UNC COMP 590-145
Refactoring & Dependency Injection
Monday, February 24, 2020
Announcements

- I'm still interested in any feedback you have; see the feedback form
- Midterm Exam 2 is Wednesday! A 1-page (double-sided) cheat sheet is allowed.
Private functions

Learning objective: enumerate the implications of making a function private in Clojure.

- You can call private functions from within their namespace
- You can't :refer private functions from another namespace
- You can't call private functions directly from another namespace
- You can call private functions indirectly:

```
((ns-resolve 'other-namespace 'private-fn) arg1 arg2 ,,,)
;; Or use the var-quote reader macro or var special form:
(#'other-namespace/private-fn arg1 arg2 ,,,)
;; a.k.a.
((var other-namespace/private-fn) arg1 arg2 ,,,)
```
Private function syntax

Learning objective: use the syntax for defining private functions in Clojure.

- Use `defn-` instead of `defn`

```
(defn- private-fn [arg1 arg2 ,,,,]
  ,,,,)
```

- Add `:^:private` metadata to `def` forms (works for data too)

```
(def ^:private private-fn ,,,)
```
Private functions and testing scope

Learning objective: *identify the scope of testing with respect to public vs. private functions.*

- Typically, only test the public interface (i.e. public functions)
- (Not an ironclad rule; sometimes testing private functions makes testing the public interface easier.)
Refactoring

Learning objective: define refactoring.

- Refactoring (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.
- Refactoring (verb): to restructure software by applying a series of refactorings without changing its observable behavior.
- Many people use the word refactoring more broadly to refer to any change to existing code. We'll stick with Fowler's definition in this class.
Refactoring and testing

- *Rely on your tests!*
- Test after every small change.
- Fast automated tests make refactoring easy!
Refactoring: flatten nested `if`s

Learning objective: identify when to use the refactoring technique.

Before:

```
(if (<= 90 grade)
  :A
  (if (<= 80 grade)
    :B
    (if (<= 70 grade)
      :C
      (if (<= 60 grade) :D :F)))))
```

After:

```
(cond
  (<= 90 grade) :A
  (<= 80 grade) :B
  (<= 70 grade) :C
  (<= 60 grade) :D
  :else :F)
```
Refactoring: introduce threading

Learning objective: *identify when to use the refactoring technique.*

```
(def people [{:name "Jeff", :age 38} ,,,])
```

Before: 
```
(prn (inc (:age (first people))))
```

After: 
```
(-> people first :age inc prn)
```
Refactoring: introduce threading (2)

Learning objective: *identify when to use the refactoring technique.*

```
(def people [{:name "Jeff", :age 38}, ...])
```

Before:  
```
(reduce + 0
  (map :age
    (filter #(< 30 (:age %))
      people)))
```

After:  
```
(->> people
  (filter #(< 30 (:age %)))
  (map :age)
  (reduce + 0))
```
Refactoring: name value

Learning objective: *identify when to use the refactoring technique.*

Before:

```lisp
(-> (filter #(= :faculty (:type %)) cs-employees)
    process-people
    make-report)
```

After:

```lisp
(let [cs-faculty (filter #(= :faculty (:type %)) cs-employees)]
  (-> cs-faculty process-people make-report))
```
Refactoring: name value/extract function

Note: you can name function values too, which is effectively the *extract function* refactoring.

Before:

```lisp
(-> (filter #(= :faculty (:type %)) cs-employees)
    process-people
    make-report)
```

After:

```lisp
(let [is-faculty? #(= :faculty (:type %))
        cs-faculty (filter is-faculty? cs-employees)]
  (cs-faculty process-people make-report))
```
Refactoring: extract function

Learning objective: *identify when to use the refactoring technique.*

Before:

```
(defn sum-of-squares [xs]
  (reduce + 0
      (map #(/* % %) xs)))
```

After (x2):

```
(def sum (partial reduce + 0))
(defn square [x] (* x x))
(defn sum-of-squares [xs]
  (sum (map square xs)))
```
Refactoring

Learning objective: apply refactoring techniques to example code.

Techniques:

- flatten nested if
- introduce threading
- name value
- extract function

Let's practice.
Refactoring practice (1)

(defn stddev [xs]
  (when (not-empty xs)
    (if (empty? (rest xs))
      0.0
      (Math/sqrt
        (* (/ 1 (dec (count xs)))
          (sum (map #(* % %)
            (sum (map #(let [mu (mean xs)]
              (map #(- % mu) xs)))))))))))

Let's extract the square function (and test!)
Refactoring practice (2)

```
(defn square [x] (* x x))
(defn stddev [xs]
  (when (not-empty xs)
    (if (empty? (rest xs))
      0.0
      (Math/sqrt
       (* (/ 1 (dec (count xs)))
        (sum (map square
            (sum (map square
                  (let [mu (mean xs)]
                    (map #(*/ % mu) xs)))))))))
Let's extract a zero-center function (and test!)
```
Refactoring practice (3)

(let [mu (mean xs)]
  (map #( - % mu) xs))

(defn stddev [xs]
  (when (not-empty xs)
    (if (empty? (rest xs))
      0.0
      (Math/sqrt
       (* (/ 1 (dec (count xs)))
        (sum (map square (zero-center xs))))))))

Let's flatten the if's (including the when) (and test!)
Refactoring practice (4)

```
(defn stddev [xs]
  (cond
    (empty? xs) nil
    (empty? (rest xs)) 0.0
    :else (Math/sqrt
      (* (/ 1 (dec (count xs)))
        (sum (map square (zero-center xs))))))))
```

Let's change calls to `empty?` to use `count` instead (and test!)
Refactoring practice (5)

```
(defn stddev [xs]
  (cond
    (= 0 (count xs)) nil
    (= 1 (count xs)) 0.0
    :else (Math/sqrt
      (* (/ 1 (dec (count xs)))
        (sum (map square (zero-center xs)))))))
```

Let's name the value \((\text{count } xs)\); we're using it in 3 places (and test!)
Refactoring practice (6)

```
(defn stddev [xs]
  (let [n (count xs)]
    (cond
      (= 0 n) nil
      (= 1 n) 0.0
      :else (Math/sqrt
        (* (/ 1 (dec n))
          (sum (map square (zero-center xs)))))))
```

Since we're using \( n \) in every predicate in \( \text{cond} \), let's use \( \text{case} \) instead (and test!)
Refactoring practice (7)

(defn stddev [xs]
  (let [n (count xs)]
    (case n
      0 nil
      1 0.0
    (Math/sqrt
      (* (/ 1 (dec n))
        (sum (map square (zero-center xs))))))))

Let's simplify the fraction: instead of multiplying by 1/N-1, let's divide by N-1 (and test!)
Refactoring practice (8)

```clojure
(defn stddev [xs]
  (let [n (count xs)]
    (case n
      0 nil
      1 0.0
      (Math/sqrt
         (/ (sum (map square (zero-center xs)))
            (dec n))))))
```

Let's introduce threading (and test!)
Refactoring practice (9)

```clojure
(defn stddev [xs]
  (let [n (count xs)]
    (case n
      0 nil
      1 0.0
      (Math/sqrt
       (- (->> xs
           zero-center
           (map square)) ; let's extract a sum-of-squares fn here
            sum
            (/ (dec n)))))
  )
)```
(defn stddev [xs]
  (let [n (count xs)]
    (case n
      0 nil
      1 0.0
      (Math/sqrt
        (->> xs
          zero-center
          sum-of-squares
          (/ (dec n))))))))
Refactoring practice: before and after

**Before:**

```clojure
(defn stddev [xs]
  (when (not-empty xs)
    (if (empty? (rest xs))
      0.0
      (Math/sqrt
       (* (/ 1 (dec (count xs)))
        (sum (map #(* % %)
               (let [mu (mean xs)]
                 (map #(-% mu) xs))))))))
```

**After:**

```clojure
(defn stddev [xs]
  (let [n (count xs)]
    (case n
      0 nil
      1 0.0
      (Math/sqrt
       (* (/ 1 (dec n))
        (sum-of-squares
         (zero-center xs)))))))
```
# Dependency Injection

Learning objective: *define dependency injection.*

*Dependency injection* is when the dependencies to a code unit are provided as arguments to the code unit, rather than being used directly.

<table>
<thead>
<tr>
<th>Without dependency injection:</th>
<th>With dependency injection:</th>
</tr>
</thead>
<tbody>
<tr>
<td>`(defn fetch-people []</td>
<td>`(defn fetch-people [db-conn]</td>
</tr>
<tr>
<td>(let [db-conn (db/get-connection)</td>
<td>(let [query ,,,]</td>
</tr>
<tr>
<td>query ,,,]</td>
<td>(execute-query db-conn query))</td>
</tr>
<tr>
<td>(execute-query db-conn query)))</td>
<td>(fetch-people (db/get-connection))</td>
</tr>
</tbody>
</table>
Dependency Injection: Analysis

Learning objective: discuss the tradeoffs of dependency injection.

Benefits

- Testing is easier: no `with-redefs` needed; just pass in stub as an argument
- Helps to isolate side effects by moving the effect to the caller

Costs

- Complicates the caller
Dependency Injection: Application

Learning objective: *modify a function to use dependency injection.*

Without dependency injection:

(ns math)
(def sum (partial reduce + 0))
(defn mean [xs]
  (/ (sum xs)
      (count xs)))

(ns user (:require math))
(math/mean [1 2 3])

With dependency injection:

(ns math)
(def sum (partial reduce + 0))
(defn mean [sum xs]
  (/ (sum xs)
      (count xs)))

(ns user (:require math))
(math/mean math/sum [1 2 3])
Dependency Injection: Case study

Is dependency injection for \texttt{sum} a good idea here?

\begin{verbatim}
(defn mean [xs]
  (/ (sum xs)
    (count xs)))
\end{verbatim}

- Probably not. The cost (complicating the caller) outweighs the benefits (before, no effects in fn body, and tests weren't complex).
- Rule of thumb: dependency injection only makes sense when the dependency contains effects.
Parting shot

- A quick question on Poll Everywhere