Empirical Analysis of Programming Language Adoption

Leo A. Meyerovich
UC Berkeley *
lmeyerov@eecs.berkeley.edu

Ariel Rabkin
Princeton University
asrabkin@cs.princeton.edu

Abstract

Some programming languages become widely popular while others fail to grow beyond their niche or disappear altogether. This paper uses survey methodology to identify the factors that lead to language adoption. We analyze large datasets, including over 200,000 SourceForge projects and multiple surveys of 1,000-13,000 programmers. Using this data, we establish baseline quantitative models of the language adoption process.

We report several prominent findings. First, language adoption does not have a long tail; a small number of languages account for most language use. Second, intrinsic features have only secondary importance in adoption. Open source library availability is the most influential factor in selecting a language for a project. Social extrinsic factors such as existing team experience also rate highly. Languages features such as performance and semantics do not. Third, developers will steadily learn and forget languages, but only of a limited variety. Developers select more varied languages if their education exposed them to different language families. Finally, when considering intrinsic aspects of languages, developers prioritize expressivity over correctness. They perceive static types as more valuable for properties such as the former rather than for correctness checking.

1. Introduction

Some programming languages succeed and others fail. Understanding this process is a foundational step towards enabling language designers and advocates to influence its outcome and overall language use. Likewise, understanding will aid developers in determining when and whether to bet on a new, experimental language. To date, the language adoption process has not been quantitatively studied in a large scale. This paper closes that gap. We use sociological methods – surveys – and software repository mining to investigate the factors that influence developer language choice.

Since little is known about the programming language adoption process, we focus on basic research questions:

How do popular languages differ from others? We found that popularity follows an exponential curve so most usage is concentrated in a small number of languages. The popular languages are used across a variety of application domains while less popular ones tend to be used for niche domains. Even in niche domains, popular languages are still more typically used. (Section 3)

Which factors most influence developer decision-making for language selection? In multiple surveys, developers listed open source libraries as the dominant factor in choosing programming languages for their most recent project. Social factors, such as existing team experience, also rate highly. (Section 4)

How do developers acquire languages? Developers rapidly and frequently learn languages. Factors such as age only play a small role. However, which languages developers learn is strongly influenced by their education, and in particular, curriculum design. (Section 5)

What intrinsic language features do developers value, and in particular, how do they perceive statically-typed functional languages? We found that developers generally value expressiveness and speed of development over language-enforced correctness. They see more value in unit tests than types, both overall and as a debugging aid. Furthermore, how features are presented strongly influences developer feelings about them, such as prioritizing class interfaces much higher than static types. (Section 6)

Our paper is organized around the above research questions. Before addressing them, we first overview our data sets and methodology. We finish the paper with discussion of the generality of our results (Section 7), related work (Section 8), and further conclusions (Section 9).

2. Methodology and Data

We now describe our data sets, methodology, and a key factor in many of our results: demographics.

* Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung.

OOPSLA '13, Indianapolis, Indiana, USA.

2.1 Data

Our paper uses four data sets for its analyses:

1. SourceForge. We wrote a crawler to download descriptions of 213,471 projects from SourceForge [1], an online repository for open source software. The years examined are 2000-2010. Of most relevance to our analysis, downloaded project metadata labels the languages used, project category (e.g., accounting), date of creation, and owners.

2. MOOC. We gained access to a survey of 1,142 students in a massive online open course (MOOC) on software-as-a-service. The survey was administered at the beginning of the course, so student beliefs would not be altered by the instructors, but they do reflect sample bias towards programmers with an interest in SaaS development. Most respondents were *not* traditional undergraduate students. Their median age was 30 and a majority (62%) described themselves as professional programmers.

The survey was primarily conducted for pedagogical purposes, not research. However, we advised the instructors on question wording, and were given access to the raw collected data. Respondents were asked if they consented to research use of their responses. We only analyze responses from adults who agreed to such use.

Respondents were randomly divided into several subsamples. Some questions were asked to every respondent (*MOOC all*) while others were only asked to one subsample. There were four subsamples; we analyze unique questions from two of them (*MOOC b* and *MOOC d*).

3. Hammer. "The Hammer Principle" is a website that invites readers to compare various items, such as programming languages, based on a multidimensional series of metrics [11]. Respondents pick a set of languages that they are comfortable with, out of a pool of 51. They picked 7 languages on average. Respondents are then shown a series of statements, such as "When I write code in this language I can be very sure it is correct." For each statement, the respondent orders the previously selected languages based on how well they match the statement. Respondent answered an average of 10 statements. The survey period was 2010-2012.

The raw data was provided by the site's maintainer, David MacIver. For each statement, we used a variant of the Glicko-2 ranking algorithm [7] to convert the sparse data of inconsistent pair-wise comparisons into a total ranking of languages. A prior publication [12] describes this analysis in further detail.

4. Slashdot. We created interactive visualizations of the Hammer data set and put them on a public website. These attracted a substantial amount of web traffic. Viewers were invited to answer a short follow-up survey. Most of the readers arrived via links from popular websites such as Slashdot and Wired; we refer to this as the "Slashdot" survey. Over 97% of the responses were collected during a two-week span in the summer of 2012.

Name	Responses	Age Quartiles	Degree	Pro.
MOOC b	166	25 - 30 - 39	51%	60%
MOOC d	415	25 - 30 - 38	51%	58%
MOOC all	1,142	25 - 30 - 38	53%	62%
Slashdot	1,679	30 - 37 - 46	55%	92%
SourceForce	266,452 people and 217,368 projects.			
Hammer	13,271			

Table 1. Overview of surveys and populations. Degree = percentage with at least a bachelor's in CS or related field. Pro = Professional programmers.

Name (and date)	Top 6 Languages
MOOC (2012)	Java, SQL, C, C++, JavaScript, PHP
Slashdot (2012)	C, Java, C++, Python, SQL, JavaScript
SourceForge (2000-2010)	Java, C++, PHP, C, Python, C#
Hammer (2010-2012)	Shell, C, Java, JavaScript, Python, Perl
TIOBE Index (Feb. 2013)	Java, C, Obj.-C, C++, C#, PHP

Table 2. Most popular languages in surveyed populations. There is substantial, but not total, overlap.

Raw anonymized data from the MOOC and Slashdot surveys are available online as are the visualization and data exploration tools for the Hammer data (and underlying correlations).[1] The accessed SourceForge webpages are publicly accessible at time of writing.

2.2 Respondent Demographics

We tracked demographic information on the MOOC and Slashdot surveys. The surveyed populations are both primarily professional developers, and a majority in each have computer science degrees. The MOOC population skews younger than Slashdot and towards fewer professionals (Table 1), though the majority is still adult professionals with degrees. Comparing the results from the MOOC and Slashdot surveys helps us tease out population-specific effects and phenomena in survey design.

Programmers are diverse. Professional developers are estimated to be a minority of all individuals who write code at work or as hobbyists [21]. Our research therefore does not exhaust the universe of programmers. However, studying professional developers is of particular interest for understanding adoption, both in terms of societal impact and understanding the relevance of best-effort practices such as technical education. We therefore emphasize the results for

[1] Currently hosted at `http://www.eecs.berkeley.edu/~lmeyerov/projects/socioplt/viz/index.html`

professionals. Our surveys are well targeted for this purpose; 80.1% of the responses in the Slashdot survey were about projects performed in the workplace.

We qualitatively validate our sample against that of previous work. Table 2 compares the six most popular languages of our surveys against the TIOBE index, which measures the volume of web search results for programming languages. The tables suggest that our survey samples are broadly in alignment with one another and with the TIOBE survey. Differences appear attributable to details in question wording. For example, programmers might use the shell regularly, but not think of it as a programming language unless prompted to include it. They might use SQL, but not consider themselves expert. We performed similar grounding comparisons against other results throughout our work.

2.3 Methodology

We applied standard methodology for iterative gathering of large-scale and cross-sectional data:

Iterative exploratory analysis. Little is known about the programming language adoption process, so our surveys are exploratory. Furthermore, they were performed in a sequence to enable the results of one survey to inform the next. We began with the pre-existing SourceForge and Hammer surveys. These led us to preliminary hypotheses, which we then investigated with the Slashdot and MOOC surveys.

Throughout the survey process, we drew upon external sources. For example, we examined adoption studies in various social sciences, such as the diffusion of innovation model by Rogers [20]. We also performed a literature survey on beliefs of prominent language designers [13]. Finally, we engaged in a series of discussions with programmers and language designers, both in industry and academia. These discussions helped frame hypotheses as well as phrase questions neutrally and broadly.

Instrument quality. We improved the survey quality of MOOC and Slashdot by applying standard techniques:

- *Pretesting.* In developing survey questions, we first designed an initial survey and tested it on several undergraduate and graduate students. We then held a discussion with about 10 graduate students in programming languages about hypotheses. We then repeatedly revised the questions, asking undergraduates, graduate students, visiting researchers and professionals how they understood each question.
- *Free response.* Each survey asked respondents if any questions were confusing and, after some individual questions, whether they had more to add. We do not report on questions that respondents flagged as confusing. To aid anonymity, we do not release the answers to free response questions.
- *Demographic questions.* We applied two techniques to detect and compensate for sample bias. First, we included a variety of demographic questions, such as for age and education. Second, we compared results from several different surveys and different populations.

Our cross-sectional survey methods have known limitations. For example, while we asked several questions about respondent's early experiences with programming languages, a longitudinal study might assist future work that explores specific hypotheses. Likewise, we focus on correlations. To support future examination of causation, we solicited information on potentially confounding factors such as developer demographics. Further discussion of our methodology and the challenges of survey research on programmers appeared at PLATEAU 2012 [12].

3. Popularity and Niches

We first examine the macro-level question of how adoption of popular languages differs from unpopular ones. We divide the analysis into three sub-questions: *What is the overall distribution of popularity? What is the relationship between languages and application domains? How do developers move between languages?*

Overall, we find that programmers use a small set of popular languages. Second, a programmer will select a language that was previously selected. Finally, a key distinction between popular and unpopular languages is that popular ones are used across application domains while unpopular ones are only used within narrow domains. The subtlety in domain specificity is that, even within an unpopular language's narrow domain, the unpopular niche language is still used less than more widely adopted languages. We now explore each point in detail.

3.1 Popularity falls off quickly

What is the distribution of popularity among different languages is a simple but central question. If the distribution is heavy-tailed (e.g., Zipfian), that would imply that, in aggregate, unpopular languages account for most of the overall usage. Such distributions are common in many social processes [4] and increase the economic viability of niche activities.

Programming language adoption does *not* appear to be heavy-tailed. Figure 1 shows that, on SourceForge, language popularity closely follows an exponential curve with a fit of $R^2 = 0.95$. We were not able to fit a power-law curve nearly as well. The consequence is that a few top languages account for most projects. The top 6 languages account for 75% of the projects and the top 20 for 95%. Just 46 languages account for 99% of all SourceForge projects. *Over half the languages that are used only account for less than 1% of the projects.* This pattern has chilling implications for the economics of introducing new languages.

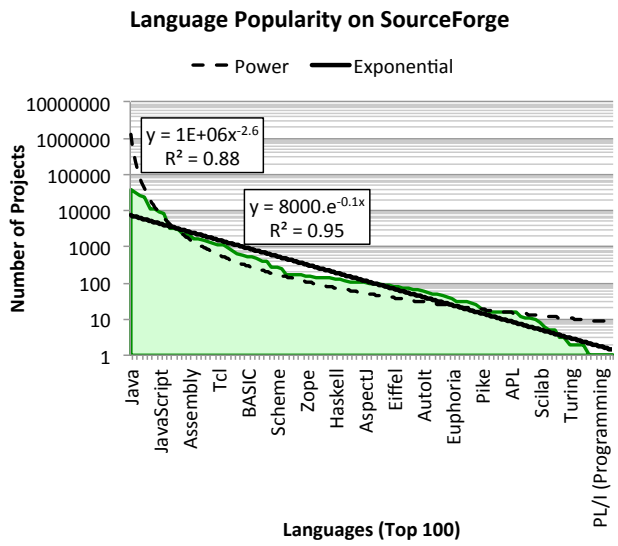

Figure 1. Language popularity fits an exponential curve better than a power law. (SourceForge).

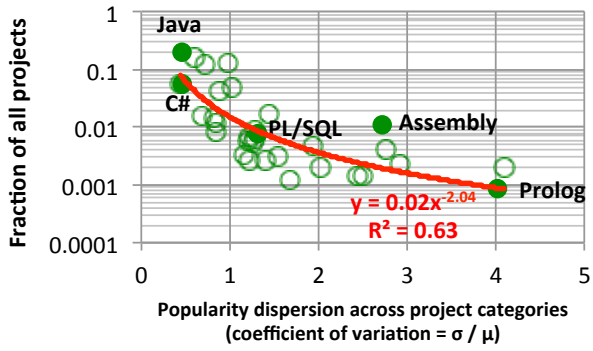

Figure 2. Dispersion of language popularity across project categories. The less popular the language, the more variable its popularity is across application domains. (SourceForge data set).

3.2 Unpopular languages are niche languages

For each language, we compared its overall popularity (as a fraction of all projects) to its popularity in each niche (e.g., accounting). We use the coefficient of variation (standard deviation divided by mean) as a metric of how a language's popularity varies across niches. Figure 2 plots the coefficient of variation against the overall popularity of a language, expressed as a fraction of all projects. Table 3 shows several point values for the mean and standard deviation.

We find that popular languages tend to have broad-based support while unpopular languages tend to be used in a few particular domains. For example, Java is a popular language and for 70% of the categories, its popularity in that category is within +/- 50% of its overall popularity. In contrast, a relatively unpopular language like Prolog is used in a handful of

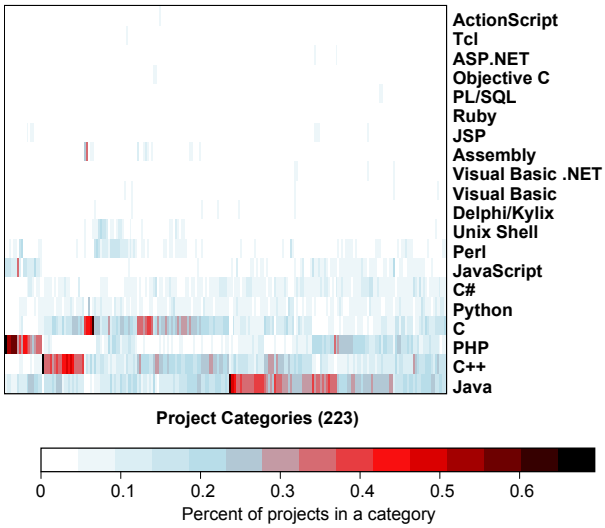

Figure 3. Fraction of projects in each language for different project categories. Y axis is the top 20 (95%) languages and X axis is project categories with at least 100 projects written in any language. Bright cells indicate high probabilities of using a particular language for a given project category. (SourceForge).

categories. For 70% of categories, the popularity within that category will only be within +/- 800% of its overall value.

There are a few outliers in our analysis. Assembly ($\mu = 0.011, \sigma = 0.03$) and Fortran ($\mu = 0.002, \sigma = 0.03$) vary in popularity across categories much more than our model predicts based on their overall popularity. In effect, they are being used as domain specific languages for numeric or low-level programming. That makes them unusual cases of domain-specific languages that are popular overall. A possible explanation is that both used to be viewed as general-purpose languages; they have held onto niches, rather than colonized them for the first time.

An outlier of the opposite nature is VBScript: our model predicts more variability across categories than actually occurs. We hypothesize that, as VBScript is a scripting language packaged with and made for Windows, it attracted a varied base of developers that were performing a range of tasks. Despite occasional outliers such as Fortran and VBScript, Figure 2 shows a clear curve that relates overall popularity to the variation in popularity across niches.

A subtlety of unpopular languages being mostly used within relatively few niches is that a niche is still more likely to be filled by projects written in more popular languages. Figure 3 illustrates this with a heatmap that shows the relative popularity of languages across niches. It illustrates that usage in each niche is skewed strongly towards the bottom rows that contain the most popular languages.

There are exceptions to popular languages dominating less popular ones in individual niches. For example, C++ is more popular than C in general but not for compilers in

Top 6 Languages

	Java	C++	PHP	C	PYTHON	C#
μ	0.205	0.167	0.119	0.140	0.063	0.062
σ	0.094	0.101	0.117	0.100	0.028	0.029
$\sigma_{\bar{x}}$	0.006	0.007	0.008	0.007	0.002	0.002

Top 25-30 Languages

	ASP	BASIC	Obj Pascal	Matlab	Fortran
μ	0.003	0.003	0.002	0.003	0.002
σ	0.004	0.004	0.003	0.008	0.009
$\sigma_{\bar{x}}$	0.001	0.001	0.000	0.000	0.000

Table 3. Mean, variance, and standard error of language popularity in different project categories. Only project categories with at least 100 projects are considered. We show the top six (75% total usage) popular languages and also languages 25-30 (1.2% total usage). (SourceForge).

particular. Likewise, the extent to which one language is more popular than another varies across niches. Figure 3 shows that PHP, C, and C++ vary across domains.

Ultimately, our analysis poses a challenge to the rising research into domain specific languages (DSLs). It implies that current popular notions of domain specialization are insufficient for acceptance by most domain users; something is missing.

3.3 Developer migration

Developers work on many projects over the course of their careers. We do not expect one language selection decision to be independent of the next, so we examine how developers move from language to language. More precisely, we ask how using a language for one project will influence a developer's selection for the next. We (incorrectly) hypothesized that developers stick to the same languages and sometimes alternate between languages within the same family [10].

We again used SourceForge data to answer this question. For each developer that contributed to multiple projects, we examined how the choice of language for a given project influenced the choice for the next one.

We present the results in Figure 4. The bright diagonal line shows that developers often keep the same language from project to project. If we select a first language uniformly at random, developers will keep to that language 18% of the time. Even more often – 52% of the time – they will switch to one of the top six languages overall. However, we do not see significant exploration within families. For example, there is a relatively low probability of switching between Scheme and LISP, or between Ruby, Python, and Perl. Instead, there is a large chance a developer will pick Java, C, or C++ for their next project, regardless of the prior language. The history independence of language selection is highlighted by the vertical bands in Figure 4.

A given prior language only occasionally correlated with the choice of a specific different language for the next

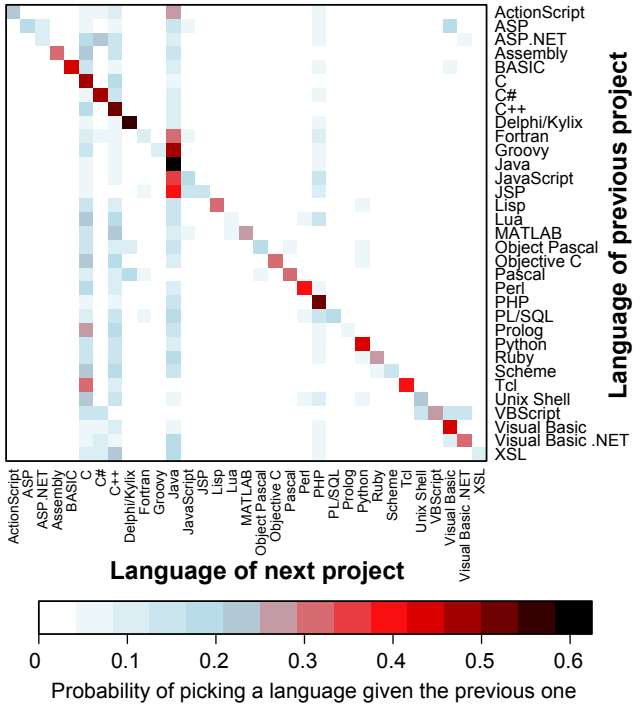

Figure 4. Probability of picking a language given the language of the previous project. Y axis shows the language of the previous project and X axis shows the language of the next: each point is the probability $p(L' = x \mid L = y)$. Languages with less than 100 projects are elided. (SourceForge)

project. Most notably, developers have high probabilities to switch between Windows scripting and application languages, such as VBScript and C#. These languages also correlate with Microsoft web-development languages such as ASP. Such correlations build upon the findings of Karus and Gall [10], who found other groupings such as WSDL and XML being used in conjunction with Java.

Overall, we found a simple and representative model for language selection: language popularity and prior use predicts over 75% of the language selection decisions in SourceForge. Despite our model's simplicity, it is effective. More complicated refinements to ours only need to address 25% of the unexplained projects.

4. Decision Making

Above, we analyzed the overall "macro"-level process of language adoption, showing results such as that most unpopular languages have their popularity concentrated in a few niches. We now offer a "micro" view: we focus on how individual developers make decisions. The central research question being investigated is *which factors influence developer selection of languages?*

Most notably, we find that the extrinsic properties of languages such as team knowledge are more influential than intrinsic ones such as static safety checks. Programming

language and software engineering research has explored these extrinsic aspects less than intrinsics, suggesting an important direction for future work. Extrinsic demographic factors, such as developer age or the size of an organization, have a noticeable effect.

4.1 Extrinsic properties dominate intrinsic ones

In the Slashdot survey, we asked respondents to rate the influence of particular factors in picking the language for their most recent project on a scale from "none" to "strong". By asking about their most recent project, we encourage reflection upon a historical event and deemphasize ideal preferences that might not be acted upon in practice. Respondents assigned individual priorities to 14 factors (Figure 5). We selected the 14 categories such that the dominant choice for picking a language would be represented. Pretesting helped form the list, and free response comments from final respondents suggest that no significant categories are missing.

A wide 30% gap separates the most and least-influential factors. The most-influential factor, the availability of open source libraries, was "strong" or "medium" for over 60% of respondents. For the least influential factor, simplicity, only 25% said the same.

A key characterization of the factors is of how they relate to intrinsic and extrinsic aspects of languages. Some of the factors, such as the language's features, depend on the language design. Others, such as whether a developer already knows the language or the ease of hiring developers who know it, are extrinsic and depend on the social context of the language. Some factors include a mix of extrinsic and intrinsic aspects. For example, the presence of libraries is a combination of social and technical factors.

We emphasize three significant results from the data:

1. **Open source libraries.** We expected library availability to be important. However, we did not expect open source libraries to be so significant overall, and especially compared to commercial libraries.
2. **Extrinsics.** Social factors rate more highly than intrinsic factors. After libraries, four of the next five factors reflect existing code or expertise with the language. Performance, rated fifth-most-important, was the only intrinsic factor in the top half. Designers and researchers interested in adoption should address extrinsic social factors.
3. **Domain specialization.** We refine the notion of domain specialization. Libraries, developer experience, and legacy code are often associated with particular application domains. Thus, the developer priority on them, taken with the result above, suggests that these are key mechanisms that tie languages to niches.

While analytic reasoning prompted our examination of particular influences on decision making, quantifying the relative importance of different factors led to key implications.

4.2 Demographic influences on priorities

We found that demographic properties of developers, an extrinsic influence, strongly correlate with language selection. Two particular forms stand out: properties of the individual and of the organization.

We found several demographic attributes of individuals to influence language selection. For example, we had expected managers to rate factors differently from workers. This was not true after controlling for respondent age. A χ^2 test for estimating responses as manager or developer for each of the 13 factors yielded a failure to differentiate with $p = 1.00$. Unique properties of managers do not drive language selection. Beyond the importance of age, we also find that, unsurprisingly, programmers working on a hobby project care less about team knowledge and legacy effects.

We examined the influence of organization size by splitting respondents into those in companies with fewer than 20 employees, and those with 20 or more. The results are shown in Figure 5 where each shading represents a different demographic slice. As organization size increases, we found that commercial libraries become more important, and open source libraries become less so (although open source is still weighted more highly). Correctness becomes a more influential factor, while simplicity, platform constraints, and development speed matter less. To our surprise, we did not see sharply different weights in differently-sized organizations for legacy code and knowledge.

5. Language Acquisition

Adopting a language requires learning it, not just using it, so we now switch focus to learning. We examine three basic questions: *How long does it take developers to learn languages? When in their careers do they learn? And how does education affect learning?*

Overall, we find that developers are limited in language learning speed and retention. Different developer demographics are more likely to learn certain languages, but overall, programmers will consistently learn the same new languages. We find that the challenge is in learning languages that use distinctive paradigms. Promisingly, computer science education correlates with increased linguistic versatility. The critical caveat, however, is that the substance of the curriculum matters: being exposed to particular paradigms matters far more than simply being a CS major.

5.1 Learning speed

We examined learning speed because it limits how many languages a programmer can use. For the language they used on their most recent project, the Slashdot survey asked respondents to estimate how long it took to learn to use the language well. Figure 6 shows the results for popular

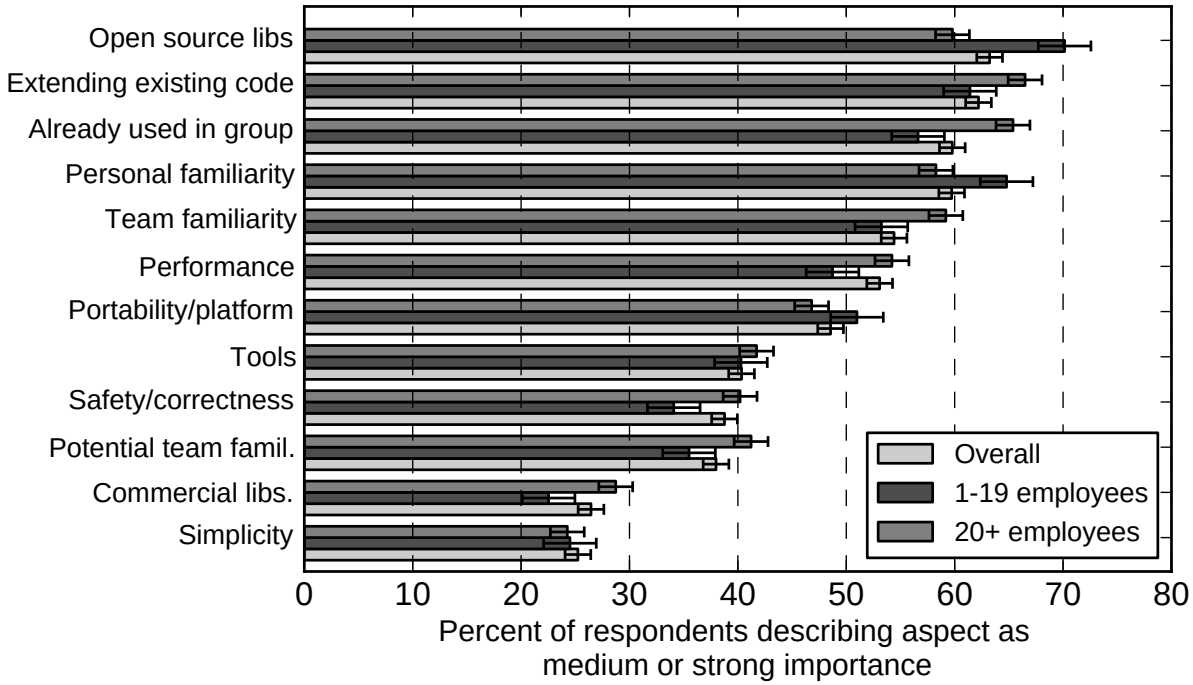

Figure 5. Strength of different factors for picking a language. Self-reported for every respondent's last project. Bars show standard error. Results broken down by company size. (Slashdot, n = 1679)

languages[2]. The question was framed as multiple choice; the y-axis labels of Figure 6 were the available options.

The most challenging language and the most approachable differ by a large ten-fold factor. Programmers report C++ as the slowest to learn while the fastest are PHP and Python. Java and C#, which are semantically similar, are between these extremes and have similar learning times.

PHP is notorious for its ad-hoc design while Python well-regarded for its simplicity. Despite their differences, both languages have comparable reported learning times of just a few months. We infer that developers report that they can "use the language well" even if they have not mastered every nuance. Developers may only need to learn a subset sufficient for completing routine work.

More fundamentally, the relative ease of learning PHP suggests that, while complexity is a barrier to adoption [20], what makes a language complicated for developers need not be what makes it complicated for researcher (e.g., convoluted formal semantics). For the same reason, what makes a language simple in a formal sense does not make it simple for developers.

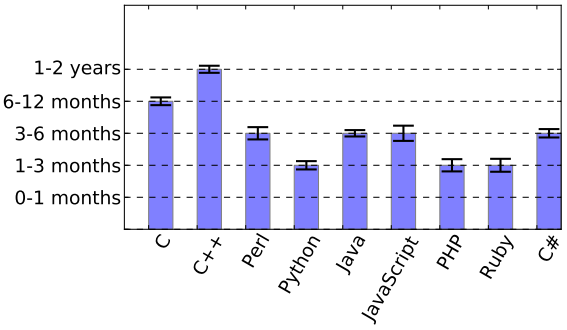

Figure 6. Reported speed of language acquisition. Bars are standard error. (Slashdot, n = 1679)

5.2 On-site language learning

We now examine the relationship between developer demographics and the languages that they know. We initially hypothesized that younger developers are much more likely to take up a new language. This would imply that they are the proper place to encourage adoption. However, our data refutes this hypothesis. We found that younger developers are

[2] We show languages for which we had at least 50 responses.

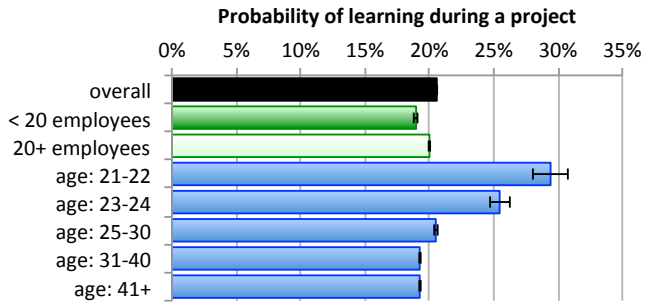

Figure 7. Probability of learning the primary language during a project. Shading denotes demographics and bars are standard error. (Slashdot, n = 1536)

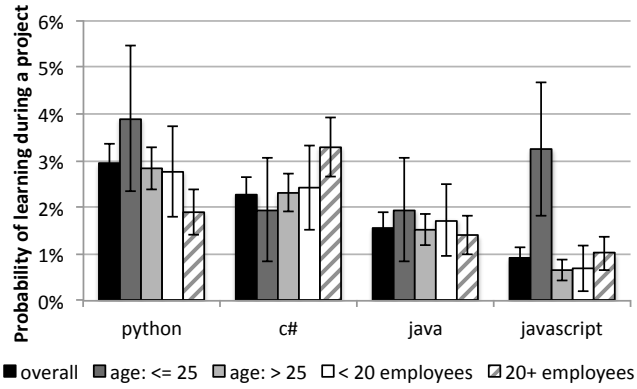

Figure 8. Probability of learning particular languages during a project. Shading denotes demographics and bars are standard error. (Slashdot, n = 1536)

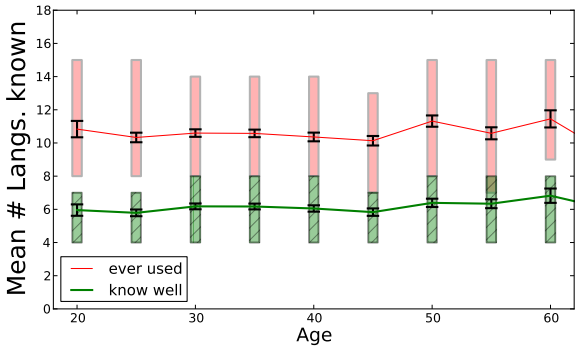

Figure 9. Number of languages by age. Developers of different ages seem to know a similar number of languages. Lightly shaded rectangles show 25th and 75th percentiles, error bars show standard error of mean. (Slashdot, n = 1679)

more likely to learn a language during a project, but only by a limited extent.

The Slashdot survey asked whether respondents learned the primary language for their last project, during the project Figure 7 shows the results, broken down by age and organization size. We found that younger developers are more prone to learn new languages. Overall, 21% of respondents learned a language for their most-recent project . That rate increases to 29% for 21-to-22-year olds. While a jump for younger programmers is not surprising due to inexperience or perhaps eagerness, we had expected to see a much higher increase in learning rate. The difference quickly tapers off, with developers aged 25-30 behaving similarly to those aged 31-40 olds (20% vs. 19%). In contrast, organization size has minimal influence here.

Developers in our sample steadily learn languages through their career. As a result, they are not limited by the languages that were popular when they were young or in school. This means that language adoption time-scales are not driven by the career timelines of developers. This finding is also relevant to employers who might otherwise target particular demographics for recruitment.

The differences between demographics are stronger when we compare learning rates across different languages (Figure 8). While age has limited impact for the overall decision to learn, it does influence which language is learned. Young programmers are several times more likely than older programmers to learn JavaScript. Likewise, workers in a big organization are less likely to learn Python and more likely to learn C#. (For C#, the difference is statistically significant at the 90% confidence level; from a two-sided t-test, p = 0.0875). Java adoption resists demographic influences. The learning rate is constant across various demographics, varying only between 1-2% for the categories shown. Its imperviousness to demographics may be due to an overall high penetration of Java; workers already knew it.

One might think, from these results, that age has an important role in language adoption. As we show in the next section, these differences in language-learning between older and younger developers have limited effect on the overall distribution of language knowledge.

5.3 Languages over time

The above analysis shows how likely a developer is to learn a language for a project. However, to understand how a developer learns over the course of a career, we must account for the fact that developers typically learn many languages. Even though the preceding section shows that young developers learn at an accelerated rate and learn different languages from older developers, we found that age has little influence on which languages a developer knows.

We asked developers to estimate the number of languages they have learned, and also to list the languages that they know well. These different questions capture different levels of knowledge and familiarity and we include both in our results. Figure 9 plots the median number of languages known by developers against their age. Both lines are remarkably

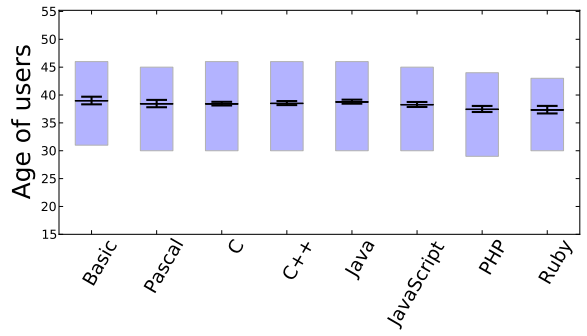

Figure 10. Mean, standard error, and 25th and 75th percentiles for ages of programmers who claim to know each language. Languages are sorted by creation date. Distributions are nearly the same for every language. (Slashdot, n = 1679)

flat. The median respondent to our survey claims to have learned ten languages, and lists six that they "know well."

There is a tension between the flat lines on Figure 9 and the fact that developers are steadily learning. Since developers of all ages are similarly likely to have learned a language for their last project, we would expect the number of languages they know to rise over time. Instead, older and younger developers report a similar degree of multilingualism. We suspect developers are forgetting – or at least, forgetting to mention – languages they no longer use regularly.

Figure 10 shows the median age for programmers who claim to know each of the indicated languages, along with the 25th and 75th percentiles of age. The distributions are all very close. This is surprising: we would have expected Basic and Pascal to skew old, and PHP to skew young. Instead, the 95% confidence interval for every language includes the overall response mean age (38). The deviations are not statistically significant.

This shows that differences in learning by age, described above, get washed out over the course of a career. Programmers of any age are equally likely to remember or newly learn older languages like Pascal. Young programmers catch up to old ones, and older ones keep learning. Popular languages do not thrive or wither based on the age breakdown of their adherents.

We observed similar age invariance patterns in the results of the MOOC survey. In that population as well, developers of different ages know a similar number of languages and a similar mix of languages. We conclude that language popularity is generally not strongly tied to age.

Age invariance suggests that developers have a limited capacity for languages: they will only maintain skill in a limited number of languages. Consequently, developers may not be willing to learn and maintain skill with many different similar languages, each with slight advantages. Ecological theories of innovations competing for adoption in resource-constrained settings become relevant [13].

5.4 Multilingualism and education

We found that particular computer science curriculums lead to graduates with linguistic versatility. In the Slashdot survey, we asked respondents which languages they learned while in school. Past school experience moderately influences the languages that respondents know later on. Our results are in Table 4. (We compare groups of languages because our survey only asked about language families learned in school, not specific languages.)

The vast majority of respondents know a compiled non-functional language, such as C or Java, regardless of major or curriculum. Likewise, dynamic languages are widely known. Education leads to increased likelihood of knowing a language, but only within a 10% increase.

Less-popular language families (assembly, functional, and mathematical languages) are much more sensitive to the form of prior education. Promisingly, developers who learned a functional or math-oriented language in school are more than twice as likely to know one later, as those who did not. Educational intervention has limits, however. For mathematical, functional and assembly languages, the large majority of developers that learned a language in that family no longer know any similar language. Likewise, simply having a general computer science education does not lead to the linguistic versatility of students who learned different language families as part of the course curriculum.

Our finding is a correlation. There can be causation in both directions. Developers who expect to use distinctive families such as assemblers might choose to study them in school. Teasing apart the causation behind our strong correlation is beyond the scope of this paper.

Ultimately, our key finding is that, to improve developer versatility, students should be exposed to diverse language families while they are still in school. Developers learn new languages throughout their career, but are less likely to learn new paradigms. Simply being a CS major does not lead to linguistic versatility, which demonstrates a measured weakness of curriculums that only focus on languages such as Java and Python.

6. Beliefs about Languages

We now turn from programmer actions to programmer beliefs. Beliefs help shape developer and manager decisions to learn and advocate languages, and thereby affect the social dynamics of adoption. Moreover, beliefs are an aspect that language designers can influence and are therefore of special importance. The central research question of this section is *how do developers perceive and prioritize different features?* Additionally, as language researchers often prioritize statically typed functional languages, we are especially curious about developer views on them.

	Overall	CS majors		learned in school	
		yes	no	yes	no
Imperative compiled (C/C++, Java, C# etc)	94%	97%	90%	95%	87%
Functional (Lisp, ML, Haskell, Scheme)	22%	24%	19%	40%	15%
Dynamic (Perl, Ruby, Python)	71%	71%	72%	78%	69%
Assembly	14%	14%	14%	20%	10%
Math (Matlab, R, SAS, Mathematica)	11%	10%	11%	31%	7%

Table 4. Probability of knowing at least one language in the indicated family. Whether developers learn a language in that family in school has more influence than being a CS major. (Slashdot, n = 1679)

Statement	**Corr.**
This language is expressive	0.87
This language excels at symbolic manipulation	0.77
This language encourages writing reusable code.	0.62
The semantics of this language are much different than other languages I know.	0.56
...	
This language has a strong static type system	0.29
...	
Libraries in this language tend to be well documented.	0.00
The resources for learning this language are of high quality	-0.02
This language is large	-0.03
I find it easy to write efficient code in this language	-0.06
...	
There are many good tools for this language	-0.14
...	
This lang. has a niche outside of which I would not use it	-0.42
This is a low level language	-0.53

Table 5. Feature desires. Correlations with the statement "This language has unusual features that I often miss when using other languages." (Hammer)

We reach several conclusions. First, developers prefer expressive languages. Second, they do not perceive the primary benefit of statically typed languages to be in bug finding. Finally, we saw symptoms of overall developer confusion in the form of gaps between how developers perceive a language, the actual design of a language, and how languages are selected in practice.

6.1 What programmers enjoy

We analyzed the Hammer survey to form initial understanding of broad beliefs. The survey asked developers to rank 51 languages against 111 different statements such as "I enjoy using this language." (Note that this is self-reported enjoyment, not an objective measure nor a measure of productivity.) After performing the Glicko-2 analysis [7] to totally order all the languages for each statement [12], we looked for statements correlated with one another.

The closest correlation with enjoying a language is "This language is expressive" (corr. 0.76). Other close correlates include "I find code written in this language very elegant" (corr. 0.73), and "I rarely have difficulty abstracting patterns I find in my code" (corr. 0.66). While open source libraries were important in previous sections that examined developer actions, "Third-party libraries are readily available, well-documented, and of high quality" only weakly correlates (corr. 0.10) with enjoyment.

We also examined correlations with the statement "This language has unusual features that I often miss when using other languages." The results are shown in Table 5). Perceived expressivity strongly correlates with this statement (corr. 0.87); developers value features that ease development. In contrast, there is no significant correlation between having unusual-but-desired features and the ease of writing efficient code. Expressivity and performance are perceived to be unrelated across actual languages.

Finally, we examined statements correlated with "This language has a strong static type system." Unsurprisingly, static types correlate strongly with statements about correctness such as "If my code in this language successfully compiles there is a good chance my code is correct." (corr. 0.85). However, languages with static types only correlate weakly with languages developers claim to enjoy (corr. 0.38) and with expressivity (corr 0.31). These are much weaker correlations than for the other statements mentioned above. Irrespective of the objective value of static typing, we begin to see it has a challenge in developer perception.

6.2 Types vs. testing

We are particularly interested in the perceived tradeoffs between static types and unit testing. We asked several questions about types and testing on the MOOC-b survey. We present only the results from the self-identified professional developers in the sample.

Table 6 displays the results. As can be seen, developers in this sample are strikingly unhappy with types. Only 36% "see the value" of static types. In contrast, 62% – nearly twice as many – see the value of unit testing. Developers are nearly half as likely to "enjoy using" static types (18%) as to enjoy unit tests (33%).

Question	Agreement
Unit testing will reveal bugs that static types miss	31%
Static types will reveal bugs that unit testing misses	7%
Unit testing will reveal more bugs that I care about than static types	19%
Static types will reveal many bugs that I simultaneously care about and are missed by unit testing	6%
I see the value of static types	36%
I see the value of unit testing	62%
I enjoy using static types	18%
I enjoy using unit testing	33%
Most of the value of unit testing is in finding bugs	33%
Most of the value of static types is in finding bugs	8%
I have used statically typed languages for large or many projects	39%
I have used unit testing for large or many projects	34%
Using types helps improve readability	45%
Using types helps improve safety	44%
Using types helps improve program modularity	19%
Using types is generally important, despite the costs	19%
Using types is rarely important	8%

Table 6. Beliefs about types and testing. Shows fraction of responses agreeing with each statement. (Results from self-identified professional developers in MOOC-b sample, n = 96)

Contrary to our initial suspicions about developer beliefs, in neither case do developers describe the value of types to be primarily in finding bugs. Only 8% of programmers think finding bugs is the chief benefit of static types, while 33% say the same about unit tests. Respondents instead find static types to be important in two areas: readability (45% agreement) and safety (44% agreement). Bug finding ranked third and modularity a distant fourth.

We suspected the survey population was biased in favor of dynamic languages because the course associated with it focused on web technologies. To check for this, we looked for correlations between developer feelings about types vs. testing and developer experience and background.

We found that CS majors were slightly ($\sim$ 5%) more likely to enjoy static types, and less likely to enjoy unit tests by a similar amount. However, a χ^2 test shows that the result is not statistically significant. We did not find the average computer science education to influence developer preferences for types vs. testing.;

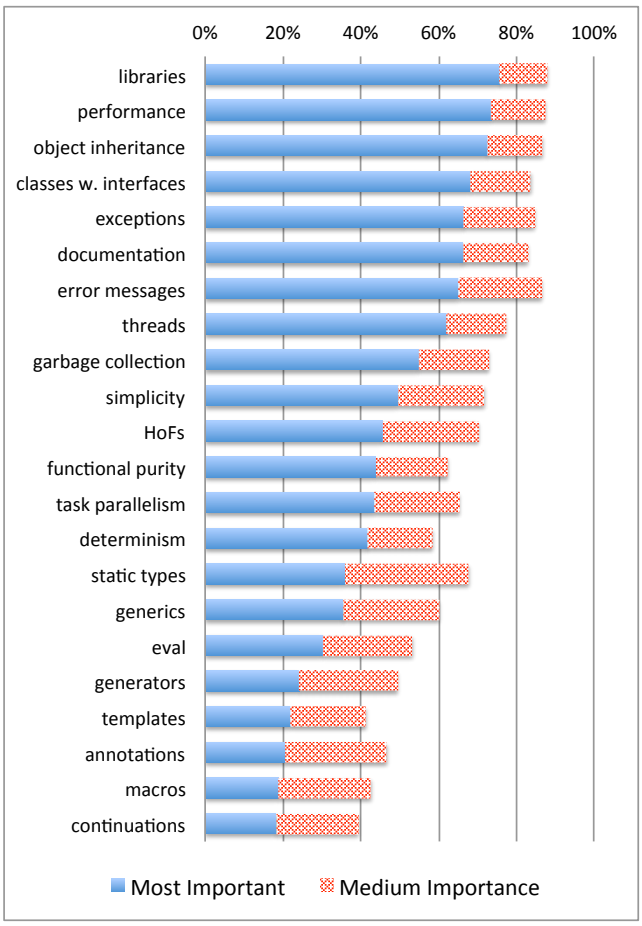

Figure 11. Feature preferences (MOOC-d data set, n = 415).

6.3 Perceived value of features

The MOOC-d survey asked developers to rate the importance of different types of language features on a scale from unimportant to crucial. The results are shown in Figure 11. As can be seen, libraries are the top-rated feature. Such perceived importance of libraries matches our finding that libraries matter in practice (Section 4), and we stress that these confirming results are largely free of effects such as priming because the questions were not asked on the same survey.

We included several options on the MOOC-d survey that represent related concepts. For example, higher-order functions are a strictly more powerful language primitive than inheritance. Interfaces are often used for type systems. Threads can be used to implement task parallelism.

While these features are similar in many ways from a semantics point of view, they had sharply different priorities with developers. For example, 72% of developers considered inheritance important or crucial; only 45% felt that way about higher-order functions. Interfaces likewise were considered far more important than static types. This is surprising, given that interfaces have little effect in dynamically-

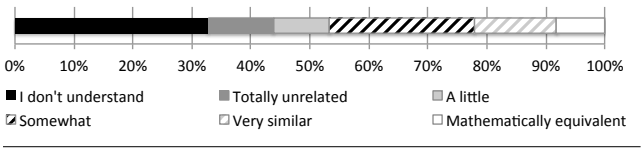

Figure 12. Higher-order functions and objects. Most developers do not see a close connection between the two. (MOOC-d data set, n = 415).

typed languages (and are sometimes even considered bad style in languages such as Python.)

Performance was ranked the second most important factor. This is significant in two ways. First, particular features used for low-level programming, such as threads, macros, static types, and templates all rate much lower. There is a gap between the importance of performance and the language features used to achieve it today. It is unclear if the gap is inherent or if language designers should look for ways to fill it. Second, given the mid-tier importance of performance in picking a language for an actual project (Figure 5), we also see a gap between perception and practice.

The survey asked programmers "How similar are higher-order functions to objects?" to gauge their beliefs about expressive or practical differences. Responses are shown in Figure 12. A third of respondents found the question confusing. Another 20% consider them either totally or largely unrelated. Only 20% reported "very similar" or "mathematically equivalent."

We conclude that a large fraction of developers are disinclined to reason about the underlying semantic power of language features. Features that appear similar to language designers do not appear so to users. (Implementation inheritance can be directly implemented atop HoFs. Methods can be replaced by function-type structure members and inheritance can be modeled succinctly by constructor delegation.)

The optional free-text responses were interesting. They included "didn't know this was possible. Sounds like it could be very useful" and "Objects require imperative programming style while functional programming is a lot less verbose and allows for recursive definition." These responses reaffirm our conclusion in the preceding paragraph.

Taken together, we see that developers' impression of feature importance is not closely tied to the underlying semantic power of the language construct in question. We infer that developer preferences here are shaped by factors extrinsic to the language, such as experience and miseducation.

7. Threats to Validity

We remind the reader of several limitations to our work and threats to its validity.

Some of our analysis is of the SourceForge repository, but SourceForge focuses on open-source development. It is possible that proprietary code bases are statistically different: Obscure languages might hang on longer inside corporate IT departments, for instance. However, open source development is a major part of all development activity and, as we showed, the most important factor in language selection.

Our survey samples, while large, are self-selected. In particular, our Slashdot survey will be biased towards highly engaged programmers who read technology blogs and are interested in programming languages. The Hammer Principle results are likely from a similar population. Our MOOC data will be biased towards programmers who wish to learn more and improve their skills. All three surveys are biased towards Americans and English-speakers. While these are important constituencies, they are not the full universe of programmers. More work is needed to see if these results hold generally; the overlap in results across our surveys suggest our work does generalize to some extent.

Our work is largely cross-sectional. While we examine some longitudinal topics using the 10 years of SourceForge data and careful phrasing of some of our survey questions, we suspect programming languages are reaching the point where we can and should examine questions that span decades. Furthermore, we examine correlations. Empirical analysis of causality is an important growing area [8] and would help elucidate several aspects of adoption.

Going forwards, we believe it would be valuable to study professionals for whom programming is a significant but not the primary job responsibility. For example, engineers and scientists often do not come from computing fields but are important classes of programmers.

8. Related Work

Relatively few studies empirically analyze language adoption. Fewer focus on developer decisions, explore cross-language phenomena, or use large data sets.

Most similar to our work is that of Chen et al. [3]. They gathered or estimated data about 17 different languages in 1993, 1998, and 2003 and then performed regression. In contrast, we examine developer actions and decision making, in much greater scale and fidelity, and with the intent of identifying and quantifying influential factors.

Our analysis of SourceForge is a variant of software engineering literature in mining software repositories. For example, Parnin et al. [16] found that only 14% of developers are responsible for incorporating generic classes into existing Java programs; most developers did not adopt this new language feature but a few became enthusiastic advocates. Surveying programmers about Java (generics) and C++ (templates), we found differences in developer perception of the same phenomena. This may entail that, as was performed in our work, adoption should be studied across languages and, for individual developers, across projects.

Others also mine repositories to understanding language feature and API adoption within an individual language or project. Okur and Dig [15] show that, given a large library of parallel constructs, 90% of usage is accounted for by 10%

of API methods. Only a limited portion of functionality has been adopted. Likewise, Vitek et al. characterize the use of laziness in the R language [14] and dynamic code evaluation in JavaScript [18]. We examine different questions.

Perhaps the most relevant repository mining research is that of Karus and Gall who investigate the propensity of open source developers to use multiple languages [10]. They found significant overlap, particularly between closely related languages such as XML Schema and XSL. Their finding does not conflict with our result that transitions between languages are mostly related to popular and past experience (Figure 4). Consider the likely case that the programmer will switch from editing a WSDL file to editing a Java file: these two languages are often part of the same ecosystem. The probability of then writing PHP is closer to the overall popularity of PHP; PHP is outside of the ecosystem.

Many of the questions we answer fundamentally differ from those in the above repository mining studies [10, 14–16, 18]. Mining exposes the "ground truth" of development practices by focusing on artifacts. We use surveys to enable more direct inquiries to humans about their decision making process. Decision making has unclear physical artifacts and is subject to perception: it is unclear how to understand decision making based on just typical repository information. Furthermore, we use surveys to reveal extrinsic data that is not in typical software repositories, such as demographics.

Small-scale surveys have been used to answer some language use questions. Datero and Galup ran a web survey to examine differences in language knowledge by gender [5]. They found modest differences. For example, within a pool of professionals, male developers were more likely to know most languages, and COBOL was the only language with a pronounced female lean. A study at a single American university found no significant bias in the languages learned by undergraduates there, however [17]. Not presented, our data shows that language selection is gender-neutral on a broader sample than the above work. However, we reported even stronger biases relating to age and organization size. The large scale of our survey has enabled regressing along many dimensions such as these.

Adoption decisions for domains beyond programming languages is studied by social psychologists, management science researchers, and other social scientists. Several leading models of adoption arose over the years, such as the Technology Acceptance Model (TAM) [6] and the Unified Theory of Acceptance and Use of Technology (UTAUT) [23]. These causal, quantitative models relate factors such as perceived ease of use to ultimate adoption decisions. Within a particular population, they predict much of the variance in an individual's desire to adopt a new technology [22]. Models that have been tuned for software development have been able to explain 63% of the variance in developer intention to use object-oriented design techniques [9]. Whereas that work aims to understand the *general* factors behind technology adoption, we seek those that are *specific to programming languages*.

Generalizing the notion of adoption even further, Rogers' seminal Diffusion of Innovation process is perhaps the most extensively studied model of adoption [20]. A 2000 study shows that this model accurately described the process by which COBOL programmers at a large financial-services company learned the C language [2]. A subsequent single-organization study looks at the decision of whether to adopt a formal development methodology [19]. In both cases, the researchers ignored the intrinsic technical attributes of the language or methodology in question, and exclusively considered social factors: we examine both. Furthermore, both cases use sample sizes much smaller than ours: 71 in the first case, and 128 in the latter. We provide a broader view by examining more factors and over wider scenarios.

Finally, various histories of programming languages provide insight into the adoption of specific languages. For example, SIGPLAN sponsors a series of conferences on the history of programming languages (HOPL). The bulk of the papers are retrospectives by language designers on a particular language or related sequence of languages. In contrast, our work seeks to compare across languages and communities. Furthermore, HOPL retrospectives tend to include deep but anecdotal analysis by language designers, while we perform quantitative data.

9. Conclusions

We have shown that unpopular languages are niche languages, and that mainstream languages are the most popular even in niches. Our surveys point to an explanation: libraries and experience tie languages to particular use areas more so than domain-specific language features. It is easy to find anecdotal examples of libraries that had major influence on language adoption within niches, such as `numpy` for numerical programming in Python, and Ruby on Rails for web applications.

Professional developers learn and forget languages throughout their careers. Language designers need not alter the college curriculum nor wait for new developers to enter the field for their language to become popular. However, educators might carefully tailor their curriculums to increase linguistic versatility across language paradigms.

Our work provides lessons for designers. Since languages grow niche-by-niche, designers should focus their marketing and library-creation efforts on particular communities. Growth comes by expanding to new domains. Turning to core language design, we observe that developers show significant unease and unenthusiasm for static typing. This suggests that today's type systems may err too much on the side of catching bad programs rather than enabling flexible development styles. Developers put much emphasis on the documentary purpose of types, suggesting one benefit researchers can build on.

Our results also help inform the broader computer science community. Since language selection is tied to libraries, legacy, and familiarity, there are history-effects and therefore potentially multiple stable equilibria. This suggests that if today's popular languages can be replaced or improved, the changes will be durable.

Beyond the immediate results of this study, our experience has implications for future empirical research of programming languages. We found that survey methods (using both new and existing surveys) to be a powerful tool for testing hypotheses about language adoption. We suspect these methods will be an increasingly valuable technique going forwards, especially given the popularity of the Internet and online courses.

Just as the availability of software repositories has enabled a large wave of empirical software engineering research, we hope our data, and the methodology we used to gather it, will support quantitative socio-technical analysis of programming languages. Of particular note are our large set of responses, tracking of respondent demographics, and solicitation of data about concrete languages and projects.

Finally, we hope our work will be just one of the first of many that analyze the socio-technical foundations of programming languages. We have examined basic questions about language adoption; going forward, there are many more questions about the increasingly social nature of programming languages [13].

Acknowledgments

We thank David MacIver for providing the Hammer data, David Patterson and Armando Fox for the MOOC data, and SourceForge for theirs. Matt Torok, Philip Guo, and Jean Yang helped publicize our Slashdot survey and provided valuable advice. Adrienne Porter Felt and Julie Wu provided helpful methodological guidance.

References

[1] Sourceforge. `http://sourceforge.net`.

[2] R. Agarwal and J. Prasad. A Field Study of the Adoption of Software Process Innovations by Information Systems Professionals. *IEEE Trans. Engr. Management*, 47, 2000.

[3] Y. Chen, R. Dios, A. Mili, L. Wu, and K. Wang. An empirical study of programming language trends. *IEEE Software*, 22:72–78, May 2005.

[4] A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51:661–703, 2009.

[5] R. Dattero and S. D. Galup. Programming languages and gender. *Communications of the ACM*, 47(1):99–102, 2004.

[6] F. D. Davis, R. P. Bagozzi, and P. R. Warshaw. User acceptance of computer technology: a comparison of two theoretical models. *Management science*, 35(8):982–1003, 1989.

[7] M. E. Glickman. Parameter estimation in large dynamic paired comparison experiments. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 48(3):377–394, 1999.

[8] S. Hanenberg. Faith, hope, and love: an essay on software science's neglect of human factors. In *Proceedings of the ACM International Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '10, pages 933–946, 2010.

[9] B. C. Hardgrave and R. A. Johnson. Toward an information systems development acceptance model: the case of object-oriented systems development. *Engineering Management, IEEE Transactions on*, 50(3):322–336, 2003.

[10] S. Karus and H. Gall. A study of language usage evolution in open source software. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, 2011.

[11] D. R. MacIver. The hammer principle. `http://hammerprinciple.com/therighttool`, 2010.

[12] L. A. Meyerovich and A. Rabkin. How not to survey developers and repositories: experiences analyzing language adoption. In *Workshop on Evaluation and usability of programming languages and tools (PLATEAU '12)*, 2012.

[13] L. A. Meyerovich and A. Rabkin. Socio-PLT: Principles for programming language adoption. In *Onward!*, 2012.

[14] F. Morandat, B. Hill, L. Osvald, and J. Vitek. Evaluating the design of the R language. In *European Conference on Object-Oriented Programming (ECOOP)*, 2012.

[15] S. Okur and D. Dig. How do developers use parallel libraries? In *FSE*, 2012.

[16] C. Parnin, C. Bird, and E. Murphy-Hill. Java generics adoption: how new features are introduced, championed, or ignored. In *Mining Software Repositories*, 2011.

[17] D. Patitucci. Gender and programming language preferences of computer programming students at moraine valley community college. Master of Science, Old Dominion University, 2005.

[18] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do: A large-scale study of the use of eval in JavaScript applications. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 52–78, 2011.

[19] C. K. Riemenschneider, B. C. Hardgrave, and F. D. Davis. Explaining software developer acceptance of methodologies: A comparison of five theoretical models. *IEEE Trans. Software Eng.*, 28, 2002.

[20] E. Rogers. *Diffusion of innovations*. Free Press., New York, NY, 1995.

[21] C. Scaffidi, M. Shaw, and B. Myers. Estimating the numbers of end users and end user programmers. In *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on*, pages 207–214, 2005.

[22] S. Sutton. Predicting and explaining intentions and behavior: How well are we doing? *Journal of Applied Social Psychology*, 28(15):1317–1338, 2006.

[23] V. Venkatesh, M. G. Morris, G. B. Davis, and F. D. Davis. User acceptance of information technology: Toward a unified view. *MIS quarterly*, pages 425–478, 2003.