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

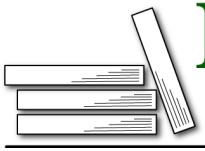
edited by Jacquelyn Carter

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

Putting It All Together

In many cases, processing your sequential data will follow a similar pattern:

1. Figure out what question you're trying to ask
2. Filter the data to remove unneeded elements
3. Transform the elements into the desired form
4. Reduce the transformed elements to the answer

The first item is often the most difficult, resting in the problem domain, or the realm of business logic. Once you have a clear question, Clojure provides the tools to process the data you have into an answer. For this section, we'll be using an online shopping cart.

Let's set the stage: in an online store, you have a catalog, a list of items for sale. These items are divided into departments. Customers place them in their carts, then checkout. This process creates a billing record. Your client has asked for a report summarizing departmental sales: for all settled carts, what is the total sales per department?

Our domain model is as follows:

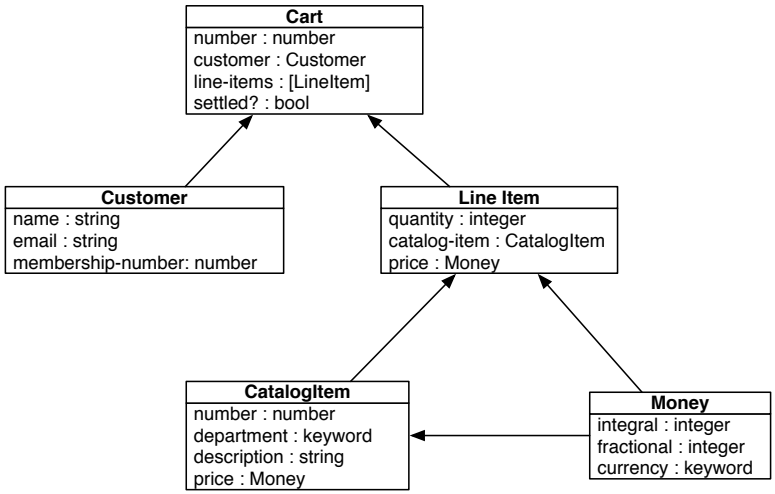
```
(require '[money :refer [make-money +$ *$])

(defrecord CatalogItem   [num dept desc price])
(defrecord Cart          [number customer line-items settled?])
(defrecord LineItem      [quantity catalog-item price])
(defrecord Customer      [name email membership-number])
```

After many checkouts, our carts might contain a vector of #Cart records as follows:

```
[#Cart{:number 116,
       :customer #Customer{:name "Danny Tanner",
                           :email "danny@fullhouse.example.com",
                           :membership-number 28374},
       :line-items [
         #LineItem{:quantity 3,
                   :catalog-item #CatalogItem{:num   664,
                                              :dept  :clothing,
                                              :desc  "polo shirt L",
                                              :price #Money{:integral  25
                                                            :fractonal 15
                                                            :currency  :usd},
                   :price #Money{:integral   75
                                 :fractional 45
                                 :currency   :usd}
         #LineItem{:quantity 1,
                   :catalog-item #CatalogItem{:num 621,
```

```
                                        :dept :clothing,
                                        :desc "khaki pants",
                                        :price #Money{:integral   35
                                                      :fractional 0
                                                      :currency   :usd},
                  :price #Money{:integral   35
                                :fractional 0
                                :currency   :usd}
    ],
  :settled? true}, ,,, ]
```

That's a pretty sizable data structure, and might be easier to understand with a class diagram:



What we're looking for is much simpler—a map of departments to dollar values, something like this:

```
{:clothing #Money{:integral 23864, :fractional 24, :currency :usd}
 :toys     #Money{:integral 11632, :fractional 77, :currency :usd}
 ,,, }
```

Let's walk through the steps to get from the contents of carts to our desired output. The first thing we need to do is find the data we care about.

Selection

The selection step of sequence processing identifies and creates a subsequence containing only the elements we're interested in, which we can obtain by using filter with our cart data.

When building your report, you only want to consider carts that have been settled. Until they are settled, they're only potential revenue, not actual revenue. Begin by reducing the size of the list, using filter.

```
(defn revenue-by-department [carts]
  (->> (filter :settled? carts)
       ,,,))
```

Using the :settled? keyword as a function, we can filter out all the carts for which :settled? is not true.

Transformation

Now that you have a sequence of settled carts, you can start separating out revenue by department. You'll discover that you don't need the cart at all—only the line-items and the catalog item it contains. You'll work one step at a time for now. Next step, create a sequence of all line items:

```
(defn revenue-by-department [carts]
  (->> (filter :settled? carts)
       (mapcat :line-items)
       ,,,))
```

The result of (mapcat :line-items ,,,) will look a lot like this:

```
[#LineItem{:quantity 3,
           :catalog-item #CatalogItem{:num   664,
                                      :dept  :clothing,
                                      :desc  "polo shirt L",
                                      :price #Money{:integral  25
                                                    :fractonal 15
                                                    :currency  :usd}},
           :price #Money{:integral   75
                         :fractional 45
                         :currency   :usd}},
 #LineItem{:quantity 1,
           :catalog-item #CatalogItem{:num 621,
                                      :dept :clothing,
                                      :desc "khaki pants",
                                      :price #Money{:integral   35
                                                    :fractional 0
                                                    :currency   :usd}},
           :price #Money{:integral   35
                         :fractional 0
                         :currency   :usd}}, ,,, ]
```

The mapcat function constructs an accumulation of the contents of the line-item vectors.

Using mapcat vs map + flatten

In place of mapcat, you could instead use map and flatten together to achieve a similar result. Whenever you are tempted to use flatten, go back one step and try to avoid creating the structure that needed to be flattened in the first place. Most commonly, this means using mapcat (to map and concatenate) rather than map.

The next step is to extract from each line item a map of the data we care about—the catalog item's :dept value, and the line-item parent's :price value. We do this with map and the helper function line-summary:

```
(defn- line-summary
  "Given a LineItem with a CatalogItem, returns a map
   containing the CatalogItem's :dept as :dept and LineItem's :price
   as :total"
  [line-item]
  {:dept  (get-in [:catalog-item :dept] line-item)
   :total (:price line-item)})

(defn revenue-by-department [carts]
  (->> (filter :settled? carts)
       (mapcat :line-items)
       (map line-summary)
       ,,,))
```

We're almost there! Now we have a sequence of maps that contain *only* the data we need to report on:

```
[{:dept :clothing :total #Money{ ,,, }
 {:dept :clothing :total #Money{ ,,, }
 {:dept :toys     :total #Money{ ,,, }
 {:dept :kitchen  :total #Money{ ,,, }
 {:dept :toys     :total #Money{ ,,, }]
```

From here, we can use group-by to construct a map with the department as the key, and a sequence of summaries as the values.

```
(defn revenue-by-department [carts]
  (->> (filter :settled? carts)
       (mapcat :line-items)
       (map line-summary)
       (group-by :dept)
       ,,,))
```

At this point, our data looks like this:

```
{:clothing [{:dept :clothing :total #Money{},
            {:dept :clothing :total #Money{}, ,,,]
 :toys     [{:dept :toys     :total #Money{},
```

```
           {:dept :toys     :total #Money{}, ,,,]
 :kitchen  [{:dept :kitchen  :total #Money{}, ,,,]}
```

We could take on an extra step to replace the summaries with a vector of the #Money{} values in the :total, but it's unnecessary. Instead, let's move on to our final step: summarizing those values.

Reduction

As with our line-summary function, you probably want to define a function to handle totaling each department that our reducing process can take advantage of.

```
(require '[money :refer [make-money +$ *$])

(defn- dept-total
  [m k v]
  (assoc m k (reduce +$ (map :total v))))

(defn revenue-by-department [carts]
  (->> (filter :settled? carts)
       (mapcat :line-items)
       (map line-summary)
       (group-by :dept)
       (reduce-kv dept-total)))
```

Within our piecewise function dept-total, we can see a microcosm of our usual sequence processing pipeline. In this case, map selects the :total from each element of the sequence, then reduce +$ sums it up.

You may find this alternative implementation of dept-total using the thread-last macro easier to read:

```
(defn- dept-total*
  [m k v]
  (assoc m k (->> (map :total v)
                  (reduce +$))))
```

That's it. We've reduced our initial vector of carts to a map of revenue by department. Our final data is in this shape, as promised:

```
{:clothing #Money{},
 :toys     #Money{},
 :kitchen  #Money{}, ,,, }
```

The data pipeline we've gone through in this section is fairly typical: select, transform, reduce. It's perhaps best for you to think of this as a unit of sequence processing—as we saw with the dept-total function, one unit of

sequence processing can enclose an entire other unit. As your practice develops, creating smooth pipelines will become more reflexive.

Another important thing to note, made obvious by our use of the thread-last macro (->>) in the revenue-by-department function: a sequence goes into and out of each step of the process. In fact, in the first three steps (filter, mapcat, and map), each element of the starting sequence could make it through all three steps successfully before the next element began, and the results would be the same. Those steps operate on a single element of the sequence at a time, without consideration for anything that's gone before or anything following. This is a good clue that using a transducer is also an option for this part of the pipeline.