination "Salyut 7"
:crew [{:name "Yuri Malyshev"
  :position "Commander"}
  {:name "Gennadi Strekalov"
  :position "Flight Engineer"}
  {:name "Rakesh Sharma"
  :position "Research Cosmonaut"}]}
(mapv (fn [mission]
   (update mission :crew
    (fn [crews]
     (mapv (fn [crew]
         (update crew :name
          string/upper-case))
     crews))))
missions)
(mapv (fn [mission]
    (update mission :crew
        (fn [crews]
            (mapv (fn [crew]
                    (mapv (fn [crew]
                            (update crew :name
                                string/upper-case))
                        crews)))))
    missions))
(mapv (fn [mission]
    (update mission :crew 
        (fn [crews]
            (mapv (fn [crew]
                    (mapv (fn [crew]
                            (update crew :name 
                                string/upper-case)
                        crews))
                crews))))
    missions))
(mapv (fn [mission]
    (update mission :crew
        (fn [crews]
            (mapv (fn [crew]
                (mapv (fn [crew]
                    (update crew :name string/upper-case)
                crews)))))))
missions)
(mapv (fn [mission]
    (update mission :crew
      (fn [crews]
        (mapv (fn [crew]
            (update crew :name
              string/upper-case))
          crews))))
missions)
Symptom:

Long, convoluted functions
Technological Solutions

Spec

Specter
<table>
<thead>
<tr>
<th>Symptoms</th>
<th>Spec</th>
<th>Specter</th>
</tr>
</thead>
<tbody>
<tr>
<td>What keys?</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>What entity?</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Deep nesting</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>Long functions</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Symptoms

What keys?

What entity?

Deep nesting

Long functions

Spec

✔

Speceter

✔

✔
Underlying Problem:

You're working at the wrong level of meaning
It’s just data
{:date #inst "1984-04-03T13:08:42.020"
:name "Soyuz T-11"
:spacecraft "Soyuz"
:destination "Salyut 7"
:crew [{:name "Yuri Malyshev"
  :position "Commander"}
  {:name "Gennadi Strekalov"
  :position "Flight Engineer"}
  {:name "Rakesh Sharma"
  :position "Research Cosmonaut"}]}
{:date #inst "1984-04-03T13:08:42.020"
:name "Soyuz T-11"
:spacecraft "Soyuz"
:destination "Salyut 7"
:crew [{:name "Yuri Malyshev"
            :position "Commander"}
            {:name "Gennadi Strekalov"
            :position "Flight Engineer"}
            {:name "Rakesh Sharma"
            :position "Research Cosmonaut"}]}
{:date #inst "1984-04-03T13:08:42.020"
:name "Soyuz T-11"
:spacecraft "Soyuz"
:destination "Salyut 7"
:crew [{:name "Yuri Malyshev"
    :position "Commander"}
  {:name "Gennadi Strekalov"
    :position "Flight Engineer"}
  {:name "Rakesh Sharma"
    :position "Research Cosmonaut"}]}
Space travel (domain)

your code

eval

Clojure Semantics

read

Clojure Syntax

Characters

Reader

Bytes
Clojure core
Libraries
Domain code
(def user-info (reagent/atom {}))

(defn favorite-color-button [color] ;; reagent component
  [:button {:on-click (fn []
                   (swap! user-info assoc :favorite-color color))
             :style {:background-color color}}
   color])

(defn favorite-color-panel []
  [:div
   [:h2 "Favorite color"]
   [:h3 "Current: " (:favorite-color @user-info)]
   [:h3 "Change it by clicking below"]
   [favorite-color-button "green"]
   [favorite-color-button "blue"]
   [favorite-color-button "red"]])
(def user-info (reagent/atom {}))

(defn favorite-color-button [color] ;; reagent component
  [:button {:on-click (fn []
    (swap! user-info assoc :favorite-color color))
    :style {:background-color color}}
  color])

(defn favorite-color-panel []
  [:div
   [:h2 "Favorite color"]
   [:h3 "Current: " (:favorite-color @user-info)]
   [:h3 "Change it by clicking below"]
   [favorite-color-button "green"]
   [favorite-color-button "blue"]
   [favorite-color-button "red"]])
Your head is up here.
Your head is up here

- Clojure core
- Libraries
- Domain code

Your code is down here
(assoc user-info :favorite-color color)

Clojure core and libraries
(defn set-favorite-color [user-info color]  
  (assoc user-info :favorite-color color))
(def user-info (reagent/atom {}))

(defn favorite-color-button [color] ;; reagent component
  [:button {:on-click (fn []
    (swap! user-info assoc :favorite-color color))
    :style {:background-color color}}
   color])

(defn favorite-color-panel []
  [:div
   [:h2 "Favorite color"]
   [:h3 "Current: " (:favorite-color @user-info)]
   [:h3 "Change it by clicking below"]
   [favorite-color-button "green"]
   [favorite-color-button "blue"]
   [favorite-color-button "red"]])
(def user-info (reagent/atom {}))

(defn favorite-color-button [color] ;; reagent component
  [:button {:on-click (fn []
      (swap! user-info set-favorite-color color))
    :style {:background-color color}}
   color])

(defn favorite-color-panel []
  [:div
   [:h2 "Favorite color"]
   [:h3 "Current: " (:favorite-color @user-info)]
   [:h3 "Change it by clicking below"]
   [favorite-color-button "green"]
   [favorite-color-button "blue"]
   [favorite-color-button "red"]])
(ns my-app.user-info)

(defn set-favorite-color [user-info color]
  (assoc user-info :favorite-color color))
(ns my-app.user-info)

(defn set-favorite-color [user-info color]
  (assoc user-info :favorite-color color))

(defn set-favorite-animal [user-info animal]
  (assoc user-info :favorite-animal animal))

(defn set-favorite-fruit [user-info fruit]
  (assoc user-info :favorite-fruit fruit))
Objection:

But Eric, isn't that a lot of code?
(def user-info (reagent/atom {}))

(defn favorite-color-button [color] ;; reagent component
  [:button {:on-click (fn []
                (swap! user-info assoc :favorite-color color))
            :style {:background-color color}}
  color])

(defn favorite-color-panel []
  [:div
   [:h2 "Favorite color"]
   [:h3 "Current: " (:favorite-color @user-info)]
   [:h3 "Change it by clicking below"]
   [favorite-color-button "green"]
   [favorite-color-button "blue"]
   [favorite-color-button "red"]])
(def user-info (reagent/atom {}))

(defn set-favorite-color [user-info color]
  (assoc user-info :favorite-color color))

(defn favorite-color-button [color] ;; reagent component
  [:button {:on-click (fn []
                  (swap! user-info set-favorite-color color))
    :style {:background-color color}}
    color])

(defn favorite-color-panel []
  [:div
   [:h2 "Favorite color"]
   [:h3 "Current: " (:favorite-color @user-info)]
   [:h3 "Change it by clicking below"]
   [favorite-color-button "green"]
   [favorite-color-button "blue"]
   [favorite-color-button "red"]])
Con:

3x the code
Con:

3x the code

Pro:

Find things in $\frac{1}{3}$ the time

Probably duplicated operations

Parts can evolve separately

Less to keep in your head
Domain code
Libraries
Clojure core
The biggest regret I hear frequently is doing data operations everywhere.
Clojure core

Libraries

Domain code

Clojure core
Domain code

Libraries

Clojure core
Clojure core

Libraries

Domain Layer 1

Domain Layer 2

Domain Layer 3

Domain Layer 4
{:date #inst "1984-04-03T13:08:42.020"
  :name "Soyuz T-11"
  :spacecraft "Soyuz"
  :destination "Salyut 7"
  :crew {
    :commander {
      :name "Yuri Malyshev"
      :favorite-color "blue"
    }
    :flight-engineer {
      :name "Gennadi Strekalov"
      :favorite-fruit :banana
    }
    :research-cosmonaut {
      :name "Rakesh Sharma"
      :favorite-animal :sheep
    }
  }}
(ns my-app.training)

(defn set-status [record training status]  
  (assoc record training status))

(ns my-app.person  
 (:require [my-app.training :as training]))

(defn set-training-status [person training status]  
  (update person :training training/set-status training status))

(ns my-app.mission  
 (:require [my-app.person :as person]))

(defn set-training-status [mission position training status]  
  (update-in mission [:crew position] person/set-training-status training status))
Law of Demeter

a given object should assume as little as possible about the structure or properties of anything else (including its subcomponents)
Objection:

But Eric, aren't you encapsulating? Isn't the point of Clojure that it's all data?
Summary:

We are encapsulating and it's still data.
- enumerate keys
- serialize to json
- print
- compare with =
- hash code
- try out data operations at REPL
Map

- enumerate keys
- serialize to json
- print
- compare with =
- hash code
- try out data operations at REPL

Entity

- domain operations
- maintain invariants
- program at a high level
- easier to understand
Stratified Design

Dishes
Basics (sauces, masalas, dough)
Ingredients (vegetables, meats, oils, spices)
Applying heat          Chopping
Chemistry (heat, acids, proteins, starches, etc)
Stratified Design

Principle:

separate things according to rates of change
Stratified Design

Principle:

each layer can only require downward

Diagram:

- Dishes
- Basics (sauces, masalas, dough)
- Ingredients (vegetables, meats, oils, spices)
- Applying heat
- Chemistry (heat, acids, proteins, starches, etc.)
- Chopping
Stratified Design

Principle:

each layer can only require downward
Stratified Design

Principle:

each layer can only require downward
Stratified Design

Principle:

be careful if you skip layers
Principle:

be careful if you skip layers
Constructors

Functions for generating entities

Benefits

- Define the entity's keys
- Check required keys
- Check types
- Check invariants
- Default values
- Calculated values
Constructors

(defn ->person [& {:keys [name
    trainings]
    :or {trainings []}}]
  (assert (string? name))
  (let [name (string/trim name)]
    (assert (not (empty? name)))
    {:name name
     :trainings trainings}))

- **Default**: `(defn ->person)`
- **Type + Required**: `& {:keys [name
    trainings] :or {trainings []}}
  (assert (string? name))` (with `v` as a named parameter)
- **Calculation**: `(let [name (string/trim name)]
  (assert (not (empty? name)))
  {:name name
   :trainings trainings})`
- **Invariant**: `name` and `trainings`
- **Key Names**: `v` (without a named parameter)
Combining Operations

Functions that combine 2 or more entities of the same type

Benefits

- Hardest type of operation first
- Constrain the design (good thing)
- Biggest potential for algebraic reasoning
(ns my-app.training)

(defn combine-training-status [a b]
  (cond
   (= :pass a) :pass
   (= :pass b) :pass
   :else       :fail))

(defn combine-trainings [a b]
  (merge-with combine-training-status a b))
Meaning

Implementation
Meaning

Data

you want to be working up here

but you're stuck down here
Code Smells
Smell:

Deep paths with get-in, update-in, assoc-in
Smell:

Large hashmaps with lots of keys
{:id 1232
 :email "eric@lispcast.com"
 :phone "504-982-9167"
 :street1 "332 Main St"
 :street2 "Apartment 54"
 :city "New Orleans"
 :state "LA"
 :zip "70117"
 :favorite-color "blue"
 :favorite-animal :orangutan
 :favorite-fruit :apple}
{:
  id 1232
  :email "eric@lispcast.com"
  :phone "504-982-9167"
  :street1 "332 Main St"
  :street2 "Apartment 54"
  :city "New Orleans"
  :state "LA"
  :zip "70117"
  :favorite-color "blue"
  :favorite-animal :orangutan
  :favorite-fruit :apple}
{:id 1232
  :contact-info {:email "eric@lispcast.com"
                 :phone "504-982-9167"
                 :address {:street1 "332 Main St"
                          :street2 "Apartment 54"
                          :city "New Orleans"
                          :state "LA"
                          :zip "70117"}}

  :favorites {:color "blue"
              :animal :orangutan
              :fruit :apple}
Smell:

Using 3rd party API results directly
Know what level of meaning you are at

- Look for semantic layers
- Avoid crossing semantic layers
- Organize your entities and operations
- Think about combining operations
- Add a layer of indirection between your service and other systems
- Use constructors to make getting it right easier
- Name operations to give things meaning