



Paul Bennett

Follow

Software, Agile and various other fleeting thoughts.

Nov 21, 2016 · 6 min read

## Clojure—the most productive environment I’ve ever used

I’ve programmed and built large systems in many different programming languages and systems. My graduate thesis was done in C, building on the kernel of Unix Version 7 that came out of Bell Labs in the 1970’s. When I came to the US in the early 80’s, I worked at the Labs and used C++. In 1991 I took a project at Morgan Stanley, where I was the only C++ programmer on a project built in Smalltalk—and when I saw what Smalltalk could do, it blew my mind. I had been struggling with various C++ IDE’s that never really lived up to their promise, and when I saw VisualWorks Smalltalk—I never wanted to do anything else. It was a total paradigm shift.

So, I worked in Smalltalk for the next 15 years or so, until around 2007, mostly in the financial services sector in NYC. I did do some Java, but the 2008 meltdown put paid to my Smalltalk career. So, what next?

Ruby seemed the nearest thing to Smalltalk at that time, and I worked at Pivotal Labs for a year or so, and a few other outfits. I also managed to find some other Smalltalk work, but no new development—they were all maintenance and performance enhancement gigs.

After that, I joined a startup, and re-wrote their existing Java app in Groovy. I had also discovered Clojure at this point, but time pressure and circumstances prevented us from using Clojure on that project. How I wish that had been different.

I didn’t get a chance to really dig in until last year, and after doing some real learning with respect to functional languages, and writing a couple of apps and a significant library (<https://github.com/wizardpb/functional-vaadin>) I can unequivocally say: Clojure is the most productive environment I’ve ever used.

Why?

There were 3 things that immediately struck me:

- Testing and Composability.
- Concurrency support.
- Multi-methods.

## Testing, Composability

Have you ever tried TDD with OO languages? How has it gone? Should be easy, right? Write a test, then red-green-refactor, you're done. Sorry, not so fast. The problem comes in 'Write a test'. The issue is not about understanding what the test should do—it's writing the fixture and setup code.

OO programs are constructed of objects, combined pieces of code and state. These objects are arranged in complicated hierarchies and networks of references between them. To test an object, the objects it references must be replaced by some kind of object that can be guaranteed to produce known results when called. These are implemented by mocks, fixtures and stubs. The problem is—setting up these can be a complicated and time consuming task, often much bigger than writing the test itself, and often much bigger than writing the code that is being tested. The result? Under the time pressure of real-world development, one of two things happens: either the tests are only half-written, or not written at all, or the architecture of the app is distorted, and designed for easy testing, NOT for a correct, robust and maintainable solution to the problem being solved. (Martin Fowler, Kent Beck and David Heinemeier Hansson had a very interesting video chat on this subject—check out <http://martinfowler.com/articles/is-tdd-dead/>). Bottom line—doing real TDD in an OO environment is difficult and time-consuming.

Not so for functional languages. Because functions are side-effect free, and generally only reference their arguments and local vars, testing them is easy—put some values in, call them, and check the result. You're done.

Now consider how this plays with the notion of function composition (which also stems from the fact that functions are side-effect free). If you know that function A is correct, and function B is correct, and

that function B will only return values that are correct for function A (all of which can be demonstrated by unit tests—especially easy when using `clojure.spec`), then you can, by the rules of function composition, assert that `A(B())` will also be correct. This means a big reduction in the amount of tests you need to write to verify your system is correct is greatly reduced.

That's a result I'm very happy with.

## Concurrency support

While Java does support concurrent programming, its model of threads, locks and volatile vars is cumbersome, error prone and very difficult to debug. Clojure goes much further, and leverages several features to make concurrent programming easier and less error prone. Firstly, its immutable data structures allow data to be shared among threads with ease—since they are immutable, threads cannot interfere with other threads. Secondly, the difference between *value* and *identity* is clearly and explicitly defined. In an OO system, these are mixed—objects have an identity, and their value is changed by mutating their state. There is no sense of an identity which is associated with different values through time. With Clojure, identity is provided by explicit constructs that can have their (now immutable) values changed *in an atomic manner, across a time span*. It is this fact that makes concurrent programming in Clojure so much easier.

There are several ways to do this.

*Atoms* allow *synchronous and independent* updates. The `swap!` function takes an atom and a function, calls the function with the current value, and sets its value to the return value of that function. If the value is changed (by another thread) during the update, the operation is retried—`swap!` is essentially a test-and-set operation.

*Refs* and a *software transactional memory* (STM) provide *synchronous and coordinated* changes. In this scheme, all updates are atomic, consistent and isolated, much as a database transaction would be. All updates and reads of a ref must be done in a `dosync` function. All function calls inside a `dosync` see a stable state, with the whole transaction being retried when an update conflict is detected.

*Agents* provide a message-based scheme allowing *asynchronous and independent* updates. Agents hold a single value, and are updated by sending a message to the agent. Messages are functions, and the agent system ensures that sometime in the future, the message function will be called with the current value of the agent and the state updated with the return value. Message execution is serialized by the agent system so that updates are applied serially.

## Multi-methods

Like OO languages, Clojure has polymorphic dispatch, implemented by multi-methods, the ability to dispatch to different versions of a function based on the arguments. Unlike OO languages, where dispatch is based solely on the type of the receiver, Clojure goes much further—further even than other Lisps, such as CLOS. Clojure multi-methods are dispatched based on the return value of a *dispatch function*. A multi-method is a set of functions, all with the same name, that share a single dispatch function. This function receives the same arguments as the function call, and returns a value—the dispatch value. Each of the functions in the multi-method set is associated with a particular dispatch value, and when a multi-method is called, the dispatch function is called, and its return value determines which version of the multi-method is invoked.

In this way, completely arbitrary, and *application-suitable* polymorphism can be defined. No longer is polymorphism strictly tied to type inheritance. Instead, the application programmer can define a dispatch scheme to directly support the problem solution. This is very powerful, and leads to much cleaner, concise code. Inheritance hierarchies can also be application-specific. These are defined by a *derive* function, based on two types of hierarchies: Java class inheritance, and application-defined relationships using the *isa?* function.

My experience is that these three things allow cleaner, faster and more correct code to be written—and that makes Clojure the most productive environment I have ever programmed in. Discovering Clojure has been the same kind of paradigm shift that moving from C++ to Smalltalk was. Right now, it's the only language I want to program in.

