Empirical Analysis of Programming Language Adoption

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

Ariel S. 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, 590,000 projects tracked by Ohloh, and multiple surveys of 1,000-13,000 programmers.

We report several prominent findings. First, language adoption follows a power law; a small number of languages account for most language use, but the programming market supports many languages with niche user bases. Second, intrinsic features have only secondary importance in adoption. Open source libraries, existing code, and experience strongly influence developers when selecting a language for a project. Language features such as performance, reliability, and simple semantics do not. Third, developers will steadily learn and forget languages. The overall number of languages developers are familiar with is independent of age. Finally, when considering intrinsic aspects of languages, developers prioritize expressivity over correctness. They perceive static types as primarily helping with the latter, hence partly explaining the popularity of dynamic languages.

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: general

General Terms Languages, Human Factors

Keywords programming language adoption; survey research

* 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2374-1/13/10. . . $15.00.
http://dx.doi.org/10.1145/2509136.2509515

1. Introduction

Some programming languages succeed and others fail. Understanding this process is a foundational step towards influencing it, assisting language designers and advocates in fostering adoption. Likewise, understanding adoption 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 at a large scale. This paper addresses that gap. We use a combination of survey research and software repository mining to investigate the factors that influence developer language choices.

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

What statistical properties describe language popularity? We begin (Section 3) with an empirical analysis of language use across many open source projects. Such a macro-scale analysis reveals what trajectories languages tend to follow. Our analysis includes the overall distribution of language use, and how it varies based on the kind of project and developer experience.

We found that popularity follows a power law, which means that most usage is concentrated in a small number of languages, but many unpopular languages will still find a user base. 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.

Which factors most influence developer decision-making for language selection? Section 4 examines the subjective motivations of developers when picking languages for specific projects. Knowing what matters to developers helps language designers and advocates address their perceived needs.

Through multiple surveys, we saw that developers value open source libraries as the dominant factor in choosing programming languages. Social factors not tied to intrinsic language features, such as existing personal or team experience, also rate highly.

How do developers acquire languages? Knowledge about the learning process is important because developers are much more likely to use a language they already know.

In Section 5, we examine how age and education shape language learning.

We found that developers rapidly and frequently learn languages. Factors such as age play a smaller role than suggested by media. In contrast, which languages developers learn is influenced by their education, and in particular, curriculum design.

What language features do developers value? Whereas Section 4 looks at how developers pick languages for specific projects, Section 6 examines their feelings about intrinsic features of languages, such as type systems. The results can help designers craft better languages and advocate them, and even influence curriculum design due by exposing knowledge gaps.

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 for debugging and overall. How features are presented strongly influences developer feelings about them: developers rank class interfaces more highly than static types.

Before addressing the above research questions, we describe our data sets and methodology. We finish the paper by discussing threats to the validity of our results (Section 7), related work (Section 8), and our conclusions (Section 9).

2. Methodology and Data

This paper is based on several different surveys and data sources: we used software repositories, surveys conducted by others, and a survey conducted by us. These were used in a sequence; the results of one analysis informed the design of the next. We began with pre-existing data from the SourceForge repository and The Hammer Principle, a long-running online survey about programming languages. These led us to preliminary hypotheses, which we investigated with surveys. Finally, we cross-validated our results with the Ohloh project database.

2.1 Data

We now describe our data sets in more detail. In chronological order, we used:

1. Open Source Repository Metadata: *SourceForge*. We wrote a crawler to download descriptions of 213,471 projects from SourceForge [2], an online repository for open source software. Of most relevance to our analysis, the downloaded metadata for each projects documents the project's languages used, primary project category (e.g., accounting), date of creation, and the project's owners. The languages are drawn from a list of 100 curated by SourceForge; the categories, from a set of 223.

The years examined are 2000-2010. This data set is a reasonable proxy for open source behavior because, for most of the analyzed period, SourceForge was the dominant software repository. For example, its modern competitor GitHub was not even created until mid 2008. Open source community behavior is important in its own right, and as our results will show, commercial developers prioritize open source when making their own adoption decisions.

2. Online Poll: *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 [13]. Respondents pick a set of languages that they are comfortable with out of a pool of 51 languages. They picked 7 languages on average. Respondents are then shown a randomly sorted 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 languages that they selected based on how well they match the statement. The survey includes 111 statements, and respondents sorted languages for an average of 10 statements each before tiring. The survey period was 2010-2012.

The raw and anonymized data was provided by the site's maintainer, David MacIver. For each statement, we used a variant of the Glicko-2 ranking algorithm [8] to convert the sparse data of inconsistent pair-wise comparisons into a total ranking of languages. A prior publication [14] describes this analysis in further detail.

The intuition is that we treat each statement as a tournament between languages. Glicko converts the sparse pairwise comparison into a total order. The Glicko family of algorithms is used in chess tournaments and online game rankings for producing a complete ranking of players without every pairwise comparison: beating a highly ranked language contributes more than beating a low-ranked language. Each player (i.e., language) has an absolute rank and a statistical confidence for it.

3. Course Surveys: *MOOC*. We gained access to a survey of 1,185 students in a massive online open course (MOOC) on software-as-a-service (SaaS). The survey was administered at the beginning of the course, so student beliefs were not be altered by the instructors, though the results 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; 1,142 said yes (96.5% of all responses). We only analyze responses from adults who agreed to research use.

Respondents were randomly divided into four subsamples. Some questions were asked to every respondent (*MOOC all*) while others were only asked to one subsample. We divided the questions to avoid fatigue in respondents from overly long surveys while still achieving enough responses for statistically significant analysis of many questions. Only

Name	Responses	Age Quartiles	Degree	Pro.
MOOC b	166	25 - 30 - 39	51%	60%
MOOC d	415	25 - 30 - 38	51%	58%
MOOC a–d	1,142	25 - 30 - 38	53%	62%
Slashdot	1,679	30 - 37 - 46	55%	92%
SourceForce	266,452 people and 217,368 projects.			
Ohloh	590,000 projects.			
Hammer	13,271			

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

Name	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#
Ohloh	2000-2013	XML, HTML, CSS, JavaScript, Java, Shell
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.

two of the subsamples (*MOOC b* and *MOOC d*) are relevant to the questions addressed in this paper.

4. Online Survey: 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 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.

5. Ohloh We cross-validated some results using Ohloh [1]. Ohloh is a website, bought by SourceForge, that tracks over 590,000 open source projects hosted on SourceForge, GitHub, and elsewhere. We used it for queries such as how many repositories contain a Java file.

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 and Ohloh API are publicly accessible at time of writing.

Not every data source is applicable to every question we ask. Table 3 summarizes how the data was used in various sections.

2.2 Respondent Demographics

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

Programmers are diverse. Professional developers are estimated to be a minority of all individuals who write code at work or as hobbyists [23]. 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 professionals.

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 [3]. 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 CSS and XML regularly, but not think of it as a programming language unless prompted to include it. They might use SQL, but not consider themselves experts. Finally, they might use a shell scripting language, but not consider it important. We performed similar grounding comparisons against other results throughout our work.

2.3 Methodology

To maximize the quality of our surveys, we applied standard methodology for iterative gathering of large-scale and cross-sectional data.

Throughout the survey design process, we drew upon external sources. At the start, we examined adoption studies in various social sciences, such as the diffusion of innovation model by Rogers [22]. We also performed a literature survey on beliefs of prominent language designers [15]. Finally, we engaged in a series of open-ended discussions with programmers and language designers. The interviewees were

[1] Currently hosted at `http://www.eecs.berkeley.edu/~lmeyerov/projects/socioplt/data/all.tar.gz` and `http://www.eecs.berkeley.edu/~lmeyerov/projects/socioplt/viz/index.html`, respectively

Section	Question	SourceForge	Ohloh	Hammer	MOOC	Slashdot
§3	What is the macro-behavior of languages and communities?	✓	+			✓+ §4.1
§4	What factors influence language choice for projects	+ §3.2 and 3.3			+ §6.2	✓
§5	How do programmers learn and lose languages				✓	✓
§6	What do programmers believe is important in a language?			✓	✓	+ §4

Table 3: **Summary of where each data source is used**. ✓= data source used in that section, + indicates that the result is cross-validated against a related result elsewhere in the paper.

primarily visitors to UC Berkeley, attendees at programming language and software engineering workshops and conferences, Bay Area startup employees, and were academic, industrial, and government programmers and language designers. These discussions covered several pre scripted questions and were otherwise driven by the participant. The interviews helped frame hypotheses as well as helped inform us on how to phrase subsequent survey questions neutrally and broadly. Overall, we received advice and input from about 30 people.

To improve the survey instruments themselves, we applied several techniques.

- *Piloting questions.* In developing survey questions, we first prepared a preliminary questionnaire 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 by asking undergraduates, graduate students, and visiting researchers and professionals about 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 preserve 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 compare results from several different surveys and different populations.

Our cross-sectional survey methods have 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 [14].

3. Popularity and Niches

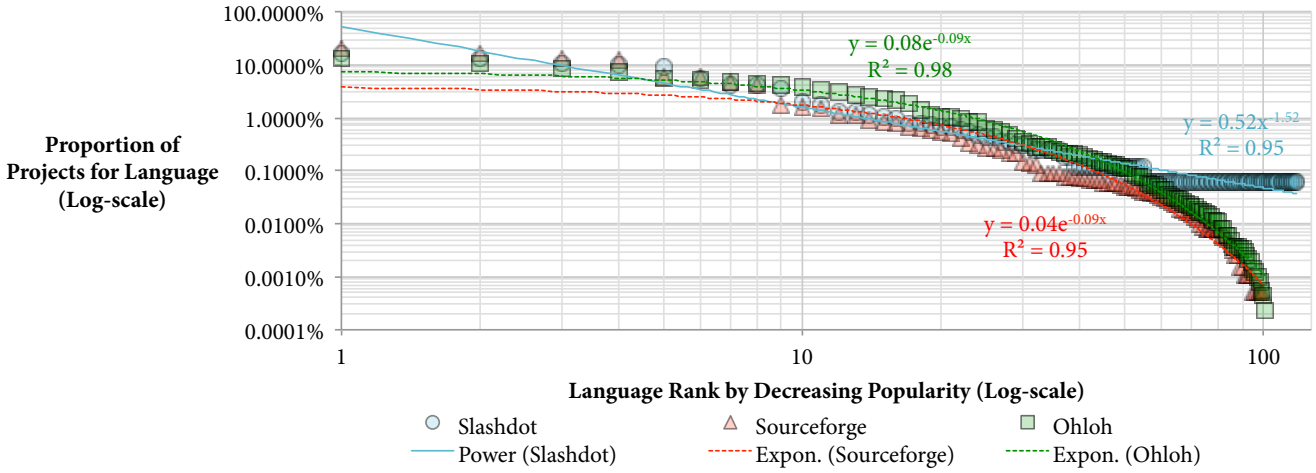
We first examine the macro-level question of how adoption of popular languages differs from unpopular ones. We divide the analysis into three sub-questions: *What is the overall distribution of popularity? What is the relationship between languages and application domains? How do developers move between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

3.1 Usage Falls Off Quickly, then Plateaus

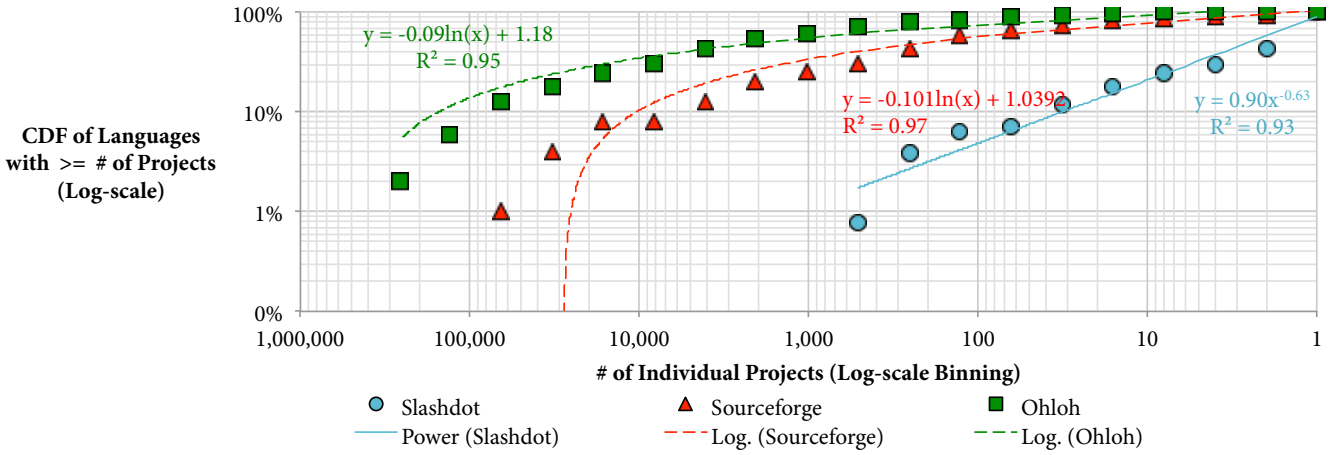
The distribution of usage across different languages indicates the risk/reward trade-off for creating a language. If the median language has significant usage, that makes creating a language a more promising endeavor than if the median language accounts for a negligible fraction of usage. (Even in the latter case, there may still be utility in building a language, but the creator has fewer grounds to expect usage.)

Figure 1a shows that a small number of languages account for most usage on SourceForge and Ohloh repositories and in the Slashdot survey results. On SourceForge, the top six languages account for 75% of the projects; the top 20 languages for 95%. General-purpose languages comprise most of the top 20 languages in all three data sets.

Some domain-specific languages also rank highly. Considering only a project's primary language, SQL is the only domain-specific language in the top 20 languages. But if we include all languages for a project, SQL, HTML, and CSS are in the top 20. We cross-validated with Ohloh's analysis of overall language use in actual repositories. Ohloh reports that XML, HTML, and CSS are the top three languages in terms of any use. SQL and `Make` are also in the top 20. That a small set of general-purpose languages dominate language use is not surprising. However, none of the language designers we interviewed suggested that a language such as CSS, a constraint-based language for web page layout, would be more popular than all of the general-purpose languages such as C and Java. The prominence of domain-specific languages among other popular languages is surprising.



(a) **Probability Mass Function**. Each point represents a language.



(b) **Cumulative Distribution Function**. Each point represents the set of languages with the number of projects listed on the x axis.

Figure 1: **Language popularity**. Slashdot survey data follows a heavy-tailed power law while curated SourceForge and Ohloh data better follow an exponential curve.

We turn now from the most popular to the least-popular languages, the tail of the popularity distribution. Our initial analysis of SourceForge shows an exponential decline in language popularity, as does cross-validation with Ohloh (Figure 1a). In contrast, popularity in the distribution's tail plateaus in the Slashdot survey of developers. The average difference in rank for languages in common between the Slashdot and SourceForge is only 6.9, suggesting that the two rankings are reporting on similar underlying usage. The disparity in tail behavior stems from the fact that the SourceForge and Ohloh language counts are based on a curated list of languages. In contrast, the Slashdot results include all responses and thus tail behavior is not filtered out.[2] Even though the Slashdot survey measures orders of magnitudes fewer projects, it yields a similar number of languages to SourceForge and Ohloh.

The different data sets do not include the same languages, however: they diverge in tail languages. Over half of the languages reported in Slashdot are not tracked by Ohloh. For example, Slashdot respondents report using SAS. Although manual verification shows SAS being used in covered projects, Ohloh does not count it. We conclude that programming language popularity has a heavy tail but that experimental artifacts can conceal this in many data sources.

The heavy tail covers many unpopular languages. For example, the least popular languages (those with only a single project) in Slashdot cover 6% of the projects (Figure 1b). Three such languages cover 0.2% of the projects. Lacking the heavy tail, SourceForge and Ohloh show an equivalent percentage of less than 0.002% for three of the least-popular languages. The unpopular languages can be quite domain-specific in practice. For example, one respondent reported developing in the Linden Scripting Language, used for the

[2] We manually inspected the Slashdot results and merged synonyms.

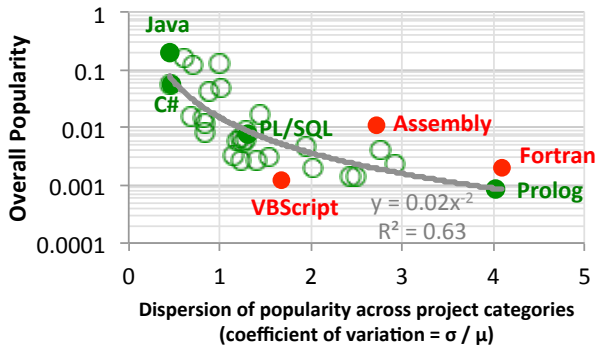


Figure 2: **Dispersion of language popularity across project categories.** The less popular the language, the more variable its popularity is across application domains. (SourceForge data set).

Second Life virtual world system. The heavy tail in our data shows that the market for languages supports creating many unpopular languages, not just extending the few relatively popular ones such as Java and Haskell.

3.2 Unpopular Languages are Niche Languages

Our next question is how the popularity of a language relates to its domain-specificity. To evaluate this, we use the project category labels provided by the SourceForge metadata. For each language, we compared its overall popularity (as a fraction of all projects) to its popularity in each category (e.g., accounting). We formalize this comparison as the coefficient of variation (standard deviation divided by mean). Figure 2 plots the coefficient of variation of language usage against the percentage of projects using them (i.e., popularity). Table 4 shows several languages in detail.

We find that popular languages receive broad-based support while unpopular languages tend to be used in a few particular domains. For example, Java is used by 20% of the projects. For 70% of the project categories, Java is used by 10–30% of the projects within that category; for those categories, Java's popularity is within 50% of its overall popularity. In contrast, a relatively unpopular language like Prolog is only used in a handful of categories. For 70% of categories, Prolog's popularity within a category will differ from its overall popularity by +/- 800% rather than Java's +/-50%.

A few outliers stand out in our analysis. Assembly ($\mu = 0.011, \sigma = 0.03$) and Fortran ($\mu = 0.002, \sigma = 0.03$) vary in popularity across categories 3-6X more than our model predicts based on their overall popularity. In effect, these are domain-specific languages for low-level or numeric programming, respectively. 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.

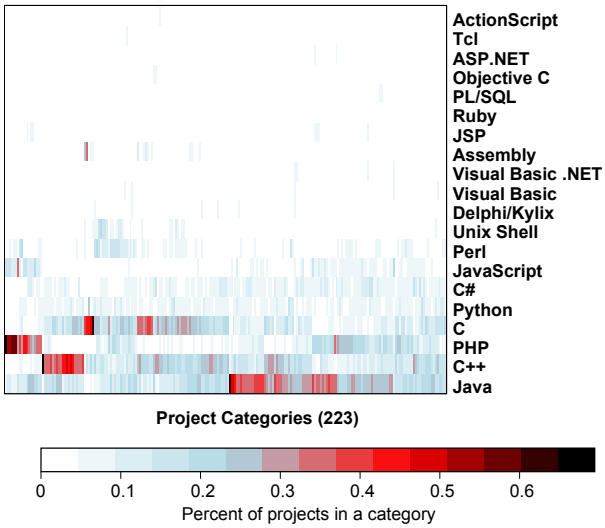


Figure 3: **Fraction of projects in each language for different project categories.** Y axis is the top 20 (95%) languages and X axis is project categories with at least 100 projects written in any language. Dark red cells indicate a high probability of using a particular language within a given project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

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 4: **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). Order of categories is arbitrary. (SourceForge).

VBScript is an outlier of the opposite sort: our model predicts more variability across categories than actually occurs. We hypothesize that, since VBScript is a scripting language packaged with and made for Windows, it attracted a varied base of developers that were performing a wide 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.

Even though unpopular languages have their usage concentrated in a few niches, they are rarely the most popu-

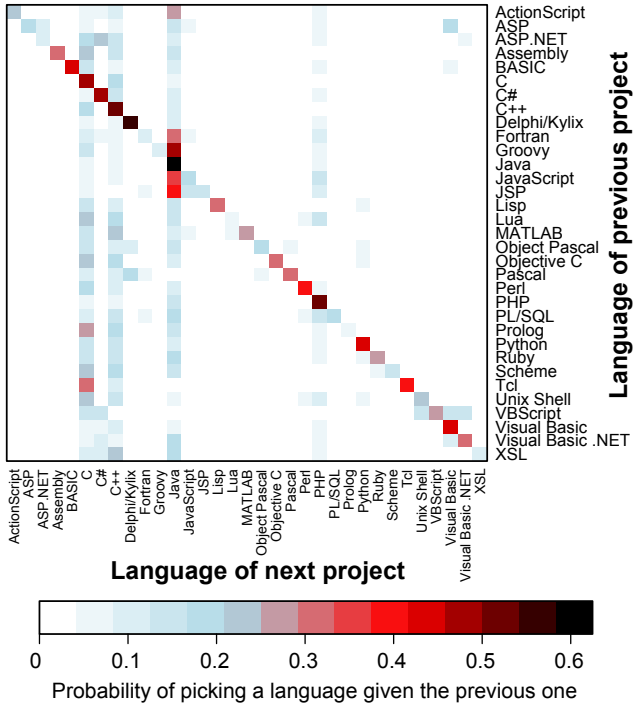


Figure 4: **Probability of picking a language given the language of the previous project.** Y axis shows the language of the previous project and X axis shows the language of the next: each point is the probability $p(L' = x \mid L = y)$. Languages with less than 100 projects are elided. (SourceForge)

lar languages in those niches: the consistently popular languages tend to win out. Figure 3 illustrates this with a heatmap showing the relative popularity of languages across niches. The languages are sorted by popularity, with the bottom rows containing the most popular languages.

There are cases where a language that is less popular overall will beat out more-popular languages in specific niches. For example, C++ is more popular than C in general but not for compilers in particular. Figure 3 shows that PHP, C, and C++ (the top three languages) vary in relative popularity across domains. This effect is largely limited to the top few languages, however. Beyond that, a niche-specific language's comparative advantage is dominated by the overall popularity of the top overall languages.

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 influences a developer's selection for the next.

Karus and Gall have analyzed the commit logs from 22 open source projects, and report that developers tend to stick to "clusters" of languages — for example, they report that only a small fraction of Java developers also use C/C++ (7%), and C/C++ developers are five times as likely as Java developers to use Perl [12]. We hypothesized that we would see a similar clustering pattern in our data.

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 project influenced the language choice for the project with the chronologically next creation date.

We present the results in Figure 4. Each row depicts the probability that a developer's next SourceForge project will use the language on the bottom, given that the developer was previously in a project that used the language labeled on the right. The bright diagonal line shows that developers often keep using the same languages. If we select a first language uniformly at random, developers will keep to that language 18% of the time. More often – 52% of the time – they will switch to one of the top six languages overall. These overall-popular languages correspond to the vertical bands in Figure 4.

A given prior language only occasionally correlated with the choice of a specific different language for the next project. Most notably, developers have high propensities 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 are also visible in the results of Karus and Gall [12], who found groupings such as WSDL and XML being used in conjunction with Java.

Notably, we do *not* see significant exploration within linguistic families. There is a relatively low probability of switching between Scheme and LISP, or between Ruby, Python, and Perl. We conclude that developer movement between languages is driven more by external factors such as the developer's background or technical ecosystem than by similarity of the underlying languages. This implies that language advocates should focus on a domain and try to convince programmers in that domain, instead of trying to convince programmers who use languages with semantic similarities to the new language.

One limitation of our result is that project participation is an imperfect proxy for language use. Some cases of language reuse may be due to developers that only *saw* a language being used on one project and then used it themselves on the next. Likewise, we only sample SourceForge projects: the effects we report may be stronger in general because they may carry through intermediate projects not reported on SourceForge.

Overall, we found a simple and representative model describes 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 would only need to address 25% of the unexplained projects.

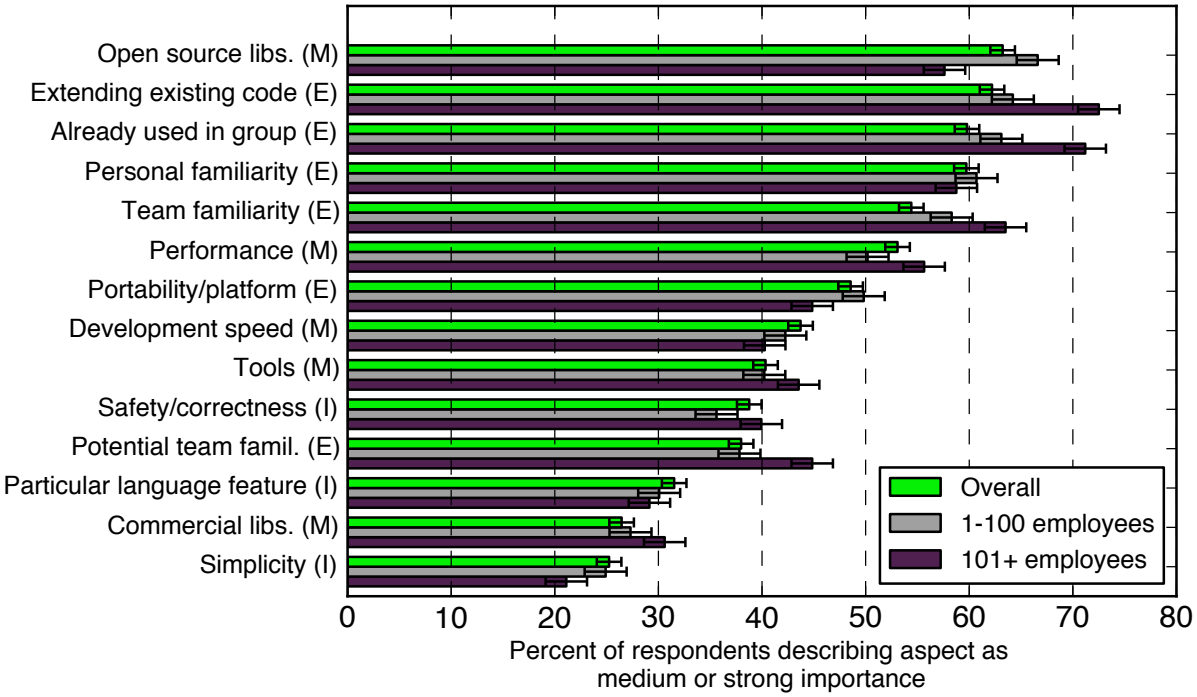


Figure 5: **Importance of different factors when picking a language**. Self-reported for every respondent's last project. Bars show standard error. E = Extrinsic factor, I = Intrinsic, M = Mixed. Shows results broken down by company size for respondents describing a work project and who indicated company size. (Slashdot, n = 1679)

4. Decision Making

Above, we analyzed the overall "macro"-level process of language adoption. One of our observations was that unpopular languages usually 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?* We break this into two parts: how do developers weigh features of languages when they are making choices, and how do demographics affect the languages that developers pick? This will help explain the observations in the previous section.

4.1 Weighing Language Features

In the Slashdot survey, we asked respondents to rate the influence of particular factors in picking the language for their most recent project. Ratings used a four-point scale from "none" to "strong". By asking about their most recent project, we encouraged reflection on a historical event and deemphasized 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 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.

We wanted to see how these results depend on a developer's work environment. To do this, we broke out those respondents who picked a work project and who indicated an organization size. We divided respondents into those working at companies with fewer than 100 employees and those with more than 100. (This threshold is closest to splitting the data set evenly, facilitating comparison.) The results for these subpopulations are also shown in Figure 5.

Some of the factors, such as the language's features or simplicity, depend on the language design, not on its user base. 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 or good development

tools is a combination of social and technical factors. Figure 5 is labeled with our judgement about whether a factor was intrinsic, extrinsic, or mixed.

We emphasize four results from the data:

1. **Open source libraries.** Open source libraries are the most influential factor for language choice overall and the most influential factor for commercial projects at small companies. They are an important factor, but not the most important factor, at large companies.
2. **Social factors outweigh intrinsics.** Existing code or expertise with the language are four of the top five factors for adoption. In contrast, intrinsic factors, such as a language's simplicity or safety, rank low. Implementation attributes, like performance and tool quality, have both intrinsic and extrinsic components. (Some languages lend themselves more easily than others to a high-performance implementation.) These mixed attributes vary in importance.
3. **Domain specialization.** Libraries, developer experience, and legacy code are all important in language selection. These factors are often associated with particular application domains. Thus, the developer emphasis on these attributes helps explain the result in Section 3.2 that less-popular languages are more niche-specific.
4. **Company size matters.** Employees at larger companies place significantly more value on legacy code and knowledge than do employees at small companies. Correctness becomes more influential for large companies, while simplicity, platform constraints, and development speed matter less. Compared to developers overall, those at larger organizations weigh commercial libraries more and open source libraries less (although open source is still weighted more highly).

These results help inform language designers seeking adoption where to focus their efforts. Developing high-value open source libraries is likely to have a large influence to language adoption, particularly for individuals and small companies. Simplicity will not attract many programmers. Smaller companies are less constrained by legacy code and experience. We suspect that they are therefore likely more willing to adopt new languages that are not backward-compatible. In contrast, a backward-compatible change to a language might be more valuable to a large company.

4.2 Demographic Influences on Language Selection

We now look at how developer demographics affect the languages that developers select. Understanding this extrinsic factor clarifies the generality of our results and its role for future empirical analysis or targeting of developers.

We observed significant correlations when distinguishing the different demographics that selected a particular language. Figure 6 shows that, for languages selected by Slashdot respondents, age and company size strongly influence

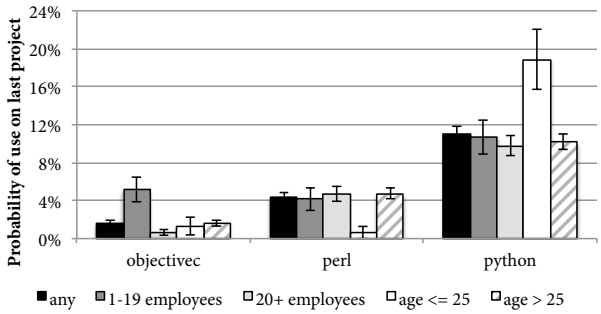


Figure 6: **Demographic influences on selecting particular languages**. Self-reported for every respondent's last project. Bars show standard error. (Slashdot, n_{any} = 1679, $n_{1\text{-}19\ employees}$ = 290, $n_{20+\ employees}$ = 790, $n_{age \leq 25}$ = 154, $n_{age > 25}$ = 1382).

whether a particular language was selected. For example, employees at small companies are 1.4-3.0 times more likely to use Objective-C than the typical respondent. Likewise, when looking at age, developers under 25 rarely use Perl but disproportionately select Python.

Compared to the breakdowns of Figure 5, we see that language selection is especially sensitive to demographic effects. Notice also that different languages are sensitive to different demographic variables: Perl and Python are age sensitive, but appear insensitive to company size; Objective-C is sensitive to company size, not to age. Our observations suggest that it is unsafe to generalize about how any particular demographic variable will correlate with language usage. Different languages will have different social dynamics.

5. Language Acquisition

Here, we switch focus from the decision to use a language at a particular instant to the process of learning languages. We examine three related questions: *How long does it take developers to learn languages? When in their careers do they learn? How does education affect learning?* The previous section demonstrated that developers prefer to use languages they already know. Understanding how quickly developers learn and what induces them to learn helps explain this adoption factor.

5.1 Learning Speed

For the language used on their most recent project, the Slashdot survey asked respondents to estimate how long it took to learn to use the language well. To "know a language well" is an intentionally imprecise and subjective standard. We showed above that when developers pick languages for a project, they are heavily influenced by the languages they believe they know: developers will be using their own subjective standard.

Figure 7 shows the results for all languages for which we had at least 50 responses. The question was framed

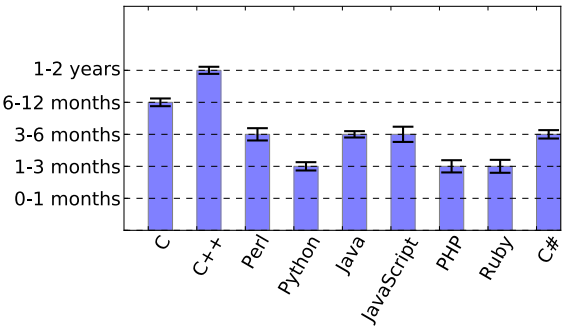


Figure 7: **Median reported speed of language acquisition.** Bars are standard error. (Slashdot, n = 1679)

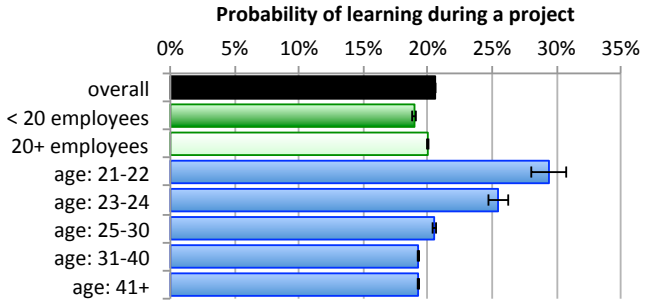


Figure 8: **Probability of learning the primary language during a project.** Shading denotes demographics and bars are standard error. (Slashdot, n = 1536)

as multiple choice; the y-axis labels of Figure 7 were the available options.

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

The relative time to learn PHP is noteworthy. PHP is notorious for its ad-hoc design while Python is well-regarded for its simplicity. Despite their differences, both languages have comparable reported learning times of just a few months. We infer that developers determine 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 [22], what makes a language complicated for developers need not be what makes it complicated for a designer or researcher (e.g., convoluted formal semantics). For the same reason, what makes a language simple in a formal sense need not make it simple for developers and otherwise adoptable.

5.2 On-site Language Learning

We now examine the relationship between developer demographics and the languages that they learn. Knowing which demographics are likely to learn helps language proponents target their outreach at those potential early adopters.

The Slashdot survey asked respondents whether, during their last project, they learned the primary language for it. Figure 8 shows the results broken down by age and organization size. We found that younger developers are more prone to learning 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%). Relative to age, organization size has minimal influence.

Our analysis shows that developers in our sample steadily learn languages throughout 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 the time scale of language adoption is not driven by the career timelines of developers.

5.3 Languages Over Time

The next sub-question we examine is how a developer's age influences the particular languages that the developer knows. Some professional recruiters claim that there are large and significant differences in the languages that older and younger developers use [11]. Our results refute this.

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. The highest median age we observed was 38.9 (BASIC and Perl); the lowest was 37.3 (Ruby). This is surprising: we would have expected changes in education over time to result in substantial variation in median programmer age. Instead, the 95% confidence interval for every language includes the overall response mean age (38). The deviations are not statistically significant. We observed similar age invariance patterns in the results of the MOOC survey. There as well, developers of different ages know a similar number of languages and a similar mix of languages.

This shows that differences in learning by age, described above, get washed out over the course of a career. In our surveys, 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. Programmers keep learning,

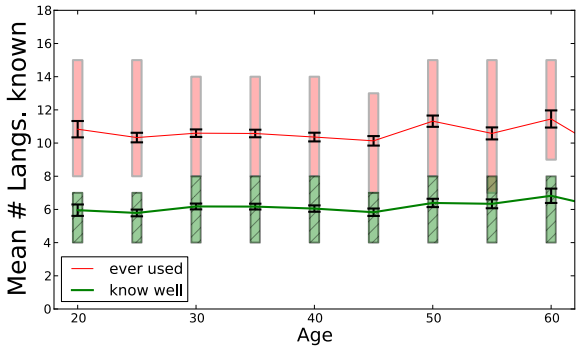


Figure 9: **Number of languages by age.** Developers of different ages seem to know a similar number of languages. Lightly shaded rectangles show 25th and 75th percentiles, error bars show standard error of mean. (Slashdot, n = 1679)

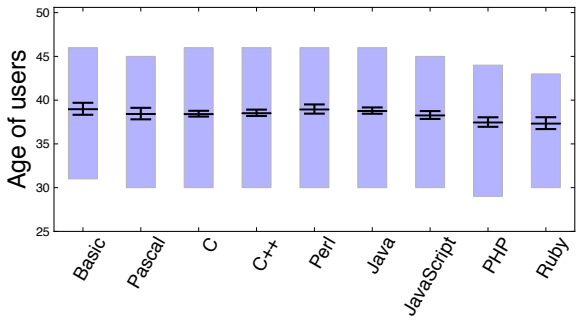


Figure 10: **Mean, standard error, and 25th and 75th percentiles for ages of programmers who claim to know each language.** Languages are sorted by creation date. Distributions are nearly the same for every language. (Slashdot, n = 1679)

and learn often enough and quickly enough that their age does not predict which languages they know.

We now look at the overall number of languages a developer knows. The Slashdot survey 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 mean number of languages known by developers against their age, along with the inter-quartile range. Both lines are remarkably flat. The mean respondent to our survey claims to have learned ten languages, and lists six that they "know well." Likewise, the upper and lower quartile range does not show any clear trend over time; the gap between the 25th and 75th percentile of number of languages does not change over time in a systematic way.

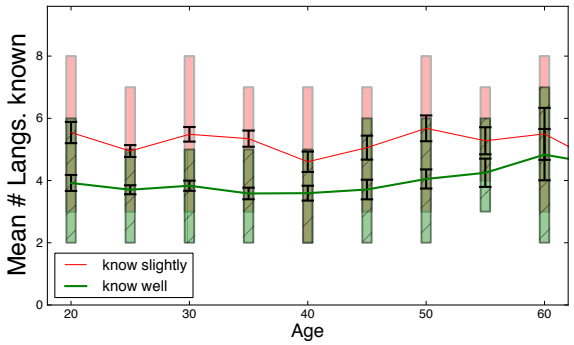


Figure 11: **Number of languages by age.** As with Figure 9; there is no clear trend with age. (MOOC, n = 1142)

The MOOC survey asked developers to list the languages they know well and the number they know slightly. The results are shown in Figure 11. Compared to the Slashdot survey, the average developer in the MOOC survey knows fewer languages. However, the results are qualitatively similar in that there is no age trend. This shows that our result is robust to both varying the detailed wording of the question and to a change in the underlying population.

There is a tension between the flat lines on Figures 9 and 11 and the fact that developers are steadily learning. Since developers of different ages are similarly likely to have learned a language for their last project, we might expect the number of languages they know to rise over time. Instead, older and younger developers report a similar degree of multilingualism. It follows that developers are losing languages as well as gaining them. They are forgetting – or at least, forgetting to mention – some languages.

This result shows that adult developers effectively have a limited capacity for languages. They typically maintain skill in a limited number of languages, and will forget as many languages as they learn. This implies that immediate developer familiarity is a limited resource for which languages must compete. Such competition is the basis for ecological theories of adoption [15].

5.4 Effects of Education

We also looked to see how a respondent's computer science education affected their subsequent programming language knowledge. The Slashdot survey asked respondents to mark which families of languages they learned while in school (e.g., assembly, functional languages, dynamic languages). Table 5 shows our results.

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. Less-popular language families (assembly, functional, and mathematical languages) are more sensitive to prior education. Promisingly, developers who learned a functional or

Language Family	Examples	Overall	For CS majors	Non-majors	Correlation of CS major vs. knowing	If taught in school	If not taught	Correlation of learned in school vs knowing
Functional	Lisp, Scheme, Haskell, ML	22%	24%	19%	0.053 (0.004 - 0.101)	40%	15%	0.262 (0.217 - 0.307)
Dynamic	Perl, Python, Ruby	79%	78%	79%	-0.008 (-0.056 - 0.040)	84%	77%	0.069 (0.021 - 0.117)
Assembly		14%	14%	14%	0.004 (-0.044 - 0.052)	20%	10%	0.138 (0.091 - 0.185)
Imperative/OO	C/C++,Java/C#	94%	97%	90%	0.134 (0.087 - 0.181)	95%	87%	0.133 (0.085 - 0.180)
Math	R, Matlab, Mathematica, SAS	11%	10%	11%	-0.020 (-0.068 - 0.028)	31%	7%	0.268 (0.223 - 0.312)

Table 5: **Probability of knowing at least one language in the indicated family, overall and grouped by major and specific educational experience.** Also shows correlation coefficients between knowing a language in that family and (a) having a CS degree, and (b) having learned a language in that family in school. Whether developers learn a language in that family in school has much more influence than being a CS major. Shown below each correlation is the 95th percentile confidence interval. (Slashdot, n = 1679)

math-oriented language in school are more than twice as likely to know one later than 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. Consequently, the correlation between education and later knowledge is relatively modest.

Notably, having been a computer science major does not lead to the same linguistic versatility of students who learned different language families as part of the course curriculum: we saw no measurable correlation between being a CS major and knowing particular programming paradigms. Our results suggest that an undergraduate curriculum that does not introduce students to a variety of languages is unlikely to result in more versatile programmers later in their careers. This demonstrates a weakness of curriculums that only focus on languages such as Java and Python.

One limitation of this finding is that it measures only correlation, and there could be causation in both directions. Developers who expect to use distinctive families such as assembly languages might choose to study them in school. Demonstrating how much causation flows in each direction is beyond the scope of this paper.

6. Beliefs about Languages

This section turns from programmer actions to programmer beliefs. This section looks at *what attributes of languages do developers like or dislike?* The question is in contrast to that of Section 4, which examines how developers pick languages within the context of a specific project.

Beliefs help shape developer and manager decisions to learn and advocate languages, and thereby affect the social dynamics of adoption. Understanding what developers value, and how developers perceive languages, can help designers both to build better-liked languages and also to advocate more effectively on behalf of their languages.

6.1 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 12. As can be seen, libraries are the top-rated feature. Such perceived importance of libraries cross-validates our finding that libraries matter in practice (Section 4).

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 are little-used in languages without static types.

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

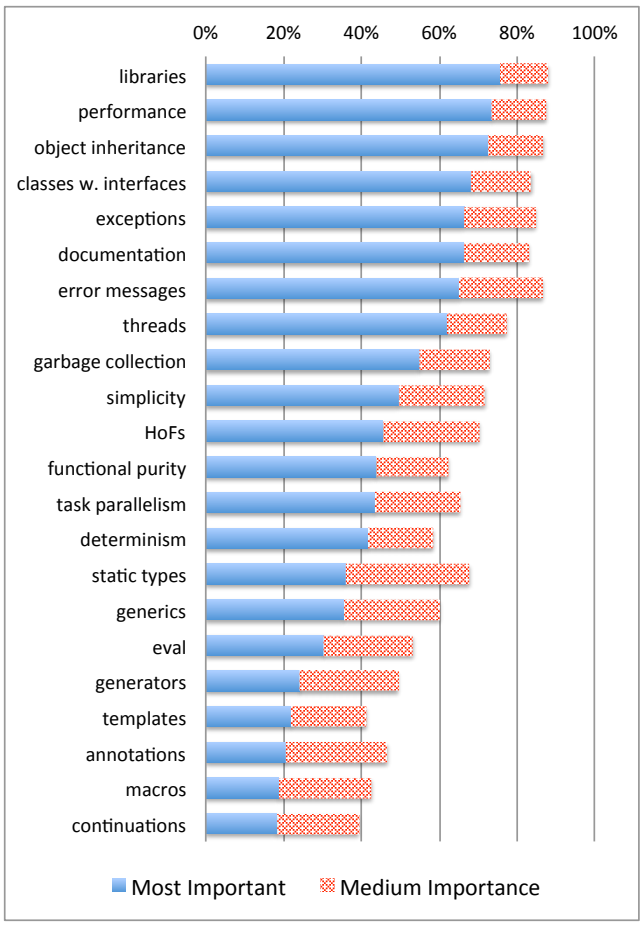


Figure 12: **Feature preferences** (MOOC-d data set, n = 415).

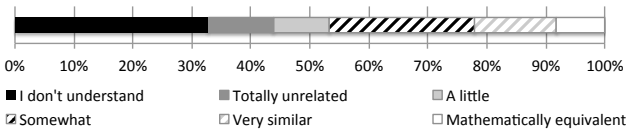


Figure 13: **Higher-order functions and objects.** Most developers do not see a close connection between the two. (MOOC-d data set, n = 415).

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 13. 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 do not report on confusing questions elsewhere; confusion in this case was expected as it is intended to reflect not understanding the underlying language semantics.)

We conclude that a large fraction of developers are disinclined to reason about the semantic power of language features. Features that appear similar to language designers do not appear so to users. For example, implementation inheritance can be implemented atop higher order functions: methods can be replaced by open functions and inheritance can be modeled with 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 observation that many developers do not perceive languages in the same terms as language designers.

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 (mis-)education.

6.2 What Programmers Enjoy

We now turn to the closely related question of what programmers enjoy in a language. This is an open-ended question, and our data is preliminary. The Hammer data set, with its large pool of languages and statements, is the data set available to us that is best suited to the inquiry. The features of a language that developers enjoy are conceptually related to the features developers value, and therefore this question serves as cross-validation for the results of Figure 12.

The Hammer survey asked developers to rank 51 languages against 111 different statements, including "I enjoy using this language." As described in Section 2, we use the Glicko-2 algorithm [8] to totally order all the languages for each such statement [14]. From these rankings, we correlated different statements. Enjoyment is the closest proxy in the data set for what we mean by "liking a language", so we looked for other attributes of languages that correlate with enjoyment.

The statement with the highest correlation with enjoying a language is "This language is expressive" (correlation 0.76). Other highly correlated statements include "I find code written in this language very elegant" (correlation 0.73), and "I rarely have difficulty abstracting patterns I find in my code" (correlation 0.66). While open source libraries significantly influence developer actions, "Third-party libraries are readily available, well-documented, and of high quality" only weakly correlates (correlation 0.10) with enjoyment.

Attributes other than "enjoyment" are also relevant to our inquiry about developer preferences. We examined correlations with the statement "This language has unusual features that I often miss when using other languages." The results

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

are shown in Table 6. Perceived expressivity strongly correlates with this statement (correlation 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.

6.3 Types vs. Testing

We close by examining the perceived tradeoffs around static types, particularly as contrasted with unit testing. Static types are a controversial choice in language design, and we hope our results help show designers how developers react.

The MOOB-b survey included a set of statements about types and testing; respondents were asked to mark if they agreed or disagreed with each. The questions were binary; yes and no were the only options. We present only the results from the self-identified professional developers in the sample. This filters out responses from students, whose experience may have been derived from small classroom assignments, rather than realistic industrial development. This leaves us with 96 responses — small compared to the other populations in this paper, but large enough for meaningful statistical analysis.

Table 7 displays the results. Even despite the large margins of error, the results are striking. Only 36% "see the value" of static types. In contrast, 62% – nearly twice as many – see the value of unit testing. Developers are nearly twice as likely to "enjoy using" unit tests (33%) as compared with static types (18%).

Question	Agreement
Unit testing will reveal bugs that static types miss	31% (+/- 9)
Static types will reveal bugs that unit testing misses	7% (+/- 5)
Unit testing will reveal more bugs that I care about than static types	19% (+/- 8)
Static types will reveal many bugs that I simultaneously care about and are missed by unit testing	6% (+/- 5)
I see the value of static types	36% (+/- 10)
I see the value of unit testing	62% (+/- 10)
I enjoy using static types	18% (+/- 8)
I enjoy using unit testing	33% (+/- 9)
Most of the value of unit testing is in finding bugs	33% (+/- 9)
Most of the value of static types is in finding bugs	8% (+/- 6)
I have used statically typed languages for large or many projects	39% (+/- 10)
I have used unit testing for large or many projects	34% (+/- 10)
Using types helps improve readability	45% (+/- 10)
Using types helps improve safety	44% (+/- 10)
Using types helps improve program modularity	19% (+/- 8)
Using types is generally important, despite the costs	19% (+/- 8)
Using types is rarely important	8% (+/- 6)

Table 7: **Beliefs about types and testing.** Shows fraction of responses agreeing with each statement, and 95th percentile confidence bounds. Results from self-identified professional developers in MOOC-b sample. (n = 96)

Contrary to our initial suspicions, developers describe neither types nor testing to be primarily about finding bugs. Only 8% of professional 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 suspect the survey population was biased in favor of dynamic languages because the course associated with it focused on software-as-a-service and therefore many web technologies written in dynamic languages. Even so, this result shows that there is a population of developers who are broadly skeptical of the benefits of static types.

We cross-validated this result using the Hammer data. We looked for statements in the Hammer dataset that correlated with "This language has a strong static type system." 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." (correlation 0.85). However, languages with static types are much less closely correlated with languages developers claim to enjoy (correlation 0.38) and with expressivity (correlation 0.31). Irrespective of the objective value of static typing, we see evidence that many developers do not value it highly.

The MOOC population is by no means representative of all programmers. The underlying course was taught in Ruby and targeting web application development. However, we believe the results are valid for dynamic language programmers. Even on this restricted population, there are useful take-aways from these results. Our data suggests that developers value the readability provided by types more than they do many other benefits. The data also suggests that developers *do not* find that types improve modularity — it contradicts claims often made on behalf of types. These observations suggest points where future language design research could better meet perceived needs.

7. Threats to Validity

This section discusses the limitations of our work. We begin with threats to validity (whether our results are accurate on the populations we sample) and then consider reliability (whether our results would apply to other populations.)

7.1 Validity

Respondents to the Slashdot survey had the opportunity to explore the Hammer data visualizations before answering the survey. This may have biased them. Because the survey questions are not closely related to the visualized data, we expect that the priming effect and consequent bias will not be large.

Not everybody who starts a survey will complete it. We lack information about respondents who declined to submit. They may therefore differ in demographics, background, or motivation from those whose responses we have.

The methodology used to extract data from the Hammer Project is novel and we do not have a rigorous analysis of the statistical margin of error. The underlying Glicko algorithm is well-documented and widely used, however.

Likewise, extracting data from SourgeForge required making assumptions and judgement calls. Projects can change categories, languages, and authors over time.

Our work is largely cross-sectional, looking at one moment in time. We do answer some questions by examining longitudinal data from the 10 years of SourceForge data and careful phrasing of some of our survey questions. 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 [9] and would help elucidate the adoption process.

7.2 Reliability

Section 3 uses data from the SourceForge repository, which hosts open-source software. It is possible that proprietary code bases are statistically different: for instance, obscure languages might hang on longer inside corporate IT departments. However, open source development is a major part of all development activities and its effects will constrain corporate development. As we showed, open source libraries have a major influence on language selection even within corporate development.

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. Because the MOOC course was focused on web-based applications and was taught in Ruby, we expect that the population will be biased towards dynamic-language users. This is a particular concern for the Hammer data, where we lack any sort of demographic information about respondents.

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 check whether these results hold in other populations. We highlighted points of agreement between our data sources. The overlap in results across our surveys suggest our work does generalize to some extent.

While many of our results are cross-validated with several data sets, some are not: different surveys asked different questions and different data sources include different metadata. The SourceForge data is our only source in Sections 3.2 and 3.3. The observations in Section 4 about the importance of legacy knowledge and code are based solely on the Slashdot data set, as are the results about language learning time in Section 5.1. In Section 6, the MOOC-d survey is our only source for the observation that developers have divergent opinions about semantically-similar features.

Going forwards, we believe it would be valuable to study professionals for whom programming is a significant but not primary job responsibility. For example, engineers and scientists often do not come from computing fields but are still 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. [5]. 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. We increase the scale and fidelity and change the intent to 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. [18] 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 adoption should be studied across languages and, for individual developers, across projects.

Others also mine repositories to understand feature and the API adoption within an individual language or project. Okur and Dig [17] 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 [16] and dynamic code evaluation in JavaScript [20]. 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 [12]. 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 [12, 16–18, 20]. 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 usage questions. Datero and Galup ran a web survey to examine differences in language knowledge by gender [6]. 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 [19]. 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) [7] and the Unified Theory of Acceptance and Use of Technology (UTAUT) [25]. 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 [24]. 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 [10]. 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 [22]. 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 [4]. A subsequent single-organization study looks at the decision of whether to adopt a formal development methodology [21]. In both cases, the researchers ignored the intrinsic technical attributes of the language or methodology in question, and considered social factors exclusively: we examined 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 make comparisons across languages and communities. Furthermore, HOPL retrospectives tend to include deep but anecdotal analysis by language designers, while we perform quantitative data analysis.

9. Conclusions

This paper has looked at programming language adoption through the lenses of four separate research questions: large-scale statistics, programmer decisions, language learning, and general beliefs about languages. These separate lines of inquiry support each other and paint a unified picture

of adoption, with lessons for language designers, advocates, educators, and employers.

In Section 3, we asked what statistical patterns language adoption obeyed. We demonstrated three claims: First, popularity falls off steeply and then plateaus according to a power law. Second, the less popular a language, the more its popularity varies from niche to niche – popular languages are consistently popular across domains of use, less-popular languages tend to have specific domains. Last, developers switch between languages based primarily on the domain and use of a language, not based on its linguistic features (syntax or semantics.)

Section 4 used survey data to show what factors influence developers when picking projects. We found that existing code, existing expertise, and open source libraries are the dominant drivers of adoption. This dovetails with the previous finding: libraries and code are niche-specific, and therefore this developer motivation helps explain the statistical findings above. The fact that developers typically do not consider particular language features important in choosing languages is likewise consonant with our statistical finding that developers do not tend to switch between semantically related languages.

One consequence of these findings, taken together, is that language designers and advocates should emphasize libraries. 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 Rails for web applications in Ruby.

Section 5 asked what causes developers to learn languages. We find that professional developers learn and forget languages throughout their careers, and that as a result, age has little to do with language choice. Some languages are easier to learn than others, and the self-reported ease with which developers learn does not seem closely related to the underlying simplicity of a language's formal semantics. Past education has moderate influence. Having been exposed to a language paradigm in school makes developers more likely to learn or remember similar languages later in their career.

In both Sections 4 and 5, we found that developers demographics strongly differ in their language preferences. We make two observations relating to this occurrence. First, empirical analysis of programs written in the same language may need to check for sample bias due to developer demographics. Second, ecological models may apply to language adoption as we hypothesized in earlier work [15]. Ecological models would predict that languages spread along demographic boundaries because the languages compete for them. Our data reveals that spread patterns exist, and that developers do indeed maintain a limited working set of languages.

Finally, we looked at developer feelings about languages when not tied to particular projects. Libraries still rate highly, supporting the results of Section 4. We also found that developers have divergent feelings about semantically similar languages features. This suggests that experience and training shape developer language perception, underscoring the results in Section 5 about the importance of education. We find that developers consider ease and flexibility to be more important than correctness. 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 emphasize the benefit of types in understanding programs, suggesting one benefit researchers can build upon.

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, which is less actively practiced than in software engineering. We found survey methods to be a powerful tool for exploring hypotheses about language adoption. We suspect these methods will be increasingly valuable going forward, especially given the popularity of the Internet and online courses. Likewise, we hope our data, and the methodology subtleties we encountered in gathering it, will support future analysis efforts. Of particular note are our large set of responses, tracking of respondent demographics, and solicitation of data about concrete languages and projects.

We have examined basic questions about language adoption. Going forward, there are many more questions about the sociotechnical nature of programming languages [15].

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. Anonymous reviewers significantly helped revise earlier drafts. Some of our anonymous reviews were among the most careful and constructive ones that we have ever received.

References

[1] Ohloh, the open source network. `http://ohloh.net`.

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

[3] Tiobe index. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`.

[4] R. Agarwal and J. Prasad. A Field Study of the Adoption of Software Process Innovations by Information Systems Professionals. *IEEE Trans. Engr. Management*, 47(3), 2000.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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)*, 2010.

[10] B. C. Hardgrave and R. A. Johnson. Toward an information systems development acceptance model: the case of object-oriented systems development. *IEEE Trans. Engr. Management*, 50(3), 2003.

[11] Q. Hardy. Technology workers are young (really young). `http://bits.blogs.nytimes.com/2013/07/05/technology-workers-are-young-really-young/`, 2013.

[12] 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)*, 2011.

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

[14] 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)*, 2012.

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

[16] 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.

[17] S. Okur and D. Dig. How do developers use parallel libraries? In *Foundations of Software Engineering (FSE)*, 2012.

[18] C. Parnin, C. Bird, and E. Murphy-Hill. Java generics adoption: how new features are introduced, championed, or ignored. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR)*, 2011.

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

[20] 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)*, 2011.

[21] 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.

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

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

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

[25] 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.