Why Clojure?

by Mike Bergman - Wednesday, December 10, 2014

http://www.mkbergman.com/1825/why-clojure/

Ten Management Reasons for Choosing Clojure for Adaptive Knowledge Apps

It is not unusual to see articles touting one programming language or listing the reasons for choosing another, but they are nearly always written from the perspective of the professional developer. As an executive with much experience directing software projects, I thought a management perspective could be a useful addition to the dialog. My particular perspective is software development in support of knowledge management, specifically leveraging artificial intelligence, semantic technologies, and data integration.

Context is important in guiding the selection of programming languages. C is sometimes a choice for performance reasons, such as in data indexing or transaction systems. Java is the predominant language for enterprise applications and enterprise-level systems. Scripting languages are useful for data migrations and translations and quick applications. Web-based languages of many flavors help in user interface development or data exchange. Every one of the hundreds of available programming languages has a context and rationale that is argued by advocates.

We at Structured Dynamics have recently made a corporate decision to emphasize the Clojure language in the specific context of knowledge management development. I'd like to offer our executive-level views for why this choice makes sense to us. Look to the writings of SD's CTO, Fred Giasson, for arguments related to the perspective of the professional developer.

Some Basic Management Premises

I have overseen major transitions in programming languages from Fortran or Cobol to C, from C to C++ and then Java, and from Java to more Web-oriented approaches such as Flash, JavaScript and PHP [1]. In none of these cases, of course, was a sole language adopted by the company, but there always was a core language choice that drove hiring and development standards. Making these transitions is never easy, since it affects employee choices, business objectives and developer happiness and productivity. Because of these trade-offs, there is rarely a truly "correct" choice.

Languages wax and wane in popularity, and market expectations and requirements shift over time. Twenty years ago, Java brought forward a platform-independent design well-suited for client-server

computing, and was (comparatively) quickly adopted by enterprises. At about the same time Web developments added browser scripting languages to the mix. Meanwhile, hardware improvements overcame many previous performance limitations in favor of easier to use syntaxes and tooling. No one hardly programs for an assembler anymore. Sometimes, like Flash, circumstances and competition may lead to a rapid (and unanticipated) abandonment.

The fact that such transitions naturally occur over time, and the fact that distributed and layered architectures are here to stay, has led to my design premise to emphasize modularity, interfaces and APIs [2]. From the browser, client and server sides we see differential timings of changes and options. It is important that piece parts be able to be swapped out in favor of better applications and alternatives. Irrespective of language, architectural design holds the trump card in adaptive IT systems.

Open source has also had a profound influence on these trends. Commercial and product offerings are no longer monolithic and proprietary. Rather, modern product development is often more based on assembly of a diversity of open source applications and libraries, likely written in a multitude of languages, which are then assembled and "glued" together, often with a heavy reliance on scripting languages. This approach has certainly been the case with Structured Dynamics' Open Semantic Framework, but OSF is only one example of this current trend.

The trend to interoperating open source applications has also raised the importance of data interoperability (or ETL in various guises) plus reconciling semantic heterogeneities in the underlying schema and data definitions of the contributing sources. Language choices increasingly must recognize these heterogeneities.

I have believed strongly in the importance of interest and excitement by the professional developers in the choice of programming languages. The code writers -- be it for scripting or integration or fundamental app development -- know the problems at hand and read about trends and developments in programming languages. The developers I have worked with have always been the source of identifying new programming options and languages. Professional developers read much and keep current. The best programmers are always trying and testing new languages.

I believe it is important for management within software businesses to communicate anticipated product changes within the business to its developers. I believe it is important for management to signal openness and interest in hearing the views of its professional developers in language trends and options. No viable software development company can avoid new upgrades of its products, and choices as to architecture and language must always be at the forefront of version planning.

When developer interest and broader external trends conjoin, it is time to do serious due diligence about a possible change in programming language. Tooling is important, but not dispositive. Tooling rapidly catches up with trending and popular new languages. As important to tooling is the "fit" of the programming language to the business context and the excitement and productivity of the developers to achieve that fit.

"Fitness" is a measure of adaptiveness to a changing environment. Though I suppose some of this can be quantified -- as it can in evolutionary biology -- I also see "fit" as a qualitative, even aesthetic, thing. I recall sensing the importance of platform independence and modularity in Java when it first came out,

results (along with tooling) that were soon borne out. Sometimes there is just a sense of "rightness" or alignment when contemplating a new programming language for an emerging context.

Such is the case for us at Structured Dynamics with our recent choice to emphasize Clojure as our core development language. This choice does not mean we are abandoning our current code base, just that our new developments will emphasize Clojure. Again, because of our architectural designs and use of APIs, we can make these transitions seamlessly as we move forward.

However, for this transition, unlike prior ones I have made noted above, I wanted to be explicit as to the reasons and justifications. Presenting these reasons for Clojure is the purpose of this article.

Brief Overview of Clojure

Clojure is a relatively new language, first released in 2007 [3]. Clojure is a dialect of Lisp, explicitly designed to address some Lisp weaknesses in a more modern package suitable to the Web and current, practical needs. Clojure is a functional programming language, which means it has roots in the lamdba calculus and functions are "first-class citizens" in that they can be passed as arguments to other functions, returned as values, or assigned as variables in data structures. These features make the language well suited to mathematical manipulations and the building up of more complicated functions from simpler ones.

Clojure was designed to run on the Java Virtual Machine (JVM) (now expanded to other environments such as ClojureScript and Clojure CLR) rather than any specific operating system. It was designed to support concurrency. Other modern features were added in relation to Web use and scalability. Some of these features are elaborated in the rationales noted below.

As we look to the management reasons for selecting Clojure, we can really lump them into two categories: a) those that arise mostly from Lisp, as the overall language basis; and b) specific aspects added to Clojure that overcome or enhance the basis of Lisp.

Reasons Deriving from Lisp

Lisp (defined as a *lis*t *p*rocessing language) is one of the older computer languages around, dating back to 1958, and has evolved to become a family of languages. "Lisp" has many variants, with Common Lisp one of the most prevalent, and many dialects that have extended and evolved from it.

Lisp was invented as a language for expressing algorithms. Lisp has a very simple syntax and (in base form) comparatively few commands. Lisp syntax is notable (and sometimes criticized) for its common use of parentheses, in a nested manner, to specify the order of operations.

Lisp is often associated with artificial intelligence programming, since it was first specified by John McCarthy, the acknowledged father of AI, and was the favored language for many early AI applications. Many have questioned whether Lisp has any special usefulness to AI or not. Though it is hard to point to any specific reason why Lisp would be particularly suited to artificial intelligence, it does embody many aspects highly useful to knowledge management applications. It was these reasons that caused us to first

look at Lisp and its variants when we were contemplating language alternatives:

1. **Open** -- I have written often that knowledge management, given its nature grounded in the discovery of new facts and connections, is well suited to being treated under the open world assumption [4]. OWA is a logic that explicitly allows the addition of new assertions and connections, built upon a core of existing knowledge. In both optical and literal senses, the Lisp language is an open one that is consistent with OWA. The basis of Lisp has a "feel" much like RDF (Resource Description Framework), the prime language of OWA. RDF is built from simple statements, as is Lisp. New objects (data) can be added easily to both systems. Adding new predicates (functions) is similarly straightforward. Lisp's nested parentheses also recall Boolean logic, another simple means for logically combining relationships. As with semantic technologies, Lisp just seems "right" as a framework for addressing knowledge problems;
2. **Extensible** -- one of the manifestations of Lisp's openness is its extensiblity via macros. Macros are small specifications that enable new functions to be written. This extensibility means that a "bottom up" programming style can be employed [5] that allows the syntax to be expanded with new language to suit the problem, leading in more complete cases to entirely new -- and tailored to the problem -- domain-specific languages (DSLs). As an expression of Lisp's openness, this extensibility means that the language itself can morph and grow to address the knowledge problems at hand. And, as that knowledge continues to grow, as it will, so may the language by which to model and evaluate it;
3. **Efficient** -- Lisp, then, by definition, has a minimum of extraneous code functions and, after an (often steep) initial learning curve, rapid development of appropriate ones for the domain. Since anything Lisp can do to a data structure it can do to code, it is an efficient language. When developing code, Lisp provides a read-eval-print-loop (REPL) environment that allows developers to enter single expressions (s-expressions) and evaluate them at the command line, leading to faster and more efficient ways to add features, fix bugs, and test code. Accessing and managing data is similarly efficient, along with code and data maintenance. A related efficiency with Lisp is lazy evaluation, wherein only the given code or data expression at hand is evaluated as its values are needed, rather than building evaluation structures in advance of execution;
4. **Data-centric** -- the design aspect of Lisp that makes many of these advantages possible is its data-centric nature. This nature comes from Lisp's grounding in the lambda calculus and its representation of all code objects via an abstract syntax tree. These aspects allow data to be immutable, and for data to act as code, and code to act as data [6]. Thus, while we can name and reference data or functions as in any other language, they are both manipulated and usable in the same way. Since knowledge management is the combination of schema with instance data, Lisp (or other homoiconic languages) is perfectly suited; and,
5. **Malleable** -- a criticism of Lisp through the decades has been its proliferation of dialects, and lack of consistency between them. This is true, and Lisp is therefore not likely the best vehicle in its native form for interoperability. (It may also not be the best basis for large-scale, complicated applications with responsive user interfaces.) But for all of the reasons above, Lisp can be morphed into many forms, including the manipulation of syntax. In such malleable states, the dialect maintains its underlying Lisp advantages, but can be differently purposed for different uses. Such is the case with Clojure, discussed in the next section.

Reasons Specific to the Clojure Language

Clojure was invented by Rich Hickey, who knew explicitly what he wanted to accomplish leveraging Lisp for new, more contemporary uses [7]. (Though some in the Lisp community have bristled against the idea that dialects such as Common Lisp are not modern, the points below really make a different case.) Some of the design choices behind Clojure are unique and quite different from the Lisp legacy; others leverage and extend basic Lisp strengths. Thus, with Clojure, we can see both a better Lisp, at least for our stated context, and one designed for contemporary environments and circumstances.

Here are what I see as the unique advantages of Clojure, again in specific reference to the knowledge management context:

6. **Virtual machine design (interoperability)** -- the masterstroke in Clojure is to base it upon the Java Virtual Machine. All of Lisp's base functions were written to use the JVM. Three significant advantages accrue from this design. First, Clojure programs can be compiled as jar files and run interactively with any other Java program. In the context of knowledge management and semantic uses, fully 60% of existing applications can now interoperate with Clojure apps [8], an instant boon for leveraging many open source capabilities. Second, certain advantages from Java, including platform independence and the leverage of debugging and profiling tools, among others (see below), are gained. And, third, this same design approach has been applied to integrating with JavaScript (via ClojureScript) and the Common Language Runtime execution engine of Microsoft's .Net Framework (via Clojure CLR), both highly useful for Web purposes and as exemplars for other integrations;
7. **Scalable performance** -- by leveraging Java's multithreaded and concurrent design, plus a form of caching called memoization in conjunction with the lazy evaluation noted above, Clojure gains significant performance advantages and scalability. The immutability of Clojure data has minimal data access conflicts (it is thread-safe), further adding to performance advantages. We (SD) have yet to require such advantages, but it is a comfort to know that large-scale datasets and problems likely have a ready programmatic solution when using Clojure;
8. **More Lispiness** -- Clojure extends the basic treatment of Lisp's s-expressions to maps and vectors, basically making all core constructs into extensible abstractions; Clojure is explicit in how metadata and reifications can be applied using Lisp principles, really useful for semantic applications; and Clojure EDN (Extensible Data Notation) was developed as a Lisp extension to provide an analog to JSON for data specification and exchange using ClojureScript [9]. SD, in turn, has taken this approach and extended it to work with the OSF Web services [10]. These extensions go to the core of the Lisp mindset, again reaffirming the malleability and extensibility of Lisp;
9. **Process-oriented** -- many knowledge management tasks are themselves the results of processing pipelines, or are semi-automatic in nature and require interventions by users or subject matter experts in filtering or selecting results. Knowledge management tasks and the data pre-processing that precedes them often require step-wise processes or workflows. The immutability of data and functions in Lisp means that state is also immutable. Clojure takes advantage of these Lisp constructs to make explicit the treatment of time and state changes. Further, based on Hickey's background in scheduling systems, a construct of transducers [11] is being introduced in version 1.70. The design of these systems is powerful for defining and manipulating workflows and rule-based branching. Data migration and transformations benefit from this design; and
10. **Active and desirable** -- hey, developers want to work with this stuff and think it is fun. SD's clients and partners are also desirous of working with program languages attuned to knowledge

management problems, diverse native data, and workflows controlled and directed by knowledge workers themselves. These market interests are key affirmations that Clojure, or dialects similar to it, are the wave of the future for knowledge management programming. Combined with its active developer community, Clojure is our choice for KM applications for the foreseeable future.

Clojure for Knowledge Apps

I'm sure had this article been written from a developer's perspective, different emphases and different features would have arisen. There is no perfect programming language and, even if there were, its utility would vary over time. The appropriateness of program languages is a matter of context. In our context of knowledge management and artificial intelligence applications, Clojure is our due diligence choice from a business-level perspective.

There are alternatives to the points raised herein, like Scheme, Erlang or Haskell. Scala offers some of the same JVM benefits as noted. Further, tooling for Clojure is still limited (though growing), and it requires Java to run and develop. Even with extensions and DSLs, there is still the initial awkwardness of learning Lisp's mindset.

Yet, ultimately, the success of a programming language is based on its degree of use and longevity. We are already seeing very small code counts and productivity from our use of Clojure. We are pleased to see continued language dynamism from such developments as Transit [9] and transducers [11]. We think many problem areas in our space -- from data transformations and lifting, to ontology mapping, and then machine learning and AI and integrations with knowledge bases, all under the control of knowledge workers versus developers -- lend themselves to Clojure DSLs of various sorts. We have plans for these DSLs and look forward to contribute them to the community.

We are excited to now find an aesthetic programming fit with our efforts in knowledge management. We'd love to see Clojure become the go-to language for knowledge-based applications. We hope to work with many of you in helping to make this happen.

---

[1] I have also been involved with the development of two new languages, Promula and VIQ, and conducted due diligence on C#, Ruby and Python, but none of these languages were ultimately selected.

[2] Native apps on smartphones are likely going through the same transition.

[3] As of the date of this article, Clojure is in version 1.60.

[4] See M. K. Bergman, 2009. " The Open World Assumption: Elephant in the Room," December 21, 2009. The open world assumption (OWA) generally asserts that the lack of a given assertion or fact being available does not imply whether that possible assertion is true or false: it simply is not known. In other words, lack of knowledge does not imply falsity. Another way to say it is that everything is permitted until it is prohibited. OWA lends itself to incremental and incomplete approaches to various modeling problems.

OWA is a formal logic assumption that the truth-value of a statement is independent of whether or not it is known by any single observer or agent to be true. OWA is used in knowledge representation to codify the informal notion that in general no single agent or observer has complete knowledge, and therefore cannot make the closed world assumption. The OWA limits the kinds of inference and deductions an agent can make to those that follow from

statements that are known to the agent to be true. OWA is useful when we represent knowledge within a system as we discover it, and where we cannot guarantee that we have discovered or will discover complete information. In the OWA, statements about knowledge that are not included in or inferred from the knowledge explicitly recorded in the system may be considered unknown, rather than wrong or false. Semantic Web languages such as OWL make the open world assumption.

Also, you can search on OWA on this blog.

[5] Paul Graham, 1993. "Programming Bottom-Up," is a re-cap on Graham's blog related to some of his earlier writings on programming in Lisp. By "bottom up" Graham means ". . . changing the language to suit the problem. . . . Language and program evolve together."

[6] A really nice explanation of this approach is in James Donelan, 2013. "Code is Data, Data is Code," on the Mulesoft blog, September 26, 2013.

[7] Rich Hickey is a good public speaker. Two of his seminal videos related to Clojure are "Are We There Yet?" (2009) and "Simple Made Easy" (2011).

[8] My Sweet Tools listing of knowledge management software is dominated by Java, with about half of all apps in that language.

[9] See the Clojure EDN; also Transit, EDN's apparent successor.

[10] structEDN is a straightforward RDF serialization in EDN format.

[11] For transducers in Clojure version 1.70, see this Hickey talk, "Transducers" (2014).
