

Extracted from:

Clojure Applied

From Practice to Practitioner

This PDF file contains pages extracted from *Clojure Applied*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2015 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The
Pragmatic
Programmers

Clojure Applied

From Practice to Practitioner

Ben Vandgrift
Alex Miller

edited by Jacquelyn Carter



Clojure Applied

From Practice to Practitioner

Ben Vandgrift
Alex Miller

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at <https://pragprog.com>.

The team that produced this book includes:

Jacquelyn Carter (editor)
Potomac Indexing, LLC (index)
Eileen Cohen (copyedit)
Dave Thomas (layout)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact rights@pragprog.com.

Copyright © 2015 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.

ISBN-13: 978-1-68050-074-5

Encoded using the finest acid-free high-entropy binary digits.

Book version: P1.0—September 2015

Validating Entities

Once we have our domain model, we need a way to validate whether our data conforms to it. Clojure's dynamic types give us great power and flexibility but also enforce fewer constraints by default. Data validation is an area in which Clojure gives us choices about when, where, and how much validation we want to provide. In areas where data is created by our code, we may want to do little or no validation, whereas we may need significant validation when accepting data from external sources.

A number of external libraries exist to provide data description and validation support. We'll focus on Prismatic's Schema² library, but you may also want to look at `core.typed`,³ `clj-schema`,⁴ `Strucjure`,⁵ or `seqex`.⁶

The Prismatic Schema library describes type metadata as data, automates descriptions, and validates data at runtime against that metadata. To add the Schema library to a project, add the following dependency in Leiningen:

```
cljapplied/project.clj
[prismatic/schema "0.4.3"]
```

Let's look at how to describe some data with Schema in the context of our recipe-manager application. This time we'll work out the details of the ingredients in the recipe:

```
cljapplied/src/ch1/validate.clj
(defrecord Recipe
  [name      ;; string
   description ;; string
   ingredients ;; sequence of Ingredient
   steps     ;; sequence of string
   servings  ;; number of servings
  ])

(defrecord Ingredient
  [name      ;; string
   quantity  ;; amount
   unit      ;; keyword
  ])
```

2. <https://github.com/prismatic/schema>
3. <https://github.com/clojure/core.typed>
4. <https://github.com/runa-dev/clj-schema>
5. <https://github.com/jamii/strucjure>
6. <https://github.com/jclaggett/seqex>

We've added comments to these records to help our colleagues (and maybe ourselves a few months down the line) understand what we expect. A particular instance of a recipe might look like this:

```
cljapplied/src/ch1/validate.clj
(def spaghetti-tacos
  (map->Recipe
    {:name "Spaghetti tacos"
     :description "It's spaghetti... in a taco."
     :ingredients [(->Ingredient "Spaghetti" 1 :lb)
                  (->Ingredient "Spaghetti sauce" 16 :oz)
                  (->Ingredient "Taco shell" 12 :shell)]
     :steps ["Cook spaghetti according to box."
             "Heat spaghetti sauce until warm."
             "Mix spaghetti and sauce."
             "Put spaghetti in taco shells and serve."]
     :servings 4}))
```

Let's use Schema to describe our recipes instead. Schema has its own version of `defrecord` that adds the ability to specify a schema for values of each field (in addition to the normal effects of `defrecord`). Schema is specified after the `:-`, which is a special keyword that Schema uses as a syntactic marker.

First, pull in the Schema namespace:

```
cljapplied/src/ch1/validate.clj
(ns ch1.validate
  (:require [schema.core :as s]))
```

Then, redefine the records using the Schema version of `defrecord`:

```
cljapplied/src/ch1/validate.clj
(s/defrecord Ingredient
  [name      :- s/Str
   quantity :- s/Int
   unit      :- s/Keyword])

(s/defrecord Recipe
  [name          :- s/Str
   description   :- s/Str
   ingredients   :- [Ingredient]
   steps        :- [s/Str]
   servings     :- s/Int])
```

Normal type hints and class names (like `String`) are valid schema descriptions, but we've used the built-in schemas like `s/Str` instead. These schemas are portable and yield the proper check in both Clojure and ClojureScript. The schema for `ingredients` is a sequence of items of type `Ingredient`. The `steps` field is a sequence of strings.

Once we've annotated our record with schema information, we can ask for an explanation of the schema, which is returned as data and printed:

```
user=> (require 'ch1.validate)
user=> (in-ns 'ch1.validate)
ch1.validate=> (pprint (schema.core/explain ch1.validate.Recipe))
(record
  ch1.validate.Recipe
  {:name java.lang.String,
   :description java.lang.String,
   :ingredients
   [(record
     ch1.validate.Ingredient
     {:name java.lang.String, :quantity Int, :unit Keyword})],
   :steps [java.lang.String],
   :servings Int})
```

We can also validate our data against the schema:

```
ch1.validate=> (s/check Recipe spaghetti-tacos)
nil
```

If the data is valid, `s/check` returns `nil`. If the data is invalid, `s/check` returns a descriptive error message detailing the schema mismatches. For example, if we passed a recipe that omitted the description and had an invalid servings value, we'd get an error message:

```
ch1.validate=> (s/check Recipe
  (map->Recipe
    {:name "Spaghetti tacos"
     :ingredients [(->Ingredient "Spaghetti" 1 :lb)
                  (->Ingredient "Spaghetti sauce" 16 :oz)
                  (->Ingredient "Taco" 12 :taco)]
     :steps ["Cook spaghetti according to box."
            "Heat spaghetti sauce until warm."
            "Mix spaghetti and sauce."
            "Put spaghetti in tacos and serve."]
     :servings "lots!"}))
{:servings (not (integer? "lots!")),
 :description (not (instance? java.lang.String nil))}
```

The error message specifies the fields that didn't conform to the schema and why they failed. These checks can be a great help in detecting invalid data passed into or between parts of your program for your domain data.

Schema also has a version of `defn` to specify schema shapes as input parameters and return types. The types are used to create a helpful docstring:

```
ch1.validate=> (s/defn add-ingredients :- Recipe
  [recipe :- Recipe & ingredients :- [Ingredient]]
  (update-in recipe [:ingredients] into ingredients))
```

```

ch1.validate=> (doc add-ingredients)
-----
ch1.validate/add-ingredients
([recipe & ingredients])
  Inputs: [recipe :- Recipe & ingredients :- [Ingredient]]
  Returns: Recipe

```

Schema can also optionally verify the runtime inputs and report schema mismatch errors by using the `s/with-fn-validation` function.

We've now looked at various trade-offs for representing domain entities, connecting entities together, and validating our entities. It's time to consider how we'll implement behavior for our domain types.

Domain Operations

We often need to define a function for our domain that can be applied to many different types of domain entities. This is particularly useful when domain entities of different types are collected together in a composite data structure.

Object-oriented languages typically address this need via *polymorphism*. Polymorphism is a means of abstraction, allowing a domain operation to be decoupled from the types to which it can be applied. This makes your domain implementation more general and provides a way to extend behavior without modifying existing code.

Clojure provides two features that allow the creation of generic domain operations: *multimethods* and *protocols*. Choosing the specific function to invoke for a generic operation is known as *dispatch*. Both protocols and multimethods can dispatch based on argument type, but only multimethods can dispatch based on argument value. We'll start by looking at how type-based dispatch compares in the two approaches and follow that with a look at value-based dispatch and how to layer protocols.

Multimethods vs. Protocols

Consider our recipe-manager application and the need to calculate an estimated grocery cost for each recipe. The cost of each recipe will be dependent on adding up the costs of all the ingredients. We want to invoke the same generic domain operation ("How much does it cost?") on entities of two specific types: Recipe and Ingredient.

To implement this domain operation with multimethods, we use two forms: `defmulti` and `defmethod`. The `defmulti` form defines the name and signature of the function as well as the *dispatch function*. Each `defmethod` form provides a

function implementation for a particular dispatch value. Invoking the multimethod first invokes the dispatch function to produce a dispatch value, then selects the best match for that value, and finally invokes that function implementation.

We need to extend our recipe-manager domain slightly to add a Store domain entity and a function that can look up the cost of an ingredient in a particular grocery store. We can sketch these without fully implementing them:

```
cljapplied/src/ch1/multimethods.clj
```

```
(defrecord Store [,,,])

(defn cost-of [store ingredient] ,,,)
```

Now we can implement our cost multimethod for both Recipes and Ingredients:

```
cljapplied/src/ch1/multimethods.clj
```

```
(defmulti cost (fn [entity store] (class entity)))

(defmethod cost Recipe [recipe store]
  (reduce +$ zero-dollars
    (map #(cost % store) (:ingredients recipe))))

(defmethod cost Ingredient [ingredient store]
  (cost-of store ingredient))
```

First the defmulti defines the dispatch function as (class entity), which produces a dispatch value based on type. If we were using maps instead of records, we would instead extract a type attribute with (:type entity) as the dispatch function.

Once the dispatch function is invoked with an entity to produce a type, that type is matched with the available defmethod implementations, and the Recipe or Ingredient function implementation is invoked.

Now consider how we might implement this same functionality with protocols. Protocols are also defined in two steps. First, the defprotocol form declares the name and a series of function signatures (but no function implementations). Then, extend-protocol, extend-type, or extend is used to declare that a type *extends* a protocol:

```
cljapplied/src/ch1/protocols.clj
```

```
(defprotocol Cost
  (cost [entity store]))

(extend-protocol Cost
  Recipe
  (cost [recipe store]
    (reduce +$ zero-dollars
      (map #(cost % store) (:ingredients recipe))))
```

```
Ingredient
(cost [ingredient store]
 (cost-of store ingredient)))
```

Here we define the `Cost` protocol, which has a single function (although it could have many). We then extend two types—`Recipe` and `Ingredient`—to the `Cost` protocol. These are both done in a single `extend-protocol` for convenience, but they could be extended separately.

Let’s compare these two approaches to type-based dispatch. Protocols are faster than multimethods for type dispatch because they leverage the underlying JVM runtime optimizations for this kind of dispatch (this is common in Java). Protocols also have the ability to group related functions together in a single protocol. For these reasons, protocols are usually preferred for type-based dispatch.

However, whereas protocols *only* support type-based dispatch on the first argument to the generic function, multimethods can provide value-based dispatch based on any or all of the function’s arguments. Multimethods and protocols both support matching based on the Java type hierarchy, but multimethods can define and use custom value hierarchies and declare preferences between implementations when there’s more than one matching value.

Thus, protocols are the preferred choice for the narrow (but common) case of type-based dispatch, and multimethods provide greater flexibility for a broad range of other cases.

Next, let’s see an example of value-based dispatch with multimethods, which isn’t covered by protocols.

Value-Based Dispatch

Although type-based dispatch is the most common case in many programs, value-based dispatch is needed in plenty of cases—and that’s where multimethods have their time to shine.

Consider a new feature in our `recipe-manager` application: building a shopping list by adding together all the ingredients in one or more recipes. Ingredients are specified with a quantity and a unit. We might have some recipes that specify spaghetti in pounds and some that specify it in ounces. We need a system that can do unit conversion. Multimethods give us the ability to provide conversions that depend on the source and target types:

```

cljapplied/src/ch1/convert.clj
(defmulti convert
  "Convert quantity from unit1 to unit2, matching on [unit1 unit2]"
  (fn [unit1 unit2 quantity] [unit1 unit2]))

;; lb to oz
(defmethod convert [:lb :oz] [_ _ lb] (* lb 16))

;; oz to lb
(defmethod convert [:oz :lb] [_ _ oz] (/ oz 16))

;; fallthrough
(defmethod convert :default [u1 u2 q]
  (if (= u1 u2)
    q
    (assert false (str "Unknown unit conversion from " u1 " to " u2))))

(defn ingredient+
  "Add two ingredients into a single ingredient, combining their
  quantities with unit conversion if necessary."
  [{q1 :quantity u1 :unit :as i1} {q2 :quantity u2 :unit}]
  (assoc i1 :quantity (+ q1 (convert u2 u1 q2))))

```

The `convert` multimethod dispatches on the *value* of the source and target types, not on their types. Adding new conversions is then a matter of supplying a `defmethod` for every source/target unit pair we allow in the system.

We also provide a fallthrough case with `:default`—when the units are the same, we can simply return the original quantity. If the units are different and we've made it to `:default`, we're attempting a conversion that wasn't defined. Since this is likely a programming error, we assert that it won't happen. Missing conversions then give us a useful error while we're testing.

Here's how this looks in practice:

```

user=> (ingredient+ (->Ingredient "Spaghetti" 1/2 :lb)
          (->Ingredient "Spaghetti" 4 :oz))
#user.Ingredient{:name "Spaghetti", :quantity 3/4, :unit :lb}

```

Here we add 1/2 pound (8 ounces) with 4 ounces and get 3/4 pound (12 ounces).

If we add new units to the system, we'll need to define conversions to and from all other units they might be combined with. In a recipe-manager application, the range of needed conversions is probably somewhat confined based on typical recipe usage.

Extending Protocols to Protocols

Both multimethods and protocols are *open systems*. Participation of a type or value in an abstraction can be specified (via `defmethod` or `extend-protocol`) separately from both the abstraction definition and the type. New participants can be dynamically added during the life of the system.

One particular case that comes up with protocols is the need to decide, at runtime, how particular concrete types should be handled in a protocol. This need commonly arises when you're creating protocols that layer over other protocols.

For example, you might need to extend the recipe manager further to calculate not only the cost of the items but also the cost of the items if bought from a particular store, including the location-specific taxes. This can be captured in a new protocol:

```
(defprotocol TaxedCost
  (taxed-cost [entity store]))
```

We already have a protocol that can make this calculation on both items and recipes of items. We'd like to layer the `TaxedCost` protocol over the existing `Cost` protocol, but this isn't allowed in Clojure:

```
(extend-protocol TaxedCost
  Cost
  (taxed-cost [entity store]
    (* (cost entity store) (+ 1 (tax-rate store))))
;;=> exception!
```

Clojure doesn't allow protocols to extend protocols because it opens up ambiguous and confusing cases for choosing the proper implementation function. However, we can provide the same effect by detecting this case for each concrete type encountered at runtime and dynamically installing the adapted protocol for that type:

```
(extend-protocol TaxedCost
  Object ;; default fallthrough
  (taxed-cost [entity store]
    (if (satisfies? Cost entity)
      (do (extend-protocol TaxedCost
          (class entity)
          (taxed-cost [entity store]
            (* (cost entity store) (+ 1 (tax-rate store))))
          (taxed-cost entity store))
        (assert false (str "Unhandled entity: " entity))))))
```

If an entity's type isn't extended to the `TaxedCost` protocol but *is* extended to the `Cost` protocol, we dynamically install an extension for the concrete type to the `TaxedCost` protocol as well. Once it's installed, we can then remake the same call and it'll now be rerouted to the just-installed implementation.

Note that this only happens on the *first* call with an unknown entity type. Thereafter the protocol has an extension, and it'll no longer route through `Object`.