How Not to Survey Developers and Repositories: Experiences Analyzing Language Adoption

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

Ariel Rabkin
Princeton University
asrabkin@cs.princeton.edu

Abstract

We present cross-sectional analyses of programming language use and reflect upon our experience in doing so. In particular, we directly analyze groups of 1,500-13,000 developers by using questionnaires and 260,000 developers indirectly so by mining 210,000 software repositories. Our analysis reveals programming language adoption phenomena surrounding developer age, birth year, workplace, and software repository preference.

We find that survey methods are increasingly accessible and relevant, but there are distinctive problems in examining developers and code repositories. We show that analyzing software repositories suffers from sample bias problems similar to those encountered when directly polling developers. Such bias limits the general validity of research claims based on analysis of software repositories. We aid future empirical researchers by describing concrete practices and opportunities to improve the results of developer and software repository surveys.

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

General Terms Languages, Human Factors

Keywords sociology, programming languages, surveys

1. Introduction

The programming language design community largely focuses on technical aspects of languages: how to efficiently implement a language, how to automatically reason about a program written in one, and how to prove properties, such as type safety, about the language itself. However, programming is about more than technical aspects. Software development is a human process carried out in a social context, and psychological and sociological factors can make the difference between a successful language and an unsuccessful one. In earlier work [9], we called for the programming language research community to devote more attention to analyzing and exploiting the social processes that surround language adoption. Here, we make a first step in achieving this.

There has been increasing interest in the social and psychological aspects of programming. Small-scale user studies are increasingly common, and the mining software repositories community even has its own long-running conference series. Many outside fields rely on large-scale cross-sectional surveys, such as telephone polls for economics. Such surveys are used throughout the technology industry today, including for software engineering research, but only infrequently as a programming language research technique. With the popularity of programming and the rise of the Internet, it is now relatively easy to do large-scale cross-sectional surveys of developers. Should we, and if so, how?

For the last two years, we have been gaining experience conducting and analyzing mass surveys of developers and repositories. Our ultimate goal is to understand the adoption process for programming languages and individual language features. This paper reflects on the research methods we used, and particularly their strengths, weaknesses, and some of the pitfalls we encountered. Our audience is programming language and software engineering researchers who might wish to pursue similar questions or methods.

In particular, this paper analyzes three topics about methodology:

- **Analyzing language adoption through big, sparse, and high-dimensional data sets.** Many of our analyses are based on such data data. Machine learning algorithms and interactive visualizations were our two main and complementary techniques for extracting research hypotheses. Furthermore, we had to take care to track uncertainty through both.
- **Survey design.** The wording of questions biases results. Researchers and practitioners do not always share the same vocabulary and basic understanding of topics, so surveys must compensate for this difference. We discuss

specific cases of the problem that we encountered and our experiences in using careful phrasing, pretesting, and soliciting free-form responses to combat it.

- **Sources of sample bias in developers.** We show how developer demographics shape results of developer surveys. We also find demographic biases in software repository surveys. Put together, these highlight a methodological dangers that weakens the generality of existing results reached by mining software repositories. Given the increasing popularity of repository mining, bias is a widespread concern.

Overall, we believe surveys are an effective methodology – if used correctly. We end on a promising note by describing emerging opportunities for performing effective survey research on programming languages. For example, while traditional student surveys insufficiently sample developers, massive open online course surveys can do better.

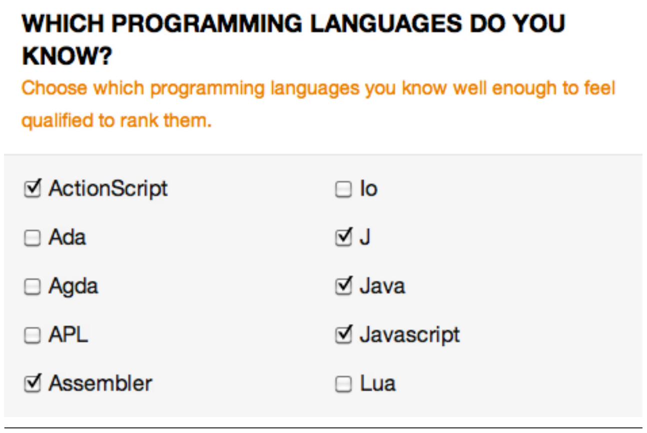
We examine four large surveys in this paper. The first section begins by discussing our reliance upon machine learning and visualization techniques for exploring a two-year survey ("Hammer") of comparative statements by programmers about languages. Hypotheses formed by interacting with the visualizations led to two further surveys. First, we arranged for adoption questions to be included as part of the entry survey for Berkeley's Massive Open Online Course (MOOC) in software engineering for software as a service [4]. Contemporaneously, our initial visualizations attracted significant social media attention, which we used to collect responses for another mass survey. As many respondents visited from the Slashdot website, we refer to this survey as the Slashdot survey. The Hammer, MOOC, and Slashdot surveys rely upon self-reporting, so we also examined 10 years of recorded activity for all projects in the SourceForge software repository.

We found a number of challenges specific to surveying programmers. Section 3 discusses how challenges in wording questions surface for this community. Following that, Section 4 looks at demographic issues.

Our conclusion is three-fold. First, surveys are a powerful but rarely utilized research instrument for basic questions in programming languages and software engineering. Second, there are many emerging and underutilized opportunities for performing surveys. Finally, survey methodologies from other fields apply to our own. We especially stress analyzing and controlling for demographics before reporting conclusions about surveys. This warning applies irrespective of whether they are directly of developers or indirectly through software repositories.

2. Hammer: Sparse High-Dimensional Data

"The Hammer Principle" is a website by David MacIver that invites readers to compare programming languages based on a series of metrics [7]. He (graciously) provided us with



WHICH PROGRAMMING LANGUAGES DO YOU KNOW?

Choose which programming languages you know well enough to feel qualified to rank them.

[✓ ActionScript]	Io
Ada	[✓ J]
Agda	[✓ Java]
APL	[✓ Javascript]
[✓ Assembler]	Lua

Figure 1. The selection phase of Hammer Data gathering

anonymized survey results. The survey is of particular interest because of its scope: it provides a significant amount of data about how developers compare languages according to properties such as correctness, speed, simplicity, job prospects, and enjoyment.

Analyzing this data helped us spot possible topics to investigate. We describe our technique here, both because we think our software will be useful to other empirical researchers and because the methodology may be of use for other domains with sparse comparative data.

Our approach was to analyze the raw data using a combination of machine learning algorithms and then examine the processed data through interactive visualizations. After the up-front investment in building the software for these steps, combined, they enabled rapid formulations and investigations of hypotheses. Tracking uncertainty through the techniques helped avoid incorrect conclusions.

Our processed data and interactive visualizations are available online[1]. We believe they can help other researchers frame hypotheses beyond those already in our own work.

2.1 The raw data

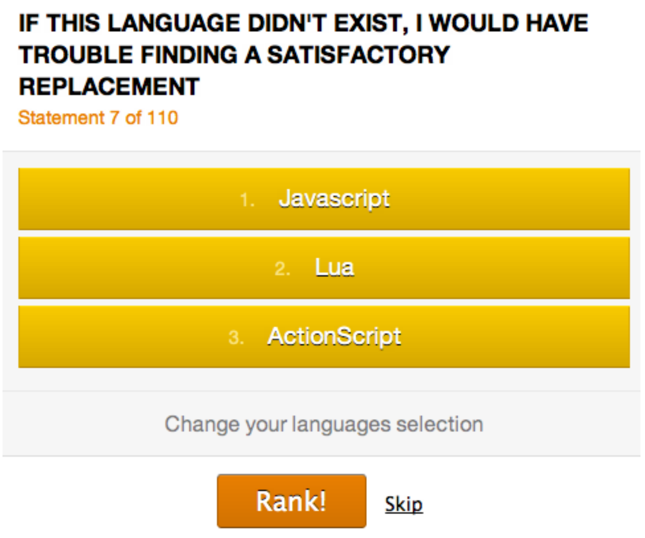
Over two years, respondents came in bursts from popular online sites such as Slashdot, Hacker News, Reddit, and Lambda the Ultimate [8]. They went through the following process:

1. **Pick languages** Respondents picked a set of languages that they "know well enough to feel qualified to rank" out of a pool of 51 (Figure 1). The average respondent picked 7 languages.
2. **Rank languages by statement** Respondents were shown a series of statements. For each statement, a respondent ordered the previously selected languages based on how well they matched the statement (Figure 2). The average respondent answered 10-11 questions.

[1] `www.eecs.berkeley.edu/~lmeyerov/projects/socioplt/viz/index.html`

Survey	Conducted by	Description	Scope
Hammer	David MacIver	Survey of developers about 50 languages and comparing 100 properties about them. No demographic information maintained.	13000 people 2 years
MOOC	David Patterson and Armando Fox (course instructors), with advice from us	Entry survey for a Massive Open Online Course in software engineering for software as a service.	1100 people
Slashdot	the authors	Survey posted by us to understand audience from Hammer visualization	1600 people 2 weeks
SourceForge	SourceForge	Project descriptions from a massive open source software repository	10 years 200000 projects 260000 people

Table 1. Data sources mentioned in this paper


IF THIS LANGUAGE DIDN'T EXIST, I WOULD HAVE TROUBLE FINDING A SATISFACTORY REPLACEMENT

Statement 7 of 110

Figure 2. The ranking phase of Hammer Data gathering

To date, over 13,000 people have filled out the survey. Individual responses are sparse and often contradictory, so we use machine learning algorithms to extract reliable information. The data set is high-dimensional (110 statements about 50 languages), so we constructed several interactive visualizations to explore the results of the statistical analysis.

2.2 Ranking algorithm

The Glicko-2 ranking algorithm computes the data for our first visualization. The visualization shows how well each statement describes each language (Figure 3). This provides a human-understandable description for each language and reveals languages that rank similarly – or fail to – based on certain properties.

The theory of preference aggregation and voting tells us that there may be no unique way to order languages by how well that statement describes them. (Imagine there are only two responses and one dimension: one respondent may rank A $>$ B and another B $>$ A.) Beyond this theoretical limit is a more particular problem: the Hammer data is sparse for unpopular languages.

The Glicko-2 ranking algorithm is designed for this sparse and inconsistent scenario [5]. It is commonly used in sports: Glicko-2 generalizes the original Elo rating system and underlies XBox's TrueSkill online player rankings.

Suppose a respondent ranks a few languages for some statement X. The languages might be given order A $>$ B $>$ C. We treat this as the outcome of 3 different matches between "players" A, B, and C in "sport" X, where A $>$ B, A $>$ C, B $>$ C. While respondents only looked at a total of 140,000 statements, these sequences expand into 4,000,000 pairwise comparisons. This was enough data for Glicko-2 to report high confidence on most rankings.

Glicko-2 proceeds as a simulation. A weak language beating a strong language gives the weak one a big boost in score and the strong one a drop, while there is little change in ranks from a strong language beating a weak one. Time and contention is factored in by tracking the deviation across matches: an occasional upset is disregarded, but if the upsets become consistent (e.g., a language was upgraded), the rank will converge on the new value. Likewise, high disagreement about a statement is reflected by a high deviation.

2.3 Interactive Visualization of Language Rankings

We used the results of the Glicko-2 analysis to build an interactive visualization. It is a form of heat map for exploring the matrix of language vs. statement (Figure 3). Each row shows how one language ranks relative to all others according to a series of statements. The size and color of a circle both indicate the rank of agreement. For example, a big green circle shows that most programmers agree that a language matches the statement better than other languages. In contrast, a small red circle shows most programmers agree the language matches the statement less than other languages. A language's row, in effect, is a fingerprint for quickly com-

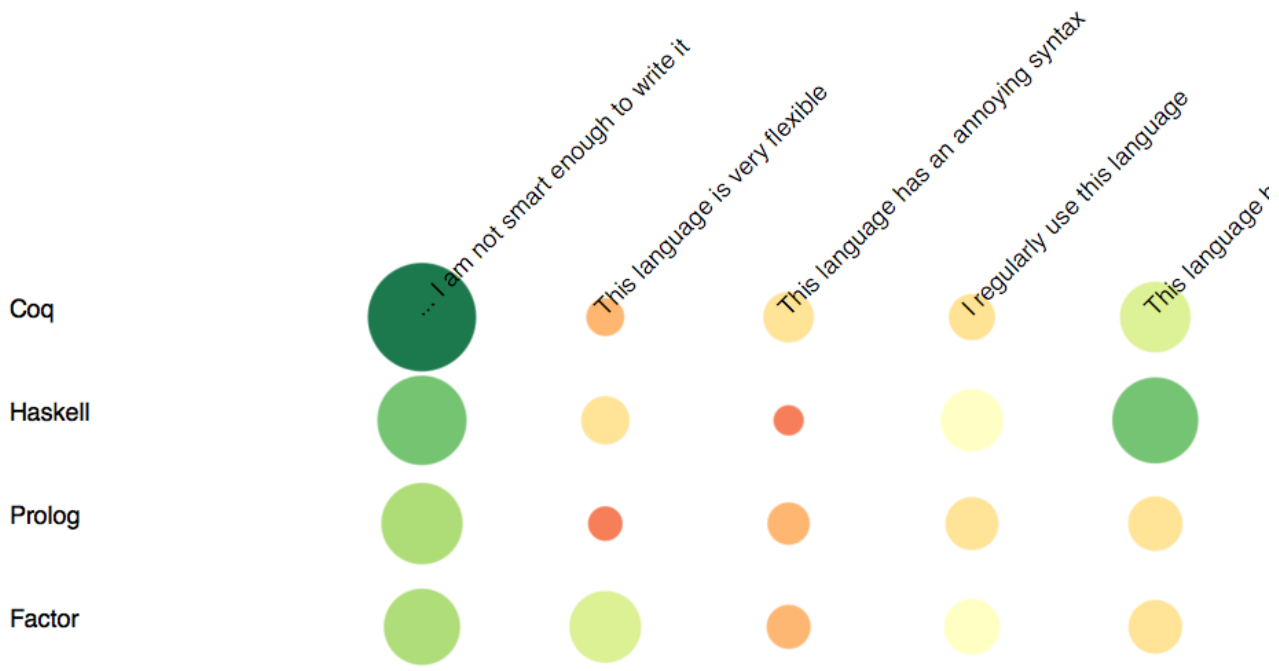


Figure 3. Interactive visualization of language rankings. Current state shows a filter for 4 particular languages across 5 particular statements sorted by match against the statement "I often feel like I am not smart enough to write this language." The clipped statement is "This language has unusual features that I often miss when using other languages."

paring languages. Circles different from their neighbors are generally of interest, as are particularly small or big ones.

Clicking on the axis of languages sorts them alphabetically, and each language and statement has a clickable toggle controlling whether it is shown. Clicking on a circle will simply sort the languages by how well they match the statement described by the circle. While simple, these three interactions enable the exploration pattern of focusing on a statement or language, finding an unusual dimension in it, and then seeing how other languages and statements compare along it.

For example, we looked up Coq's fingerprint and found that the strongest statement programmers make (the biggest circle in the row) is that they do not feel smart enough to write in it. Curious about how Coq relates to other languages in this problem area, we clicked on the circle. Figure 3 shows that the two languages that most often prompt these feelings of inadequacy are Coq and Haskell, followed by, with nearly equal reservations, Prolog and Factor. Visual inspection quickly revealed programmers believe these languages are inflexible, have an acceptable syntax, and they infrequently use them. Except for Haskell, the languages did not have particularly desirable features.

The visualization also led to tweaking our ranking algorithm. In particular, we found REBOL to rank high in categories that we did not expect. The basic reason is that it had a high deviation; it ranked highly but with little confidence. This can be due to conflicting responses or overly sparse data. Our solution was two-fold. First, we decrease the strength of contentious results by ranking based on "`rawScore` $- 3 * \sigma$", similar to XBox player rankings. Second, we visualize contentious statements using translucent circles. Hovering over a circle shows both the raw score and its deviation.

2.4 Correlations and Clustering

As our research interest was more about general language phenomena than particular languages, we next analyzed the correlations between different statements. We treat one language's final statement rankings as a single high-dimensional observation and measure the correlation coefficients across statements. Finally, we reused the interactive heat map for navigating the matrix of statement correlation coefficients.

We found many surprises. For example, we investigated correlations with the desirable statement that "Third-party libraries are readily available, well-documented, and of high quality."At a coefficient coefficient of 0.10, libraries only weakly correlate with static type systems . This conflicts with long-held beliefs about the nature of modularity by the functional programming community [6]. We saw that, instead, the most strongly correlated statement is that "there are many good tools for this language." Tools are weakly and *negatively* correlated with correctness, which is the strongest correlation with static types. Libraries and tools do not require functional programming nor types. A mystery arises, however. Tools and types are both strongly correlated with debugging and maintenance: if tools and types share similar benefits, why is only the former correlated with libraries?

Another surprise is that terse languages anti-correlate with annoying syntax: given long-standing critiques of "write-once" languages, we expected the reverse. This case is also interesting in that, as language researchers, we were more actively seeking semantic phenomena. The visualization highlighted surprising phenomena about a topic we did not think to consider. For both the weak reuse of typed code and legibility of terse code, the visualization highlighted properties that dispute communal wisdom and phenomena that we would have otherwise overlooked.

Finally, we computed the k-means clusterings of statements and languages. We use this both to expose further relationships and to simplify the earlier visualizations.

Consider analyzing the first cluster of statements shown in Figure 4. The numbers indicate distance from the center of the cluster: the average distance of 3.3 is good. The first and last statements about helpful conventions and dogma pairing together are unsurprising: social conventions often arise to solve problems. Clustering the second statement about simple debugging with the others about convention is surprising. Norvig [10] and others view patterns and other conventions as symptoms of linguistic defects, yet programmers rank languages with them as easy to debug, which is not a defect. Language-level research for incorporating patterns may therefore be a case of the *streetlight effect*, focusing on what is already known and ignoring what is not. Clustering helps form hypotheses.

A subtlety of the clustering is that, for the language rankings used to cluster statements, the languages did not have to rank highly for the statements to go in the cluster. They could be similar at any value, as long as it is consistent. The visualization therefore shows which languages support the cluster, and their rank for the center-most statement. Not shown, for the above clustering, languages within 5% of the center have average rank 43 with standard deviation 10. The rank is not high relative to other statements – languages are generally not considered overly dogmatic – but on the scale of dogma that languages exercise (according to the ranking visualization), it is.

We also used the clusterings to improve the original ranking visualization. Showing the matrix of all 50 languages and 111 statements was overwhelming and slow. Instead, by default, we only show the center-most items from the statement and language clusters. For example, of the statements in Figure 4, we would only show the first about conventions and fourth about well-organized libraries. If a user wants to explore a particular family of languages or statements, the clustered features can be expanded.

3. Framing the Right Questions

The Hammer Principle data was illuminating, but it left us eager to verify our results by direct survey methods. We suggested the inclusion of some questions on the routine course entry survey for the Berkeley Massive Open Online Course in engineering Software-as-a-Service applications and obtained access to the results.

3.1 Names for language features

Software development is a technical field, and therefore has a technical vocabulary. However, researchers and practitioners do not always share the same jargon. As a result, care in wording questions is important.

One question on the MOOC survey asked developers how often they created their own generic classes in Java. (A generic class is one that has a type parameter, such as `class Foo<T extends Collection>`.) In our sample, 40% said they did this often or sometimes. This result is hard to believe, since a survey of existing code by Chris Parnin et al. found that only 14% of developers are actually responsible for doing so [11]. (Anecdotal evidence and our personal experience agrees with the study)

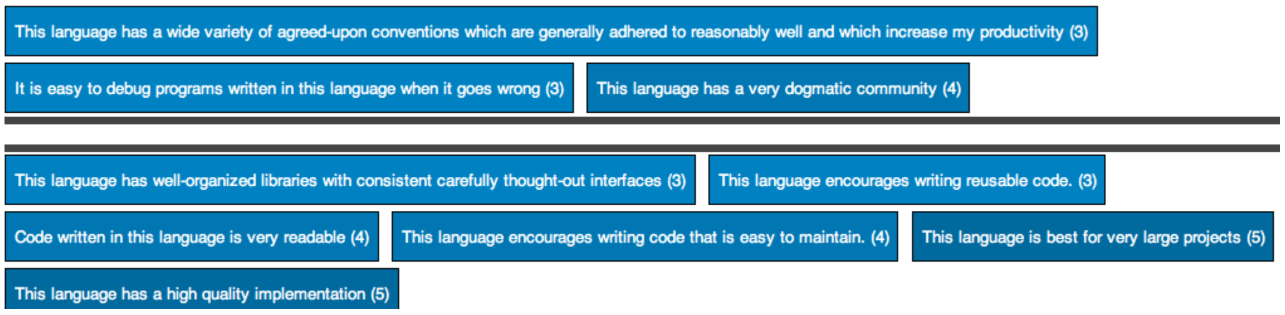
There are two possible interpretations of this dichotomy. It may be that professional developers work very differently from open source developers. We suspect, however, that developers may have misunderstood the question. We had separately asked developers about *creating* generic classes and about *using* generic classes, such as instantiating the Java standard library `ArrayList<T>`. We think this distinction is not as clear to developers as it is to us.

For another example, consider the concept of determinism, that a piece of code should produce the same behavior each time it is invoked. We asked developers how important this was. We found that 35% of developers (117 out of 335) responded "don't know or no opinion." We are unsure, from this data, whether programmers truly have no opinion, or whether the word is unfamiliar to them. To mitigate this problem, surveys should be careful to define constructs and examine the level of understanding by respondents.

3.2 Knowing a language

A basic question about programming language adoption is how many languages developers know. We asked this question in different ways on different surveys. The Slashdot survey (Figure 5) asked developers to list the number of languages they knew well, and separately, to estimate the total number of languages they know. Our concurrent submission has more details about developer language acquisition over time. Here, we focus on ambiguities and limitations of the survey method.

As can be seen, there is virtually no age trend; developers of all ages list an average of six languages they know well, and claim to have ever learned about ten. The lack of change might suggest that developers learn their languages early, and never learn more. However, we have separate counter evidence from the same survey that developers routinely do learn languages: old developers are as likely as young developers to know newer languages, such as Ruby. We


This language has a wide variety of agreed-upon conventions which are generally adhered to reasonably well and which increase my productivity (3)

It is easy to debug programs written in this language when it goes wrong (3)

This language has a very dogmatic community (4)

This language has well-organized libraries with consistent carefully thought-out interfaces (3)

This language encourages writing reusable code. (3)

Code written in this language is very readable (4)

This language encourages writing code that is easy to maintain. (4)

This language is best for very large projects (5)

This language has a high quality implementation (5)

Figure 4. Interactive visualization showing two statement clusters from a k-means analysis.

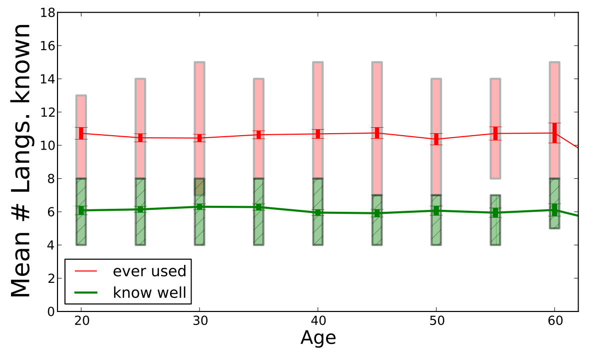


Figure 5. Developers of different ages seem to know a similar number of languages. Lightly shaded rectangles show 25th and 75th percentiles, darker solid bars show standard error of mean. (Slashdot) Questions were "What programming languages do you know well? List the ones you know best in the beginning." and "About what number of languages have you used in your life?"

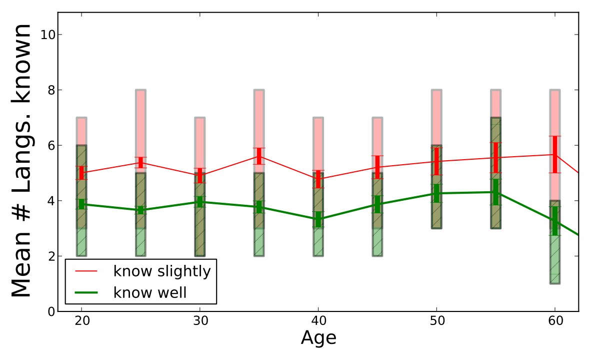


Figure 6. Same format as Figure 5, but for MOOC survey. Question wording was: "List the languages in which you are proficient, starting from ones in which you are most expert" and "Which other programming languages have you used at least once or twice? [with a list of checkboxes and a free response box]"

suspect that developers are forgetting languages, or at least, forgetting to mention them. The conflicting results suggest that asking developers to estimate the number of languages is not a reliable technique.

We tried using explicit prompts, and even checkboxes for specific popular languages, on the MOOC survey. This does not seem to have prevented the forgetfulness bias.

We may be encountering two broader methodological problems. First, we are unconvinced that different developers interpret "knowing" a language the same way. There must be some minimum level of mastery or comfort before a developer will claim to "know assembly." Is this level constant for developers of different experience levels? Are senior developers more or less assertive than junior developers? Second, there may be effects from memory and engagement. Showing a list of languages may help remind respondents of forgotten languages, but may cause fatigue. (We showed lists of languages next to all the language-count questions, for both surveys. The MOOC survey had explicit checkboxes, plus a free response field.)

To close with a more positive note: The MOOC survey also asked developers about the languages they know. There, developers were asked to list the languages they knew well, and to separately list the languages they knew slightly. The results are shown in Figure 6. As can be seen, the result is broadly similar to that of Figure 5. The data is has more variable and has wider distributions, which may be due to the fact that MOOC students are a highly varied population composed of several constituencies. Despite the noise, we see that there still is no clear trend over age.

This similarity suggest that the methodological limitations we highlight are not necessarily fatal. Results do appear to be reproducible, across populations and across slight variation in the way questions are asked. However, even with something as simple as the number of languages a developers knows well, we already see sources of variability and, depending on phrasing, get a 50% difference in the average answer.

3.3 Techniques for improving question quality

We used two primary techniques to limit misunderstanding: pretesting and including free-response questions. This subsection discusses these techniques.

Pretesting We pretested questions in several ways.

We started by discussing the intended survey topic with academic and industrial language designers, both in groups and individually. This was especially helpful for soliciting topics to analyze and, for concrete questions, possibilities to consider. We would show the survey to respondents and ask them to vocalize their understanding of each question and to explain their answer. For example, we had initially asked respondents with college degrees how many years it had been since they left school. Pretesting caught a bug in our thinking – some students graduate, spend time in industry, and then return to school. We rephrased the question to account for cases like this. Finally, we ran pilots of the surveys. This caught formatting and survey software issues as well as helped create surveys of an acceptable length.

Free response To detect if a bad question slipped through pretesting, we routinely offered respondents a free-response field and also invited them to point out any questions they found confusing.

Free-response answers revealed several points of confusion. For example, one question asked if respondents majored in "computer science or similar field." Some respondents marked no, and indicated that they majored in "software engineering". The next time we asked that question, we were clearer about what counted as related. This is important because, in concurrent work, we found that education influenced language use.

Another example of a mistake caught by free-response is that the MOOC survey asked about college experiences. Two-thirds of responses to the MOOC survey were from outside the United States, and as one commenter mentioned, not all countries use "college" to denote the same institutions. This is a mistake that we could not necessarily have caught by in-person pretesting with local students and developers.

4. Asking the Right People

Above, we analyzed challenges in finding *what* to ask. Here, we turn to the question of *who*.

4.1 Hobbyists versus professionals

Not all developers are alike. The same word might describe hobbyist users of Visual Basic, semi-skilled PHP developers, domain experts in scientific computing, and expert distributed systems programmers at a large Internet services company. Asking about "the average developer" or "the true population statistic" requires picking out some subset of the people who have ever programmed and defining them as the population of interest.

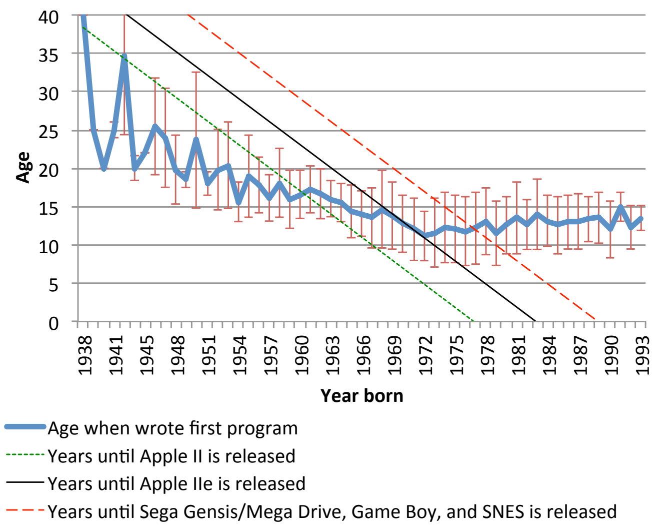


Figure 7. Age at time of first "Hello World" over time. Ages decrease until people born in 1972, after which ages climb until leveling in 1982.

	MOOC	Slashdot
CS major	53%	55%
Professional Developer	62%	92%
Male	84%	97%

Table 2. Demographics from MOOC and Slashdot surveys

On our Slashdot survey, we asked developers how quickly they learned the language they used for their last project. We found that developers learned faster for hobby projects than for work projects. For work, only half the users learned within three months whereas it was over 60% for hobby projects.

One explanation might be that developers work harder on their own projects. Another explanation is to observe that not all developers will program for fun. Developers who program as hobbyists are likely to be biased towards the developers who enjoy programming more, and who may consequently be better at it.

This shows that professionals and hobbyists are not interchangeable. The hobbyists may be drawn from those developers who are the most fluent and capable. A consequence of this is that open-source development, with a heavy hobbyist contingent, may not be a reliable proxy for closed-source development, which is usually conducted by professionals. This is a threat to the generality of research that tries to extrapolate from open source to all kinds of programming.

4.2 Other demographic challenges

There are other demographic biases introduced by cross-sectional surveys. We show demographic information about MOOC and Slashdot in Table 2. As can be seen, the MOOC respondents are mostly professional developers, and are 84%

male, which corresponds roughly to the unfortunate demographics of our field. The Slashdot respondents were 97% male. This does *not* correspond to the demographics of programmers. It does not even correspond to the demographics of Slashdot, which is only about 63% male[12]. Gender bias has definite consequences. In contrast, 42% of female respondents in the MOOC sample are professional developers, and 65% for male respondents.

We suspect that mentioning a survey via social networking and attaching an impersonal request for responses will bias towards the most opinionated and self-confident respondents. These will skew male. Explicitly asking every member of a population to respond to a survey (the MOOC methodology) mitigated the male bias, and we suspect, the opinionated-respondent bias.

The gender example is significant in another way. None of our research questions directly concerned gender. Instead, the fact that our responses are biased by gender gives us a warning that there are other biases creeping in. The question serves as something of a warning light for other methodological limitations. Because we do not know the actual cause of these biases, we cannot compensate with techniques such as regression.

Demographics change over time. In Figure 7 , we show how the average age of a developer's first programming experience has changed over a 55-year window. Unsurprisingly, it falls off as computers become more prevalent. More surprisingly, the trend has several inflection points in the 1980s. We very tentatively think these correlate with changes in popular computer equipment, but verifying this is left as future work.

On a more optimistic note, we see that the age of first programming has stabilized. If the society of programmer stabilizes, both cross-sectional and longitudinal studies will have more predictive utility.

4.3 Sample bias in work environments

In the Slashdot survey, we examined demographic influences on language selection. Respondents were asked, for the language they used on their last project, what factor mattered most. Reported in our concurrent work, we found that social factors, such as open source libraries and familiarity, mattered most. Correctness, commercial libraries, simplicity, and language features were the least important. However, the results varied between different subsamples.

One of the most noticeable phenomena was considering workplaces of 1 employee, 2-4, 5-9, 10-19, and 20 or more. We sliced on finer granularities for large companies, such as 20-100, but the graduations were less significant. Essentially, up to a threshold, the bigger a company, the more social factors influence language preferences. For example, development speed matters the most for companies of 2-4 employees. The size of the organization developing code influences the decision-making of those writing it.

Demographic effects are not always obvious. For example, managers may have different priorities than their underlings. As expected, we saw an increase in social concerns for managers, though it was slight. However, we then compared managers against other respondents of the same age, and the differences disappeared. Older employees, not just managers, prioritize social factors slightly more than other employees. That is not to say being a manager does not matter. For example, managers generally started programming later than their peers, with the gap only closing at the end of high school. For age, we saw the most important distinction was being under 20 versus over 20. Finally, we note that we are not just characterizing software development managers but ones that read Slashdot and Wired.

We see that, not only will employment bias basic results, but so do factors such as age, job role, and the nature of the employer.

4.4 Sample bias in open source repositories

Early in our work, we analyzed the SourceForge software repository to quantitatively analyze actual language use. We encountered substantial sample bias, some of which we discuss here.

We found that the population dynamics on SourceForge changed over time. Figure 8 illustrates several trends by showing, for each language, which months it was most used in. The rise (and fall) of each language varies. For example, D spiked in 2007, and Scheme plummeted in early 2009. We also found that the best single predictor for the language of a project is the language used on the previous one, which is true 30% of the time. The changes in popularity and the tendency to reuse a language, together, mean that the year in which a project is developed in a language reveals artifacts about the developer. Sampling from popular years vs. unpopular ones may correspondingly sample experienced and unexperienced language users. *The year a project is written in a particular language matters in understanding the type of developer.*

Significantly, overall use of SourceForge waxed and then waned. The middle of 2006 began a long-running boom that began to collapse in 2009. A major reason is the rise of alternative repositories. GitHub launched in April of 2008, reached 40,000 repositories by early 2009 (25% of the shown SourceForge projects), and 1 million total repositories by July of 2010. In April of 2011, it hit 2 million repositories.[2] Bitbucket also launched in 2008, and Google Project Hosting has been available since at least 2007. These repositories vary by language, license, and version control system for the hosted projects. *The repository selected for a project matters, as does the point in time when the data is collected.* For example, projects in 2009 can be labeled

[2] https://github.com/blog/455-100-000-users, https://github.com/blog/936-one-million, https://github.com/blog/841-those-are-some-big-numbers

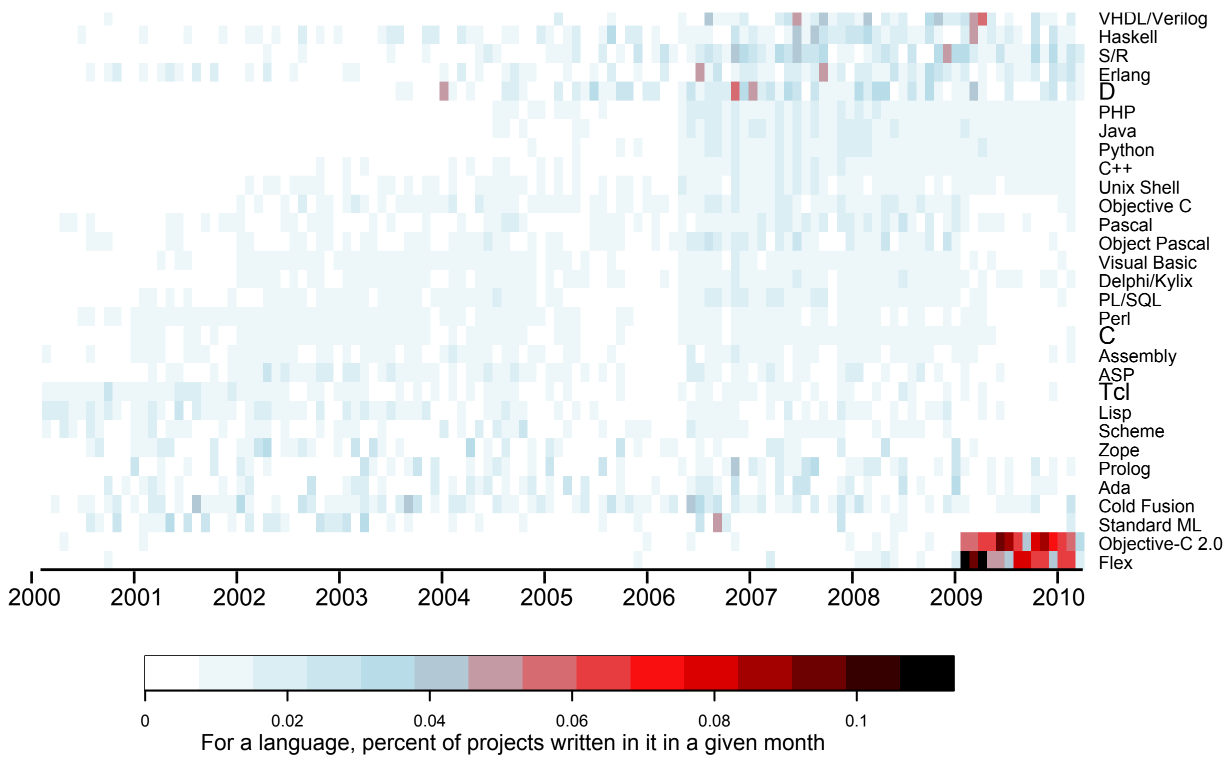


Figure 8. Relative language use over time. (SourceForge data set). Each row is independently normalized to itself. Languages with fewer than 100 projects are elided, and for space limitations, as are 20 of the remaining.

as belonging to early adopters based on which repository is selected.

Some languages, such as Perl and Python, have language-specific code repositories. Projects not in those repositories may be more likely to be created by an estranged language user, and therefore one who deviates from linguistic norms.

We see that the language, repository, year, and developer history are all considerations when analyzing an open source project.

5. Related Work

Surveys are commonly performed for software engineering research, but much less so for understanding programming languages.

Ambitiously, Chen et al [2] model programming language adoption via statistical regression. A web-based survey of developers was used as ground truth for the experiment. Unfortunately, they include no information about the demographics of the surveyed population — or even about the sample size.

There has been some experience with targeted surveys of developers within particular organizations. A 2000 study by Agarwal et al. looked at factors that explain how eagerly COBOL programmers at a large financial-services company learned the C language [1]. Riemenschneider et al. looked at how programmers inside an organization do or do not adopt a formal development methodology [13]. Both studies had comparatively small samples sizes – 71 and 128 developers, respectively.

As mentioned, mining software repositories has become a standard technique in software engineering research. A whole conference series exists on the topic. Zeller et al. offer a methodological critique of such techniques [14]. They emphasize the importance of grounding studies in predictive theories and of using sensible abstraction levels. We are focused on a different class of methodological error: those due to inappropriate generalizations from particular data sources.

For a substantial bibliography on sociological research on programming language usage and adoption, we refer the reader to our earlier work [9].

6. Conclusion: A Delicate Opportunity

Throughout this paper, we have presented quantitative results about programming language use, from tools benefiting library building more than types, to terse languages being legible, to social factors most influencing on language selection and even more so at large companies. Surveys, both sent directly to developers and indirectly gathered through repositories, have a lot to teach us about the nature of programming languages.

However, as analyzed in this paper, using surveys as a research instrument is challenging. Inappropriate use will limit the generality of perceived results. We discussed three common challenges to our work, and how we overcame them:

Sparse, contentious data Especially when used for initial qualitative analysis, data sets will be big, sparse, and high-dimensional. Machine learning helps find relationships, but care must be taken to track certainty. Time taken to build interactive visualizations is well-spent as it yields a tool for building hypotheses and checking any analysis.

Phrasing We found that there was significant risk of framing questions in ways that developers could not easily understand. Terminology should be assumed suspect. Pretesting helped us reduce this risk; we found that the larger and more diverse the pretest audience, the bigger the benefit. Errors will still happen, such as cultural ones due to the global nature of modern software development. Free form answers provide outlets for qualitative feedback, and critical to survey instrument quality, will reveal errors. Manually reading feedback from thousands of respondents is not always fun, but it is important.

Sample bias is a problem with most surveys: different demographics have a different probability of participating, and no recruitment strategy is perfect. Instead of focusing just on preventing bias, we found it important to also detect when and how it invariably happened. Demographic questions enable the use of regression to account for some forms of bias, and at a more fundamental level, expose that a bias even exists. We show how sample bias influences analysis, *including the popular research technique of mining software repositories.*

We suspect that programming language and software engineering research will be increasingly able to benefit from survey data. Thanks to the rise of massive online courses that target adult practitioners, it is increasingly feasible for the academic community to directly survey large numbers of professional developers. Specific language communities are increasingly receptive to surveys as well. For example, there is an annual survey of developers working in the Clojure language [3], and similar efforts occur in the Scala community. Popular languages have supporting communities, and we found community leaders to be supportive of work such as our own.

We hypothesize, as part of overall data-driven trends for best practices in software, that surveys will become standard practice for programming language research and practice. We hope our experiences are useful to others planning to analyze such promising but sensitive data.

Acknowledgments

We thank David MacIver for graciously offering us the data from the Hammer Principle website and David Patterson and Armando Fox for access to the survey of their MOOC course. Anonymous reviewers provided helpful editing suggestions.

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.

References

[1] 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.

[2] 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.

[3] C. Emerick. `http://cemerick.com/2012/08/06/results-of-the-2012-state-of-clojure-survey`, 2012.

[4] A. Fox and D. Patterson. Software engineering for saas. `https://www.coursera.org/course/saas`, 2012.

[5] 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.

[6] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

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

[8] D. R. MacIver. Personal Communication, 2012.

[9] L. A. Meyerovich and A. Rabkin. Socio-PLT: Principles for Programming Language Adoption. In *Onward!*, October 2012.

[10] P. Norvig. Design patterns. `http://norvig.com/design-patterns/`, March 1998.

[11] 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 '11, 2011.

[12] Quantcast. Slashdot Traffic and Demographic Statistics. `http://www.quantcast.com/slashdot.org#!demo&anchor=panel-GENDER`, June 2012.

[13] 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.

[14] A. Zeller, T. Zimmermann, and C. Bird. Failure is a four-letter word: a parody in empirical research. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, 2011.