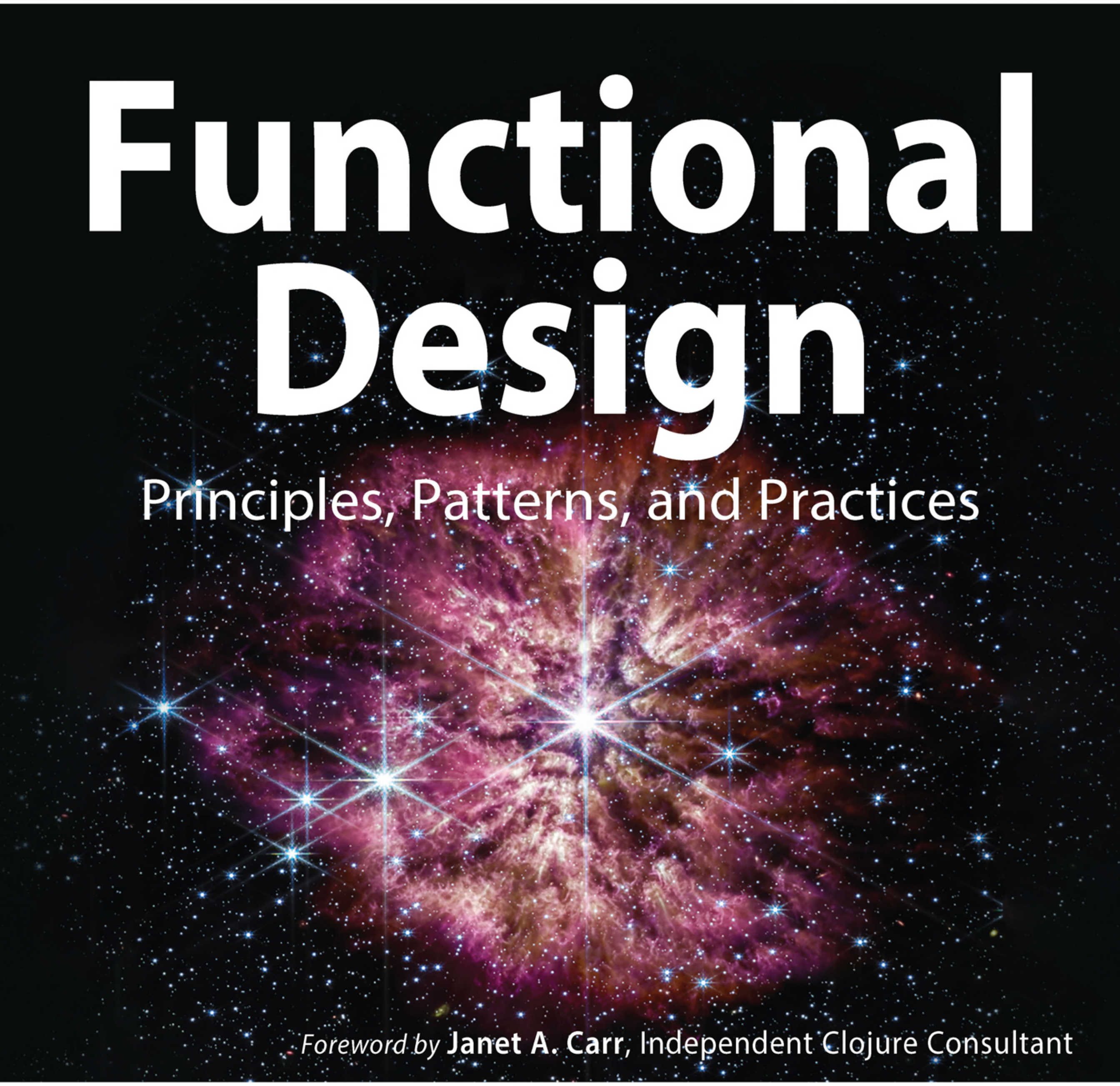
Robert C. Martin Series

Functional Design

Principles, Patterns, and Practices



Foreword by **Janet A. Carr**, Independent Clojure Consultant

Robert C. Martin

Functional Design

```
    for (Driver d : drivers) {
      if (!d.getRumors().equals(rumors))
        return true;
    }
    return false;
  }
}
```

A quick perusal of this code will convince you that it is written in a very traditional OO style and that the objects encapsulate their own state relatively well.

Clojure

When writing the Clojure version I did not start out with a design sketch. Rather, I depended upon my TDD tests to help me with the design. The tests are as follows:

```
(ns gossiping-bus-drivers-clojure.core-spec
  (:require [speclj.core :refer :all]
            [gossiping-bus-drivers-clojure.core :refer :all]))

(describe "gossiping bus drivers"
  (it "drives one bus at one stop"
    (let [driver (make-driver "d1" [:s1] #{:r1}[4])
          world [driver]
          new-world (drive world)]
      (should= 1 (count new-world))
      (should= :s1 (-> new-world first :route first))))

  (it "drives one bus at two stops"
    (let [driver (make-driver "d1" [:s1 :s2] #{:r1})
          world [driver]
          new-world (drive world)]
      (should= 1 (count new-world))
      (should= :s2 (-> new-world first :route first))))
```

4. #{. . .} represents a set in Clojure. A *set* is a list of items that has no duplicates.

```
(it "drives two buses at some stops"
  (let [d1 (make-driver "d1" [:s1 :s2] #{:r1})
        d2 (make-driver "d2" [:s1 :s3 :s2] #{:r2})
        world [d1 d2]
        new-1 (drive world)
        new-2 (drive new-1)]
    (should= 2 (count new-1))
    (should= :s2 (-> new-1 first :route first))
    (should= :s3 (-> new-1 second :route first))
    (should= 2 (count new-2))
    (should= :s1 (-> new-2 first :route first))
    (should= :s2 (-> new-2 second :route first))))

(it "gets stops"
  (let [drivers #{{:name "d1" :route [:s1]}
                  {:name "d2" :route [:s1]}
                  {:name "d3" :route [:s2]}}]
    (should= {:s1 [{:name "d1" :route [:s1]}
                   {:name "d2" :route [:s1]}]
              :s2 [{:name "d3", :route [:s2]}]}
             (get-stops drivers)))
  )

(it "merges rumors"
  (should= [{:name "d1" :rumors #{:r2 :r1}}
            {:name "d2" :rumors #{:r2 :r1}}]
           (merge-rumors [{:name "d1" :rumors #{:r1}}
                          {:name "d2" :rumors #{:r2}}])))


(it "shares gossip when drivers are at same stop"
  (let [d1 (make-driver "d1" [:s1 :s2] #{:r1})
        d2 (make-driver "d2" [:s1 :s2] #{:r2})
        world [d1 d2]
        new-world (drive world)]
    (should= 2 (count new-world))
    (should= #{:r1 :r2} (-> new-world first :rumors))
    (should= #{:r1 :r2} (-> new-world second :rumors))))
  (it "passes acceptance test 1"
    (let [world [(make-driver "d1" [3 1 2 3] #{1})
                 (make-driver "d2" [3 2 3 1] #{2})
                 (make-driver "d3" [4 2 3 4 5] #{3})]]
      (should= 6 (drive-till-all-rumors-spread world))))

  (it "passes acceptance test 2"
    (let [world [(make-driver "d1" [2 1 2] #{1})
                 (make-driver "d2" [5 2 8] #{2})]]
          (should= :never (drive-till-all-rumors-spread world))))
  )
```

There are some interesting similarities between the Java tests and the Clojure tests. They are both quite wordy; although the Clojure tests contain half as many lines. The Java version has 12 tests whereas the Clojure version has only 8. This difference has a lot to do with the way the two different solutions were partitioned. The Clojure tests also play pretty fast and loose with the data.

Consider, for example, the `"merges rumors"` test. The `merge-rumors` function expects a list of drivers; however, the test does not create completely formed drivers. Rather, it creates abbreviated structures that look like drivers as far as the `merge-rumors` function is concerned.

The solution is all contained in a single, very short file:

```
(ns gossiping-bus-drivers-clojure.core
  (:require [clojure.set :as set]))

(defn make-driver [name route rumors]
  (assoc[5] {} :name name :route (cycle[6] route) :rumors rumors))

(defn move-driver [driver]
```

5. `assoc` adds elements to a map. `(assoc {} :a 1)` returns `{:a 1}`.

6. `cycle` returns a lazy (and "infinite") list that simply repeats the input list endlessly. Thus, `(cycle [1 2 3])` returns `[1 2 3 1 2 3 1 2 3 …]`.

```
  (update[7] driver :route rest))

(defn move-drivers [world]
  (map move-driver world))

(defn get-stops [world]
  (loop [world world
         stops {}]
    (if (empty? world)
      stops
      (let [driver (first world)
            stop (first (:route driver))
            stops (update stops stop conj driver)]
        (recur (rest world) stops)))))

(defn merge-rumors [drivers]
  (let [rumors (map :rumors drivers)
        all-rumors (apply set/union[8] rumors)]
      (map #(assoc % :rumors all-rumors) drivers)))

(defn spread-rumors [world]
  (let [stops-with-drivers (get-stops world)
        drivers-by-stop (vals[9] stops-with-drivers)]
    (flatten[10] (map merge-rumors drivers-by-stop))))

(defn drive [world]
  (-> world move-drivers spread-rumors))

(defn drive-till-all-rumors-spread [world]
  (loop [world (drive world)
```

7. The `update` function returns a new map with one element changed. `(update m k f a)` changes the `k` element of `m` by applying the function `(f e a)`, where `e` is the old value of element `k`. Thus, `(update {:x 1} :x inc)` returns `{:x 2}`.

8. The `union` function is from the `set` namespace. Notice the `ns` at the top aliases the `clojure.set` namespace to just `set`.

9. `vals` returns a list of all the values in a map. `keys` returns a list of all the keys in a map.

10. The `flatten` function turns a list of lists into a list of all the elements. So `(flatten [[1 2][3 4]])` returns `[1 2 3 4]`.

```
         time 1]
    (cond
      (> time 480) :never
      (apply = (map :rumors world)) time
      :else (recur (drive world) (inc time)))))
```

This solution is 42 lines, whereas the Java solution is 145 lines spread among five files.

Both solutions have the concept of a Driver, but I made no attempt to encapsulate the concepts of Route, Stop, and Rumor into independent objects. They all just happily live within the Driver.

Worse, the Driver "object" is not an object in the traditional OO sense. It has no methods. There is one method in the system, `move-driver`, that operates on a single Driver, but it's just a little helper function for the more interesting `move-drivers` function.

Six out of the eight functions take only the `world` as an argument. Thus, we might say that the only true object in this system is the `world`, and it has five methods. But even that is a stretch.

Even if we decide that the Driver is a kind of object, it is not mutable. The simulated `world` is nothing more than a list of immutable Drivers. The `drive` function accepts the `world` and produces a new `world` in which all the Drivers have been moved one step, and Rumors have been spread at any stop where more than one Driver has arrived.

That `drive` function is an example of an important concept. Notice how the `world` passes through a pipeline of functions. In this case there are only two, `move-drivers` and `spread-rumors`, but in larger systems the pipeline can be quite long. At each stage along that pipeline the `world` is modified into a slightly new form.

This tells us that the partitioning of this system is not about objects, but about functions. The relatively unpartitioned data passes from one independent function to the next.

You might argue that the Java code is relatively straightforward, whereas the Clojure code is too dense and obscure. Believe me when I say that it does not take very long to get comfortable with that density and that the perceived obscuration is an illusion based on unfamiliarity.

Is the lack of partitioning in the Clojure version a problem? Not at its current size; but if this program were to grow the way most systems grow, that problem would assert itself with a vengeance. Partitioning OO programs is a bit more natural than partitioning functional programs because the dividing lines are much more obvious and pronounced.

On the other hand, the dividing lines we chose for the Java version are not guaranteed to lead to an *effective* partitioning. The warning is in the `drive` function of the Clojure program. It seems likely that a better partitioning of this system might lie along the different operations that manipulate the world, rather than things like Routes, Stops, and Rumors.

Conclusion

We saw some differences in the Prime Factors and Bowling Game katas; but the differences were relatively minor. The differences in the Gossiping Bus Drivers kata were much more pronounced. This is likely because that last kata was a bit larger than the first two (I'd say twice the size), and also because it was a true finite state machine.

A *finite state machine* moves from state to state, taking actions that depend upon the incoming events and the current state. When such systems are written in an OO style, the state tends to be stored in mutable objects that have dedicated methods. But in a functional style, the state remains