Fluent, Composable Error Handling

Brian Beckman

September 9, 2013

Contents

1 Introduction

Consider a program composed of *serially dependent computations*, any of which produces either a value to feed to the next computation in-line, or an error. If any computation in the sequence produces an error, no downstream computations should be attempted and the error should be the result of the entire sequence.

We show a sequence of solutions in Java and Clojure for this program. We strive for fluent style, which minimizes the number of temporary variables

to name and manage. This style directly mimics the abstract data flow of the solution.

We find that fluent style is only available in Java if errors are propagated by exception. The reason is that the code for handling errors is in an independent *catch* block, and the non-error code path can be written without explicitly handling errors.

With functional programming in general and Clojure in particular, we can have fluent code without exceptions because we can abstract error handling in a monad.

2 Motivating Example Problem

As a concrete example, suppose we get an authorization token, do a database lookup, do a web-service call, filter the results of that call, do another web-servie call, and then combine the results. The data flow of our program resembles that in figure 1.

2.1 Fluent Solution in Java, No Error Handling

In the C++ – like languages, including JavaScript and Java, we might keep intermediate results in instance variables and model the flow as methods that produce objects from objects, that is, as **transforms**. This style is called **fluent style** because the text of the program resembles the flow in the diagram.

Imagine a *main* program like the following, in which transforms are on their own lines, indented from their *sources* by one 4-space tab stop. The **source** of a transform is an expression that produces an object to feed downstream.

```
public static void main(String[] args) {
    Computation databaseResults = new Computation()
        .authorize()
        .readDatabase();
    String result = databaseResults
        .callWebService()
        .filterResults()
        .combineResults(databaseResults
            .callOtherWebService());
    System.out.println(result); }
```

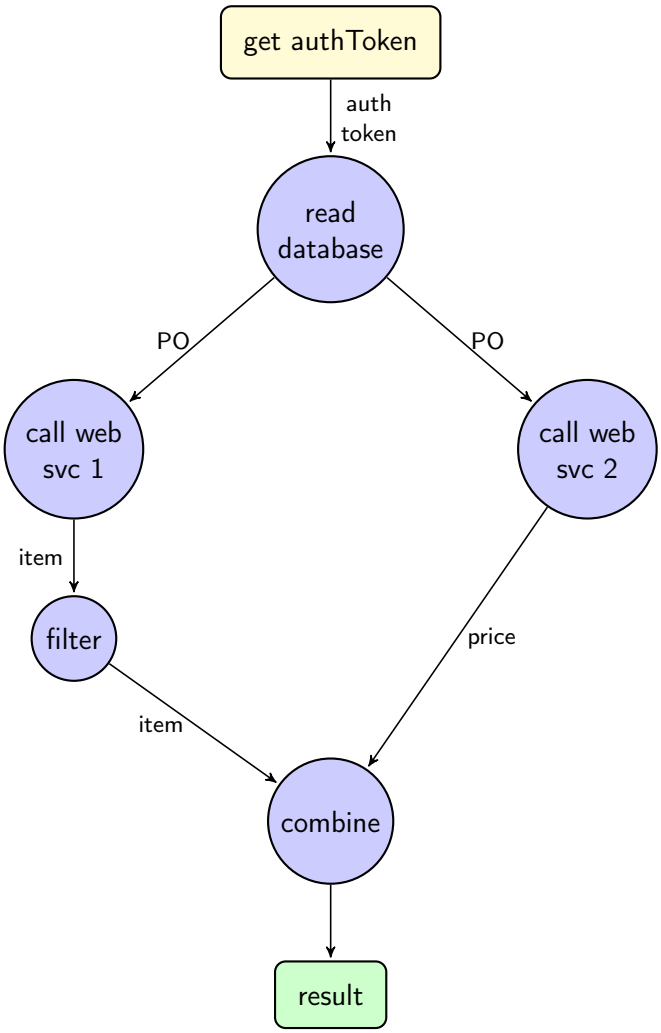

Figure 1: Serially dependent computations

Notice that we save the *Computation* produced by reading the database in its own local variable, namely *databaseResults*. We do so because the dataflow branches from that result: we need the result twice, once for calling the first web service and once for calling the second web service. If not for this branching and recombining of the dataflow, we might have written the entire program as one, fluent expression with no intermediate variables. In any event, the correspondence between the diagram of the program and the desired code is obvious. The code *looks like* the diagram. Changes to the diagrammatic specification propagate straightforwardly to changes in the code, all due to fluent style.

Also note the non-idiomatic compressed style, minimizing blank lines and lines with just one closing brace. We adopt this style to save space in this paper; production versions of such code would have more white space.

The following is a complete program that mocks out the database and web-service calls as static JSON objects encoded in strings, and can be compiled and executed, even online in some sandbox like `http://www.compileonline.com/compile_java_online.php`.

The most important thing to note about this code is that each method, *e.g.*, *authorize*, *readDatabase*, etc., takes an object – implicitly as *this* – and returns an object. This convention enables the fluent style by chaining transforms with dot. It so happens that we return *this* from each method. We could create a new object in each transform, but it would not simplify the code and would not give any performance advantage (quite the opposite). Instead, we propagate values through instance variables, namely *authToken*, *databaseResults*, etc.

```
public class Computation {
    private String authToken;
    private String databaseResults;
    private String webServiceCallResults;
    private String filteredWebServiceCallResults;
    private String otherWebServiceCallResults;

    public Computation () {}
    public Computation authorize() {
        authToken = "John's credentials";
        return this; }
    public Computation readDatabase() {
        databaseResults = "{\"name\":\"John\", \"PO\":\"421357\"}";
        return this; }
```

```
    public Computation callWebService() {
        webServiceCallResults =
            "[{\"item\":\"camera\"}, {\"item\":\"shoes\"}]";
        return this; }
    public Computation filterResults() {
        filteredWebServiceCallResults =
            "[{\"item\":\"camera\"}]";
        return this; }
    public Computation callOtherWebService() {
        otherWebServiceCallResults = "{\"price\":\"420.00\"}";
        return this; }
    public String combineResults(Computation other) {
        return "{[" + filteredWebServiceCallResults +
            "," + otherWebServiceCallResults + "]}"; }

    public static void main(String[] args) {
        Computation databaseResults = new Computation()
            .authorize()
            .readDatabase();
        String result = databaseResults
            .callWebService()
            .filterResults()
            .combineResults(databaseResults
                .callOtherWebService());
        System.out.println(result);
}   }
```

2.2 Fluent Solution in Java, with Exceptions

The program above has *no* error handling. At this point, let us agree that we *must* have error handling in real-world programs.

One of the better techniques for error handling in fluent style is with exceptions. If each sub-computation is responsible for throwing its own exception, then a single try-catch suffices to get error details out of the overall sequence, leaving the essential dataflow unchanged. Our main routine has minimal changes, and becomes simply

```
public static void main(String[] args) {
    try {
        Computation databaseResults = new Computation()
```

```
            .authorize()
            .readDatabase();
        String result = databaseResults
            .callWebService()
            .filterResults()
            .combineResults(databaseResults
                .callOtherWebService());
        System.out.println(result); }
    catch (Exception e) {
        System.out.println(e.getMessage());
}   }
```

noting, in passing, that we ignore resource management (database connections, sockets, file handles, etc.) in this paper.[1]

Let's give each mocked sub-computation a 10% chance of erroring, and our entire sample becomes the following:

```
import java.util.Random;
public class Computation {
    private String authToken;
    private String databaseResults;
    private String webServiceCallResults;
    private String filteredWebServiceCallResults;
    private String otherWebServiceCallResults;
    private static Random random = new java.util.Random();
    private static Boolean randomlyError() {
        return random.nextDouble() < 0.10; }

    public Computation () {}
    public Computation authorize() throws Exception {
        if (randomlyError()) { throw new Exception("auth errored"); }
        authToken = "John's credentials";
        return this; }
    public Computation readDatabase() throws Exception {
        if (randomlyError()) { throw new Exception("database errored"); }
        databaseResults = "{\"name\":\"John\", \"PO\":\"421357\"}";
        return this; }
    public Computation callWebService() throws Exception {
```

[1] Idiomatically, resources can be handled in a *finally* clause or with Java 7's Automatic Resource Management (ARM). See `http://bit.ly/15GYkMh`

```
        if (randomlyError()) { throw new Exception("ws1 errored"); }
        webServiceCallResults =
            "[{\"item\":\"camera\"}, {\"item\":\"shoes\"}]";
        return this; }
    public Computation filterResults() throws Exception {
        if (randomlyError()) { throw new Exception("filter errored"); }
        filteredWebServiceCallResults =
            "[{\"item\":\"camera\"}]";
        return this; }
    public Computation callOtherWebService() throws Exception {
        if (randomlyError()) { throw new Exception("ws2 errored"); }
        otherWebServiceCallResults = "{\"price\":\"420.00\"}";
        return this; }
    public String combineResults(Computation other) throws Exception {
        if (randomlyError()) { throw new Exception("combine errored"); }
        return "{[" + filteredWebServiceCallResults +
            "," + otherWebServiceCallResults + "]}"; }

    public static void main(String[] args) {
        try {
            Computation databaseResults = new Computation()
                .authorize()
                .readDatabase();
            String result = databaseResults
                .callWebService()
                .filterResults()
                .combineResults(databaseResults
                    .callOtherWebService());
            System.out.println(result); }
        catch (Exception e) {
            System.out.println(e.getMessage());
}   }   }
```

2.3 Fluency Lost Without Exceptions

Error handling with exceptions is debatable,[2] especially in Java, where runtime exceptions need not be declared,[3] but the alternative of checked excep-

[2] http://www.joelonsoftware.com/items/2003/10/13.html
[3] http://bit.ly/1e5P6Cg

tions can be considered harmful.[4]

Worse yet, the semantics of composed locks and exceptions are black magic. The fundamental reason is that an exception thrown from inside a lock leaves the program in an indeterminate state for other threads, with the lock summarily abandoned. There are expert techniques for mitigating this,[5] but a defensible way out is just to eschew exceptions.

But, rather than join the debate, just imagine that we have decided against exceptions for whatever reason and try to write reasonable code.

Add a private *String* field, *errorResult*, and let every method set the error result if and only if it errors. We must change *combineResults*; it can no longer return just a *String*, but rather a *Computation*, because it may, itself, produce an error. Furthermore, we lose the fluent style because every call must be individually checked.

A particularly nasty way to do this is as follows:

```
public static String computation () {
    Computation c1 = new Computation();
    Computation c2 = c1.authorize();
    if (c2.errorResult.isEmpty()) {
        Computation c3 = c2.readDatabase();
        if (c3.errorResult.isEmpty()) {
            Computation c4 = c3.callWebService();
            if (c4.errorResult.isEmpty()) {
                Computation c5 = c4.filterResults();
                if (c5.errorResult.isEmpty()) {
                    Computation c6 = c3.callOtherWebService();
                    if (c6.errorResult.isEmpty()) {
                        Computation c7 = c5.combineResults(c6);
                        if (c7.errorResult.isEmpty()) {
                            return c7.getResult(); }
                        else {return c7.errorResult;} }
                    else {return c6.errorResult;} }
                else {return c5.errorResult;} }
            else {return c4.errorResult;} }
        else {return c3.errorResult;} }
    else {return c2.errorResult;} }
public static void main(String[] args) {
    System.out.println(computation()); }
```

[4] http://bit.ly/9NyrdD

[5] http://bit.ly/qOO1r

This is so intolerable as to barely deserve criticism, despite the fact that its working set is optimized for the positive path![6] We've lost any correspondence between the program text and the program specification, /i.e.,/ the diagram in figure 1. All options for nesting and placement of curly braces are ludicrous. Changing the computation graph would entail a sickening amount of work. Code like this is best left to automatic code generators, if we tolerate it at all.

The prevailing style, nowadays, is to reverse error branches and to return as early as possible from the main routine. I have seen and reviewed many instances of this style in shipped code from pre-eminent shops. Despite the fact that multiple returns were condemned in the dogma of structured programming and are lethal in code that manages resources,[7] the justification for this is three-fold:

- it results in linear code that can be read from top to bottom
- edits to the computation graph entail just adding or subtracting a localized block of a few lines of code and adjusting a few temporary variables
- modern compilers can reverse the branches *again* in the generated code automatically after profiling[8]

This alternative[9] is the following:

```
public static String computation() {
    Computation c1 = new Computation();
    Computation c2 = c1.authorize();
    if (! c2.errorResult.isEmpty()) {return c2.errorResult;}
    Computation c3 = c2.readDatabase();
    if (! c3.errorResult.isEmpty()) {return c3.errorResult;}
    Computation c4 = c3.callWebService();
    if (! c4.errorResult.isEmpty()) {return c4.errorResult;}
    Computation c5 = c4.filterResults();
    if (! c5.errorResult.isEmpty()) {return c5.errorResult;}
    Computation c6 = c3.callOtherWebService();
    if (! c6.errorResult.isEmpty()) {return c6.errorResult;}
```

[6] The error branches are all at addresses far from the common-case, non-error branches, which are clustered together for maximum locality.

[7] `http://bit.ly/sAvDmY`

[8] `http://bit.ly/QkXSM`

[9] favored in the previously cited Joel-on-Software blog

```
    Computation c7 = c5.combineResults(c6);
    if (! c7.errorResult.isEmpty()) {return c7.errorResult;}
    return c7.getResult(); }
public static void main(String[] args) {
    System.out.println(computation()); }
```

This, at least, gets rid of the ludicrous nesting, but exposes another deep weakness: we have a proliferation of temporary variables just to hold the intermediate *Computations*. Why bother with this when we have no hope of fluent style? Let's go to

```
public static String computation() {
    Computation c1 = new Computation();
    c1.authorize();
    if (! c1.errorResult.isEmpty()) {return c1.errorResult;}
    c1.readDatabase();
    if (! c1.errorResult.isEmpty()) {return c1.errorResult;}
    c1.callWebService();
    if (! c1.errorResult.isEmpty()) {return c1.errorResult;}
    c1.filterResults();
    if (! c1.errorResult.isEmpty()) {return c1.errorResult;}
    c1.callOtherWebService();
    if (! c1.errorResult.isEmpty()) {return c1.errorResult;}
    c1.combineResults(c1);
    if (! c1.errorResult.isEmpty()) {return c1.errorResult;}
    return c1.getResult(); }
public static void main(String[] args) {
    System.out.println(computation()); }
```

Edits to the graph now entail even easier edits to the source. The whole program at this point is the following:

```
import java.util.Random;
public class Computation {
    private String errorResult;
    private String result;
    private String authToken;
    private String databaseResults;
    private String webServiceCallResults;
    private String filteredWebServiceCallResults;
```

```
private String otherWebServiceCallResults;
private static Random random = new java.util.Random();
private static Boolean randomlyError() {
    return random.nextDouble() < 0.10; }

public Computation () {errorResult=""; result="no result";}
public Computation authorize() {
    if (randomlyError()) { errorResult = "auth errored"; }
    authToken = "John's credentials";
    return this; }
public Computation readDatabase() {
    if (randomlyError()) { errorResult = "database errored"; }
    databaseResults = "{\"name\":\"John\", \"PO\":\"421357\"}";
    return this; }
public Computation callWebService() {
    if (randomlyError()) { errorResult = "ws1 errored"; }
    webServiceCallResults =
        "[{\"item\":\"camera\"}, {\"item\":\"shoes\"}]";
    return this; }
public Computation filterResults() {
    if (randomlyError()) { errorResult = "filter errored"; }
    filteredWebServiceCallResults =
        "[{\"item\":\"camera\"}]";
    return this; }
public Computation callOtherWebService() {
    if (randomlyError()) { errorResult = "ws2 errored"; }
    otherWebServiceCallResults = "{\"price\":\"420.00\"}";
    return this; }
public Computation combineResults(Computation other) {
    if (randomlyError()) { errorResult = "combine errored"; }
    result = "{[" + filteredWebServiceCallResults +
        "," + otherWebServiceCallResults + "]}";
    return this;}
public String getResult() {return result;}
public static String computation() {
    Computation c1 = new Computation();
    c1.authorize();
    if (! c1.errorResult.isEmpty()) {return c1.errorResult;}
    c1.readDatabase();
    if (! c1.errorResult.isEmpty()) {return c1.errorResult;}
```

```
        c1.callWebService();
        if (! c1.errorResult.isEmpty()) {return c1.errorResult;}
        c1.filterResults();
        if (! c1.errorResult.isEmpty()) {return c1.errorResult;}
        c1.callOtherWebService();
        if (! c1.errorResult.isEmpty()) {return c1.errorResult;}
        c1.combineResults(c1);
        if (! c1.errorResult.isEmpty()) {return c1.errorResult;}
        return c1.getResult(); }
    public static void main(String[] args) {
        System.out.println(computation());
}   }
```

3 Let's Do Better

Looking back, the main benefits of fluent style are

- direct correspondence between the program specification and the program text – the text *looks like* the diagram
- edits to the specification and edits to the code are parallel
- minimal number of temporary variables

But we only have fluent style if we use exceptions. Without exceptions, we're essentially writing assembly language: storing and combining intermediate results in temporary variables and checking for errors after every step. It's possible to do much better, with and without exceptions, by going *functional*.

In Java, our fundamental modeling tool is the *mutable*, *stateful object*. Stateful object programming has many disadvantages:

- modeling dataflow is awkward
- concurrency entails locks, which are complex
- concurrency with exceptions is very difficult
- composing stateful objects, even without concurrency, is difficult: the operational semantics of even a sequential program requires temporal reasoning, well outside the capabilities of compilers and programming tools

It's worthwhile to emphasize a point we left unstated. Why did we use a new instance variable in the object for each intermediate state? Why not use just a single variable for every non-error result? After all, we used a single variable for the error result?

The reason is that we wanted the individual methods that update non-error state to be as independent as possible. Though our mocks don't do so, in a real program, each intermediate computation would use the result of its predecessors: *readDatabase* would use the *authToken*, the web-service calls would use *databaseResults* and so on. By using a separate, named variable for each intermediate result, the correctness of our individual sub-computations would be easier to verify by inspection. If we had re-used a single *String* variable, the temporal flow forced by the dependencies would be even more obscured, and our program would be even more difficult to understand and maintain. It's definitely worth a few more named variables to make our program easier for the next programmer tasked with reading our code. Because the only tool we have is mutable state, it's hard to do better than a sequence of mutable state variables mirroring our sequence of sub-computations.

The essence of the problem is that we are modeling a *flow* of data through *transforms* as a flow of data through mutable variables. If, instead, we invert the paradigm to make the *transforms* the focus, we sidestep this problem. Doing so requires a language with first-class transforms, that is, *functions*. Mutable state variables become immutable function parameters. Thread-safety becomes automatic and locks do not arise.

Fluency is immediate with exceptions, as before. We can achieve fluency for errors-as-return-values with a *monad*. Despite the name, monads are not exotic or complicated. They furnish a straightforward generalization of function composition that is fundamental for manageable concurrent and distributed programming, even with onlymutable, stateful objects.

C# has first-class functions, a.k.a. *lambda expressions*, as does C++ 11 and as will Java 8. In the mean time, we can use *Clojure*, a Java-compatible functional language.

3.1 Fluent Functional Solution With Exceptions

We may write the program with only one intermediate variable, as before. This variable holds results of the database read, which we must use for each of the two web-service calls. Even this variable can be eliminated *via memoization*, *common sub-expression elimination*, *lambda lifting*, *the state monad*, or *parallel composition*, but that is for another time and place. For

now, write the flow directly as a sequential composition of function calls *via* Clojure's `->`, using its `let` syntax for the one intermediate variable, as follows.

```
(try
  (let [db-results ; be the result of the composition
        (-> (computation)
            authorize
            read-database
            )]
    (-> db-results
        call-web-service
        filter-ws
        (combine (-> db-results
                     call-other-web-service))))
  (catch Exception e (.getMessage e)))
```

This looks very much like the fluent Java solution. In Java, we have fluent streams of items connected by dots. In Clojure, we have the same fluent streams of items headed by arrows.

The rest of the program is as follows. First, declare a namespace for our symbols to inhabit:

```
(ns temp-1.core)
```

Define a private symbol *computation* to be a function of no arguments that produces an empty hash-map.

```
(defn- computation [] {})
```

Define a private symbol *randomly-error* to be a function of no arguments that produces *true* if a random double-precision number is less than 0.10.

```
(defn- randomly-error [] (< (rand) 0.10))
```

Define several private symbols to be mocks of functions that produce our application-specific data, or, with 10% probability, throw an exception. These all use the three-term *if*-branch form, which is just the same as Java's

$\langle e_1 \rangle ? \langle e_2 \rangle : \langle e_3 \rangle$ operator, taking an expression of Boolean type and two arbitrary expressions only one of which is evaluated.

```
(defn- authorize [computation]
  (if (randomly-error) (throw (Exception. "auth errored"))
                       {:auth-token "John's credentials"}))
(defn- read-database [auth-token]
  (if (randomly-error) (throw (Exception. "database errored"))
                       {:name "John", :PO 421357}))
(defn- call-web-service [database-results]
  (if (randomly-error) (throw (Exception. "ws1 errored"))
                       [{:item "camera"}, {:item "shoes"}]))
(defn- filter-ws [web-service-call-results]
  (if (randomly-error) (throw (Exception. "filter errored"))
                       [{:item "camera"}]))
(defn- call-other-web-service [database-results]
  (if (randomly-error) (throw (Exception. "ws2 errored"))
                       [{:price 420.00M}]))
```

The last application function is a bit more complicated: it takes two arguments and combines them.

```
(defn- combine [filtered-web-service-results
                other-web-service-call-results]
  (if (randomly-error) (throw (Exception. "combine errored"))
      (concat filtered-web-service-results
              other-web-service-call-results)))
```

Several improvements are notable in this first attempt:

- first, as stated, with only one exception, state variables have become immutable function parameters, purely local to each transform
- the one remaining intermediate variable is itself immutable, removing any need for temporal reasoning – we only need to understand dependencies, and they are explicit in the code
- the code is shorter, less repetitive, less noisy

- the values of each mock can be modeled directly as hash-maps, arrays, integers, and decimal numbers like $420.00M$, as opposed to JSON objects encoded in strings
  - such direct modeling removes the implied need, unstated in our Java solution, for JSON serialization and parsing
  - such direct modeling also means that we do not need direct Java interop; our computation "constructor" just returns an empty hash-map
  - if we did need needed to interface with an exising Java class, we would only need to *import* the class and change our constructor call from `(computation)` to `(Computation.)`, shorthand for `(new Computation)`

We continue to use Java's native *Exception* class, and this illustrates Java interop again.

3.2 Fluent Error Handling Without Exceptions

The improvements above alone justify the Clojure solution over the Java solution. But the case is really obvious when we get down to error handling without exceptions. In Clojure, we use a variation of **the Maybe monad**.[10]

Here is the main code

```
(let [db-results
      (>-> (computation)
           authorize
           read-database)]
  (>-> db-results
       call-web-service
       filter-ws
       (combine (>-> db-results
                     call-other-web-service))))
```

It looks *just like* the non-monadic code, just with a different arrow. The prerequisite helpers are as follows:

```
(defn- computation [] (with-em-result {}))
```

[10] `http://bit.ly/WV02FF`

```
(defn- authorize [computation]
  (with-em-result
    (if (randomly-error) {:error "auth errored"}
        {:auth-token "John's credentials"})))
(defn- read-database [auth-token]
  (with-em-result
    (if (randomly-error) {:error "database errored"}
        {:name "John", :PO 421357})))
(defn- call-web-service [database-results]
  (with-em-result
    (if (randomly-error) {:error "ws1 errored"}
        [{:item "camera"}, {:item "shoes"}])))
(defn- filter-ws [web-service-call-results]
  (with-em-result
    (if (randomly-error) {:error "filter errored"}
        [{:item "camera"}])))
(defn- call-other-web-service [database-results]
  (with-em-result
    (if (randomly-error) {:error "ws2 errored"}
        [{:price 420.00M}])))
```

All but *combine* are straightforward, simply wrapping their results in a *with-em-result*. In fact, this wrapping is one of only two things we must learn about monads: it puts values in boxes.

```
(defn- combine [other-ws-results-val]
  (fn [filtered-ws-results-val]
    (with-em-result
      (if (randomly-error)
        {:error "combine errored"}
        (concat filtered-ws-results-val
                other-ws-results-val)))))
```

All but *combine* are straightforward, simply wrapping their results in a *with-em-result*. In fact, this wrapping is one of only two things we must learn about monads: it puts values in boxes.

3.2.1 Monads Are Just Values In Boxes

An instance of a monad is just a value in a box. Every monad has an operator, *m-result*, which takes a value and produces a value in a box. All

monads work this way: take a value and put it in a box. The box can be arbitrarily complicated inside, and each type of monad has its own type of box. But all monads have this *m-result* operator in common: you take values and put them in boxes.

Our *with-em-result* macro above is just shorthand *m-result* for our error-propagating monad.

```
(defmacro with-em-result [expr]
  `(with-monad if-not-error-m (m-result ~expr)))
```

Monads have one more essential operator, *m-bind*. This takes two arguments: a value-in-a-box and a function-from-value-to-value-in-a-box. The signature of this function is just like the signature of *m-result*: it takes a value and returns a value-in-a-box. Here we see again the fundamental simplifying idea: we're always taking values and putting them in boxes.

The implementations of *m-result* and *m-bind* are particular to each monad. You package your application-dependent logic in your functions-from-value-to-value-in-a-boxes.

Why do we have a variation of the standard *Maybe* monad?. First, note 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. 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.

```
(defmonad if-not-error-m
  [m-result (fn [value] value)
   m-bind   (fn [value f]
              (if-not (:error value)
                (f value)
                value))
  ])
```

3.3 Getting Rid of the Last Variable

4 Code

```
(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/algo.monads "0.1.4"]
                 [org.clojure/data.zip    "0.1.1"]
                 [dk.ative/docjure        "1.6.0"]
                ]
  :repl-options {:init-ns ex1.core})


(ns ex1.core
  (:use clojure.algo.monads))


(defmonad if-not-error-m
  [m-result (fn [value] value)
   m-bind   (fn [value f]
              (if-not (:error value)
                (f value)
                value))
   m-zero   {:error "unspecified error"}
   m-plus   (fn [& mvs]
              (first (drop-while :error mvs)))

   ])


(ns ex1.core-test
  (:require [clojure.test        :refer :all]
            [ex1.core            :refer :all]
            [clojure.algo.monads :refer :all]))
```

```
(deftest exception-throwing-test
  (testing "exceptions are thrown"
    (is (thrown? ArithmeticException (/ 1 0)))
    (is (thrown-with-msg? ArithmeticException #"Divide by zero" (/ 1 0)))
    ))

(deftest comprehension-test
  (testing "sequence monad and comprehension"
    (is (= (domonad sequence-m
                    [a (range 5)
                     b (range a)]
                    (* a b))
           (for [a (range 5)
                 b (range a)]
             (* a b)))
        "Monadic sequence equals for comprehension")))

(defn- divisible? [n k]
  (= 0 (rem n k)))

(def ^:private not-divisible?
  (complement divisible?))

(defn- divide-out [n k]
  (if (divisible? n k)
    (recur (quot n k) k)
    n))

(defn- error-returning-check-divisibility-by [k n]
  (let [q (divide-out n k)]
    (if (= q n)
      {:error (str n ": not divisible by " k)}
      q)))

(defn- exception-throwing-check-divisibility-by [k n]
  (let [q (divide-out n k)]
    (if (= q n)
      (throw (Exception.
              (str {:error (str n ": not divisible by " k)})))
      q)))

(defn- best-small-divisor-sample [a2]
  (try
    (->> a2
        (exception-throwing-check-divisibility-by 2)
        (exception-throwing-check-divisibility-by 3)
        (exception-throwing-check-divisibility-by 5)
        (exception-throwing-check-divisibility-by 7))
    (catch Exception e (.getMessage e)))
```

5 References

6 Conclusion