Dynamic Languages—ready for the next challenges, by design.

David Ascher, PhD, ActiveState[1]
Version 1
July 27, 2004

Abstract

Dynamic languages are high-level, dynamically typed open source languages. These languages, designed to solve the problems that programmers of all abilities face in building and integrating heterogeneous systems, have proven themselves both despite and thanks to their independence from corporate platform strategies, relying instead on grassroots development and support. Ideally suited to building loosely coupled systems that adapt to changing requirements, they form the foundation of myriad programming projects, from the birth of the web to tomorrow's challenges.

Introduction

There is a category of programming languages which share the properties of being high-level, dynamically typed and open source. These languages have been referred to in the past by some as "scripting languages,"[2] and by others as "general-purpose programming languages". Neither moniker accurately represents the true strengths of these languages. We propose the term *dynamic languages* as a compact term which evokes both the technical strengths of the languages and the social strengths of their communities of contributors and users.

While many of the arguments presented in John Ousterhout's landmark paper on scripting are as valid as when they were written, changes in the information technology landscape and maturation of thinking about open source lead us to reexamine his argument. This paper will argue that many of the pressures on software systems, such as the push for standards-compliant open systems and the competitive advantages granted to customizable systems[3], combined with a shift from CPU-bound systems to network-bound systems, have propelled dynamic languages into a new, critical role. In addition to their traditional role in support of scripting tasks, these programming languages have

---

[1] Corrections, additions and other feedback is encouraged at DavidA@ActiveState.com. The author gratefully acknowledges technical and editorial input from Matt Herdon, Kevin Altis, Greg Wilson, James McGill, Trent Mick, Jeff Hobbs, Cameron Laird, Eric Promislow, Craig Woods, Shantel Shave, and Emily Pickett.

[2] John K. Ousterhout, March 1998, "Scripting: Higher Level Programming for the 21st Century", *IEEE Computer*; also available at: www.tcl.tk/doc/scripting.html

[3] See Tim O'Reilly coverage of the topic of the customizability of open source in "The Open Source Paradigm Shift" (tim.oreilly.com/opensource/paradigmshift_0504.html)

demonstrated an unequaled ability to build a diverse set of important software systems.

We believe this shift in importance warrants replacing the term "scripting language" with one that better describes the languages' nature and impact, and suggest the use of the term *dynamic languages*. The choice of the word "dynamic" over "scripting" is a pragmatic one—the original term has tended to minimize the broad range of applicability of the languages in question. The new term reflects the belief that the real-world value of these languages derives more from their dynamics (technical and social) than their approachability.

In what follows, we present the essential characteristics of dynamic languages as they contrast with other language categories. Popular dynamic languages are briefly surveyed, followed by an analysis of their emergent properties in current technical, social, economic, and legal contexts. We suggest software environments where they are most and least appropriate. After discussing some popular beliefs about these languages, we explore the future of these languages , touching both on key upcoming challenges, as well as opportunities for growth.

Language Categories

Among the hundreds of programming languages available, a relatively small number are widely used. These can be grouped into a few broad categories. The categorization used in this paper is deliberately not based on strictly technical features of the languages, but instead on a combination of technical, social, business, and use-in-practice features.

Legacy languages

Legacy languages, such as Cobol, Fortran, and PL/I, are important because no matter how much one would like to at times, the past can't be wished away, especially in corporate IT systems. Few IT strategies can effectively accept a "closed world" hypothesis; hence, it is important when considering a new language to evaluate its ability to be bridged to preexisting systems.

System languages

System languages include C, C++, and, more recently, Java and C#. These languages are characterized by strong typing (as explained in Ousterhout (1998)), the ability to build tightly-coupled efficient systems, and, especially for Java and C#, a tight binding between the language and the underlying platforms (the Java Runtime Environment and .NET Common Language Runtime respectively). One consequence of the tight integration between the language and the platform is that situations which require breaking the "closed world" assumption can be problematic.

Proprietary languages

We use the term "proprietary languages" to refer to languages which share many technical features with dynamic languages, but which are owned, controlled, and evolved by corporations. The prototypical example is Visual Basic, which is high-level and adaptable for both scripting tasks and building applications, but whose evolution is driven directly by Microsoft's platform plans. For example, the evolution of Visual Basic from version 6 to Visual Basic .NET caused considerable frustration among its users, but makes sense from the Microsoft point of view because Microsoft believes that all of its users should move to using the .NET framework, something that required deep changes in VB6.

Dynamic languages

Described in detail in the next section, dynamic languages are defined as high-level, dynamically typed, and open source, developed by a grassroots community rather than a corporation or consortium.

Moderation In All Things

Before we discuss the strengths and weaknesses of dynamic languages, a note about the extent of the claims being made. The topic of programming language choice often leads to heated arguments where categorical positions are stated, often in the face of clear evidence that more moderate approaches may be more rational. A primary argument in this paper is that dynamic languages play an extremely effective and crucial role as part of an overall pragmatic programming language strategy. Some situations may be best served by a single-language approach, whether dynamic or not; however, many situations are best addressed with a combination of system, proprietary and dynamic languages, with connections to legacy systems. There is no silver bullet in the world of programming languages.

What is a Dynamic Language?

For the purposes of this paper, the term *dynamic languages* refers to high-level, grassroots, open source programming languages with dynamic typing, including but not limited to Perl, PHP, Python, Tcl, and Ruby. We will first cover each of the three definitional criteria (high-level, grassroots open source, and dynamic typing). We will then briefly introduce each of the currently popular languages. By focusing on the most popular languages, we'll be able to identify: 1) properties which emerge from combinations of the properties of the language and the network effect exhibited by all successful open source projects; and 2) the particular challenges of building contemporary software systems, taking

into account market, technical, and legal issues.

Criterion 1: High-level

The ever-increasing diversity of software systems has pushed programming language evolution along several dimensions which are generally referred to by a catch-all term: "high-level"[4]. This evolution is evident in: 1) a bias toward more abstract built-in data types, from associative arrays to URIs; 2) particular syntactic choices emphasizing readability, concision and other "soft" aspects of language design; 3) specific approaches to typing of variables, variously referred to as "loosely typed," "dynamically typed," or "weakly typed," in clear opposition to "static typing"; 4) automation of routine tasks such as memory management and exception handling; and finally 5) a tendency to favor interactive interpreter-based systems over machine-code-generating compiler models. Somewhat tied to each of these trends is the notion that, as computers become faster and humans have more to do in the same amount of time, newer programming languages should fit with human constraints rather than with computational ones. Thus, high-level languages aim to require less from the human programmer, and more from the computer. This leads, generally, to languages that are easier to use and slower to execute (naturally, there are exceptions to this generalization).

Criterion 2: Grassroots open source

The term term "open source" is used in at least three ways: The legal usage refers to open source software *licenses* which encourage code sharing; the methodological usage refers to a development model characterized by loose networks of self-organizing pools of volunteer developers; and the sociological usage refers to the communities which form around specific software projects, characterized by close relationships between users and developers.

Given the recent adoption of various open source licenses by traditionally proprietary software behemoths, it's worth noting that all of the successful dynamic languages to date are "old fashioned" open source, meaning that an individual released an early version of the language to "the net", attracted a following of users and contributors, and built a community of peers. While the licensing aspects of an open source project make no distinction between individual and corporate creators, the nature of the original creator (biological or corporate) has massive impact on the language's adoption and evolution , for legal as well as psychological reasons. Most likely, contributors to Perl, Python, etc. would have been neither as enthusiastic to help "pitch in," nor as quickly accepted as contributors, had the language creators been corporations rather than individuals.

On the other hand, it's also clear that corporations have learned how to run successful open-source

[4] Even system languages tend to become higher level over time, but as a whole, dynamic languages are higher level than system languages.

projects. The Eclipse IDE framework, originated at IBM, has been quite successful at gathering input from organizations, particularly educational institutions.

While each of the successful dynamic languages have chosen different specific licenses, it is far from accidental that none selected the more extreme GPL license used by the Linux kernel. All of the successful language communities have deliberately picked licenses that fit equally well with corporate requirements for non-viral licenses and the Free Software Foundation's goals (although clearly not the tactics, given the license differences). In general, the language communities view themselves as on the "liberal" side of the open source debate (inasmuch as any large group can be described as having a consistent opinion), and aren't compelled to pick sides on the morality of proprietary licenses. This approach has served them well, with significant successes both within the Linux and Windows communities.

Criterion 3: Dynamically typed

The strongest *technical* difference between dynamic languages and most of their competitors is that the typing systems (in layman's terms, the mechanisms by which programming languages refer to the *kinds* of objects being manipulated) are more dynamic than static. Being dynamic is an asset if one needs to be able to change quickly. Thus, being *dynamically typed* makes sense if the nature of objects being manipulated is either unknown or unpredictable. This tends to be the case in systems which: 1) are not precisely specified (the problems addressed aren't yet well understood); 2) are evolving fast (due to changing standards or changes of opinion); or 3) need to interact with other systems which change unpredictably (for example, third party web applications). In addition to dynamic typing, dynamic languages often build in other dynamic behaviors, such as loading arbitrary code at runtime, runtime code evaluation, and more.

Popular Dynamic Languages

While the preceding three criteria are useful in understanding what we define as a dynamic language, what's important is not their intrinsic features so much as their extrinsic behaviors in the broader information technology ecosystem. There, it's important to consider separately those dynamic languages that have proven to be widely adopted.

Hundreds of programming languages exist, with dozens of new ones developed every year. Our focus is on the impact, successes, and future of programming languages from a pragmatic, market-reflecting point of view, rather than a more academic "state of the art" perspective. Therefore, we look at languages that have achieved a certain degree of popular success, rather than languages that, although technically significant, have had less influence on the market.

Perl

Perl is often referred to as "the duct tape of the Internet." It arose from the need to extend the capabilities of Unix command-line tools into a more general-purpose programming system. Perl's strength at processing text and its accessibility to a broad range of users led to its massive success concurrent with the growth of the web. Its affinity for processing text files has meant both that it is it used in many such situations, and that a multitude of popular tools are built in Perl. Thus, it is a language that many IT managers can safely assume their staff know. In addition to being used for daily sysadmin or "glue" tasks, Perl has been successfully used in a tremendous variety of larger systems, from enterprise-class mail processing to world-class websites such as Amazon.com.

Python

Python, of the same generation as Perl, embodies a preference for clean design and clarity over concision. Akin to a dynamic, less verbose version of Java, Python has found particular affinity with seasoned programmers who are looking for rapid ways of building flexible systems. As such, Python is often used in prototyping contexts such as scientific computation and GUI application design, as well as in high-performance systems. Two notable, recent Python-powered successes include the BitTorrent peer-to-peer system, with over 1.5 million downloads *per month*, and the SpamBayes Bayesian anti-spam classifier, which delivers world-class results using advanced mathematics. In both cases, a key benefit of Python lies in its ability to "stay out of the way" of the programmers implementing sophisticated algorithms.

PHP

PHP, unlike Perl and Python which were very broad in scope, focused from its inception on a single task: building dynamic websites. It's safe to say that it has succeeded, with the latest Netcraft surveys finding PHP installed on over 16 million domains. PHP combines a syntax that is easy for even novice web designers to learn, with a rich library of modules that capitalize on the fact that most websites need to do similar things (talk to databases, cache images, process forms, etc.). PHP is now considered the most serious competition to the web strategies of both Microsoft (with ASP.NET) and Sun (with J2EE).

Tcl

Tcl (short for Tool Command Language), designed with application integration in mind, has found applicability across a wide variety of platforms and application domains. It has been particularly successful at GUI applications (through its Tk toolkit), automation in general, and test automation in

particular. Its small code size has led to it being deployed in a variety of embedded contexts; for example, Tcl is part of Cisco's IOS router operating system, and as such is included in all high-end Cisco routers and switches. A different, but equally important, example of Tcl usage is AOLserver, America Online's web server—yet another example where a scripting language runs some of the largest and busiest production environments in the world.

JavaScript/JScript/ECMAScript

The language that is technically referred to as ECMAScript, but more commonly known by the name of its Netscape-authored implementation, JavaScript, deserves special mention at this point. It certainly qualifies as a dynamically-typed language, is quite high-level, and has at least two open source implementations. Exceedingly popular, it is supported by all major web browsers, and, as a result, is part of a huge number of websites. Significant applications have been built using it, especially on the client-side of the web transaction, such as webmail interfaces and blogging tools. It is worth noting, however, that JavaScript is unlike the languages previously mentioned in two significant respects. First, because it was defined as the language of the browser, it had to combine strict security requirements (e.g. a JavaScript program can't, as a rule, access files on disk) with odd user interface challenges. For example, it is "better" for a JavaScript program to fail quietly in the case of a programming error, and this behavior can make it a significant challenge to build large systems in JavaScript. Furthermore, and more critically, JavaScript has suffered from *too much* corporate interest. The design of the language was one of the battlefields between Microsoft and Netscape, and it can be argued that the resulting language is a casualty of war. Even with open-source implementations, the language did *not* evolve according to normal open source mechanisms; instead, evolution was governed by the politics of the ECMA standards process, under considerable pressures from both major browser vendors. As a result, JavaScript is effectively unchanging (a polite word for 'dead'), and web designers are pondering moves to other technologies such as Macromedia's Flash, Microsoft's proposed XAML, or Mozilla's XUL frameworks.

Ruby, Groovy, Prothon, others

The languages described above are simply the most popular today. Depending on when you read this, their relative popularity may have shifted due to evolution of the languages, the market requirements, fashion-like "buzz", importance of various platforms, etc. New languages are also sure to emerge. Some notable "up-and-coming" languages include Ruby, which provides a blend of Perl and Python-inspired features; Prothon, which aims to be a "better" version of Python; and Groovy, still under specification, which aims to be the standard dynamic language on top of the Java platform. It's much too early to tell whether any or all of these languages will achieve the success of Perl.

What's reassuring is that, because of the market dynamics at play, the winners will win because they are better at doing something that many people value.

Properties of Dynamic Languages

More important than the differences among the languages noted above are their commonalities.

Technical purity

Dynamic languages were designed to solve the technical problems faced by their inventors, not to address specific goals identified as part of a 'strategic plan' to influence buyers of IT solutions. As such, they have a "pure" focus on solving technical problems, with no agenda to push a particular platform, operating system, security model, or other piece of the IT stack (this focus is true of most successful grassroots open source projects). The value of technical purity is most notable in comparison to competing proprietary languages where it is clearly not exhibited, viz. Visual Basic's recent changes. Note that technical purity should not be confused with a more academic notion of purity. The successful dynamic languages all embrace the pragmatic constraints of the real-world, such as integration with 'foreign' systems and backwards-compatibility, even though those constraints often make the technical details much "messier." The crux is that they are pure in *intent*, in that they do not serve a non-technical agenda.

Optimizing person-time, not computer-time

The driving forces for the creation of each major dynamic language centered on making tasks easier for *people*, with raw computer performance a secondary concern. As the language implementations have matured, they have enabled programmers to build very efficient software, but that was never their primary focus. Getting the job done fast is typically prioritized above getting the job done so that it runs faster. This approach makes sense when one considers that many programs are run only periodically, and take effectively no time to execute, but can take days, weeks, or months to write. When considering networked applications, where network latency or database accesses tend to be the bottlenecks, the folly of hyper-optimizing the execution time of the wrong parts of the program is even clearer. A notable consequence of this difference in priority is seen in the different types of competition among languages. While system languages compete like CPU manufacturers on performance measured by numeric benchmarks such as LINPACK, dynamic languages compete, less formally, on productivity arguments and, through an indirect measure of productivity, on how "fun" a language is. It is apparently widely believed that fun languages correspond to more productive programmers—a hypothesis that would be interesting to test.

Open source, deeply

All of the successful dynamic languages have, according to our definition, a primary implementation which is open source. This simple fact has in practice meant that the open source implementations have been the *de facto* language definition. An important consequence of the open source nature of the primary implementation has been that modifications to the language by third parties have been easier to adopt into the mainstream code base than if any kind of contractual relationship had been necessary. Any engineer anywhere can "tweak" the language to his or her heart's content, without having to ask anyone for permission. This ease with which experiments can be performed by anyone is without equal. Maintaining any significant modifications in the face of a language under constant change is a maintenance burden, and it is widely understood that it's best to contribute modifications back to the main code base. The resulting phenomenon of naturally aggregating improvements from "anybody" is (relatively) unencumbered by bureaucracy, either of the nondisclosure-agreement-signing kind or of the standards-body kind. While a challenge for organizations that require the use of standards-based technologies, this has allowed the languages to evolve quickly, and to incorporate feedback from stakeholders of all sizes[5]. It is worth noting that academics (university students in particular) have been able to convert ideas into implementations with remarkable efficiency through open source, a process that tends to be quicker than either academic publication or the traditional industrial model of getting the idea reified in a product.

One of the ways in which dynamic languages are *deeply* open source is the almost total transparency about how the languages are evolved. The bug lists and patch review processes are public, and most conversations about the evolution of the language occur on public mailing lists, subject to the scrutiny of all. There is no hiding behind firewalls or membership in an organization.

Evolution by meritocracy and natural selection

Dynamic languages evolve along two, often orthogonal, directions. The core of the language is often controlled by a tight-knit, extremely competent set of individuals who are in charge of the language's basic tenets. These teams are meritocratic rather than democratic, and consider user-suggested changes only inasmuch as they don't present a deviation from the aesthetic or philosophical principles of the language design. It is through this rather autocratic process that the languages have managed to remain "true to their core" over 10+ years of evolution. In contrast, the *capabilities* of the languages (rather than their *style*) has most effectively grown through extensions, libraries, and modules. In that area, individual contributions are equally valued, and frenetic market competition rewards authors of important and useful modules, giving massive feedback to all

[5] To some degree, the Java Community Process is an attempt by a corporation to replicate this successful part of the open source development model.

contributors. Programming languages are unique among open source projects in that the gap between users and authors is minute; users can give valuable design feedback because, just like the library creators, they write software and understand the perspective of the software designer.

It is through that bazaar of module distribution that practical usefulness emerges, because for a dynamic language to support a new technology, the language itself rarely needs to change; all that's needed is someone to write a special-purpose module. Thus, as soon as a library, file format, or Internet protocol becomes "useful enough," the language communities build language-specific modules to support it. It is the ability of open source communities to distribute the workload to those who first feel the need to scratch a particular itch, that makes them able to compete effectively against multi-million dollar efforts. It is worth noting that dynamic typing is an important edge in the race to "embrace and extend" new technologies; for example, if a web application changes the schema of the data being transmitted, clients written in a dynamically typed language will require fewer changes than their statically typed counterparts, all other things being equal.

Platform neutrality

Dynamic languages have naturally been platform-neutral. Building a programming language that is limited to a particular platform is anathema to *language* designers; the language designer tends to believe that "everyone" would be better off by using their language, or if not everyone, then at least everyone trying to solve a particular kind of problem with a particular background. These goals tend to define target audiences which span all platforms (e.g. the programming challenges of web designers should be mostly independent of the underlying platform). While all of the popular dynamic languages were born with individual platform "biases," they also embrace the notion that they should work as well as possible on all platforms. Over time, each language evolves to fit an ever-increasing set of target operating systems, naturally covering Linux and other Unix variants, and various Windows platforms, but also reaching into more esoteric platforms like mainframes, supercomputers, phones, and various embedded devices. Operating systems and platforms, more generally, are seen as "just another context to operate in." Whether or not each use case is supported is simply a matter of perceived need and volunteer time.

This approach has both negative and positive consequences. Dynamic languages cannot fit frameworks such as .NET or the JVM as well as languages explicitly designed to fit them. On the other hand, dynamic language communities are free from the need to restrict themselves to specific platforms definitions, and have tended to embrace a wider variety of platforms. It's important to note that platform neutrality doesn't mean "cross-platform at any cost", where a feature must be available on all platforms before it is available on any. Instead, platform-specific language extensions (typically through libraries or modules) are developed by users who have needs for particular platform support.

It is thus possible to write cross-platform programs using dynamic languages (most are), and it is equally possible to write programs which fully exploit platform-specific features.

Languages you can build a plan on because users determine the language plan

Since the effort required for maintenance of the language is borne by the users of the language, the decision to end support for a platform is closely tied to the disappearance of users of that language on that platform. Business drivers which accelerate unnecessary changes, such as the idea of forcing customer upgrades because of a requirement for revenue, don't exist for technologies such as dynamic languages which are volunteer-driven and free. Importantly, corporate users of dynamic languages who have investments in particular ports find it relatively cheap to maintain these ports either directly or through funding vehicles such as specialized vendors.

When to use Dynamic Languages

We've mentioned some successful deployments of dynamic languages and some of their observable properties. We've stated that system languages are also important pieces of the IT puzzle. When should one consider using a dynamic language ? There are, naturally, no simple answers that cover every possible scenario. In particular, any policy that prescribes a particular language is incompatible with a value-based approach to language choice. If we assume an environment in which language choice is possible, however, some areas have shown to be ideal for the use of dynamic languages.

Scripting tasks

Scripting is certainly an arena where dynamic languages are without equal. Whether the specific task involves simple text processing, database exploration, or gluing together existing tools, scripting languages have the right blend of ease-of-use, rapid development support, and rich interfaces to support these scenarios.

Prototyping

A different use is the construction of complex systems, especially if the requirements aren't well-specified ahead of time. The domain can vary considerably, from process automation to scientific research to GUI development. If the programmer isn't sure at the onset of the project how the final application will look or act, then the rapid development capability of dynamic languages leads to higher productivity and better end-point quality. If it's "cheap and easy" to correct a mistake, you correct it more often, leading to shorter projects or better software (or both). Furthermore,

experienced users of dynamic languages tend to take on more ambitious projects, because the cost of failure is lower. The rapid edit-compile-run-test cycle exhibited by dynamic languages has made them favorites of agile development methods, which favor iterative approaches over top-down models.

Ideally suited for loose coupling

As argued by many observers[6], always-on networks, mobile devices and open networking protocols allow for a radically new way of building software systems, focused less on the PC and more on the power of coordinating and aggregating network resources, e.g. through the use of web services[7]. In this new model, deeply integrated platforms are not as valuable as components (using the broadest definition) accessible through open interfaces.

Steering computational tasks

The scientific computing area, known for its obsessive pursuit of optimization, outrageous supercomputing facilities and need for always-increasing computing power, may seem odd to associate with dynamic languages. Indeed, most serious scientific computations are done using system or legacy languages that have benefited from decades of optimization work. However, those optimizations are typically restricted to specific computations (linear algebra, numeric optimization, etc.). In many cases, the "scientific" part of scientific computation involves a great deal of what other disciplines would call prototyping—trying out an idea to "see if it works." This experimentation needs to be done by computer scientists in collaboration with non-computational scientists, the specialists in physics, biology, chemistry, climatology, or other disciplines who drive the science behind the computations. Given this environment, it is not surprising to learn that dynamic languages are routinely used as part of a holistic approach to scientific computation, in which computational scientists build flexible systems that are then easily scripted by domain experts.

Business logic

The distinction between variable, high-level domain-specific choices and optimized, constant building blocks occurs in every application domain. For example, in many corporate applications a distinction is made between "business logic" (e.g. what data should be collected from the user, what kinds of reports should be generated), and the "back-end code," (e.g. database or network calls,

---

[6] See, for example, David Stutz' reference to "the loosely coupled mindset that today's leading edge developers apply to work and play." (www.synthesist.net/writing/onleavingms.html)

[7] Jon Udell, Adam Bosworth, and others have written about the benefit of loose coupling web-services architectures; see e.g. www.infoworld.com/articles/fe/xml/02/06/10/020610feappdevtci.xml

communication with other subsystems). In this regard, dynamic languages share the same benefits as languages such as Visual Basic: rapid development, easily learned by occasional programmers, well suited for end-user scriptability, and forgiving to programming errors.

Advanced technologies

Because of the language design aspects that strive to minimize the human effort required to express computational ideas in code, dynamic languages are deeply appreciated by people writing complex or sophisticated systems, be they nuclear scientists, network engineers, or web architects. System languages tend to require more discipline of the "bookkeeping" variety than do dynamic languages, be that through requirements for explicit type annotation, explicit memory management, interface definitions prior to implementation, etc. While useful in systems that require specific guarantees of robustness, the scaffolding needed by those language approaches can get in the way of seeing the sculpture as a whole.

A widely-held (but hard to test) belief is that the rate of coding errors per line of code is roughly independent of programming language, regardless of the level of the language[8]. Casual inspection of high-level language code contrasted with equivalent systems code will show that dynamic languages are more concise. A given task requires fewer lines of code to execute in a high-level language than in a lower-level language,[9] and thus should have fewer errors. In addition, this suggests that high-level languages make it easier for a programmer to keep a larger part of their program in working memory. Given this, the success of dynamic languages in the scientific and engineering communities at large is not surprising; those kinds of users need to focus on the complexities of the business logic, and worrying about the details of the optimized memory pools is detrimental to getting the important work done.

When Not to Use Dynamic Languages

Only the most zealous advocates of dynamic languages will recommend their use in all situations. There are software contexts that seem plainly inappropriate for their use.

Some high-performance tasks

While we've argued that dynamic languages can be used to build high performance systems, even those applications rely on code written in more static languages to do key parts of the work. Several

[8] A playful analogy would be to suggest that the rate of typos a person produces depends on the skills of the individual writer rather than the characteristics of the natural language (i.e. French vs. English) used.

[9] There are references for this particular claim, such as An Empirical Comparison of Seven Programming Languages, IEEE Computer, October 2000.

kinds of tasks, such as some numeric computations, machine code generation, or low-level hardware interfacing, are best done in programming languages where the concepts being manipulated (be they numbers, bytes, pixels, or memory addresses) are expressed in a language optimized for them. In most of these cases, there is no ambiguity about the requirements—a mathematical routine should do the same operation as it did when it was first invented—or the types of objects manipulated. It *makes sense* to use a language like Fortran or C++ (which benefit from decades of optimization research) to implement it. There is a diverse set of such tasks where performance is the overwhelming concern, and where dynamic languages would result in unacceptable results. In many cases, combining a "steering layer" written in a dynamic language and optimized components written in other languages can lead to a system with the flexibility of a dynamic language approach and the effective performance of a low-level language.

Small memory systems

Dynamic languages, because they are high-level and interpreted, require more machinery to execute than either lower-level languages or languages that get compiled to machine code or equivalent. Thus, generally speaking, they are inappropriate choices for very small memory systems. It is interesting to consider, however, that what were once considered very small memory systems, such as phones and TV set-top boxes, are now laden with enough memory to run much larger applications.

Myths about dynamic languages

Given that much of what is said above is "old news," one must wonder why dynamic languages haven't garnered more visibility among the mainstream, especially in the media and corporate boardrooms. In addition to the commercial forces at play (some of the competing languages are actively promoted by marketing organizations with advertising budgets and PR firms), and acknowledging that the technical communities at the core of these open source languages tend to do a poor job of presenting their ideas to non-technical audiences, it must be noted that part of the problem has been a lack of challenge to persistent myths or misconceptions surrounding dynamic languages. We examine some of these critically.

Myth: "You can't build real applications with scripting languages"

While Ousterhout (1998) should be credited for widely publicizing the strengths and value of languages such as Perl, Tcl, Python etc., some believe that by adopting the moniker "scripting language," he unwittingly facilitated the propagation of one of the biggest criticisms of these

languages—that they are only useful for small, simple, automation tasks, and shouldn't be considered for the serious programming challenges that professional programmers routinely face. While rigorous objective analyses on the topic are hard to find, there is an abundance of anecdotal evidence suggesting that professional programmers can, and have, built world-class systems using these languages.

The world-wide web, arguably the most successful IT project of the last decade, is substantially based on dynamic languages. At every stage of the web's growth, from homegrown "home pages" (which were most often powered by Perl) to today's mission-critical websites (a large percentage of which are written in PHP and Perl), dynamic languages have been critical components of identifying new challenges, prototyping architectures, and building scalable, robust systems. It could be claimed that, without high-level languages, a project with the combined risk and size of Yahoo! would never have been started, let alone completed. Web applications of all kinds, such as the Mailman list management software, the Bugzilla bug tracking system, the Typepad/Moveable Type blogging system, or the Gallery photo archival system, are all powered by dynamic languages. Google uses Python in a variety of systems. The social software site Friendster.com recently shifted from a JSP architecture to one based on PHP, specifically to address performance problems.

Myth: "Dynamic languages are brittle"

Dynamic typing has in practice meant that the compiler is unable to make strong statements about the types of objects at compile time[10]. This does mean that exceptional conditions may occur at runtime. Instead of looking on this as a critical weakness of dynamic languages, it can be argued that this has led to systems which are more robust to runtime failures than statically-typed counterparts. Because runtime failures happen more often, defensive mechanisms have been built to deal with them, and the overall system is more stable. As a result, applications written in dynamic languages tend to fail more gracefully than those written in lower-level languages. For example, writing code that robustly deals with possible network outages is orders of magnitude easier with a dynamic language than with a language such as C. This ability to effectively deal with exceptional situations will only become more important as systems become more interconnected.

Myth: "You can't build large systems with dynamic languages"

The above sentence is usually followed by an argument as to why tight coupling, strong typing, and strict interface checking are key to building large systems. Smalltalk experts, who have been

[10] Advanced techniques such as whole-program analysis and type inference offer the promise of removing this restriction, but those approaches have not yet been successfully used in real-world applications of dynamic languages.

building large systems for decades, probably chuckle at that argument more than any others. Building large systems *does* present different challenges than building smaller systems. The importance of infrastructure components such as error handling, logging, and performance monitoring are key, as are design-time concerns such as architectural soundness and scalability planning, and development-time issues such as multi-tiered testing strategies, iterative development, proper planning and documentation, and so on. These challenges are orthogonal to the language choice, and certainly quite large systems have been built with dynamic languages.

Myth: "There's no innovation in open source"

This myth has received airtime recently with executives from some proprietary software vendors accusing the open source community of producing clones rather than building innovative software. We'll leave it to others to defend the work done in the domains of operating systems or productivity applications. The argument that open source produces no innovative work certainly doesn't hold much water when it comes to programming languages. Not only have programming languages typically come out of academic research efforts (which are effectively open source), but open source language designers have continued to innovate, even though that innovation has occured through different mechanisms than those of proprietary languages.

Unlike languages such as Java and C#, which are the focus of serious, goal-directed, funded research efforts, dynamic languages evolve in a more spontaneous (but not necessarily worse) way. Academics worldwide find it easy to get the implementations, understand them (with direct help from the maintainers), experiment with changes, and argue for language changes. Much academic language research therefore looks at the dynamic languages as a fertile ground on which to develop next generation approaches.

In general, the spirit of cooperation that pervades open source makes for rapid experiments and rapid implementations. Examples include the Stackless implementation of Python, which is proving to be exceptionally useful in some high-performance contexts; Tcl's virtual filesystem, which is still unique in the flexibility it offers developers looking to distribute their applications effortlessly; and the Perl 6 effort, which is the focus for considerable design and engineering work toward building a fast register-based virtual machine with unparalleled flexibility. Perl is an interesting project to contrast with proprietary languages. Technically speaking, the shift from Perl 5 to Perl 6 is probably as significant as the shift from Visual Basic 6 to Visual Basic .NET. Indeed, the architects of Perl 6 don't expect it to be backwards-compatible with Perl 5 (just like VB.NET isn't backwards-compatible with VB6). However, unlike VB, there is every reason to believe that the Perl 5 language will continue to be developed, supported, documented, and used for years. The investment in Perl 5 *by the community* will ensure its long-term health, as no one has a strong commercial interest in "forcing

upgrades."

Myth: Dynamic languages aren't well supported

This myth has been fading in recent years as the benefits of open source support systems have become more well known, thanks to the success of Linux. The dynamic language communities have organized a variety of support mechanisms, from professional trainers to peer-support online discussion groups to vendors offering enterprise support contracts, to contractors able to modify the languages to fit particular customer needs, and, in some cases, shepherd the changes back into the core language distribution.

A related point is the availability of books and other teaching or reference resources. Book publishers compete fiercely for shelf-space to cover dynamic languages. It is rare *not* to see books on dynamic languages among the top-sellers in the Programming category on sites like Amazon.com.

Myth: "Dynamic languages don't have good tools"

This myth deserves two answers. The first is that there *are* tools for dynamic languages, but the providers of these tools are either not commercial vendors (open source projects tend to spawn complementary open source projects) or they are not the same tool vendors that target proprietary or systems languages. ActiveState (the author's employer) has been vigorously competing in the dynamic languages tools market for seven years, along with many others. The tools can be found if you look for them, and some equal or exceed the quality and features of large commercial vendor tools. The second answer is that the tools for dynamic languages aren't the same as the tools for systems languages. If you define a tool as a piece of software that helps you build a system better or faster, then the diversity of software available on the Internet targeted at dynamic language programmers is awe-inspiring (browse through search.cpan.org for a Perl-centric example). Because of the positive feedback cycle evident among dynamic language programmers, there are thousands of libraries, modules, packages, and frameworks available for use, most under open source licenses. When the open source community doesn't provide the answer to a commonly felt pain among dynamic language programmers, companies such as ActiveState, NuSphere, Wing, Zend, and others jump in with commercial offerings.

Myth: Dynamic languages don't fit with .NET, Java, System X

Interoperability is a natural consequence of the decentralized development model of dynamic languages. All of the major languages have interfaces to well-established frameworks, be they COM, CORBA, etc. More recent platforms haven't been ignored either; there are successful ports of the

dynamic languages to Java (Jacl and Jython in particular) and interesting projects and products targeting the .NET platform (IronPython, PerlNET). History seems to argue that as soon as a real and well-defined need is articulated, it's simply a matter of time before the right talent emerges from the volunteer community (usually without forewarning) to lead the effort to meet that need.

What about Java?

Java, especially when seen as "organized opposition to Microsoft," is interesting to contrast with the dynamic languages, since a superficial look at the language could lead one to group them together. There are technical and non-technical reasons why Java isn't considered a dynamic language.

Statically typed and security focused

Foremost, Java is statically typed. A Java programmer needs to specify the type of each variable, as well as the particular interfaces each class implements; any deviation from these declarations causes (intended!) syntax errors or compilation failures. The choice of static typing in Java wasn't done for arbitrary reasons, naturally—it was done because it is far easier to optimize programs for which type information is know and guaranteed by the system. Additionally, it is far easier to make security guarantees about statically-typed languages, and one will recall that the need for "verifiable" code is at the foundation of both the Java Virtual Machine architecture and the .NET Common Language Runtime. It *is* possible to implement dynamically typed languages on top of such systems (Jython is a Python implementation for Java, and Groovy is a new dynamically-typed language for Java), but Java-the-language is far from that.

Not as loose

Java's design makes it easy to build highly integrated Java applications, and harder to build interfaces between Java systems and non-Java systems. This is somewhat related to a feeling that there is a "Java way" of doing any given task. Contrasting that with the dynamic language model where there are multiple ways to do any one thing, one understands why the Java approach is simpler to manage and also possibly blinkered—changing the officially supported way of doing any one thing becomes a significant effort—in the dynamic languages world, for better of worse, it happens (or doesn't!) as part of a brutal natural selection process.

There are more subjective difference between the dynamics of the Java community and those of the dynamic language communities. Dynamic language communities are *looser* than the Java community. This is true at many levels—the definition of "community member" is fuzzier in the dynamic

language world. There is no equivalent to the formalized Java Community Process (JCP), which, while designed to be inclusive, effectively raises the bar compared to the informal models used by the grassroots open source communities[11]. While the JCP is more broadly accepting of organizations than some other standards-defining bodies, it still requires a financial commitment from companies, effectively filtering out many possible contributors. This may be by design (e.g. to ensure "committed" contributors). Regardless, it does narrow the scope of the self-defined community.

Challenges for Dynamic Languages

As discussed earlier, dynamic languages are not appropriate in all contexts and their future success is not necessarily guaranteed. It is worth asking whether the organizational behaviors that have spawned them are appropriate for long-term success, both individually and as a category.

Lack of strategic vision

To date, dynamic languages have not been driven by strategic plans. In fact, most successful open source projects (Mono and Gnome being notable exceptions) have enjoyed success in spite of a lack of a long-term plan, let alone a clearly defined vision. The pragmatic, tactical approach to fix what's broken today as opposed to anticipate the problems of tomorrow, has, when combined with the selection processes inherent in the open source ecosystem, led to a survival of the fittest for *today's* problems, rather than rewarding those with the most compelling vision for *future success*. It's worth asking if the lack of a plan is guaranteed to be a winning approach in the long term.

A good example to highlight here is the different approaches toward newer standards such as SOAP, evident in dynamic languages vs. Java and C#, for example. The dynamic language communities are generally content with letting "someone else" worry about the standards-definition process, and are confident that they'll be able to support them when they are defined and stable. In contrast, Microsoft and Sun have committed significant resources to *defining* the standard, for clearly competitive, non-altruistic reasons. It is reasonable to expect that the resulting standards have been more influenced by how well they fit with those languages than with languages that got involved late in the standard-definition process. In this case, the combination of strategic planning and the resources of large corporations clearly resulted in shifts in the standard toward more strongly-typed languages. An interesting counter-spin is that dynamic language enthusiasts tend to prefer a

[11] It is interesting to note, for example, that Perl, Python, and Tcl have all picked an "implementation-driven" IETF-style model for controlling language enhancements, rather than a "spec-first" W3C-style model. The focus on "rough consensus and working code" tends to trump futuristic perspectives in the dynamic language evolution game.

different approach to web services over SOAP (namely REpresentational State Transfer, known as REST), which (they claim) is simpler, more pragmatic, portable, robust, and less resource-intensive.

No real/formal budget

Given the importance of programming languages in shaping IT, and the effective success of dynamic languages, it is stunning to realize that these languages have effectively succeeded with *no budget*. Certainly real value is invested, through sponsored work by individual companies, to some degree through the various organizations that support the languages, and predominantly through the volunteer labor that goes in on a daily basis. The fact remains, however, that there is no budget either for significant marketing activities, or, more problematically, to engage in long-term technical projects.

If one guesses the budget supporting either the language-related aspects of the .NET framework project at Microsoft or the Java-related projects at Sun and IBM, and compares it with the "sweat and tears" budget of developer groups in the dynamic language communities, it's hard to bet on the "little guy." However, even greater inequalities have existed in the operating system or database sectors, where the open source alternatives have made tremendous strides, showing that traditional budgeting and investment models should not be applied blindly to open source efforts.

One important advantage that open source can bring to bear in such competitive battles is that its costs of failure and limits on success are negligible. If an effort to re-engineer the Python virtual machine fails, all that's lost is volunteer time. This makes it possible to entertain doing several such experiments simultaneously, and pick the winner. Similarly, there are no limits on success—there is no "cost of sales" to worry about with open source success stories, nor are there support costs. The dynamics of open source success tend to scale the pool of talented contributors and the support bandwidth along with the success. Still, ask any dynamic language lead if he could use two man-years of dedicated work on the language and the answer will always be yes.

Lack of a marketing department

Budgetary constraints aside, it is worth noting that the market influence that dynamic languages have had is the accomplishment of a wide pool of people with quite narrow technical skills. While a few programmers can also turn a good phrase or design a nice logo, it's fair to say that there is no marketing department with the coordination, plan-based activities, and, again, budget with which to influence decision makers. Clearly the reward mechanisms which have led to a growing pool of technical talent in each language community have not led to a sizable pool of marketing talent. Technologists are their own worst enemies in cases such as these—they believe that the better

technologies will "win", in the face of centuries of data showing that sometimes it's the technology with the better ads that wins. While dynamic languages will undoubtedly survive without marketing, it is interesting to contemplate how different the software world would be in the absence of marketing (or, failing that, with less asymmetric promotion).

Legal stability / Patent threats

One of the most vague but real threats to open source in general is the unequal position of open source communities in the face of legally savvy corporate opposition. Specifically, the risk of patent and other intellectual property attacks against open source projects is worth considering seriously. The current state of software patents (especially in the United States) makes it disproportionately easy for larger corporations to claim (and receive) software patents for inventions that can be independently developed by open source developers. Volunteer developers, as a rule, have no direct economic interest in developing the software, hence no interest in acquiring such patents (even if the cost weren't prohibitive for most individuals). It is possible for patent-holding corporations to bring suits against commercial distributors of dynamic languages, commercial users of dynamic languages and, least likely but most threatening, against the individual contributors to those languages. The asymmetry evident in the relative legal arsenals on both sides of that divide is worrisome[12].

FORECAST

The history of dynamic languages is a source of inspiration for the future of dynamic languages. A look at the last 15 years and the impact of dynamic languages on other languages suggests a few trends.

Embracing new development methodologies

Dynamic languages tend to be adopted by programmers who are resistant to "following the pack" (especially if they detect a marketing-driven impetus behind the pack motion). It is therefore not surprising to note that there is significant overlap between dynamic language enthusiasts and proponents of novel development methodologies: many of the Agile methods are routinely adopted and defined by people working in languages such as Smalltalk, Ruby, Perl, Python (and, it must be said, Java). Many of the artifacts and scaffolding systems required by methodologies such as Extreme Programming are often written in a dynamic language, even if the main code base is not. This makes sense: it's an excellent application domain for these languages, where performance matters less than ease or speed of development and maintenance.

[12] As discussed by David Stutz in a trip report on the MIT/Sloan Free/Open Source Software Conference: www.synthesist.net/writing/osspatents.html

Life on the edge

Dynamic languages were created to address computing needs that mainstream languages ignored or couldn't address effectively due to their design limitations. Thus the need to process text to respond to networking requests (as in the CGI protocol, the foundation of the dynamic web) led to the success of languages such as Perl. The increase in the capabilities of routers and switches has provided fertile ground for Tcl in the 21st century. The explosion of database-backed websites developed and maintained by non-engineers led to the sustained explosion of PHP use worldwide. The latest twist on the web, blogging, is powered at least as much by dynamic languages as by more traditional languages. The need for rapid development on more powerful mobile platforms is an interesting avenue of growth for Python. In each of these cases, adventurous people exploring new technologies have used the strengths of dynamic languages to let them build systems that, in later generations, become more well-specified, and hence more appropriate for reimplementation in system languages. The dynamic languages' affinity for loosely-defined, rapidly changing requirements is evident in their past, and one can expect it to be advantageous in the future. To put it simply, the ease with which people can "hack something up" with dynamic languages makes them ideal for the frontier, wherever it is at any given time.

New languages

It is equally clear from studying the past that no specific language has a good reason to expect to be the dominant language in the future. Even within the dynamic language category, popularity has shifted from one to the other as a function of time, language evolution, and primarily, different use cases. The lack of a commercial outlook means that dynamic languages do fairly little to actively bind their users to long-term commitments—a consequence of which is that users of a dynamic language routinely learn new languages, and, over the course of a career, build expertise in several languages.

New features

Programming languages evolve under various pressures: bug fixing, the wishes of users (which often are simply asking for "feature matching" from other languages), and the more intellectual pressures of language designers, who look for new architectures, or new syntactic or semantic approaches, to increase either the breadth of the language (i.e. support new machine architectures) or its suitability for particular tasks. There are opposing pressures, as noted above, such as requirements for backwards compatibility, which grow in importance with mainstream adoption (mainstream users tend to be much more conservative than early adopters). Each of the major dynamic languages has undergone massive revisions in the last decade, leading to much more full-featured languages, while growing the user base consistently.

New economic and legal model

As has been argued elsewhere[13], the existence of open source implementations of a technology encourages the commodification of that technology. While this phenomenon has been widely noted in the operating systems, web server, and database markets, few analysts have noted that there have been no serious efforts at defining new strictly proprietary languages—Java is following a modified open source model (the Java Community Process), and even Microsoft has placed C# under the auspices of an international standards body (ECMA). The languages themselves are not seen as revenue sources—the revenue models lie in the technologies that the languages rely on.

Just as the commercial vendors have changed their distribution model and seem to have moved toward the open source model, open source communities have been educated on the legal issues around software distribution, from patents and the need to establish clear intellectual property ownership, to the legal risks to which the various actors (contributors, distributors, users) are exposed. Evidence of this maturation is the formation of non-profit umbrella organizations with legal guardianship over the languages, revised license agreements, and more formalized paperwork surrounding contributions from third parties. For example, both the Perl and Python communities actively built non-profit foundations (similar to the Apache Foundation) with appropriate legal status, advisors, and sufficient enough assets that they are both launching targeted funding programs. The budgets involved are still relatively small, but the significance of the accomplishments should not be diminished.

Conclusion

The process by which programming languages are chosen is an interesting one. Individuals tend to follow the advice of peers, as well as being influenced by what they perceive as trends, whether it's for status or for employability. However, these choices are easily reconsidered upon trying a language—working with a language that is a poor fit is typically painful enough to convince people to revisit their original choice. This dynamic is at the heart of the popular success of dynamic languages—sooner or later, programmers find one or more such language that they *like*, or, put differently, that they are able to use productively.

Unlike individuals, organizations choose languages following a very different process, where trends are probably even more important, but the process of "correcting" earlier choices is much rarer, because the choice of language is often made by non-programmers. From the perspective of high-level managers, which programming language should be used within an organization is

[13] See David Stutz's essay "Some Implications of Commodification" (www.synthesist.net/writing/commodity_software.html)

typically seen as a "low-level" consequence of a more important decision on a "platform strategy" or "technology strategy." That high-stakes decision is the focus of tremendous battles among giants such as Microsoft (which argues for its .NET/Longhorn strategy), Sun Microsystems (which promotes a Java-centric strategy), and a variety of other players now arguing for Linux and a more heterogeneous technology stack. These policy choices tend to limit the programming languages available to the programmers who will actually implement the software, and, unfortunately, by nature of being strategic and the result of long-term forecasting, these policies often ignore the painful realities that programmers face *today*. It is not surprising, therefore, that even in organizations that have clearly defined mandates to use "only" .NET or Java, individual programmers, often motivated primarily by a desire to "get things done," have for years used dynamic languages to solve particular kinds of tasks.

Just as Linux was suddenly recognized as a significant platform choice after years of being "snuck in through the back door", high-level open source programming languages are becoming recognized by mainstream analysts as key pieces of an effective approach to building software. It could be to help keep legacy systems running while in the middle of a multi-year transition to a newer system; to integrate subsystems resulting from mergers and acquisitions; to rapidly provide interfaces to customers or partners who demand more flexible integration ; or just to quickly dig through some log files to identify an intruder and fix a network problem. Dynamic languages let people build and maintain important software better and faster. The strengths of these languages derive from their open source nature, from their pragmatic approach, and from their constant evolution in response to real user needs. Ignoring them is equivalent to ignoring the hammer in your tool chest because you've just been sold a fancy screwdriver.

Further Reading

This paper claims little in the way of original thought. Rather, it attempts to present an aggregate view of common trains of thought among a wide set of people, including programmers, language designers, language historians, managers, analysts, and reporters. Readers are encouraged to explore areas of specific interest on the internet. Some recommended sites are listed below:

Websites

The dynamic languages have language-specific communities, which center on a few websites, and from which most relevant technical content can be reached:

- Perl: www.perl.org, use.perl.org, cpan.perl.org, www.perl.com
- Python: www.python.org, www.pythonology.com
- Tcl: www.tcl.tk

- PHP: www.php.net
- Ruby: www.ruby-lang.org
- Prothon: www.prothon.org
- Groovy: groovy.codehaus.org

In addition, several of the sites that are part of the O'Reilly Network (www.oreillynet.com) contain relevant material, as do many of the mailing lists archived on ActiveState's ASPN site (aspn.ActiveState.com/Mail).

Blogs

As with many recent technological trends, the best analyses of important shifts due to open source in general and dynamic languages in particular are those provided by individual commentators and journalists. Notable blogs on related topics include: Jon Udell (weblog.infoworld.com/udell), Sam Ruby (www.intertwingly.net/blog) , Tim O'Reilly (tim.oreilly.com), David Stutz (www.synthesist.net/writing), and Ted Leung (www.sauria.com/blog).