Extracted from:

Clojure Applied

From Practice to Practitioner

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Clojure Applied

From Practice to Practitioner

Ben Vandgrift
Alex Miller

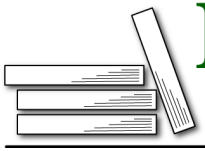
edited by Jacquelyn Carter

Clojure Applied

From Practice to Practitioner

Ben Vandgrift
Alex Miller

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina



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*.

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

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: B1.0—April 8, 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 where Clojure gives us choices about when, where, and how much validation we wish to provide. In areas where data is created by our code, we may wish 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 Schema[2] 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:

```
[prismatic/schema "0.3.2"]
```

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:

```
(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
     ])
```

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:

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

```
(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 just a special keyword Schema uses as a syntactic marker.

```
(require '[schema.core :as s])

(s/defrecord Ingredient
    [name     :- String
     quantity :- s/Int
     unit     :- s/Keyword])

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

Normal type hints and class names (like String) are valid schema descriptions. 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=> (clojure.pprint/pprint (s/explain Recipe))
(record
 user.Recipe
 {:name java.lang.String,
  :description java.lang.String,
  :ingredients
  [(record
    user.Ingredient
    {:name java.lang.String, :quantity Int, :unit Keyword})],
  :steps [java.lang.String],
  :servings Int})
```

We can also validate our data against the schema:

```
user=> (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 would get an error message:

```
user=> (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 did not 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.

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

user=> (doc add-ingredients)
-------------------------
user/add-ingredients
([recipe & ingredients])
  Inputs: [recipe :- Recipe & ingredients :- [Ingredient]]
  Returns: Recipe
```

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

We've now looked at various tradeoffs for representing domain entities, connecting entities together, and how to validate our entities. Next it's time to consider how we will 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 will 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 cost of each ingredient. We wish 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 will first invoke the dispatch function to produce a dispatch value, then select the best match for that value, and finally invoke 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 will just sketch these without fully implementing them:

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

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

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

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

(defmethod cost Recipe [recipe store]
  (reduce +$ zero-dollars
    (map cost (: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 are used to declare that a type *extends* a protocol.

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

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

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

Here we define the protocol Cost 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 as they leverage the underlying JVM runtime optimizations for this kind of dispatch (this is very 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, while 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 have the ability to define and use custom value hierarchies and declare preferences between implementations when there is 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 is not covered by protocols.

Value-based Dispatch

While type-based dispatch is the most common case in many programs, there are plenty of cases where value-based dispatch is needed 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 as follows:

```
(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] [_ _ oz] (* oz 16))

;; oz to lb
(defmethod convert [:oz :lb] [_ _ lb] (/ lb 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 multimethod convert dispatches on the *value* of the source and target type, 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 have made it to :default, then we are 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 are 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 added 1/2 pound (8 ounces) with 4 ounces and got 3/4 pound (12 ounces).

If we add new units to the system, we will 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. This means that 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 creating protocols that layer over other protocols.

For example, you might need to extend the recipe manager further to not only calculate 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 would like to layer the TaxedCost protocol over the existing Cost protocol but this is not allowed in Clojure:

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

Clojure does not 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) (tax-rate store))))
          (taxed-cost entity store))
      (assert (str "Unhandled entity: " entity)))))
```

If an entity's type is not extended to the TaxedCost protocol but it *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 re-make the same call and it will now be re-routed to the just-installed implementation.

Note that this only happens on the *first* call with an unknown entity type—after that, the protocol has an extension and it will no longer route through Object.