Extracted from:

Programming Clojure

Second Edition

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

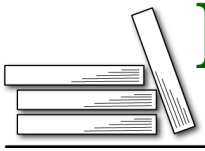
Programming Clojure

Second Edition

Stuart Halloway
Aaron Bedra

Foreword by Rich Hickey, creator of Clojure

Edited by Michael Swaine



Pragmatic Bookshelf

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 *http://pragprog.com*.

The team that produced this book includes:

Michael Swaine (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-934356-86-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—April 2012

One piece of Clojure's solution to the expression problem is the protocol. Protocols provide a flexible mechanism for abstraction that leverages the best parts of interfaces by providing only specification, not implementation, and by letting datatypes implement multiple protocols. Additionally, protocols address the key weaknesses of interfaces by allowing nonintrusive extension of existing types to support new protocols.

The following are the strengths of protocols:

- Datatypes can implement multiple protocols.
- Protocols provide only specification, not implementation, which allows implementation of multiple interfaces without the problems associated with multiple-class inheritance.
- Existing datatypes can be extended to implement new interfaces with no modification to the datatypes.
- Protocol method names are namespaced, so there is no risk of name collision when multiple parties choose to extend the same extant type.

The defprotocol macro works just like definterface, but now we are able to extend existing datatypes to implement our new abstraction.

```
(defprotocol name & opts+sigs)
```

Let's redefine IOFactory as a protocol, instead of an interface.

```
(defprotocol IOFactory
  "A protocol for things that can be read from and written to."
  (make-reader [this] "Creates a BufferedReader.")
  (make-writer [this] "Creates a BufferedWriter."))
```

Notice we can include a document string for the protocol as a whole, as well as for each of its methods. Now let's extend java.io.InputStream and java.io.OutputStream to implement our IOFactory protocol.

We use the extend function to associate an existing type to a protocol and to provide the required function implementations, usually referred to as *methods* in this context. The parameters to extend are the name of the type to extend, the name of the protocol to implement, and a map of method implementations, where the keys are keywordized versions of the method names.

```
(extend type & proto+mmaps)
```

The make-reader implementation for an InputStream just wraps the value passed to it in a BufferedReader.

src/examples/protocols.clj
```
(extend InputStream
  IOFactory
  {:make-reader (fn [src]
                  (-> src InputStreamReader. BufferedReader.))
   :make-writer (fn [dst]
                  (throw (IllegalArgumentException.
                          "Can't open as an InputStream.")))})
```

Similarly, the implementation of make-writer for an OutputStream wraps its given input in a BufferedWriter. And since you can't write to an InputStream or read from an OutputStream, the respective implementations of make-writer and make-reader throw IllegalArgumentExceptions.

```
(extend OutputStream
  IOFactory
  {:make-reader (fn [src]
                  (throw
                   (IllegalArgumentException.
                    "Can't open as an OutputStream.")))
   :make-writer (fn [dst]
                  (-> dst OutputStreamWriter. BufferedWriter.))})
```

We can extend the java.io.File type to implement our IOFactory protocol with the extend-type macro, which provides a slightly cleaner syntax than extend.

```
(extend-type type & specs)
```

It takes the name of the type to extend and one or more specs, which includes a protocol name and its respective method implementations.

```
(extend-type File
  IOFactory
  (make-reader [src]
    (make-reader (FileInputStream. src)))
  (make-writer [dst]
    (make-writer (FileOutputStream. dst))))
```

Notice that we create an InputStream, specifically, a FileInputStream, from our file and then make a recursive call to make-reader, which will be dispatched to the implementation defined earlier for InputStreams. We use the same recursive pattern for the make-writer method, as well as for the methods of the following remaining types.

We can extend the remaining types all at once with the extend-protocol macro:

```
(extend-protocol protocol & specs)
```

This takes the name of the protocol followed by one or more type names with their respective method implementations.

```
(extend-protocol IOFactory
  Socket
  (make-reader [src]
    (make-reader (.getInputStream src)))

  (make-writer [dst]
    (make-writer (.getOutputStream dst)))

  URL
  (make-reader [src]
    (make-reader
     (if (= "file" (.getProtocol src))
       (-> src .getPath FileInputStream.)
       (.openStream src))))

  (make-writer [dst]
    (make-writer
     (if (= "file" (.getProtocol dst))
       (-> dst .getPath FileInputStream.)
       (throw (IllegalArgumentException.
               "Can't write to non-file URL"))))))
```

Now let's put it all together.

```
(ns examples.io
  (:import (java.io File FileInputStream FileOutputStream
                    InputStream InputStreamReader
                    OutputStream OutputStreamWriter
                    BufferedReader BufferedWriter)
           (java.net Socket URL)))

(defprotocol IOFactory
  "A protocol for things that can be read from and written to."
  (make-reader [this] "Creates a BufferedReader.")
  (make-writer [this] "Creates a BufferedWriter."))

(defn gulp [src]
  (let [sb (StringBuilder.)]
    (with-open [reader (make-reader src)]
      (loop [c (.read reader)]
        (if (neg? c)
          (str sb)
          (do
            (.append sb (char c))
            (recur (.read reader))))))))

(defn expectorate [dst content]
  (with-open [writer (make-writer dst)]
    (.write writer (str content))))

(extend-protocol IOFactory
```

```
  InputStream
  (make-reader [src]
    (-> src InputStreamReader. BufferedReader.))

  (make-writer [dst]
    (throw
     (IllegalArgumentException.
      "Can't open as an InputStream.")))

  OutputStream
  (make-reader [src]
    (throw
     (IllegalArgumentException.
      "Can't open as an OutputStream.")))

  (make-writer [dst]
    (-> dst OutputStreamWriter. BufferedWriter.))

  File
  (make-reader [src]
    (make-reader (FileInputStream. src)))

  (make-writer [dst]
    (make-writer (FileOutputStream. dst)))

  Socket
  (make-reader [src]
    (make-reader (.getInputStream src)))

  (make-writer [dst]
    (make-writer (.getOutputStream dst)))

  URL
  (make-reader [src]
    (make-reader
     (if (= "file" (.getProtocol src))
       (-> src .getPath FileInputStream.)
       (.openStream src))))


  (make-writer [dst]
    (make-writer
     (if (= "file" (.getProtocol dst))
       (-> dst .getPath FileInputStream.)
       (throw (IllegalArgumentException.
               "Can't write to non-file URL"))))))
```