Fluent, Composable Error Handling

Brian Beckman

September 7, 2013

Contents

1 Introduction

Consider a program composed of a sequence of *serially dependent* computations, any of which produces either a value to feed to the next computation or an error value. If any component in the sequence produces an error value, that value should be the result of the entire sequence and all computations following the erroring computation should be abandoned.

The desired behavior is similar to that of the Maybe monad,[1] the difference being that *Maybe* just produce *Nothing* if anything goes wrong. The consumer of the computation doesn't know what stage of the pipeline failed nor any details at all about the error. *Maybe* suppresses all that. Such a situation is not tolerable in the real world. Consider the example of a database retrieval followed by a few web-service calls followed by a filter and transformation followed by a logging call followed by output to UI components. If something goes wrong in this sequence of computations, we need to know exactly where and as much detail as we can get about the failure. But we certainly don't want any computations downstream of the failure to be attempted.

1.1 How-To's

This is a literate program.[2] That means that source code *and* documentation spring from the same, plain-text source files. That gives us a fighting chance of keeping knowledge and source coherent.

This file is named *ex1.org*. It's an outline in plain text with rather obvious structure. Top-level headlines get a single star; second-level headlines get two stars, and so on; LaTeX can be freely written anywhere; source-code examples abound to copy-and-paste, and text explaining how to build and run the source is nearby.

You can edit the file with any plain-text editor. Emacs offers some automation in generating the typeset output, *ed1.pdf*, and the source code of the application right out of the *org* file. To generate source code, issue the emacs command `org-babel-tangle`. To generate documentation, issue the emacs command `org-latex-export-to-pdf`.

If you don't want to use emacs, by all means use *any* plain-text editor. We are working on a batch process via *make* so that you can just clone the repo, make whatever edits you like, type *make*, and have a complete PDF file and a complete directory full of source code.

2 Tangle to Leiningen

2.1 Files in the Project Directory

In our example, the top-level directory doesn't have a name – put our *org* file in that directory. The Leiningen project directory will have the same

[1]`http://en.wikipedia.org/wiki/Monad_(functional_programming)`#The$_{\text{Maybemonad}}$
[2]See `http://en.wikipedia.org/wiki/Literate_programming`.

name as our *org* file. Our *org* file is named `ex1.org` and we want a directory tree rooted at `ex1` exactly as above.

Start with the contents of the project directory, `ex1`. Each org-mode babel source-code block will name a file path – including sub-directories – after a `:tangle` keyword on the `#+BEGIN_SRC` command of org-mode.

2.1.1 .Gitignore

First, we must create the `.gitignore` file that tells `git` not to check in the ephemeral *ejecta* of build processes like `maven` and `javac`. When we gain more confidence and adoption with tangle and LaTeX, we will even ignore the PDF file and the generated source tree, saving *only* the *org* file in the repository.

2.1.2 README.md

Next, we produce a `README.md` in `markdown` syntax for the entire project:

```
# ex1
A Clojure library designed to do SOMETHING.
## Usage
TODO
## License
Copyright © 2013 TODO
```

2.1.3 project.clj

Next is the `project.clj` file required by Leiningen for fetching dependencies, loading libraries, and other housekeeping. If you are running the Clojure REPL inside emacs, you must visit this file *after tangling it out of the org file*, and then run

```
M-x nrepl-jack-in
```

in that buffer (see more in section 3).

```
(defproject ex1 "0.1.0-SNAPSHOT"
  :description "Project Fortune's Excel Processor"
  :url "http://example.com/TODO"
  :license {:name "TODO"
            :url "TODO"}
  :dependencies [[org.clojure/clojure  "1.5.1"]
                 [org.clojure/data.zip "0.1.1"]
                 [dk.ative/docjure     "1.6.0"]
                ])
```

2.2 The Documentation Subdirectory

Mimicking Leiningen's documentation subdirectory, it contains the single file `intro.md`, again in `markdown` syntax.

```
# Introduction to ex1
TODO: The project documentation is the .org file that produced
this output, but it still pays to read
http://jacobian.org/writing/great-documentation/what-to-write/
```

2.3 Core Source File

By convention, the core source files go in a subdirectory named `./ex1/src/ex1`. This convention allows the Clojure namespaces to map to Java packages.

The following is our core source file, explained in small pieces. The *org* file contains a spec for emitting the tangled source at this point. This spec is not visible in the generated PDF file, because we want to individually document the small pieces. The invisible spec simply gathers up the source of the small pieces from out of their explanations and then emits them into the source directory tree, using another tool called *noweb*.[3] This is not more complexity for you to learn, rather it is just a way for you to feel comfortable with literate-programming magic.

2.3.1 The Namespace

First, we must mention the libraries we're using. This is pure ceremony, and we get to the meat of the code immediately after. These library-mentions correspond to the `:dependencies` in the `project.clj` file above. Each `:use` or `:require` below must correspond to either an explicit dependency in the

[3] `http://orgmode.org/manual/Noweb-reference-syntax.html`

`project.clj` file or to one of several implicitly loaded libraries. Leiningen loads libraries by processing the `project.clj` file above. We bring symbols from those libraries into our namespace so we can use the libraries in our core routines.

To ingest and compile raw Excel spreadsheets, we use the built-in libraries `clojure.zip` for tree navigation and `clojure.xml` for XML parsing, plus the third-party libraries `clojure.data.zip.xml` and `dk.ative.docjure.spreadsheet`. The following brings these libraries into our namespace:

```
(ns ex1.core
  (:use [clojure.data.zip.xml :only (attr text xml->)]
        [dk.ative.docjure.spreadsheet] )
  (:require [clojure.xml :as xml]
            [clojure.zip :as zip]))
```

2.3.2 Data Instances

Next, we create a couple of data instances to manipulate later in our unit tests. The first one ingests a trivial XML file and the second one converts the in-memory data structure into a *zipper*,[4] a very modern, functional tree-navigation facility. These instances will test our ability to freely navigate the raw XML form of Excel spreadsheets:

```
(def xml (xml/parse "myfile.xml"))
(def zippered (zip/xml-zip xml))
```

2.3.3 A Test Excel Spreadsheet

Finally, we use `docjure` to emit a test Excel spreadsheet, which we will read in our unit tests and verify some operations on it. This code creates a workbook with a single sheet in a rather obvious way, picks out the sheet and its header row, and sets some visual properties on the header row. We can open the resulting spreadsheet in Excel after running `lein test` and verify that the `docjure` library works as advertised.

[4]`http://richhickey.github.io/clojure/clojure.zip-api.html`

```
(let [wb (create-workbook "Price List"
                          [["Name"       "Price"]
                           ["Foo Widget" 100]
                           ["Bar Widget" 200]])
      sheet (select-sheet "Price List" wb)
      header-row (first (row-seq sheet))]
  (do
    (set-row-style!
      header-row
      (create-cell-style! wb
        {:background :yellow,
         :font       {:bold true}}))
    (save-workbook! "spreadsheet.xlsx" wb)))
```

2.4 Core Unit-Test File

Unit-testing files go in a subdirectory named `./ex1/test/ex1`. Again, the directory-naming convention enables valuable shortcuts from Leiningen.

As with the core source files, we include the built-in and downloaded libraries, but also the `test framework` and the `core` namespace, itself, so we can test the functions in the core.

```
(ns ex1.core-test
  (:use [clojure.data.zip.xml :only (attr text xml->)]
        [dk.ative.docjure.spreadsheet]
  )
  (:require [clojure.xml :as xml]
            [clojure.zip :as zip]
            [clojure.test :refer :all]
            [ex1.core :refer :all]))
```

We now test that the zippered XML file can be accessed by the *zipper* operators. The main operator of interest is `xml->`, which acts a lot like Clojure's *fluent-style* [5] *threading* operator `->`.[6] It takes its first argument, a zippered XML file in this case, and then a sequence of functions to apply. For instance, the following XML file, when subjected to the functions `:track`, `:name`, and `text`, should produce `'("Track one" "Track two")`

[5] `http://en.wikipedia.org/wiki/Fluent_interface`

[6] `http://clojuredocs.org/clojure_core/clojure.core/-%3E`

```
<songs>
  <track id="t1"><name>Track one</name></track>
  <ignore>pugh!</ignore>
  <track id="t2"><name>Track two</name></track>
</songs>
```

Likewise, we can dig into the attributes with natural accessor functions [7]#+name: docjure-test-namespace

```
(deftest xml-zipper-test
  (testing "xml and zip on a trivial file."
    (are [a b] (= a b)
      (xml-> zippered :track :name text) '("Track one" "Track two")
      (xml-> zippered :track (attr :id)) '("t1" "t2"))))
```

Next, we ensure that we can faithfully read back the workbook we created *via* `docjure`. Here, we use Clojure's `thread-last` macro to achieve fluent style:

```
(deftest docjure-test
  (testing "docjure read"
    (is (=

      (->> (load-workbook "spreadsheet.xlsx")
           (select-sheet "Price List")
           (select-columns {:A :name, :B :price}))

      [{:name "Name"      , :price "Price"}, ; don't forget header row
       {:name "Foo Widget", :price 100.0  },
       {:name "Bar Widget", :price 200.0  }]

      ))))
```

[7]Clojure treats colon-prefixed keywords as functions that fetch the corresponding values from hashmaps, rather like the dot operator in Java or JavaScript; Clojure also treats hashmaps as functions of their keywords: the result of the function call `({:a 1} :a)` is the same as the result of the function call `(:a {:a 1})`

3 A REPL-based Solution

To run the REPL for interactive programming and testing in org-mode, take the following steps:

1. Set up emacs and nRepl (TODO: explain; automate)
2. Edit your init.el file as follows (TODO: details)
3. Start nRepl while visiting the actual |project-clj| file.
4. Run code in the org-mode buffer with `C-c C-c`; results of evaluation are placed right in the buffer for inspection; they are not copied out to the PDF file.

```
[(xml-> zippered :track :name text)        ; ("Track one" "Track two")
 (xml-> zippered :track (attr :id))]       ; ("t1" "t2")

(->> (load-workbook "spreadsheet.xlsx")
     (select-sheet "Price List")
     (select-columns {:A :name, :B :price}))

(run-all-tests)
```

4 References

5 Conclusion

Fu is Fortune.