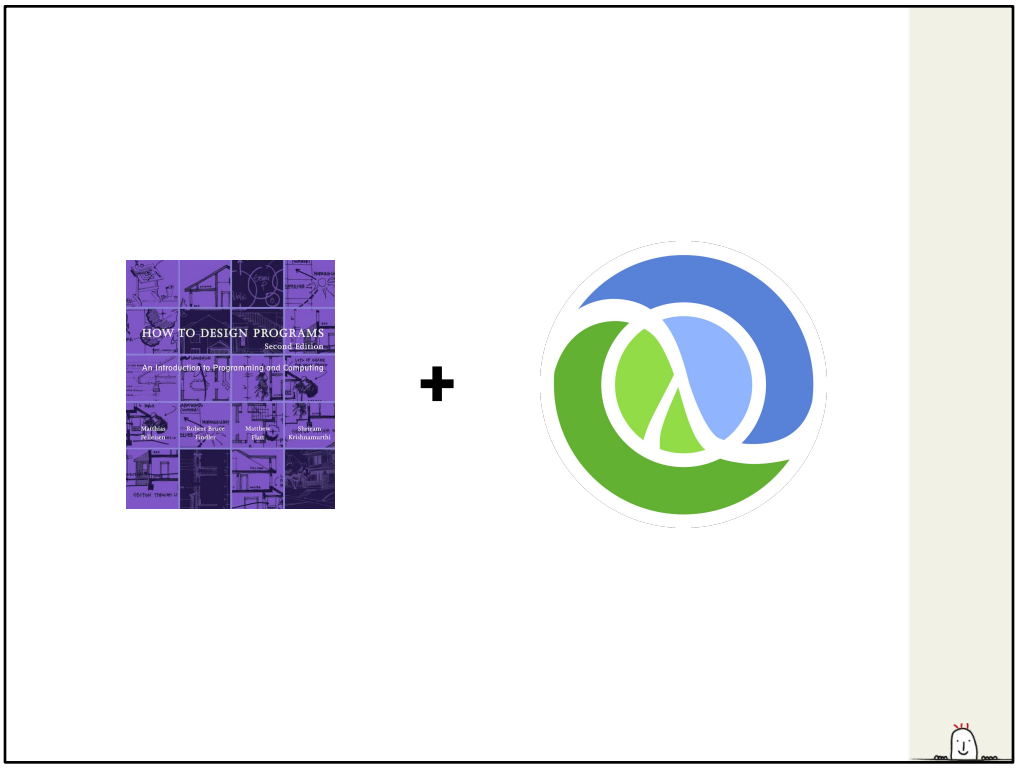
BOB
2021

Design Recipes
+
clojure.spec

Diego SÁNCHEZ
Leandro DOCTORS

- Hi, I'm Leandro DOCTORS.
- And I'm Diego SÁNCHEZ.
- And today we're presenting our workshop on Combining clojure.spec with design recipes.



- We are here to combine two things:
  - the Design Recipes from the book “How to Design Programs” (by Felleisen and others)
  - with the power of *clojure.spec* for both software specifications & generative testing

Preliminaries

- Before we begin, we would like to state a few things...

- First of all
  - we are all here today to share our insight
  - (both the one we bring from home, and the one we all may gain during this session)

Approach

- No silver bullet
- Methods and Paradigms over Tools
- We are not experts on clojure.spec
- Suggestions are welcome!

- Second, With respect to how do we approach this session:
  - To begin with, you have to be aware that, by itself, attending this session will not solve all of your problems (it may help tackling a few, though :)
  - Second, the tools we propose using are just the best means *we* could find.
    - Perhaps, *you* can find other tools that suit your own style better. That's fine.
  - Third, code improvement suggestions are very welcome!
  - And, finally: please be aware that this is the first iteration of this tutorial.
    - Your feedback (on any of its aspects) is greatly appreciated to improve it for future attendants!
    - We would love to hear your experiences!
      - Trying out the Recipes with clojure.spec? Let us know!
      - Trying out the Recipes with something else? Let us know!

Solving Problems

- Let’s first ask ourselves…
- How do we normally solve programming problems?

- We normally solve programming problems by Trial and Error (some people call it “Tinker until It Works”).
- We aim our darts, and we hope to hit the bullseye...

What does it take to hit the bullseye?

- But… what do we need to actually achieve this?

Hitting the bullseye

- Feedback
- Reproducibility

- We need two things:
  - #1) In case our solution doesn't work yet (for whatever reason), we have to make sure that (over time) we will eventually converge into yielding a working solution.

    (In short, "Am I improving my aim?")
  - #2) When our solution finally "works" (again, whatever that means), we have to make sure that we will be able to apply it again next time we stumble upon the same problem.

    (In short, "How can I win again?")
- In order to specifically achieve (#1), we need *feedback*.
- In order to specifically achieve (#2), we need *reproducibility*.
- However, in order to achieve both of them at the same time, we need something more...

Hitting the bullseye

"Darts" by Bogdan Suditu is licensed with CC BY 2.0.
Original: https://www.flickr.com/photos/8726888@N08/2377844553
License: https://creativecommons.org/licenses/by/2.0/

- **Systematic** Feedback
- **Systematic** Reproducibility

- We need a *systematic* process, so that we can get both feedback and reproducibility *in a continuous way*.
- But... What is systematic?

Systematic

- Carried out using a planned, ordered procedure
- Methodical, Regular and Orderly

Wiktionary: https://en.wiktionary.org/wiki/systematic

- This is Wiktionary's definition of Systematic.
- Clearly, being systematic enables us to obtain both feedback and reproducibility.

Systematic Software Design

- What?
  - *Systematic* thought, analysis, planning, and understanding.
  - End to end.
- How?
  - *Systematically* yelding Intermediate Products...
  - ...via Design Recipes.

Systematic Software Design:

- comprises systematic thought, analysis, planning, and understanding.
- From the very beginning. At every stage. And for every step.

How do we implement Systematic Software Design?

- By *Systematically* yielding *Intermediate Products* for our programs...
- via *Design Recipes*
- The existence of these intermediate products is very important, as it enables self-empowerment (we can be sure we are making progress - even before asking others for help).
- But… What is a Design Recipe?

Design Recipes

- A Design Recipe is...
  - a set of development guidelines, divided into “baby steps”.
  - (all design recipes share a single design process.)
  - Design Recipes are useful for developing anything (from isolated functions to full programs).
  - Let’s quickly see what those steps are...

Design Recipes: Steps

1. Problem Analysis --> Data Definitions
2. Signature, Purpose Statement, Header
3. Examples
4. Template
5. Definition (Coding)
6. Testing

- These are all the steps of the basic Design Recipe.
- As you can see, they cover the whole development workflow, end to end.

Applying the Basic Recipe

- Now, let’s apply those steps to a simple problem.

0. Problem Statement

- “Compute the area of a square.”

- We want to compute the area of a square.

1. Problem Analysis → Data Definitions

```
;; We will use natural numbers to represent both the square's
side length and area.
```

- Firstly, we decide how to represent the Entities we discovered within our “Core Domain”.
- In this case, they are side “length” and “area”.

2. Signature, Purpose Statement, Header

```
;; Length --> Area
;; Compute the area of a square with
side length 'len'.
(defn area-of-square
     [len]
     ...)
```

Arguments:

```
(s/def ::length
       nat-int?)

(s/def ::area
       nat-int?)
```

Function signature:

```
;; Length --> Area
(s/fdef area-of-square
       :args :len ::length
       :ret ::area)
```

- In step 2, we start shaping the interface of our function.
- Here, clojure.spec is particularly helpful in expressing how we conceive the Entities of our “Core Domain”, allowing an incredible amount of detail.
- We will revisit these concepts later on.

3. Examples

```
;; given: (= length 2), expect: (= area 4)

;; given: (= length 7), expect: (= area 49)
```

- In step 3, we produce the examples for our program.
- Eventually, they will become test cases.

4. Template

```
(defn area-of-square
     [len]
     (... len ...))
```

- In Step 4, we craft the outline for our function.
- Naïvely, at this point we only know that “something” happens to the parameter we pass into the function.

5. Coding

```
(defn area-of-square
     [len]
     "compute the area of a square of side `len`."
     (* len len))
```

- Only in step 5 we get to write the code that actually implements our function.

6. Testing

```
(deftest ^:stable test-area-computation
        (testing "Area computation"
                 ;; A square of side 2 should have an area of 4.
                 (is (= 4 (area-of-square 2)))
                 ;; A square of side 7 should have an area of 49.
                 (is (= 49 (area-of-square 7)))))
```

- Finally, in step 6, we get the code for our tests.
- As they directly correspond to our examples, some may realize it is quite straightforward writing the tests *before* the function code itself.

6. Testing

```
(deftest ^:unstable exercise-area-of-square
        (testing "Exercising the function `area-of-square`"
                (is (= {:total 1, :check-passed 1}
                       (stest/summarize-results (stest/check `area-of-square))))))
```

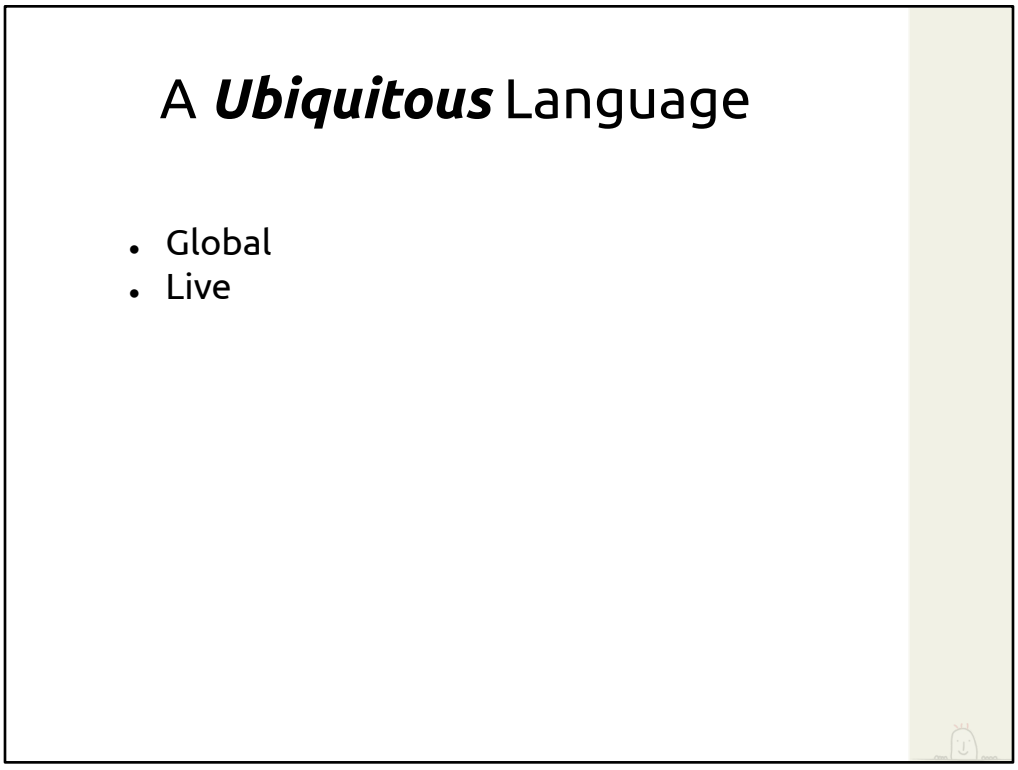
- And, when we add generative tests to the mix, we may notice that our CI infrastructure may become “unstable”.
- In this case, it is convenient if we label them accordingly to easily skip them when we need to do so.

Design Recipes
&
Domain-Driven Design*

- We previously mentioned concepts such as “Core Domain” and “Entities”.
- For those into Domain-Driven Design, these terms may sound familiar.
- Let’s see how Design Recipes implicitly embrace and foster these (and others!) DDD concepts and practices.
- An important note: the DDD concepts covered here have been abridged to fit within the time limits of this workshop.

An *Expressive* Challenge

- First of all, before we can start thinking about working out the solution, we have to think about how expressive we *need* to become.
- And by “expressive” we mean not only how to convey the eventual solution, but also about how to convey our understanding of the problem.
- We have to make sure to enable *all stakeholders* (and not only the development team) to express ourselves so that:
  - when we *think* we all agree on something… we *actually* agree on that.
  - (as you may already know, that is definitely something non-trivial to achieve)
- In order to get such a deep understanding, we need something very peculiar...



- We need a globally shared mental model.
- This is, a set of concepts & vocabulary
  - structured around the domain model,
  - shared *by all stakeholders*
  - to connect all the activities of the team with the software product.
- Let's explain how this "Ubiquitous Language" should be:
  - First, by "global", we mean:
    - *globally* available, *globally* shared, and *globally* used throughout-our-project
    - (from our wiki, through all of our documentation, to our code)
  - And, by "live", we mean that
    - our Ubiquitous Language will certainly evolve during the lifecycle of our software product.
    - (Here, the advise of "Version Control everywhere" certainly *does* apply. Plus, it may have the added value of traceability.)

A ***Ubiquitous*** Language

- “Length”
- “Area”

- As you may have already guessed, some terms of the Ubiquitous Language for our toy example would be:
  - “length”, and
  - “area”
- All the significance of these concepts (and, probably, others) should be conveyed by the terms of the Ubiquitous Language specifically crafted for this project.
- Why is this so important? Because (as we have already said) we-will-express our Core Domain model *using these-very-same-terms*.
- But, what is the Core Domain?

Core Domain & Entities

```
(s/def ::length
       nat-int?)

(s/def ::area
       nat-int?)
```

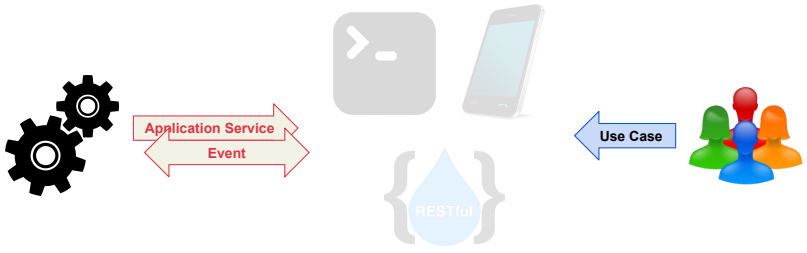
- The Core Domain is the *essence* of how our solution will tackle our problem.
  - The Core Domain is what makes our product worth writing.
  - (If we can't identify our Code Domain, our product will be irrelevant.)
  - The Core Domain is to be described in terms of the Ubiquitous Language.
- Our Entities are representations of domain concepts, properties and/or relationships, normally implemented via software constructs.
  - What we did in step 2 was defining an aspect of our core domain through defining our entities' data.
- But.. Do entities' data *by themselves* bring value to our product?
- Or, in terms of our toy example: Could an "Length" Entity add *any* value to our product, just by sitting idle for as long as it fancies?

Value ←→ Transformations

```
(s/fdef area-of-square
        :args :len ::length
        :ret ::area)
```

- Well, not really. What really brings value to the table is *the relationships* across, between, and within our entities (in this case, length and area).
- How could we do that? Through *Data Transformations* within our product.
- Once again, Data Transformations can be easily modelled in our Functional world using... functions :-)

Notes on Delivery Mechanisms



- Before we continue, there is one thing we have to explain very clearly:
- The sum of our Use Cases should be supported by the sum of our Application Services and Events.
- No matter which Delivery Mechanisms we provide to our users as a means to achieving their goals (a CLI, a GUI, a REST endpoint...), our first priority has to be designing the Services our Application provides.
- This is because both the Application Services and the Domain Events are the only ones that can *directly* interact with our Core Domain Entities.
- So, it is much less likely to change due to non-Domain-related causes than, let's say, the current fashionable framework or API.
- And, as you may have guessed by now, in the Functional world, we can express our Application Services, Events and Use Cases as nothing more... than simple functions :-)

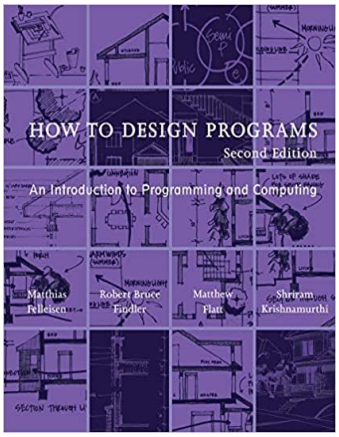
Let’s solve a Problem!

- OK, enough chit-chat.
- Now, let’s get our hands dirty! shall we?

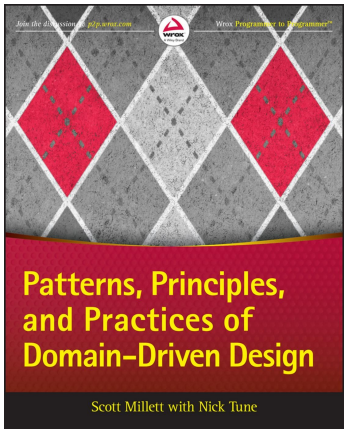
Sharing Insight

References

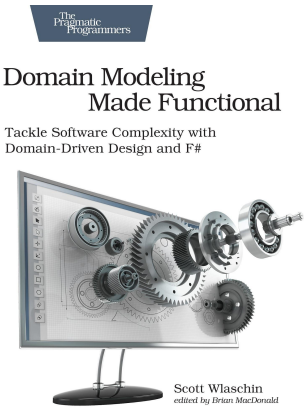
Felleisen, Findler, Flatt, & Krishnamurthi - The MIT Press - 2018. How to Design Programs: an Introduction to Programming and Computing.



Tune, N., & Millett, S. (2015). Patterns, Principles, and Practices of Domain-Driven Design. Wiley.



Scott Wlaschin (2015). Domain Modeling Made Functional: Tackle Software Complexity with Domain-Driven Design and F# 1st Edition



Thank you!

Design Recipes + clojure.spec

diego.sanches@gmail.com

@dsminotauro

ldoctors@gmail.com

ingdiegosanchez

diegosanchez

leandrodoctors

allentiak