The Pragmatic Bookshelf

PragPub

The First Iteration

IN THIS ISSUE

Clojure Is Hot!

Welcome to our Clojure issue.

Issue #25
July 2011

Contents

FEATURES

DEPARTMENTS

Published monthly in PDF, mobi, and epub formats by The Pragmatic Programmers, LLC, Dallas, TX, and Raleigh, NC. E-Mail support@pragprog.com, phone +1-800-699-7764. The editor is Michael Swaine (michael@pragprog.com). Visit us at http://pragprog.com for the lowdown on our books, screencasts, training, forums, and more.

ISSN: 1948-3562

Up Front

Clojure Is Hot

by Michael Swaine

In this Clojure issue, four authors take on the Clojure language from four perspectives. And for balance, we also have non-Clojure articles by Jeff Langr, Tim Ottinger, Dan Wohlbruck, and John Shade.

Welcome to our first-ever Clojure issue.

Like most good inventions, this JVM-friendly Lisp dialect draws on the work of many but is the brainchild of one person—in this case, Rich Hickey. So although we've covered it before (in November 2010 [U1] and July 2009 [U2]), we decided that it was time to devote a whole issue to Clojure. And we came to this conclusion on thermal grounds: Clojure is hot.

According to Chas Emerick, commenting on his 2010 State of Clojure Survey [U3], "It seems clear that the Clojure community is growing, and growing fast." Also, "very few people have come directly from Common Lisp and Scheme," suggesting that the growth of Clojure is not just movement within the Lisp community. The metablog Planet Clojure now lists more than 250 blogs. Is Clojure the new Ruby? The last programming language? The future? It's been called all these things.

Here are some resources so you can test the temperature and educate yourself: clojure.org [U4], Clojure/core [U5], Rich Hickey's Clojure Reading List [U6], Planet Clojure metablog [U7], Clojure.conj [U8], disclojure: public disclosure of all things clojure [U9], clojure google group [U10], and Clojure links on Twitter [U11]

A Lively Community

One bit of evidence of vitality in a language is lively action in user groups. And Clojure has a lot of user group activity. So when I thought of putting together a Clojure issue, it seemed the logical thing to tap the user groups for articles. Not just because the lively and technically sophisticated discussions in user group forums led me to think they would be a good source of articles, which proved to be true, but also because I thought an open call to user group members might be a good way to get a feeling for the kind of discussion going on in the Clojure community.

But I wanted to hedge my bet by drawing on tested talent, so I solicited an article from Aaron Bedra of Relevance, who is working with Stuart Halloway on the second edition of Programming Clojure [U12].

So the calls went out, the articles came in, and here's the line-up:

Jeff Héon from the Montreal Clojure User Group leads off with an introduction to Clojure that highlights its capabilities for data manipulation. It's a gentle intro that should let the reader new to Clojure get to know enough about the language to decide if it is worth pursuing further.

Then we swing to the other extreme. Steven Reynolds of the Clojure Houston User Group follows up with a deep dive into the internal representation of

some Clojure collections. He illustrates the backing data for objects like a physician using an MRI to see the internals of their patient.

Aaron Bedra finds the pragmatic way between these extremes, walking you through the development of some Unix services in Clojure, with the knowledge and clarity that he's putting into the next edition of Programming Clojure [U13].

Then Ambrose Bonnaire-Sergeant of the Seattle Clojure User Group walks you through the creation of a small Clojure DSL, starting with common building blocks like conditionals and motivating more advanced mechanisms that Clojure uniquely provides, like Multimethods and ad-hoc hierarchies.

But Wait, There's More...

Because there is more to life, even the coding life, than Lisp dialects, we've included a few other goodies. Jeff Langr and Tim Ottinger follow up last issue's article on pair programming with a detailed list of benefits of pairing—benefits to the individual programmers, to the team, and to the management or the project. And Dan Wohlbruck takes us back in time to the birth of the Unix operating system, which celebrates its 37th birthday this month.

John Shade weighs in on a different birthday celebration, with some pointed comments on the industry's most celebrated centenarian. Of course there's the latest Events Calendar, telling you about where our authors will be appearing and other notable events, and Choice Bits, where you'll learn that it's good to be Branson.

And last but not least—no, actually it is least—we've added a page at the end of the issue in which we hint at things to come. It's called "But Wait, There's More..." In an open-ended way, it give the issue a sense of, uh, closure.

External resources referenced in this article:

[U1] http://pragprog.com/magazines/download/17.HTML

[U2] http://pragprog.com/magazines/download/1.HTML

[U3] http://cemerick.com/2010/06/07/results-from-the-state-of-clojure-summer-2010-survey/

[U4] http://clojure.org/

[U5] http://clojure.com/

[U6] http://www.amazon.com/Clojure-Bookshelf/lm/R3LG3ZBZS4GCTH

[U7] http://planet.clojure.in/

[U8] http://clojure-conj.org/

[U9] http://disclojure.org/

[U10] http://groups.google.com/group/clojure

[U11] https://twitter.com/clojurelinks

[U12] http://pragprog.com/shcloj2

[U13] http://www.pragprog.com/refer/pragpub25/titles/shcloj2/

Choice Bits

It's Good to Be Branson

The Virgin Tweeter, lessons learned, and other projectiles lobbed from the tweeting trenches.

What's Hot

Top-Ten lists are passé—ours goes to 11. These are the top titles that folks are interested in currently, along with their rank from last month. This is based solely on direct sales from our online store.

1	NEW	Designed for Use
2	2	Agile Web Development with Rails
3	1	CoffeeScript
4	9	iOS Recipes
5	5	The RSpec Book
6	11	Exceptional Ruby
7	4	Programming Ruby 1.9
8	NEW	Hello, Android
9	NEW	Seven Languages in Seven Weeks
10	6	HTML5 and CSS3
11	3	Crafting Rails Applications

The Virgin Tweeter

It's not hard to produce interesting tweets. Just tell us what you did today. Like this guy.

- *At Cape Canaveral to see off astronaut Mark Kelly @ShuttleCDRKelly at the last shuttle launch: http://virg.co/shuttle* — @richardbranson
- *Joan and I were honoured to have dinner with the Queen and @BarackObama at a state banquet last night: http://virg.co/palace* — @richardbranson
- *On top of the sub at the @virginoceanic launch. http://twitpic.com/4hceg0* — @richardbranson
- *I am guest editing @BigIssueSA—special edition on #entrepreneurship with @virginunite. It's out now. http://virg.co/bigissue* — @richardbranson
- *Spent yesterday walking through beautiful Marrakech—it's a wonderful time to visit Morocco. http://virg.co/morocco* — @richardbranson
- *Swimming the Irish Sea to help raise £1m for Cancer Research UK. Here's how you can join me: http://virg.co/swim @JoinTheSwim #jointheswim* — @richardbranson
- *Tempted to give the climbing wall a go—best not until the knee is better.* — @richardbranson
- *I have bought Pluto & intend to reinstate it as a planet. This could herald a new age in space tourism. http://ow.ly/4qX0s #virginpluto* — @richardbranson

Lessons Learned

If you are receptive, deep insights will come to you every day. Be sure to share them on Twitter.

- *Has anybody noticed that as languages grow in complexity, the best programmers switch to simpler languages?* — @unclebobmartin
- *OK, this is SO WEIRD: Don't wear polarized sunglasses when using the iPad. When it's in portrait orientation, the screen goes black!!* — @Pogue
- *The lesson here is probably not to buy a lawn mower at WalMart unless you want all the oil to leak out onto your garage floor overnight.* — @tottinge
- *This morning, I am reminded of the research paper "Goals Gone Wild": http://bit.ly/eE7CEn* — @estherderby
- *Read about Kepler's laws & Schrodinger's cat for bedtime stories past few days to kids. Started with resistance, but now they're intrigued.* — @venkat_s
- *Children are asleep and wife is on vacation in Istanbul—perfect opportunity to continue with compiler project: tweaking lexical analyzer.* — @staffannoteberg
- *In my day we managed our own memory and we LIKED IT.* — @atomicbird
- *Commitment—(noun) : Opening a carton of Häagen-Dazs and throwing away the lid.* — @bketelsen
- *"We think that dating is a problem to be solved using data and analytics," says OKCupid's Sam Yagan. http://oreil.ly/ity1Nu /gg* — @OReillyMedia
- *Product Blog: How to use Basecamp to plan a wedding. http://bit.ly/kFd2Jz* — @37signals
- *Won't go to my old bar, the Glenview House again. Someone bought it. It is no longer a dive. No foosball; No pool; Couples on dates!?!?* — @jwgrenning
- *WWDC keynote: Royal Wedding for nerds.* — @petermaurer
- *It's not the space I like; it's the lack of people in it.* — @invalidname
- *I want stickers that say "As Seen On Failblog", and place them on unsuspecting passers-by.* — @PragmaticAndy
- *You know, just because your BnB directory or grilled cheese sandwich shop is tied to the web doesn't make you a tech startup.* — @dannysullivan
- *Reading about how more women in a group make for a smarter group hbr.org/2011/06/defend…* — @JeniT
- *Whenever I feel bad about cheesy stuff I've done, I remind myself that Justin Timberlake was in N'Sync.* — @ginatrapani
- *Just has a sneezing fit. Counted 32 sneezes. It's still going on. 35 now. Lawks.* — @stephenfry
- *We've done more between breakfast and lunch than some folks do between brunch and afternoon snack.* — @scottdavis99
- *Life is too short to remove USB safely.* — @bigoudii

This month we followed Abigail A, Chris Adamson, Richard Branson, Esther Derby, Scott Davis, Stephen Fry, James Grenning, Tom Harrington, Andy Hunt, Chris Johnson, Brian Ketelsen, Uncle Bob Martin, Peter Maurer, Staffan Nöteberg, O'Reilly Media, Tim Ottinger, David Pogue, Venkat Subramaniam, Danny Sullivan, Jeni Tennison, 37signals, and Gina Trapani. You can follow us at www.twitter.com/pragpub [U1].

External resources referenced in this article:

[U1] http://www.twitter.com/pragpub

Clojure Building Blocks

An Introduction to Clojure and Its Capabilities for Data Manipulation

by Jean-François "Jeff" Héon

Jeff introduces Clojure fundamentals and uses them to show why you might want to explore this language further.

I mainly use Java at work in an enterprise setting, but I've been using Clojure at work for small tasks like extracting data from log files or generating or transforming Java code. What I do could be done with more traditional tools like Perl, but I like the readability of Clojure combined with its Java interoperability. I particularly like the different ways functions can be used in Clojure to manipulate data.

I will only be skimming the surface of Clojure in this short article and so will present a simplified view of the concepts. My goal is for the reader to get to know enough about Clojure to decide if it is worth pursuing further using longer and more complete introduction material already available.

I will start with a mini introduction to Clojure, followed by an overview of sequences and functions combination, and finish off with a real-world example.

Ultra Crash Course

Clojure, being a Lisp dialect, has program units inside lists. A function call will be the first element of a list, optionally followed by parameters.

For setup instructions, look here [U1]. Clojure programs can be run as a script from the command line, as a file from your IDE, or precompiled and packaged to be run as a normal Java jar. They can also be simply loaded or typed in the REPL, the interactive development shell. The REPL might be invoked from your IDE or simply called from the command line, provided you have java 1.5 or higher installed:

```
java -cp clojure.jar clojure.main
```

I invite you to follow along with a REPL on a first or second read and try the examples and variations. You can display the documentation of a function with the doc function.

Entering the following at the REPL:

```
(doc +) ;In Clojure, + is a function and not an operator
```

will echo the documentation. For the article, I precede REPL output with the > symbol.

```
>-------------------------
clojure.core/+
([] [x] [x y] [x y & more])
  Returns the sum of nums. (+) returns 0.
```

For the curious, you can also display the source of a function with source.

```
(source +) ;Try it yourself.
```

First, let's start with the mandatory addition example.

```
(+ 2 4 6)
> 12
```

Values can be associated to a symbol with def.

```
(def a 3)
>#'user/a
```

The REPL will write the symbol name preceded by the namespace, #'user/a, in this case.

Typing "a" will return back its value.

```
a
>3
```

The symbol is bound to the result of the expression after its name.

```
(def b (* 2 a))
>#'user/b
b
>6
```

The str function will concatenate the string representation of its arguments.

```
(str "I have " b " dogs")
>"I have 6 dogs"
```

We can also string together characters. You'll notice that character literals in Clojure are preceded by a backslash.

```
(str \H \e \l \l \o)
>"Hello"
```

It is common to manipulate data as collections, be it lists, vectors, or whatever. The apply function will call the given function with the given collection unpacked.

```
(def numbers [2 4 6]) ;define a vector with 3 integers
>#'user/numbers
```

(I will omit the echoing of the symbol name for the remainder of the article.)

```
(apply + numbers) ;Sames as  (+ 2 4 6)
>12
```

Vectors are accessed like a function by passing the zero-based index as an argument.

```
(numbers 0)
>2
```

Sequences

Clojure has many core functions operating on sequences [U2]. A sequence allows uniform operations across different kinds of collections [U3], be it a list, a vector, a string, etc. In our examples, we will be using mostly vectors, an array-like data structure with constant-time access [U4].

For example, the take function will return the n first elements.

```
(take 3 [1 2 3 4 5]) ;Take 3 from a vector
>(1 2 3)
(take 3 "abcdefg") ;Take 3 from a string
>(\a \b \c)
```

If you were expecting to get back the string "abc", you might be disappointed by the result, as I was the first time I tried. What happened here? Operations producing sequences, like take, do not return elements in the original collection data type, but return a sequence of elements. That is why calling take on a string returns a sequence of characters. This means that take on the vector did not return a vector, but a sequence.

Let's define a test vector to explore more sequence manipulations.

```
(def days-of-the-week ["sunday", "monday", "tuesday",
     "wednesday", "thursday", "friday", "saturday"])
```

Oops! I forgot to capitalize the days. Let's use map, which applies a function to each element of a collection and returns a sequence of the results. For example, the following returns a sequence of our numbers incremented by one.

```
(map inc numbers)
>(3 5 7)
```

First let's develop a function to capitalize a word. Note that there already exists a capitalize function in the clojure.string namespace, but we'll roll our own to demonstrate a few points. We'll develop our function incrementally using the REPL.

We'll start by getting the first letter of a word. The function first will create a sequence over the given collection and return the first element.

```
(first "word")
>\w
```

Let's use a bit of Java interop and call the static function toUpperCase from the Java Character class.

```
(java.lang.Character/toUpperCase (first "word"))
>\W
```

So far so good. Now let's get the rest of our word.

```
(rest "word")
>(\o \r \d)
```

What happens if we want to string our capitalized word together?

```
(str (java.lang.Character/toUpperCase (first "word")) (rest "word"))
> "W(\\o \\r \\d)"
```

We get back the string representation of the first argument, the letter W, concatenated with the string representation of the sequence of the rest of the word.

We need to use a variant of the function apply, which takes an optional number of arguments before a sequence of further arguments.

```
(apply str (java.lang.Character/toUpperCase (first "word"))
     (rest "word")) ;Same as (str \W \o \r \d)
>"Word"
```

Now let's make a function from our trials and tribulations.

```
(defn capitalize [word]
        (apply str (java.lang.Character/toUpperCase
                    (first word)) (rest word)))
```

The first line defined the function named capitalize taking one parameter named word. The second line is simply our original expression using the parameter.

Let's try it out.

```
(capitalize (first days-of-the-week))
> "Sunday"
```

Good. We're ready to capitalize each day of the week now.

```
(def capitalized-days (map capitalize days-of-the-week))
capitalized-days
>("Sunday" "Monday" "Tuesday" "Wednesday"
     "Thursday" "Friday" "Saturday")
```

Map is an example of a high-order function, which has one or more functions in its parameter list. It's a convenient way of customizing a function's behavior via another function instead of using flags or more involved methods like passing a class containing the desired behavior inside a method.

Notice that the original collection is left untouched.

```
days-of-the-week
> ["sunday" "monday" "tuesday" "wednesday"
     "thursday" "friday" "saturday"]
```

Clojure collections are persistent, meaning they are immutable and that they share structure. Let's add a day to have a longer weekend.

```
(conj capitalized-days "Jupiday")
>("Jupiday" "Sunday" "Monday" "Tuesday"
     "Wednesday" "Thursday" "Friday" "Saturday")
```

Adding Jupiday has not modified the original collection capitalized-days, which is guaranteed not to ever change, even by another thread. The longer week was not produced by copying the 7 standard days, but by keeping a reference to the 7 days and another to the extra day. Various collection "modifications", which really return a new data structure, are guaranteed to be as or almost as performant as the mutable version would be [U5].

Filtering operations can be done with the filter high-order function, which return a sequence of elements satisfying the passed-in function.

```
(filter odd? [0 1 3 6 9])
>(1 3 9)
```

When a function passed to an higher function is simple and only used once, there is no need to give it a name. We can define the function in-place. We just use fn instead of defn and forego specifying a name.

For example, here is another way of capitalizing our week days using an anonymous function.

```
(map (fn [word] (apply str (java.lang.Character/toUpperCase
     (first word)) (rest word))) days-of-the-week)
>("Jupiday" "Sunday" "Monday" "Tuesday"
     "Wednesday" "Thursday" "Friday" "Saturday")
```

Another handy sequence operation is reduce. It applies a function between the first two elements of a vector and then applies the function with the result and the 3rd element and so on.

```
(reduce * [1 2 4 8]) ;Same as (* (* (* 1 2) 4) 8)
> 64
```

Another form of reduce takes a parameter as the first value to combine with the first element.

```
(reduce * 10 [1 2 4 8])  ;Same as (* (* (* (* 10 1) 2) 4) 8)
> 640
```

Let's sum the number of characters for each day.

```
(reduce (fn [accumulator element]
     (+ accumulator (count element))) 0 days-of-the-week)
> 50
```

We can redefine the previous anonymous function using syntactic sugar.

```
#(+ %1 (count %2))
```

Note that we can omit the number 1 from the usage of the first argument.

```
#(+ % (count %2))
```

Here is an example to extract the word three in three languages from a vector of vectors.

```
(map #(% 3) [["Zero" "One" "Two" "Three"]
    ["Cero" "Uno" "Dos" "Tres"]["Zéro" "Un" "Deux" "Trois"]])
>("Three" "Tres" "Trois")
```

Composition of Functions

Let's explore function assembly with a wild example: capitalize and stretch.

Let's define our additional function.

```
(defn stretch [word]
        (apply str (interpose " " word)))
```

And test.

```
(stretch "word")
>"w o r d"
```

This would be a standard way of combining stretch and capitalize.

```
(map (fn [word] (stretch (capitalize word))) days-of-the-week)
>("S u n d a y" "M o n d a y" "T u e s d a y" "W e d n e s d a y"
     "T h u r s d a y" "F r i d a y" "S a t u r d a y")
```

Clojure also provides the comp function, which produce a new function from the successive application of the functions given.

```
(map (comp capitalize stretch) days-of-the-week)
>("S u n d a y" "M o n d a y" "T u e s d a y" "W e d n e s d a y"
     "T h u r s d a y" "F r i d a y" "S a t u r d a y")
```

Had we wanted to keep a capitalize-n-stretch function, we could have associated the result of the composition to a symbol.

```
(def capitalize-n-stretch (comp capitalize stretch))
(capitalize-n-stretch "Hello")
>"H e l l o"
```

We can compose more than one function together and we can even throw in anonymous functions into the mix.

```
(map (comp inc (fn [x] (* 2 x)) dec) numbers)
>(3 7 11)
```

We can produce a new function by partially giving arguments.

```
(def times-two (partial * 2))
(times-two 4) ;Same as (* 2 4)
>8
```

We can revisit our compose example differently.

```
(map (comp inc (partial * 2 ) dec) numbers)
>(3 7 11)
```

A Real-World Example

Here is an example of a real function I wrote to collect all the referenced table names for a specific schema. The SQL statements are peppered in various Java files. I call the extract-table-names function for each file, and a corresponding .out file is produced with the referenced table names, uppercased, sorted, and without duplicates. After processing the file, the name of the file and the table count is returned to be displayed by the REPL. The goal is not for you to understand all the program, just to have a feel of it.

```
(ns article
  (:use [clojure.string :only [split-lines join upper-case]]))
  ;Import a few helper functions
;;Extract table names matching MySchema for a given line
(defn extract[line]
  (let [matches (re-seq #"(\s|\")+((?i)(MySchema)\.\w+)" line)]
  ;We're using a regular expression
    (map #(% 2) matches)))
    ;Extract the table name (third item in each match)
(defn extract-table-names [file-path file-name]
  "Extract MySchema.* table names from the java file
   and write sorted on an out file."
  (let [file (slurp (str file-path file-name ".java"))
  ;Get the file
        lines (split-lines file)
        ;Split the file by lines
        names (remove nil? (flatten (map extract lines)))
        ;Extract and remove non-matches
        cleaned-names (-> (map upper-case names) distinct sort)
        ;Uppercased, distinct only and sorted
        ]
        ;Write the file with unique sorted table names
        (spit (str file-path file-name ".out")
          (join "\n" cleaned-names))
    (str file-name ". Table count: " (count cleaned-names))))
;Usage example
(extract-table-names "/DataMining/" "DataCruncher")
```

I've also used Clojure to extract running time statistics of our system and then generate distribution charts with Incanter [U6], a wonderful interactive statistical platform.

This conclude my brief tour of data manipulation with Clojure. There is a lot more to sequences than what I've shown. For example, they are realized as needed, in what is referred to as lazy evaluation [U7]. There is an excellent summary of functions in the sequence section of the Clojure cheatsheet [U8]. Clojure functions can also be combined in other interesting ways like the thread-first or thread-last macros [U9].

About the Author

Jean-François "Jeff" Héon has been fascinated with programming ever since His parents got him a Commodore 64 in High School. He loves nagging his co-workers about new languages and frameworks. Jeff is most happy spending time with His wonderful wife and kid.

Send the author your feedback [U10] or discuss the article in the magazine forum [U11].

External resources referenced in this article:

[U1] http://clojure.org/getting_started

[U2] http://clojure.org/sequences

[U3] http://clojure.org/data_structures#Data%20Structures-Collections

[U4] http://clojure.org/data_structures#Data%20Structures-Vectors%20%28IPersistentVector%29

[U5] http://www.infoq.com/presentations/Value-Identity-State-Rich-Hickey

[U6] http://incanter.org/

[U7] http://en.wikipedia.org/wiki/Lazy_evaluation

[U8] http://clojure.org/cheatsheet

[U9] http://vimeo.com/8474188

[U10] mailto:michael@pragprog.com?subject=building-blocks

[U11] http://forums.pragprog.com/forums/134

Clojure Collections

Looking Deep Inside Clojure Data Collections

by Steven Reynolds

Steven explains the benefits of immutability and explores how Clojure's data collections handle it.

Clojure embraces a functional programming style, controlling mutability tightly. Unless you take special steps to permit it, data collections in Clojure are not mutable; they cannot be changed.

Why bother with immutability? Clojure does so for two key reasons. Having a network of references to mutating objects is fundamentally very complex. Complexity is the enemy of software development. Secondly, such a network is exquisitely difficult to make correct while allowing concurrency.

In the familiar imperative or Object Oriented programming styles, when a structure is updated, it is mutated. The application holds a stable reference to a collection and the collection itself is changed. Clojure instead generally uses collections that cannot be changed; the update operations return a new version of the collection. Whenever a version of a collection is created, that version of the collection must remain accessible (because it cannot be changed). Hence these type of collections are sometimes called *persistent*. Of course, old versions of the collection can be garbage collected when there are no references to them.

The interesting challenge is to ensure that, when a new collection must be returned, Clojure doesn't need to copy the entire collection. Excessive copying causes performance degradation.

Lists

Clojure contain a fairly classical representation for lists. The next code sample creates some lists and exports a graph of each of them.

```
(defn list-ex []
  (let [w '(1 2 3)          ; Figure 1
        x (rest a)          ; Figure 2
        y (conj  a '(3 4)) ; Figure 3
        z (cons 1 a)        ; Figure 4
        lsaver (PersistentListSaver.)
        csaver (ConsSaver.)]
    (. lsaver save w "list_before.dot")
    (. lsaver save x "list_after_rest.dot")
    (. lsaver save y "list_after_conj.dot")
    (. csaver save z "list_after_cons.dot")
    ))
```

The original list, w, is created in the let form and contains the elements 1, 2, and 3. It is shown in Figure 1. The second list, x, is the rest of w. This is all of w except its head element (1 in this case).

PersisentListSaver is a Java class that uses reflection to dump the Clojure list to a graph. This graph is then drawn using GraphViz.

If you look at the list in Figure 1, you can see the representation has a first that contains the head of the list, and a rest that contains other elements.

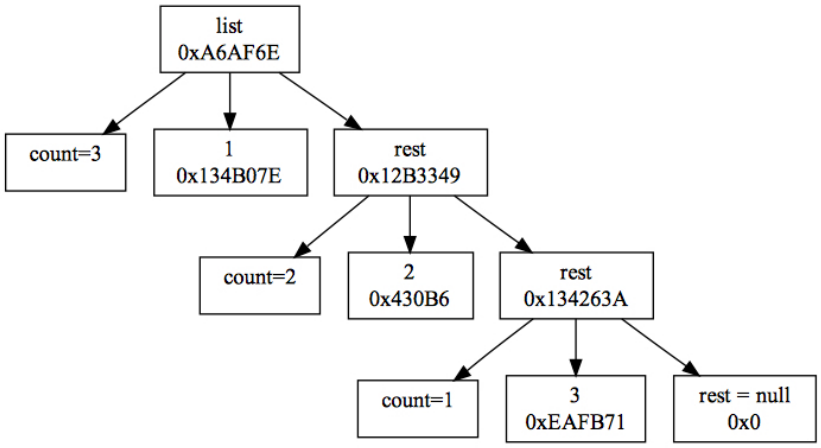


Figure 1: Clojure List

As you can see in Figure 2, when Clojure takes the rest of the original list, it doesn't need to copy any data. The hex numbers in the graph nodes are the result from calling System.identityHashCode on the node. The Java contract of this hash code is that it will always return the same number for the same Object. The usual JDK implementation is to return the address of the Object.

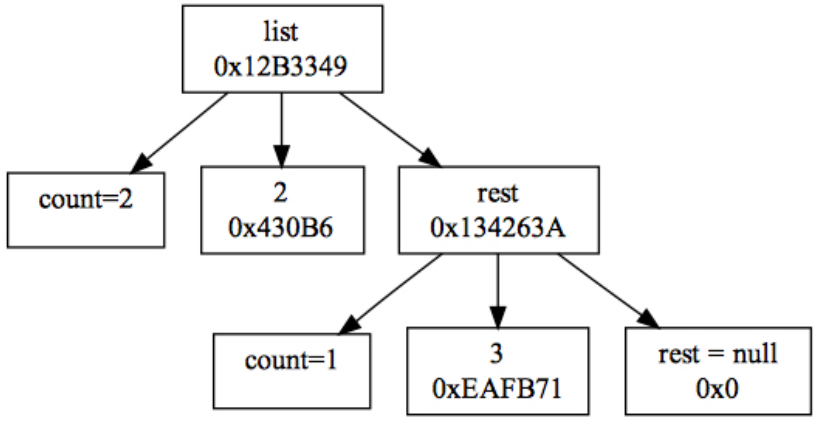


Figure 2: Clojure List from Figure 1 after rest applied

The graphs are simplified: some details left out, and element nodes are shown inline rather than with a separate node. In reality primitives are not stored in Clojure (or Java) collections. They're always storing a boxed Object. That boxing is left out of the graphs for greater clarity and to save paper.

If you add elements to a list, Clojure also arranges to share data. The code sample adds 3 and 4 to the list w using conj and saves this new list in y. The conj function adds elements to the collection at the most efficient location.

If you compare Figure 3 to Figure 1, you can see that conj added the elements at the front. Again, all the preexisting elements are shared.

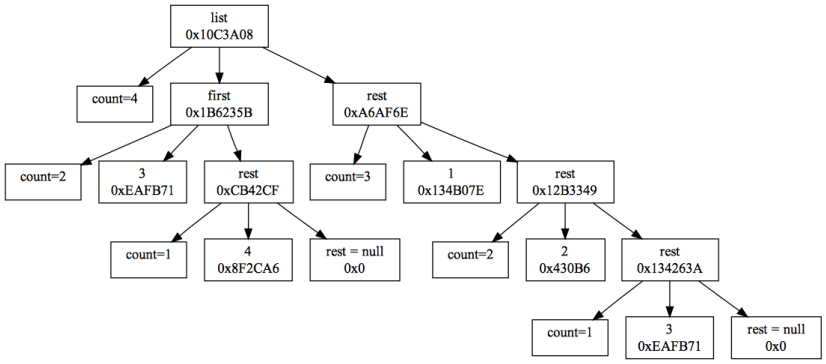


Figure 3: Clojure List from Figure 1 after conj applied

When you add elements to a list using the cons function (list z), Clojure creates a mixed structure with a Cons cell, a first element, and an ISeq, as shown in Figure 4. This shares elements, but future operations will not be quite as efficient. The cons function is specified to always add to the front of any data collection.

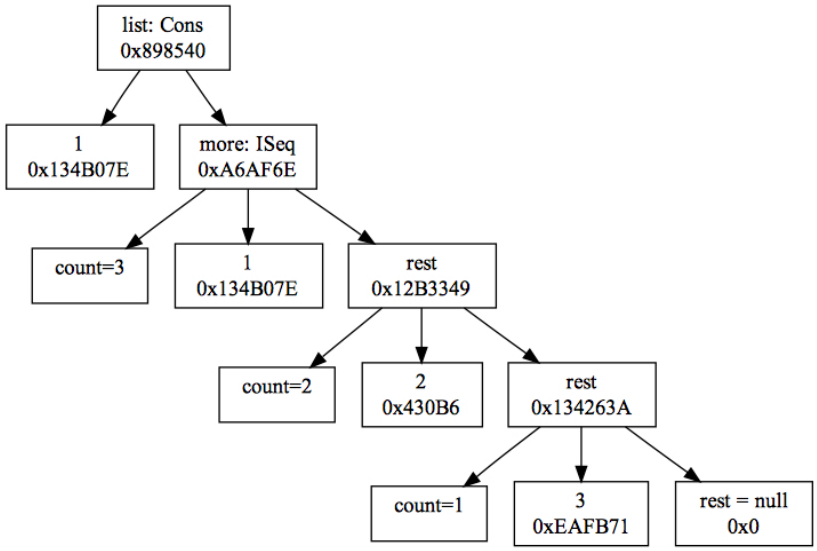


Figure 4: Clojure List from Figure 1 after cons applied

Maps

Another very important type of collection is maps. The next example shows a small map.

```
(defn amap-assoc []
  (let [x '{1 "one" 2 "two"}        ; Figure 5
        y (assoc  x 3 "three")      ; Figure 6
        pamsaver (PersistentArrayMapSaver.)]
    (. pamsaver save x "amap_before_assoc.dot")
    (. pamsaver save y "amap_after_assoc.dot")
    ))
```

In Clojure, small maps are stored in a simple structure backed by an array that contains the keys and values interleaved. You can see this in Figure 5. Lookups are done by a linear scan of the array. That's fast when the map is small. If you add an element to the map (use the function assoc), Clojure just makes a new map without sharing (Figure 6).

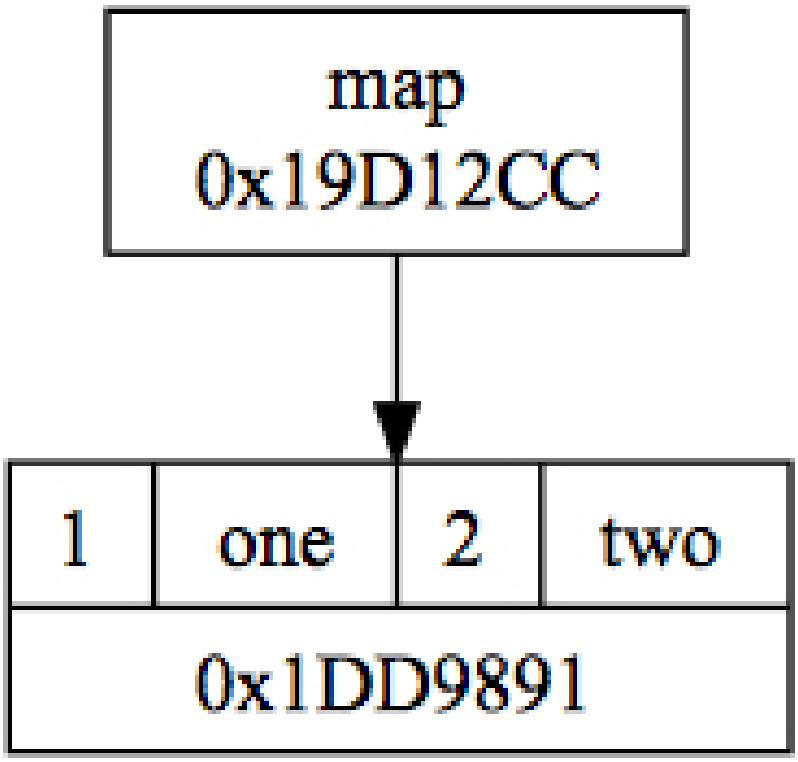


Figure 5: Clojure Array Map

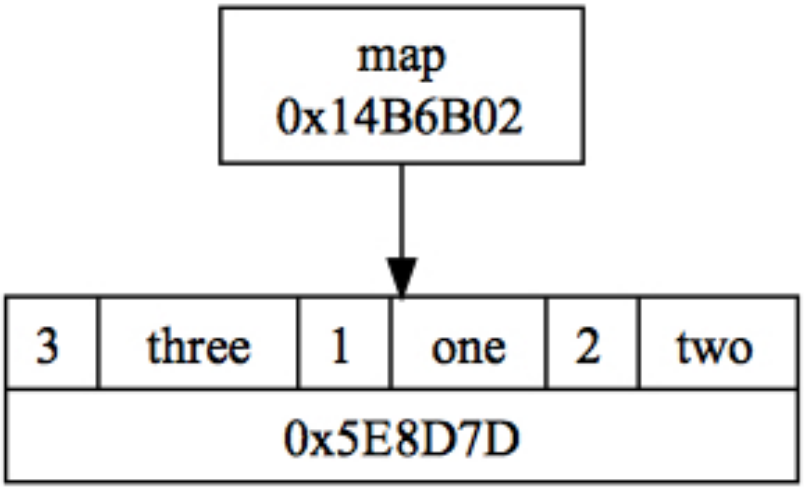


Figure 6: Clojure Array Map from Figure 5 after assoc applied

Longer maps are stored in a PersistentHashMap. They stored as a tree with a 32-way branching factor. PersistentHashMap uses a sparse array to store data at each tree level. The bitmap indicates which elements of the nominal full array are actually present in the realized sparse array.

You can read more about Clojure's PersistentHashMap at the Wikipedia article "Hash array mapped trie [U1]" and at the blog [U2] by arcanesentiment. The Clojure implementation is based on a paper [U3] by Phil Bagwell. This data structure is technically a trie rather than a tree because the keys can be variable length, for example a String.

The next example shows a small hash map.

```
(defn hmap-assoc []
  (let [x (hash-map 1 "one" 2 "two")      ; Figure 7
        y (assoc  x 3 "three")            ; Figure 8
        phmsaver (PersistentHashMapSaver.)]
    (. phmsaver save x "hmap_before_assoc.dot")
    (. phmsaver save y "hmap_after_assoc.dot")
    ))
```

In Figures 7 and 8, you can see the maps before and after an element is added. In this example, only one level of the trie is needed. If you look at Figure 8, you can see that the new map is not able to share any elements with the old map.

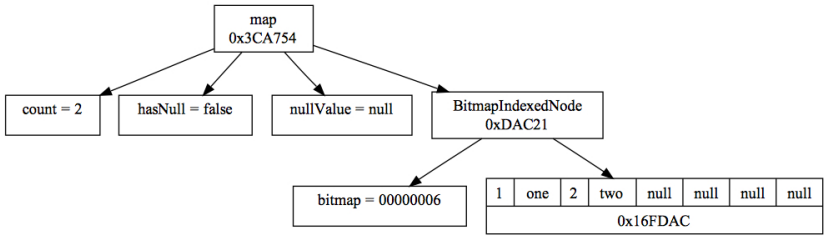


Figure 7: Clojure Hash Map

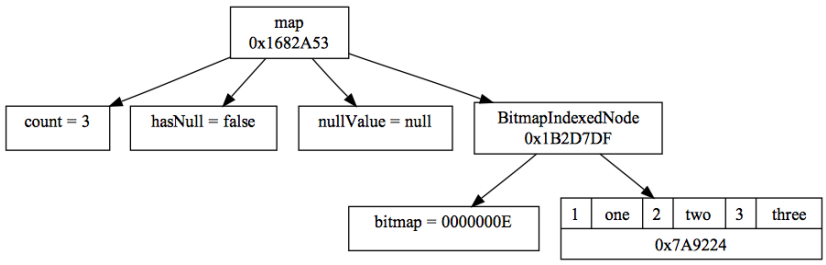


Figure 8: Clojure Hash Map from Figure 7 after assoc applied

Figure 9 shows a more realistic larger hash map that has 17 elements. I removed most of the nodes so that the figure will fit on the page. With a 32-way branching factor, this map instance only needs two levels to store its data; two levels create room for 32*32 elements.

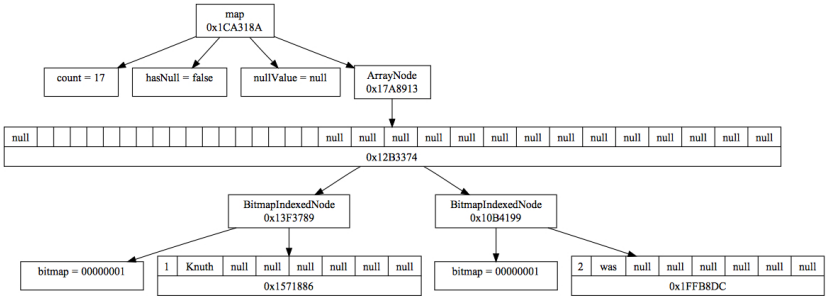


Figure 9: Larger Clojure Hash Map

If you add the key/value {18 "physics"} to this larger map, you get Figure 10. This figure shows that most of the hash map was shared. The new node is of course different, also the nodes above the new node are necessarily different. This technique is well known and is called path copying.

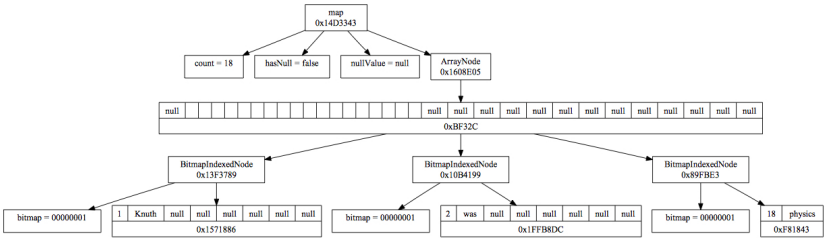


Figure 10: Clojure Hash Map from Figure 9 after assoc applied

Transient Collections

Clojure also has transient versions of the collections. These collections are allowed to change, and so can be faster in some situations. In particular, a transient is used when a collection is built up from another collection. That transient is converted to persistent when it is returned. That approach provides a nice speedup in a way that is invisible to the application code.

Conclusion

I have shown the internal representation of some Clojure collections; I believe that it is very illuminating to see the backing data for objects. It is like a physician using an MRI to see the internals of their patient. The graphs do show internal representations, and these representations will likely change with future versions of Clojure. You should not depend on the details. These graphs were made with Clojure 1.2.

The Clojure data collections are very sophisticated, and play a key role in how Clojure can both avoid mutability and also have excellent performance.

About the Author

Steven is a technical lead and Product Manager at INT [U4] where he works on several Java toolkits. These toolkits build GUI displays of complex geoscience and GIS d ata. Steven has previously worked on database monitoring applications and a peer-to-peer distributed toolkit (InterAgent) that for a time became Sun's JMS implementation. Steven got his degrees in Electrical Engineering from the University of Texas and then from Rice University. Steven has spoken at Houston Techfest and JavaOne several times.

Send the author your feedback [U5] or discuss the article in the magazine forum [U6].

External resources referenced in this article:

[U1] http://en.wikipedia.org/wiki/Hash_array_mapped_trie

[U2] http://arcanesentiment.blogspot.com/2008/08/array-mapped-hash-tries-and-nature-of.html

[U3] http://lampwww.epfl.ch/papers/idealhashtrees.pdf

[U4] http://www.int.com/

[U5] mailto:michael@pragprog.com?subject=collections

[U6] http://forums.pragprog.com/forums/134

Create Unix Services with Clojure

Clojure's Affinity for Java Opens a Lot of Doors

by Aaron Bedra

Aaron is the coauthor (with Stuart Halloway) of the forthcoming *Programming Clojure, Second Edition*. Here he gives a practical, hands-on experience with Clojure.

There are many reasons for turning to Clojure to solve problems. Clojure is a general purpose programing language, based on Lisp, that runs on the Java Virutual Machine. It would take up all of your attention, and a lot of your time to go through them all, so you should browse the videos [U1] for yourself. One of Clojure's selling points is its ability to interoperate with any Java code. Billions of dollars have been invested into building supporting infrastructure and libraries around the Java stack, and it is trivial for you to tap into them using Clojure. This article will dive right in to writing Cloure code with a dash of Java interop.

For this article, we will use the Leiningen [U2] build tool. There are fantastic installation instructions right on the github project page. Leinigen works on Linux, OS X, and Windows, so you should be covered. Before starting this article, make sure you have the current (1.5.2 or greater) version of Leiningen installed.

In this example we will be building an application to test the availability of websites. The goal here is to check to see if the website returns an HTTP 200 OK response. If anything other than our expected response is received, it should be noted. Let's start by creating a new project.

```
lein new pinger
```

Open your project.clj file and modify the contents to match what we are going to be working on. Be sure to update Clojure to the latest version.

```
(defproject pinger "0.0.1-SNAPSHOT"
  :description "A website availability tester"
  :dependencies [[org.clojure/clojure 1.3.0-beta1]])
```

Grab the dependencies by running lein deps.

```
lein deps
```

First we need to write the code that connects to a url and captures the response code. We can accomplish this by using Java's URL Library.

```
(ns pinger.core
  (:import (java.net URL)))

(defn response-code [address]
 (let [connection (.openConnection (URL. address))]
   (doto connection
     (.setRequestMethod "GET")
     (.connect))
   (.getResponseCode connection)))

(response-code "http://google.com")
-> 200
```

Now let's create a function that uses response-code and decides if the specified url is available. We will define available in our context as returning an HTTP 200 response code.

```
(defn available? [address]
  (= 200 (response-code address)))

(available? "http://google.com")
=> true

(available? "http://google.com/badurl")
=> false
```

Next we need a way to start our program and have it check a list of urls that we care about every so often and report their availability. Let's create a main function.

```
(defn -main []
  (let [addresses '("http://google.com"
                    "http://amazon.com"
                    "http://google.com/badurl")]

(while true
  (doseq [address addresses]
    (println (available? address)))
  (Thread/sleep (* 1000 60)))))
```

In this example we create a list of addresses (two good and one bad), and use a simple while loop that never exits to simulate a never-ending program execution. It will continue to check these urls once a minute until the program is terminated. Since we are exporting a -main function, don't forget to add :gen-class to your namespace declaration.

```
(ns pinger.core
  (:import (java.net URL))
  (:gen-class))
```

Now that we have the fundamentals in place we need to tell leningen where our main function is located. Open up project.clj and add the :main declaration:

```
(defproject pinger "0.0.1-SNAPSHOT"
  :description "A website availability tester"
  :dependencies [[org.clojure/clojure "1.3.0-beta1"]]
  :main pinger.core)
```

It's time to compile our program into a jar and run it. To do this, run

```
lein uberjar
java -jar pinger-0.0.1-SNAPSHOT-standalone.jar
```

```
true
false
true
```

You should see your program start and continue to run until you press ctrl-c to stop it.

Adding real continuous loop behavior

A while loop that is always true will continue to run until terminated, but it's not really the cleanest way to obtain the result as it doesn't allow for a clean shutdown. We can use a scheduled thread pool that will start and execute the

desired command in a similar fashion as the while loop, but with a much greater level of control. Create a file in the src directory called scheduler.clj and enter the following code:

```
(ns pinger.scheduler
  (:import (java.util.concurrent ScheduledThreadPoolExecutor TimeUnit)))

(def ^:private num-threads 1)
(def ^:private pool (atom nil))

(defn- thread-pool []
  (swap! pool (fn [p] (or p (ScheduledThreadPoolExecutor. num-threads)))))

(defn periodically
  "Schedules function f to run every 'delay' milliseconds after a
  delay of 'initial-delay'."
  [f initial-delay delay]
  (.scheduleWithFixedDelay (thread-pool)
                           f
                           initial-delay delay TimeUnit/MILLISECONDS))

(defn shutdown
  "Terminates all periodic tasks."
  []
  (swap! pool (fn [p] (when p (.shutdown p)))))
```

This code sets up a function called periodically that will accept a function, initial-delay, and repeated delay. It will execute the function for the first time after the initial delay then continue to execute the function with the delay specified thereafter. This will continue to run until the thread pool is shut down. Since we have a handle to the thread pool, we can do this gracefully via the shutdown function.

Let's update our application to take advantage of the scheduling code as well as make the -main function only responsible for calling a function that starts the loop.

```
(defn check []
  (let [addresses '("http://google.com"
                    "http://google.com/404"
                    "http://amazon.com")]
    (doseq [address addresses]
      (println (available? address)))))

(def immediately 0)
(def every-minute (* 60 1000))

(defn start []
  (scheduler/periodically check immediately every-minute))

(defn stop []
  (scheduler/shutdown))

(defn -main []
  (start))
```

Make sure to update your namespace declaration to include the scheduler code:

```
(ns pinger.core
  (:import (java.net URL))
  (:require [pinger.scheduler :as scheduler])
  (:gen-class))
```

Not everything in the previous sample is necessary, but it makes for more readable code. Adding the start and stop functions makes it easy to work interactively from the REPL which will be a huge advantage should you choose to extend this example. Give everything one last check by running lein uberjar and executing the jar. The program should function exactly as it did before.

Logging

So far we have produced a program capable of periodically checking the availability of a list of websites. It is, however, lacking the ability to keep track of what it has done and to notify us when a site is unavailable. We can solve both of these issues with logging. There are a lot of logging options for Java applications, but for this example we will use log4j. It gives us a real logger to use, and it gives us email notification. This is great because we will have the ability to send email alerts when a website isn't available. In order to do this we will need to pull log4j and mail into our application. To make it easier to take advantage of log4j we will also pull in clojure.tools.logging. Open your project.clj file and add clojure.tools.logging, log4j, and mail:

```
(defproject pinger "0.0.1-SNAPSHOT"
  :description "A website availability tester
  :dependencies [[org.clojure/clojure "1.3.0-beta1"]
                 [org.clojure/tools.logging "0.1.2"]
                 [log4j "1.2.16"]
                 [javax.mail/mail "1.4.1"]]
  :main pinger.core)
```

and pull the dependencies in with leiningen:

```
lein deps
```

The great part about the clojure logging library is that it will use any standard Java logging library that is on the classpath so there is no additional wiring required between log4j and your application. Create a folder in the root of your project called resources. Leiningen automatically adds the contents of this folder to the classpath, and you will need that for your log4j properties file. Create a file under the resources directory named log4j.properties and add the following contents:

```
log4j.rootLogger=info, R, email

log4j.appender.R=org.apache.log4j.RollingFileAppender
log4j.appender.R.File=logs/pinger.log
log4j.appender.R.MaxFileSize=1000KB
log4j.appender.R.MaxBackupIndex=1
log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=%d{ISO8601} %-5p [%c] - %m%n

log4j.appender.email=org.apache.log4j.net.SMTPAppender
log4j.appender.email.SMTPHost=localhost
log4j.appender.email.From=system@yourapp.com
log4j.appender.email.To=recipient@yourapp.com
log4j.appender.email.Subject=[Pinger Notification] - Website Down
log4j.appender.email.threshold=error
log4j.appender.email.layout=org.apache.log4j.PatternLayout
log4j.appender.email.layout.conversionPattern=%d{ISO8601} %-5p [%c] - %m%n
```

This sets up standard logging to pinger.log and will send an email notification for anything logged as error, which in our case is when a website doesn't respond

with an HTTP 200 response or when an exception is thrown while checking the site. Make sure to change the email information to something that works in your environment.

Let's update the code and add logging. The goal here is to replace any println statements with log messages. Open core.clj, add the info and error functions from clojure.tools.logging into your namespace declaration, and create a function to record the results.

```
(ns pinger.core
  (:import (java.net URL))
  (:use [clojure.tools.logging :only (info error)])
  (:require [pinger.scheduler :as scheduler])
  (:gen-class))
...

(defn record-availability [address]
  (if (available? address)
    (info (str address " is responding normally"))
    (error (str address " is not available"))))
```

Also update check to reflect the changes:

```
(defn check []
  (let [addresses '("http://google.com"
                    "http://google.com/404"
                    "http://amazon.com")]
    (doseq [address addresses]
      (record-availability address))))
```

Rebuild and try your program again. You should notice a newly created logs directory that you can check for program execution. You should also notice an email come in with an error message. If you get a "connection refused" error on port 25, you will need to set up a mail transport agent on your machine to enable mail sending. You now have a way to notify people of a website failure!

Configuration

We have hard-coded our list of websites to monitor, and that simply won't work! We need a way to give a list of sites to monitor from some external source. We could use a properties file, database or webservice to accomplish this. For ease of explanation we will go with a properties file. Create a file named pinger.properties in the root of the application and add the following to it:

```
urls=http://google.com,http://amazon.com,http://google.com/404
```

We need a way to load this file in and create a collection of sites to feed into the check function. Create a file named config.clj in the src directory:

```
(ns pinger.config
  (:use [clojure.java.io :only (reader)])
  (:import (java.util Properties)))

(defn load-properties []
  (with-open [rdr (reader "pinger.properties")]
    (doto (Properties.)
      (.load rdr))))

(def config (load-properties))
```

As long as pinger.properties is on the classpath, the previous example will read pinger.properties into a Java properties object. Since we don't want to do this every time we run the website checking routine, we create a var to hold the value for us. All we have left to do is get the url's attribute and put it into a list. Add the following function into the config namespace:

```
(defn urls []
  (str/split (.get config "urls") #","))
```

Make sure to require clojure.string in your namespace declaration

```
(ns pinger.config
  (:use [clojure.java.io :only (reader)])
  (:require [clojure.string :as str])
  (:import (java.util Properties)))
```

Finally, update the check function in core.clj to use the new configuration function.

```
(ns pinger.core
  (:import (java.net URL))
  (:use [clojure.tools.logging :only (info error)])
  (:require [pinger.scheduler :as scheduler]
            [pinger.config :as config])
  (:gen-class))
...

(defn check []
  (doseq [address (config/urls)]
    (record-availability address)))
```

Rebuild your application with leiningen and give it a try. Remember to put pinger.properties on the classpath:

```
java -cp pinger.properties:pinger-0.0.1-standalone.jar pinger.core
```

Wrapping Up

We now have what we need to succeed. In this example we covered:

- Using Java's URL to check a website to see if it was available
- Using Java's ScheduledTheadPoolExecutor to create a periodically running task
- Using log4j with clojure.tools.logging to send error notifications
- Using Java's property system for configuration
- Using leiningen to create standalone executable jars

There are quite a few things you could do to expand on this example. We could redefine what it means for a website to be available by adding requirements for certain HTML elements to be present, or for the response to return in a certain time to cover an SLA. Try adding to this example and see what you can come up with.

About the Author

Aaron Bedra is a developer at Relevance [U3] and a member of Clojure/core [U4], where he uses Clojure to solve hard, important problems. He is the co-author of Programming Clojure, Second Edition [U5].

Send the author your feedback [U6] or discuss the article in the magazine forum [U7].

External resources referenced in this article:

[U1] http://clojure.blip.tv

[U2] http://github.com/technomancy/leiningen

[U3] http://thinkrelevance.com/

[U4] http://clojure.com

[U5] http://www.pragprog.com/refer/pragpub25/titles/shcloj2/

[U6] mailto:michael@pragprog.com?subject=services

[U7] http://forums.pragprog.com/forums/134

Growing a DSL with Clojure

Clojure Makes DSL Writing Straightforward

by Ambrose Bonnaire-Sergeant

From seed to full bloom, Ambrose takes us through the steps to grow a domain-specific language in Clojure.

Lisps like Clojure are well suited to creating rich DSLs that integrate seamlessly into the language.

You may have heard Lisps boasting about code being data and data being code. In this article we will define a DSL that benefits handsomely from this fact.

We will see our DSL evolve from humble beginnings, using successively more of Clojure's powerful and unique means of abstraction.

The Mission

Our goal will be to define a DSL that allows us to generate various scripting languages. The DSL code should look similar to regular Clojure code.

For example, we might use this Clojure form to generate either Bash or Windows Batch script output:

Input (Clojure form):

```
(if (= 1 2)
  (println "a")
  (println "b"))
```

Output (Bash script):

```
if [ 1 -eq 2 ]; then
  echo "a"
else
  echo "b"
fi
```

Output (Windows Batch script):

```
IF 1==2 (
  ECHO a
) ELSE (
  ECHO b
)
```

We might, for example, use this DSL to dynamically generate scripts to perform maintenance tasks on server farms.

Baby Steps: Mapping to Our Domain Language

I like Bash, so let's start with a Bash script generator.

To start, we need to expose some parallels between Clojure's core types and our domain language.

So which Clojure types have simple analogues in Bash script?

Strings and numbers should just simply return their String representation, so we will start with those.

Let's define a function emit-bash-form that takes a Clojure form and returns a string that represents the equivalent Bash script.

```
(defn emit-bash-form [a]
  "Returns a String containing the equivalent Bash script
  to its argument."
  (case (class a)
    java.lang.String a
    java.lang.Integer (str a)
    java.lang.Double (str a)
    nil))
```

The case expression is synonymous here with a C or Java switch statement, except it returns the consequent. Everything in Clojure is an expression, which means it must return something.

```
user=> (emit-bash-form 1)
"1"
user=> (emit-bash-form "a")
"a"
```

Now if we want to add some more dispatches, we just need to add a new clause to our case expression.

Echo and Print

Let's add a feature.

Bash prints to the screen using echo. You've probably seen it if you've spent any time with a Linux shell.

```
ambrose@ambrose-desktop> echo asdf
asdf
```

clojure.core also contains a function println that has similar semantics to Bash's echo.

```
user=> (println "asdf")
asdf
;=> nil
```

Wouldn't it be cool if we could pass (println "a") to emit-bash-form?

```
user=> (emit-bash-form (println "asdf"))
asdf
;=> nil
```

At first, this seems like asking the impossible.

To made an analogy with Java, imagine calling this Java code and expecting the first argument to equal System.out.println("asdf").

```
foo( System.out.println("asdf") );
```

(Let's ignore the fact that System.out.println() returns a void).

Java evaluates the arguments before you can even blink, resulting in a function call to println. How can we stop this evaluation and return the raw code?

Indeed this is an impossible task in Java. Even if this were possible, what could we expect do with the raw code?(!)

System.out.println("asdf") is not a Collection, so we can't iterate over it; it is not a String, so we can't partition it with regular expressions.

Whatever "type" the raw code System.out.println("asdf") has, it's not meant to be known by anyone but compiler writers.

Lisp turns this notion on its head.

Lisp Code Is Data

A problem with raw code forms in Java (assuming it is possible to extract them) is the lack of facilities to interrogate them. How does Clojure get around this limitation?

To get to the actual raw code at all, Clojure provides a mechanism to stop evaluation via the *tick*. Prepending a tick (aka *quote*) to a code form prevents its evaluation and returns the raw Clojure form.

```
user=> '(println "a")
;=> (println "a")
```

So what is the type of our result?

```
user=> (class '(println "a"))
;=> clojure.lang.PersistentList
```

We can now interrogate the raw code as if it were any old Clojure list (because it is!).

```
user=> (first '(println "a"))
;=> println
user=> (second '(println "a"))
;=> "a"
```

This is a result of Lisp's remarkable property of code being data.

A Little Closer to Clojure

Using the tick, we can get halfway to a DSL that looks like Clojure code.

```
(emit-bash-form
  '(println "a"))
```

Let's add this feature to emit-bash-form. We need to add a new clause to the case form. Which type should the dispatch value be?

```
user=> (class '(println "a"))
clojure.lang.PersistentList
```

So let's add a clause for clojure.lang.PersistentList.

```
(defn emit-bash-form [a]
  (case (class a)
    clojure.lang.PersistentList
     (case (name (first a))
       "println" (str "echo " (second a))
       nil)
    java.lang.String a
    java.lang.Integer (str a)
    java.lang.Double (str a)
    nil))
```

As long as we remember to quote the argument, this is not bad.

```
user=> (emit-bash-form '(println "a"))
"echo a"
user=> (emit-bash-form '(println "hello"))
"echo hello"
```

Multimethods to Abstract the Dispatch

We've made a good start, but I think it's time for some refactoring.

Currently, to extend our implementation we add to our function emit-bash-form. Eventually this function will be too large to manage; we need a mechanism to split this function into more manageable pieces.

Essentially emit-bash-form is dispatching on the type of its argument. This dispatch style is a perfect fit for an abstraction Clojure provides called a *multimethod*.

Let's define a multimethod called emit-bash. Here is the complete multimethod.

```
(defmulti emit-bash
  (fn [form]
    (class form)))
(defmethod emit-bash
  clojure.lang.PersistentList
  [form]
  (case (name (first a))
    "println" (str "echo " (second a))
    nil))
(defmethod emit-bash
  java.lang.String
  [form]
  form)
(defmethod emit-bash
  java.lang.Integer
  [form]
  (str form))
(defmethod emit-bash
  java.lang.Double
  [form]
  (str form))
```

A multimethod is actually fairly similar to a case form. Let's compare this multimethod with our previous case expression. defmulti is used to create a new multimethod, and associates it with a dispatch function.

```
(defmulti emit-bash
  (fn [form]
    (class form)))
```

This is very similar to the first argument to case.

```
(case (class form)
  ...)
```

defmethod is used to add "clauses," known as methods. Here java.lang.String is the "dispatch value," and the method returns the form as-is.

```
(defmethod emit-bash
  java.lang.String
  [form]
  form)
```

This is similar to adding clauses to our case expression.

```
(case (class form)
  java.lang.String form
  ...)
```

Notice how the multimethod is like a more flexible case expression.

We can put methods wherever we like; anyone who can see the multimethod can add their own method from their own namespace. This is much more "open" than a case form, in which all clauses are required to be in the same code form.

Notice how this compares to Java inheritance, where modifications can only occur in a single namespace, often not one that you control. This common situation highlights some advantages of separating class definitions from implementation inheritance.

Compared to case, multimethods also have an important advantage of being able to add new dispatches without disturbing existing code.

So how can we use emit-bash? Calling a multimethod is just like calling any Clojure function.

```
user=> (emit-bash '(println "a"))
"echo a"
```

The dispatch is silently handled under the covers by the multimethod.

Extending our DSL for Batch Script

Let's say I'm happy with the Bash implementation. I feel like starting a new implementation that generates Windows Batch script. Let's define a new multimethod, emit-batch.

```
(defmulti emit-batch
  (fn [form] (class form)))
(defmethod emit-batch clojure.lang.PersistentList
  [form]
  (case (name (first a))
    "println" (str "ECHO " (second a))
    nil))
(defmethod emit-batch java.lang.String
  [form]
  form)
(defmethod emit-batch java.lang.Integer
  [form]
  (str form))
(defmethod emit-batch java.lang.Double
  [form]
  (str form))
```

We can now use emit-batch and emit-bash when we want Batch and Bash script output respectively.

```
user=> (emit-batch '(println "a"))
"ECHO a"
user=> (emit-bash '(println "a"))
"echo a"
```

Ad-hoc Hierarchies

Comparing the two implementations reveals many similarities. In fact, the only dispatch that differs is clojure.lang.PersistentList!

Some form of implementation inheritance would come in handy here.

We can tackle this with a simple mechanism Clojure provides to define global, ad-hoc hierarchies.

When I say this mechanism is simple, I mean non-compound [U1]; inheritance is not compounded into the mechanism to define classes or namespaces but rather is a separate functionality.

Contrast this to languages like Java, where inheritance is tightly coupled with defining a hierarchy of classes.

We can derive relationships from names to other names, and between classes and names. Names can be symbols or keywords. This is both very general and powerful!

We will use (derive child parent) to establishes a parent/child relationship between two keywords. isa? returns true if the first argument is derived from the second in a global hierarchy.

```
user=> (derive ::child ::parent)
nil
user=> (isa? ::child ::parent)
true
```

Let's define a hierarchy in which the Bash and Batch implementations are siblings.

```
(derive ::bash ::common)
(derive ::batch ::common)
```

Let's test this hierarchy.

```
user=> (parents ::bash)
;=> #{:user/common}
user=> (parents ::batch)
;=> #{:user/common}
```

Utilizing a Hierarchy in a Multimethod

We can now define a new multimethod emit that utilizes our global hierarchy of names.

```
(defmulti emit
  (fn [form]
    [*current-implementation* (class form)]))
```

The dispatch function returns a vector of two items: the current implementation (either ::bash or ::batch), and the class of our form (like emit-bash's dispatch function).

current-implementation is a dynamic var, which can be thought of as a thread-safe global variable.

```
(def ^{:dynamic true}
  "The current script language implementation to generate"
  *current-implementation*)
```

In our hierarchy, ::common is the parent, which means it should provide the methods in common with its children. Let's fill in these common implementations.

Remember the dispatch value is now a vector, notated with square brackets. In particular, in each defmethod the first vector is the dispatch value (the second vector is the list of formal parameters).

```
(defmethod emit [::common java.lang.String]
  [form]
  form)
(defmethod emit [::common java.lang.Integer]
  [form]
  (str form))
(defmethod emit [::common java.lang.Double]
  [form]
  (str form))
```

This should look familiar. The only methods that needs to be specialized are those for clojure.lang.PersistentList, as we identified earlier. Notice the first item in the dispatch value is ::bash or ::batch instead of ::common.

```
(defmethod emit [::bash clojure.lang.PersistentList]
  [form]
  (case (name (first a))
    "println" (str "echo " (second a))
    nil))
(defmethod emit [::batch clojure.lang.PersistentList]
  [form]
  (case (name (first a))
    "println" (str "ECHO " (second a))
    nil))
```

The ::common implementation is intentionally incomplete; it merely exists to manage any common methods between its children.

We can test emit by rebinding *current-implementation* to the implementation of our choice with binding.

```
user=> (binding [*current-implementation* ::common]
         (emit "a"))
"a"
user=> (binding [*current-implementation* ::batch]
         (emit '(println "a")))
"ECHO a"
user=> (binding [*current-implementation* ::bash]
         (emit '(println "a")))
"echo a"
user=> (binding [*current-implementation* ::common]
         (emit '(println "a")))
#<CompilerException java.lang.IllegalArgumentException:
    No method in multimethod 'emit' for dispatch value:
    [:user/common clojure.lang.PersistentList] (REPL:31)>
```

Because we didn't define an implementation for [::common clojure.lang.PersistentList], the multimethod falls through and throws an Exception.

Multimethods offer great flexibility and power, but with power comes great responsibility. Just because we can put our multimethods all in one namespace doesn't mean we should. If our DSL becomes any bigger, we would probably separate all Bash and Batch implementations into individual namespaces.

This small example, however, is a good showcase for the flexibility of decoupling namespaces and inheritance.

Icing on the Cake

We've built a nice, solid foundation for our DSL using a combination of multimethods, dynamic vars, and ad-hoc hierarchies, but it's a bit of a pain to use.

```
(binding [*current-implementation* ::bash]
  (emit '(println "a")))
```

Let's eliminate the boilerplate. But where is it?

The binding expression is an good candidate. We can reduce the chore of rebinding *current-implementation* by introducing with-implementation (which we will define soon).

```
(with-implementation ::bash
  (emit '(println "a")))
```

That's an improvement. But there's another improvement that's not as obvious: the quote used to delay our form's evaluation. Let's use script, which we will define later, to get rid of this boilerplate:

```
(with-implementation ::bash
  (script
    (println "a")))
```

This looks great, but how do we implement script? Clojure functions evaluate all their arguments before evaluating the function body, exactly the problem the quote was designed to solve.

To hide this detail we must wield one of Lisp's most unique forms: the macro.

The macro's main drawcard is that it doesn't implicitly evaluate its arguments. This is a perfect fit for an implementation of script.

```
(defmacro script [form]
  '(emit '~form))
```

(That first ' should really be a backtick. The editor had a brainfreeze and couldn't figure out how to get a backtick through the build system intact.)

To get an idea what is happening, here's what a call to script returns and then implicitly evaluates.

```
(script (println "a"))
=>
(emit '(println "a"))
```

It isn't crucial that you understand the details, rather appreciate the role that macros play in cleaning up the syntax.

We will also implement with-implementation as a macro, but for different reasons than with script. To evaluate our script form inside a binding form we need to drop it in before evaluation.

```
(defmacro with-implementation
  [impl & body]
  '(binding [*current-implementation* impl]
    ~@body))
```

(Again, that ' should really be a backtick.)

Roughly, here is the lifecyle of our DSL, from the sugared wrapper to our unsugared foundations.

```
(with-implementation ::bash
  (script
    (println "a")))
=>
(with-implementation ::bash
  (emit
    '(println "a"))
=>
(binding [*current-implementation* ::bash]
  (emit
    '(println "a")))
```

It's easy to see how a few well-placed macros can put the sugar on top of strong foundations. Our DSL really looks like Clojure code!

Conclusion

We have seen many of Clojure's advanced features working in harmony in this DSL, even though we incrementally incorported many of them. Generally, Clojure helps us switch our implementation strategies with minimum fuss.

This is notable when you consider how much our DSL evolved.

We initially used a simple case expression, which was converted into two multimethods, one for each implementation. As multimethods are just ordinary functions, the transition was seamless for any existing testing code. (In this case I renamed the function for clarity).

We then merged these multimethods, utilizing a global hierachy for inheritance and dynamic vars to select the current implementation.

Finally, we devised a pleasant syntactic interface with a two simple macros, eliminating that last bit of boilerplate that other languages would have to live with.

I hope you have enjoyed following the evolution of our little DSL. This DSL is based on a simplified version of Stevedore [U2] by Hugo Duncan [U3]. If you are interested in how this DSL can be extended, you can do no better than browsing the source code of Stevedore.

About the Author

Ambrose Bonnaire-Sergeant is a Computer Science student at the University of Western Australia. He is passionate about functional languages, Clojure being his current favourite. In his spare time, Ambrose likes to learn new programming languages, play his Clarinet and sing in local Choirs. If you are in Western Australia and are looking to start a Clojure or Functional Programming User group, you can contact Ambrose at abonnairesergeant@gmail.com [U4].

This article was written in Vim using Meikel Brandmeyer's VimClojure plugin. See more of Meikel's work here [U5].

Send the author your feedback [U6] or discuss the article in the magazine forum [U7].

External resources referenced in this article:

[U1] http://blip.tv/clojure/stuart-halloway-simplicity-ain-t-easy-4842694

[U2] https://github.com/pallet/stevedore

[U3] http://hugoduncan.org/

[U4] mailto:abonnairesergeant@gmail.com

[U5] http://kotka.de/

[U6] mailto:michael@pragprog.com?subject=growing

[U7] http://forums.pragprog.com/forums/134

Pair Programming Benefits

Two Heads Are Better than One

by Jeff Langr, Tim Ottinger

Two heads are better than one, and four hands are better than two.

Pair programming is touted as a way of building a better system: two heads are better than one, they say, and thus two heads will usually produce a higher-quality system. Follow the rules of pairing (see last month's article, Pair Programming in a Flash[U1]), and you'll have an even better chance of realizing this potential.

Review

A colleague and friend of ours said that he despises pairing, but he does it all the time and teaches others to do it. The reason? "It sure makes the code nicer."

The review element of pairing is essential: Unlike manufactured products, code product not only ends up in the consumer's hands, it also stays beneath the programmers' collective feet. As Uncle Bob Martin says, the primary input to a programmer is yesterday's code. Code can serve as a good foundation, or a constant hindrance.

Good code can make it easier to track down the source of a defect. Bad code obscures important details, and duplication scatters them all over the code base. Bad code leaves you scratching your head when your system has crashed and customers (and VPs) are screaming for you to get it back up.

Is this increase in quality enough of a reason to consider throwing two people at the problem? Laurie William's book *Pair Programming Illuminated* goes into considerable detail on the costs and benefits of pairing. The statistic that is most quoted from this book is that pairs produce higher-quality code in 15% more time than individuals. For that additional cost, what other returns on investment can pairing produce?

In the remainder of this article, we'll present our list of benefits (a few of which are the same as outlined in the Williams book) that we've accrued over the past 10+ years of pairing experience. Nothing comes free, of course; there are most certainly costs and other considerations to take into account when considering pairing.

We pair because it makes the code better, and makes us better.

Team and System Benefits

- The value of increased system quality can't be diminished. Allowing slap-happy programmers to run roughshod over a system will drag down future productivity, compounding costs every minute that it's allowed to continue. What do you really know about the quality of product your team members produce?

- Pairing rotation expands the sphere of knowledge of all developers on a team. This broader knowledge increases the potential for individuals to recognize duplicate logic across the code base. Increased awareness of other parts of the system can also help contribute to a better overall system design.
- We tout the team room concept as one of the best ways to increase collaboration and productivity. However, it's not without trade-offs. A room populated with a whole team can be noisy and distracting at times. Pairing can help: A focused pair can more easily block out distractions than an individual. People are also less likely to interrupt a pair deep in work and conversation than an individual sitting alone.
- A set of programmers each doing their own thing in a private office or cube does not a true team make. A real team collaborates closely, and team members understand each other as individuals. Pairing is a great way to get there.
- At some level, standards are useful beasts (although it's possible to go too far with them). But without appropriate mechanisms in place, standards begin to quickly fall by the wayside until they're no longer valuable. The peer pressure of pairing can help ensure that we continue to adhere to basic team agreements.

Programmer Benefits

- Pairing helps prevent pigeonholing. Not only will you move throughout all responsibilities on your team, but you'll also be more free to move to other teams, as your managers learn that they will not be devastated by your departure.
- As a new hire in a pairing environment, you don't spend week one (or month one) sitting and reading out-of-date documentation or fearing a code base that you can barely begin to understand on your own. Instead, you get to jump right in and wet your feet with live production code. The rest of the team doesn't resent having to take time out from "their" work to answer your endless questions about the system--they can instead work with you directly, because that's how the team has chosen to work.
- We don't know about you, but our experiences with ex post facto reviews in lieu of pairing have usually been far from enjoyable. We find that they take a lot of time and distract us from "our" work, which means we typically give them short shrift. We suspect most other programmers feel the same way. When code "in review" cannot be committed to the main development line, it rots while the version control system marches on. Waiting for a code review may subject a programmer to a very costly merge.
- We love learning new things about software development. We think we're pretty good at programming, yet rarely a day goes by when we don't learn something new and significant--even from the most junior programmers on the team.
- If you're the team's rock star, pairing can give you mentoring and teaching opportunities that you've never had before, plus the respect you deserve.

Invariably, a great programmer on any team (whether outgoing or quiet) becomes revered by the team. If you have the skills alone, you have the skills paired too.

- If you are the weakest player on the team, you will find that pairing gives you an opportunity to learn from your teammates. In addition, as the partner shares the keyboard and ensures that you're doing test-first work, you will find that it's harder to make a mistake that gets through to integration (let alone release). You have a safer working/learning environment.
- Pairing is enjoyable and sustainable. Lest you think we only consult with teams, not drinking our own Kool-Aid, both of us have paired daily for extended periods as part of software development teams. We appreciate the social and personal growth aspects of pairing immensely.
- When you are tired, frustrated, less well, hungover, underslept, low on biorhythms or feeling unlucky, you are far more likely to stay engaged and productive if you are pairing. Partners look out for you. Your worse days pairing won't look like your worse days as a solo programmer.
- Accomplishment is the ultimate motivator. Working in pairs allows you to participate in successes more often than solo work does.

Management/Project Management Benefits

- We promote expertise, not specialty. The increased team member knowledge gained from pair rotation reduces your risk of depending on team specialists. Most team members will end up with competency in most areas of your system. Loss of an expert does not devastate your team's productivity while you secure a replacement.
- New hires usually represent a drain on productivity. We've been in shops where new hires weren't trusted to work alone for months (and in one place, years). With pairing in place, however, a new hire almost immediately becomes a productive team contributor.
- Not only do you need not worry about losing team members, you can use pairing as part of a larger "cross-pollination" strategy. If you manage multiple pairing teams, you can swap team members with negligible negative impact to the teams involved (see previous bullet). Temporarily swapping team members can reinvigorate both teams by introducing new perspectives or techniques.
- Individual capabilities are usually all over the map in a typical team. Planning and estimation is tougher because of these disparities. Pairing instead begins to produce a more-leveled team: Under-performers are pulled up by their more-capable team members, producing a team that has a better long-term chance for success. The leveling produces a more predictable rate of development, which in turn can improve the quality of estimates.
- No one can hide in a team that's pairing. It's tough for team members to go off and surf the net when their peers are depending on them to

contribute via pairing. An engaged team is status quo when frequent pair-swapping is common.

- Your technologies of choice become far less important as new hire criteria. It can be tough to find a qualified Clojure developer, for example, but if you already have a team who is well-versed in Clojure and pairing, it's a non-issue. Instead of technologies, you concern yourself with primarily two things: attitude and aptitude. Can this candidate work well with my team (and does he or she want to work in this manner), and does he or she have the chops to quickly learn the technologies and contribute?
- Interviews themselves become simpler. A few minutes of relaxed pairing with team members is often all it takes to determine if a candidate is up to it. No dumb puzzles or whiteboard programming sessions required!

Ultimately, what is the value of a true team that works well together, collaborates, continuously improves the code base, and encourages each member to improve? That's the kind of team that you can foster with healthy pairing. The bean counters might not get it, but the benefits to all involved—be they programmer, manager, customer, or business—warrants serious consideration.

About Jeff

Jeff Langr has been happily building software for three decades. In addition to co-authoring Agile in a Flash [U2] with Tim, he's written over 100 articles on software development and a couple books, *Agile Java* and *Essential Java Style*, and contributed to Uncle Bob's *Clean Code*. Jeff runs the consulting and training company Langr Software Solutions from Colorado Springs.

About Tim

Tim Ottinger is the other author of Agile in a Flash [U3], another contributor to *Clean Code*, a 30-year (plus) software developer, agile coach, trainer, consultant, incessant blogger, and incorrigible punster. He writes code. He likes it.

Send the authors your feedback [U4] or discuss the article in the magazine forum [U5].

External resources referenced in this article:

[U1] http://pragprog.com/magazines/2011-06/pair-programming-in-a-flash

[U2] http://www.pragprog.com/refer/pragpub25/titles/olag/Agile-in-a-flash

[U3] http://www.pragprog.com/refer/pragpub25/titles/olag/Agile-in-a-flash

[U4] mailto:michael@pragprog.com?subject=agile-reflections

[U5] http://forums.pragprog.com/forums/134

When Did That Happen?

The UNIX Operating System

by Dan Wohlbruck

UNIX turns 37 this month, and Dan flashes back to the 70s to see how it all began.

In July 1974, an article appeared in the Communications of the ACM that described Bell Labs' UNIX operating system. The article, written by Dennis M. Ritchie and Ken Thompson and titled "The UNIX Time-Sharing System," is one of the most famous in all of computer science. On the anniversary of that paper the story of the origin of UNIX is worth recalling.

UNIX derived its name and foundational concepts from an ambitious multi-sponsor project intended to produce a time sharing system called MULTICS. Thompson and Ritchie, along with a few others, represented Bell Labs on the MULTICS project team.

At the 1965 Fall Joint Computer Conference, two of the MULTICS project's three sponsors submitted a paper called "An Introduction and Overview of the MULTICS System." The authors, F. J. Corbato from the Massachusetts Institute of Technology and V. A. Vyssotsky from Bell Laboratories, Inc., introduced MULTICS by saying, "MULTICS is a comprehensive, general-purpose programming system which is being developed as a research project.... One of the overall design goals is to create a computing system which is capable of meeting almost all of the present and near-future requirements of a large computer utility."

The paper spelled out what these requirements were: to "run continuously and reliably 7 days a week, 24 hours a day... be capable of meeting wide service demands: from multiple man-machine interaction to the sequential processing of absentee-user jobs; from the use of the system with dedicated languages and subsystems to the programming of the system itself; and from centralized bulk card, tape, and printer facilities to remotely located terminals." And it had to be adaptable: "Because the system must ultimately be comprehensive and able to adapt to unknown future requirements, its framework must be general, and capable of evolving with time."

Given the state of computing in 1965, the requirements were ambitious. Many in the audience believed that MULTICS would never be completed.

From the Ashes of MULTICS

Indeed, as the development effort wore on, Bell Labs became frustrated with progress on MULTICS. Ritchie recalled, "For computer science at Bell Laboratories, the period 1968-1969 was somewhat unsettled. The main reason for this was the slow, though clearly inevitable, withdrawal of the Labs from the MULTICS project. To the Labs' computing community as a whole, the problem was the increasing obviousness of the failure of MULTICS to deliver promptly any sort of usable system, let alone the panacea envisioned earlier."

Richie and some of his colleagues felt the pain directly. "We were among the last Bell Laboratories holdouts actually working on MULTICS, so we still felt some sort of stake in its success. More important, the convenient interactive computing service that MULTICS had promised to the entire community was in fact available to our limited group.... Thus, during 1969, we began trying to find an alternative to MULTICS."

Ultimately, because of the apparent obviousness of failure, Bell Labs and the few remaining technicians assigned to it withdrew from the MULTICS project and decided to go their own way.

With this background of disappointment, Ken Thompson and Dennis Ritchie began to design and write an operating system for Bell Labs. As a spin on the name MULTICS, they called it UNIX, and their goals were less ambitious and the effort was more cost-effective. And most important, Thompson and Ritchie delivered. When UNIX went operational in 1970, it worked.

UNIX was first installed on a PDP-7 but in late 1970 it was ported to a PDP-11. Ritchie wrote about a part of the process this way: "Every program for the original PDP-7 Unix system was written in assembly language, and bare assembly language it was—for example, there were no macros. Moreover, there was no loader or link-editor, so every program had to be complete in itself."

The porting experience underscored the need for high-level language support. "Thompson decided that we could not pretend to offer a real computing service without Fortran, so he sat down to write a Fortran in TMG. As I recall, the intent to handle Fortran lasted about a week."

Thompson scrapped the plan to implement a Fortran and instead defined and implemented a compiler for a language he called B. The influences for B were the BCPL language (Basic Common Programming Language designed at Cambridge University and, according to Richie, "Thompson's taste for spartan syntax, and the very small space into which the compiler had to fit." B wasn't the ultimate programming language. It compiled (slowly) to simple interpretive code that ran rather slowly. But it made work at the Labs easier. "Once interfaces to the regular system calls were made available," Ritchie said, "we began once again to enjoy the benefits of using a reasonable language to write what are usually called systems programs: compilers, assemblers, and the like."

The Essential Language Component

B's successor was, of course, C. And C was a critical element of the success of UNIX.

Ritchie started working on C in 1971, and by 1973 it was solid and powerful enough that it made sense to rewrite the UNIX kernel in C. "It was at this point," Ritchie said, "that the system assumed its modern form; the most far-reaching change was the introduction of multi-programming. There were few externally-visible changes, but the internal structure of the system became much more rational and general. The success of this effort convinced us that C was useful as a nearly universal tool for systems programming, instead of just a toy for simple applications."

By 1979 virtually all Unix utilities and most UNIX application programs were written in C. And UNIX had proven itself to be the answer to the ambitious challenge laid down by the planners of MULTICS fourteen years earlier.

"It seems certain, Ritchie said, "that much of the success of Unix follows from the readability, modifiability, and portability of its software that in turn follows from its expression in high-level languages."

After co-authoring UNIX, in 1989 Thompson and Ritchie received the NEC Prize for significant contributions to computer technology and, in 1998, they were awarded the U.S. National Medal of Technology for the development of the UNIX operating system.

It all started in over 40 years ago—and that's when it happened.

About the Author

Dan Wohlbruck has over 30 years of experience with computers, with over 25 years of business and project management experience in the life and health insurance industry. He has written articles for a variety of trade magazines and websites. He is currently hard at work on a book on the history of data processing.

Like this article? Hate it? Want to remind us about Ada Lovelace's translation and expansion of Luigi Menabrea's notes on Charles Babbage's Analytical Engine, and make the case that Ada really wrote the first computer book? Send the author your feedback [U1] or discuss the article in the magazine forum [U2].

External resources referenced in this article:

[U1] mailto:michael@pragprog.com?subject=history

[U2] http://forums.pragprog.com/forums/134

Calendar

Author sightings, partner events, and other notable happenings.

Open and Clojure

May I, your humble editor, draw your attention to a couple of upcoming conferences?

OSCON

Clojure/conj

First, O'Reilly's big open source bash, OSCON [U1]. Mostly because I will be there, rubbing elbows and bending them in my painstaking research into open source technology and Oregon craft beers.

To me, OSCON is proof that the concept of the big technology conference is not dead in the head. It would be easy to conclude otherwise. Regional conferences and online gatherings have sucked up a lot of the milkshake of the big shindigs. A lot of those big conferences have in fact gone away, and some of those that survive don't seem very lively. Not mentioning any names. But OSCON always has a solid lineup of speakers who have something relevant to say, and the attendees always seem savvy to me. Maybe it's because it's an *open source* conference.

And Portland is a great town for a conference. It's a town of neighborhoods. You can visit it for a couple of days and feel a sense of ownership, even though—or because—you've spent all your time in and around Powell's World of Books in the Pearl district or hanging out on Alberta Street. And of course it's the home to more excellent craft beers than anyplace else on the planet. I speak as a proud Oregonian since 1999.

Second, Clojure/conj. This November 10–12 Raleigh, NC, event falls outside the three-month window of our events calendar, but well inside our issue

theme. Rich Hickey will be speaking. Michael Focus. Stuart Halloway. Chris Houser. Aaron Bedra. Do I really have to say more, if you're into Clojure?

The fun actually starts on November 7 with classes taught by members of clojure/core and runs on through the conference. Class topics include: Functional programming, Lisp syntax, The sequence library, Concurrency, Java interop, Multimethods, Macros, OO Revisited, and The Clojure ecosystem.. Details of the conference will emerge on the website [U2] as the date grows closer.

Author Appearances

Who's where, and what for.

July 9–14 **"Implied Aesthetics: A Sensor-based Approach towards Mobile Interfaces"**
Daniel Sauter, author of Mobile Processing [U3]
HCI International 2011 [U4], Orlando, Florida

July 12 **"Driving Technical Change"**
Terence Ryan, author of Driving Technical Change [U5]
Uberconf [U6], Denver, CO

July 12–15 **"Uber Groovy," "Grails: Bringing Radical Productivity to the JVM," Apprenticeship Birds of a Feather**
Dave Klein, author of Grails: A Quick-Start Guide [U7]
UberConf [U8], Denver, CO

July 13–15 **"Java is from Mars, Ruby is from Venus"**
Paolo Perrotta, author of Metaprogramming Ruby [U9]
Conferencia Rails [U10], Madrid, Spain

July 18–21 **iPhone Studio**
Daniel Steinberg, co-author of iPad Programming [U11]: A Quick-Start Guide for iPhone Developers and author of Cocoa Programming [U12]: A Quick-Start Guide for Developers
Pragmatic Studios [U13], Reston, VA

July 19 **"Just Enough C For Open Source Projects"**
Andy Lester, author of Land the Tech Job You Love [U14]
Software Craftsmanship McHenry County's monthly meeting [U15], McHenry County, IL

July 22 **"Practical Test Automation"**
Jared Richardson, author of Ship It! [U16]
Southern Fried Agile [U17], Charlotte, NC

July 22–23 **Facilitating**
Rachel Davies, co-author of Agile Coaching [U18]
Agile Coaches Gathering [U19], Bletchley Park, UK

July 25–29 **"Projects and Community With Github"**
Andy Lester, author of Land the Tech Job You Love [U20]
OSCON [U21], Portland, OR

July 27 **"Building and Maintaining a Project Community with Github"**
Andy Lester, author of Land the Tech Job You Love [U22]
OSCON [U23], Portland, OR

July 27 **"Lightning Fast Clojure"**
Aaron Bedra, co-author of Programming Clojure, Second Edition [U24]
OSCON [U25], Portland, OR

July 29–30 **"Playfulness at Work: a Real Serious Message(tm) with Ruby as the Medium"**
Ian Dees, author of Scripted GUI Testing with Ruby [U26] and coauthor of Using JRuby [U27]
Cascadia Ruby Conf [U28], Seattle, WA

Aug 3–5 **"I Dunno, Probably Something Cool About JRuby"**
Ian Dees, author of Scripted GUI Testing with Ruby [U29] and coauthor of Using JRuby [U30]
JRubyConf [U31], Washington, DC

Aug 8–11 **Celebrating, we assume**
Andrew Hunt, author of Pragmatic Thinking and Learning [U32]
Agile/XP 2011 10th Anniversary [U33], Salt Lake City, UT

Aug 8–12 **Facilitating Open Jam**
Rachel Davies, co-author of Agile Coaching [U34]
Agile2011 [U35], Salt Lake City, UT

Aug 11 **"The Only Agile Tools You'll Ever Need"**
Jeff Langr, co-author of Agile in a Flash [U36] (with Tim Ottinger) and *Agile Java*
Agile 2011 [U37], Salt Lake City, UT

Aug 12–13 **Keynote: "Your Code: The Director's Cut" plus sessions "Mac OS X for iOS Developers" and "The Key to Blocks"**
Daniel Steinberg, co-author of iPad Programming [U38]: A Quick-Start Guide for iPhone Developers and author of Cocoa Programming [U39]: A Quick-Start Guide for Developers
Cocoa Conf [U40], Columbus, OH

Aug 12–13 **"Introduction to AV Foundation" and "Advanced AV Foundation"**
Chris Adamson, author of iPhone SDK Development [U41]
Cocoa Conf [U42], Columbus, OH

Aug 12–13 **Emcee**
Dave Klein, author of Grails: A Quick-Start Guide [U43]
CocoaConf [U44], Columbus, OH

Aug 22–25 **iPhone Studio**
Daniel Steinberg, co-author of iPad Programming [U45]: A Quick-Start Guide for iPhone Developers and author of Cocoa Programming [U46]: A Quick-Start Guide for Developers
Pragmatic Studios [U47], Denver, CO

Sept 2 **"Pomodoro Technique—Can you focus for 25 minutes?"**
Staffan Nöteberg, author of Pomodoro Technique Illustrated [U48]
Reaktor Dev Day [U49], Helsinki, Finland

Sept 7–9 **"Unleashing your Inner Hacker"**
Aaron Bedra, co-author of Programming Clojure, Second Edition [U50]
Heartland Developers Conference [U51], Omaha, NE

Sept 7–9 **Opening Keynote**
Rachel Davies, co-author of Agile Coaching [U52]
ALE2011 [U53], Berlin, Germany

Sept 9–11 **"Hello Groovy!," "Grails: Bringing Radical Productivity to the JVM"**
Dave Klein, author of Grails: A Quick-Start Guide [U54]
New England Software Symposium [U55], Boston, MA

Sept 15–16 **"Surfing the Agile Wave"**
Rachel Davies, co-author of Agile Coaching [U56]
Agile on the Beach [U57], Falmouth, UK

Sept 16 **Keynote**
Andrew Hunt, author of Pragmatic Thinking and Learning [U58]
Innovate Virginia [U59], Richmond VA

Sept 16 **"Building Analytics with Clojure"**
Aaron Bedra, co-author of Programming Clojure, Second Edition [U60]
Innovate Virginia [U61], Richmond VA

Sept 16 **"Tightening Your Feedback Loop"**
Jared Richardson, author of Ship It! [U62]
Innovate Virginia [U63], Richmond VA

Sept 18–20 **"Clojure Part 2: Building Analytics with Clojure"**
Aaron Bedra, co-author of Programming Clojure, Second Edition [U64]
Strangeloop [U65], St. Louis, MO

Sept 19 **"Skynet: A Scalable, Distributed Service Mesh in Go"**
Brian Ketelsen, author of The Go Programming Language [U66]
Strangeloop [U67], St. Louis, MO

O'Reilly Events

Upcoming events from our friends at O'Reilly.

July 12 **O'Reilly Strata Online Conference: "In this Strata OLC, we'll look at the rapidly growing field of personal analytics. We'll discuss tool stacks for recording lives, and hear surprising stories about what happens when introspection meets technology."**
Strata [U68], online

July 25–29 **O'Reilly Open Source Convention: "Join today's open source innovators, builders, and pioneers as they gather at the Oregon Convention Center in Portland, Oregon, to share their expertise and experience, explore new ideas, and inspire each other."**
OSCON [U69], Portland, OR

July 30–31 **Maker Faire Detroit: "Our mission at Maker Media, a division of O'Reilly Media and home to MAKE Magazine, Maker Faire and the host of other inspirational and instructional Maker Media brands, is to unite, inspire, inform, and entertain a growing community of highly imaginative and resourceful people who undertake amazing projects in their backyards, basements, and garages. We call these people 'Makers.'"**
Maker Faire Detrtoit [U70], Dearborn, MI

Sept 17–18 **World Maker Faire**
World Maker Faire [U71], New York, NY

Sept 22–23 **O'Reilly Strata Conference: Making data work: "With hardcore technical sessions on parallel computing, machine learning, and interactive visualizations; case studies from finance, media, healthcare, and technology; and provocative reports from the leading edge, Strata Conference showcases the people, tools, and technologies that make data work."**
Strata NY [U72], New York, NY

USENIX Events

What's coming from our USENIX friends.

July 11–12 **The 2nd ACM SIGOPS Asia-Pacific Workshop on Systems "will be a forum for systems researchers and practitioners across the world to present their work in computer systems (broadly defined) and for locals in Asian/Pacific region to meet, interact, and collaborate with top researchers in the field."**
APSys 2011 [U73], Shanghai, China

Aug 8–12 **The 20th USENIX Security Symposium. "USENIX Security '11 brings together researchers, practitioners, system administrators, system programmers, and others interested in the latest advances in the security of computer systems and networks." Not to mention a bunch of collocated conferences: 2011 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections, 4th Workshop on Cyber Security Experimentation and Test, USENIX Workshop on Free and Open Communications on the Internet, 5th USENIX Workshop on Offensive Technologies, 2nd USENIX Workshop on Health Security and Privacy, 6th USENIX Workshop on Hot Topics in Security, and Sixth Workshop on Security Metrics.**
USENIX Security '11 [U74], San Francisco, CA

Welcome to Your Next Decade, Pioneer

July 9 **Browser pioneer Marc Andreessen enters his 40s.**

July 16 **VisiCalc co-creator Dan Bricklin enters his 60s.**

Aug 9 **Alfred Aho, co-creator of AWK language, enters his 70s.**

Aug 10 **Jan Rajchman, inventor of the Selectron tube memory, would have turned 100 today.**

Aug 11 **WWII British computer pioneer Tom Kilburn would have turned 90 today.**

Aug 12 **The IBM PC was released 30 years ago today.**

Sept 9 **C creator Dennis Ritchie enters his 70s.**

External resources referenced in this article:

[U1] http://www.oscon.com/oscon2011

[U2] http://clojure-conj.org/

[U3] http://pragprog.com/refer/pragpub25/titles/dsproc

[U4] http://www.hcii2011.org/

[U5] http://pragprog.com/refer/pragpub25/titles/trevan/driving-technical-change

[U6] http://uberconf.com/conference/denver/2011/07/home

[U7] http://pragprog.com/refer/pragpub24/titles/dkgrails/grails

[U8] http://uberconf.com/conference/denver/2011/07/home

[U9] http://pragprog.com/refer/pragpub25/titles/ppmetr/metaprogramming-ruby

[U10] http://conferenciarails.org/

[U11] http://pragprog.com/refer/pragpub24/titles/sfipad

[U12] http://pragprog.com/refer/pragpub24/titles/DSCPQ

[U13] http://pragmaticstudios.com

[U14] http://pragprog.com/refer/pragpub25/titles/algh/land-the-tech-job-you-love

[U15] http://mchenry.softwarecraftsmanship.org/

[U16] http://pragprog.com/refer/pragpub25/titles/prj/ship-it

[U17] http://southernfriedagile.com/

[U18] http://pragprog.com/refer/pragpub25/titles/sdcoach/agile-coaching

[U19] http://www.agilecoachesgathering.org/

[U20] http://pragprog.com/refer/pragpub24/titles/algh/land-the-tech-job-you-love

[U21] http://www.oscon.com/oscon2011

[U22] http://pragprog.com/refer/pragpub25/titles/algh/land-the-tech-job-you-love

[U23] http://www.oscon.com/oscon2011/public/schedule/detail/19110

[U24] http://pragprog.com/refer/pragpub25/titles/shcloj2/

[U25] http://www.oscon.com/oscon2011/public/schedule/detail/20020

[U26] http://pragprog.com/refer/pragpub25/titles/idgtr/scripted-gui-testing-with-ruby

[U27] http://pragprog.com/refer/pragpub25/titles/jruby/using-jruby

[U28] http://cascadiarubyconf.com/

[U29] http://pragprog.com/refer/pragpub25/titles/idgtr/scripted-gui-testing-with-ruby

[U30] http://pragprog.com/refer/pragpub25/titles/jruby/using-jruby

[U31] http://jrubyconf.com/

[U32] http://pragprog.com/refer/pragpub25/titles/ahptl/pragmatic-thinking-and-learning

[U33] http://agile2011.agilealliance.org/

[U34] http://pragprog.com/refer/pragpub25/titles/sdcoach/agile-coaching

[U35] http://agile2011.agilealliance.org/

[U36] http://www.pragprog.com/refer/pragpub24/titles/olag/Agile-in-a-flash

[U37] http://agile2011.agilealliance.org/

[U38] http://pragprog.com/refer/pragpub24/titles/sfipad

[U39] http://pragprog.com/refer/pragpub24/titles/DSCPQ

[U40] http://cocoaconf.com

[U41] http://www.pragprog.com/refer/pragpub24/titles/amiphd/iphone-sdk-development

[U42] http://cocoaconf.com

[U43] http://pragprog.com/refer/pragpub25/titles/dkgrails/grails

[U44] http://cocoaconf.com

[U45] http://pragprog.com/refer/pragpub24/titles/sfipad

[U46] http://pragprog.com/refer/pragpub24/titles/DSCPQ

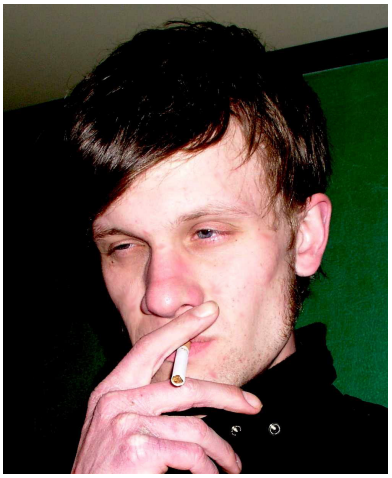
[U47] http://pragmaticstudios.com
[U48] http://pragprog.com/refer/pragpub25/titles/snfocus/pomodoro-technique-illustrated
[U49] http://reaktordevday.fi/en/
[U50] http://pragprog.com/refer/pragpub25/titles/shcloj2/
[U51] http://www.heartlanddc.com/?page_id=411
[U52] http://pragprog.com/refer/pragpub25/titles/sdcoach/agile-coaching
[U53] http://ale2011.eu/
[U54] http://pragprog.com/refer/pragpub25/titles/dkgrails/grails
[U55] http://nofluffjuststuff.com/conference/boston/2011/09/home
[U56] http://pragprog.com/refer/pragpub25/titles/sdcoach/agile-coaching
[U57] http://www.agileonthebeach.com/
[U58] http://pragprog.com/refer/pragpub25/titles/ahptl/pragmatic-thinking-and-learning
[U59] http://innovatevirginia.com/
[U60] http://pragprog.com/refer/pragpub25/titles/shcloj2/
[U61] http://innovatevirginia.com/
[U62] http://pragprog.com/refer/pragpub25/titles/prj/ship-it
[U63] http://innovatevirginia.com/
[U64] http://pragprog.com/refer/pragpub25/titles/shcloj2/
[U65] https://thestrangeloop.com/sessions/clojure-part-2-building-analytics-with-clojure
[U66] http://pragprog.com/refer/pragpub25/titles/bkgo
[U67] https://thestrangeloop.com/
[U68] http://strataconf.com/strata-july2011
[U69] http://www.oscon.com/oscon2011
[U70] http://makerfaire.com/detroit/2011
[U71] http://makerfaire.com/newyork/2011
[U72] http://strataconf.com/stratany2011
[U73] http://apsys11.ucsd.edu/
[U74] http://www.usenix.org/events/sec11/

Shady Illuminations

The Three Virtues of IBM

by John Shade

IBM is 100 years old. That's old, especially when you remember that it cryogenically freezes its CEOs at age 60.



Last month IBM celebrated its 100th birthday. Did you break out the bubbly?

One hundred years. That's old. That's three digits. IBM is so old that if I made up some dumb joke about the wrecking ball [U1] tearing down the old Endicott Building [U2], IBM would get it.

One hundred years. IBM doesn't let its CEOs get this old. It cryogenically freezes them at 60, and keeps them around because you're not allowed to leave.

Although you know, technically, last month was really the 100th anniversary of the formation of a company named C-T-R. That later became IBM. But we're not going to quibble over a name change, right?

And of course Tom Watson wasn't in charge yet, and until he was running things it wasn't really IBM. Not the IBM of story [U3] and song [U4]. In fact, Watson named it IBM in 1924, so that makes IBM technically 87. But we're not going to let petty technicalities overshadow the celebration, right?

A Virtual Orgy

And celebration there was: a virtual orgy of congratulation. Mostly self-congratulation. IBM's web sites were taken over by lists of the company's key patents, innovations, and milestones, temporarily bumping aside the white papers on how IBM's cloud services are reshaping business. Watson (the program, not the man) was named Person of the Year by the Webby Awards and Lisa Kudrow. One journalist strained to imagine what it must have been like for IBM as a startup 100 years ago. Ignoring the fact that IBM was never a startup, but was formed by the merger of three established firms.

Or, actually, that's how C-T-R was formed. IBM was formed by replacing the letterhead. But there I go raining on the party again.

And I'm going to go on doing it. Forgive me if I'm not as impressed as I'm supposed to be by this list of patents and inventions. If a significant percentage of the people who have ever worked in tech have been your employees, if you're one of the biggest companies in the field and for decades were bigger than all the rest of the field put together, if you've been around for a century, and if you take credit for all the innovations of people working for you, then shouldn't the lion's share of the patents and inventions and such be associated with you? If they're not, what the heck have you been doing for the past hundred years?

Or 87 in base ten.

Of course a lot of the innovation in IBM these days is buying young companies in order to connect their products and services with IBM's customer base. And

that's not taking credit for the innovations of your employees. That's taking credit for the innovations of other companies' employees. So that's different.

A Virtuoso Performance

But let's be fair. IBM has racked up a pretty impressive record.

In the 1990s IBM achieved the biggest losses in the history of business.

It was once a giant, bigger than all of its competitors combined. The Seven Dwarfs, they called IBM's competitors, while IBM was Snow White, because of the integrity of its sales staff and their impressive personal hygiene. Today, IBM is just one tech megacorporation among many, although its salespeople are still clean.

It has sold off whole market segments: Personal computers. Disk drives. Printers. Copiers and duplicators. Phones. Satellites.

What other company could have run up this remarkable record?

Of course there were some positive achievements along the way, too.

The floppy disk. The UPC barcode. The social secutity system. The Apollo space missions. A little googling tells me that those are a few of the trophies on the IBM mantle. (Although I'd never heard that IBM had invented the Apollo space missions. That's sort of disppointing.)

I suppose the bottom line is, you can't argue with success. Or so they say. I think you can. I think it's survival that you can't argue with.

And IBM has survived. Shrunken to a shriveled homunculus of its former obesity, it still survives. Prospers, even. It will probably survive another 87 years, and if it does, I predict that it will be because of IBM's three virtues.

The Three Virtues

1. IBM knows how to articulate its vision.

Thomas Watson, Sr. was the consummate salesman, and IBM has never lost sight of the need to sell—ideas, technology, products, services. It has occasionally lost sight of its market. "Today we define our space as enterprise and global," current IBM CEO Sam Palmisano said recently. "We don't sell to consumers. We have in the past, but we don't anymore."

Like the personal computer business. That was selling to consumers. Do you see how cavalierly Palmisano dismisses a whole industry and a whole era of IBM's history?

As he should. IBM should never have been in the personal computer business. That the company was eventually able to see this and exit from what at the time looked like the center of the tech universe is impressive.

Palmisano tells the story of a Harvard professor who while researching a book interviewed IBMer around the world. According to Palmisano, she was impressed that they all said what he said—the message had gotten through. Get 400,000 people all saying the same thing and you have power.

And you have clear articulation of vision. When management can communicate the vision to the press, the customers, and the staff with no cognitive leakage, that's impressively articulate.

2. IBM has always shown respect for intelligence: it likes bright people.

That seems to be a value across the board. IBM hired the greatest designers and architects to design its buildings: Eero Saarinen, Charles and Ray Eames, Paul Rand, Ludwig Mies van der Rohe. IBM branded itself internally and externally with the mantra "Think." It prides itself on pushing the limits of machine intelligence, pitting computers against chess grandmasters and Jeopardy champions. Tom Watson, Jr., promoted non-discrimination policies explicitly for the competitive advantage this gave IBM in hiring purely based on smarts.

Brightness is an IBM value and virtue.

3. And then there's the boring but earnest image.

If an IBM junior executive had shown up on your doorstep in the 1950s, imagining for the moment that you were alive in the 1950s and had a doorstep, your immediate reaction would have been to look him up and down [U5] and ask, "where's the other one [U6]?"

Bill Gates and Larry Ellison and Steve Jobs and Scott McNealy and Filo and Yang and Brin and Page and other tech founders infused their companies with their personalities, and the companies are interesting as a result. IBM reflects the personalities of the two Watsons, the human ones, and is boring as a result. "We have learned not to confuse charisma with leadership," some IBMer said somewhere. I don't remember where. I should have taken notes, but honestly, it was just too boring.

That's IBM: Boring but earnest. It's an image and a value that the company promotes and embraces. And it's not just the dress code. (Wikipedia: "A dark (or gray) suit, white shirt, and a 'sincere' tie was the public uniform for IBM employees for most of the 20th century.") The dress code has relaxed, but the image hasn't changed. I'd call it *clean*.

Articulate and bright and clean: that's a storybook [U7], man.

About the Author

John Shade was born under a cloud in Montreux, Switzerland, in 1962. Subsequent internment in a series of obscure institutions of ostensibly higher learning did nothing to brighten his outlook. Nor to make it cleaner or more articulate, for that matter. Send the author your feedback [U8] or discuss the article in the magazine forum [U9].

External resources referenced in this article:

[U1] http://en.wikipedia.org/wiki/Maynard_G._Krebs

[U2] http://www-03.ibm.com/ibm/history/exhibits/endicott/endicott_intro.html

[U3] http://www.ibmandtheholocaust.com/

[U4] http://www-03.ibm.com/ibm/history/multimedia/everonward_trans.html

[U5] http://www.bobbemer.com/DRESS.HTM

[U6] http://christianity.about.com/od/jehovahswitnesses/a/Why-Do-Jehovahs-Witnesses-Practice-Door-To-Door-Evangelism.htm

[U7] http://articles.cnn.com/2007-01-31/politics/biden.obama_1_braun-and-al-sharpton-african-american-presidential-candidates-delaware-democrat?_s=PM:POLITICS

[U8] mailto:michael@pragprog.com?subject=shade

[U9] http://forums.pragprog.com/forums/134

But Wait, There's More...

Coming Attractions

This may be the last page, but that doesn't mean it's the end of the conversation. Keep up with what's going on in the Pragmatic universe by subscribing to our newsletter and drop by the Pub again next month for more Pragmatism. We'll keep a seat warm and a mug chilled and a candle burning in the window.

Coming Soon in PragPub

In the queue: We're brainstorming with screencasting guru Miles Forrest and something cool may come out of that. Frequent contributor Brian Tarbox recently attended a writers' conference and discovered how writing code is like writing dialog. We're talking to someone about an article on specs2, the specifications library from Scala. We're trying to get an article on HTML5 geolocation from Brian Hogan, but he's crazy busy. Trevor Burnham has another CoffeeScript article in mind. And we hope to introduce a new feature next issue that will give you insight into who we are and what we do.

Coming Soon on the Bookshelf

These books are now in print: *iOS Recipes: Tips and Tricks for Awesome iPhone and iPad Apps*, *CoffeeScript: Accelerated JavaScript Development*, and *Programming Concurrency on the JVM: Mastering Synchronization, STM, and Actors*.

But to Really Be in the Know...

...you need to subscribe to our weekly newsletter. It'll keep you in the loop, it's a fun read, and it's free. All you need to do is create an account on pragprog.com (email address and password is all it takes) and select the checkbox to receive newsletters.

In the Meantime

While you're waiting for the next issue of the newsletter or of *PragPub* to come out, you can follow us on twitter at @pragpub, @PragmaticAndy, or @pragdave. Or on Facebook at facebook.com/PragmaticBookshelf [U1]. Or email us at michael@pragprog.com [U2].

External resources referenced in this article:

[U1] http://www.facebook.com/PragmaticBookshelf

[U2] mailto:michael@pragprog.com