Understanding Conversational Programmers: A Perspective from the Software Industry

Parmit K. Chilana[1], Rishabh Singh[2], Philip J. Guo[3]

[1]Management Sciences
University of Waterloo
Waterloo, ON Canada
pchilana@uwaterloo.ca

[2]RiSE group
Microsoft Research
Redmond, WA USA
risin@microsoft.com

[3]Computer Science
University of Rochester
Rochester, NY USA
pg@cs.rochester.edu

ABSTRACT

Recent research suggests that some students learn to program with the goal of becoming *conversational programmers*: they want to develop programming literacy skills not to write code in the future but mainly to develop conversational skills and communicate better with developers and to improve their marketability. To investigate the existence of such a population of conversational programmers in practice, we surveyed professionals at a large multinational technology company who were not in software development roles. Based on 3151 survey responses from professionals who never or rarely wrote code, we found that a significant number of them (42.6%) had invested in learning programming on the job. While many of these respondents wanted to perform traditional end-user programming tasks (e.g., data analysis), we discovered that two top motivations for learning programming were to improve the efficacy of technical conversations and to acquire marketable skillsets. The main contribution of this work is in empirically establishing the existence and characteristics of conversational programmers in a large software development context.

Author Keywords

Conversational programmers; programming literacy; non-CS majors; technical conversations.

ACM Classification Keywords

K.3.2 Computers and Education: Computer and Information Science Education—*literacy, computer science education*.

INTRODUCTION

Momentum around the importance of programming literacy [31] has catalyzed several specialized initiatives to teach coding skills to a broad audience. From introducing programming to elementary school children [5] to launching introductory programming university courses for students outside computer science (CS) [11] to creating free massive online programming courses (e.g., *Coursera, edX, Udacity*) hundreds of academic, government, and industry efforts around the world are trying to prepare a tech-savvy workforce of the future.

Learning and teaching programming has also been a key theme in human-computer interaction (HCI) and computing education research for over three decades [25]. For example, numerous projects have contributed insights into the struggles of novice programmers [13,16], proposed designs for improving the usability of complex development environments [26,28], and even invented new programming languages that simplify programming concepts for learners of all ages [29]. While these research findings and experience reports from classrooms have helped us better understand the barriers to learning programming, many of the insights are based on the assumption that learners will eventually write code (e.g., as a professional developer or a domain-specific end-user programmer [20]). But, is this always the case?

Recent research shows that some students in non-CS fields such as management may not aspire to write code as an end-user programmer or a professional programmer, but are still strongly interested in taking programming classes [6]. These students were termed *conversational programmers* because they wanted to develop only conversational skills in programming literacy to be able to: 1) aid technical conversations with professional software developers in the future; or 2) enhance their marketability in the software industry.

Unfortunately, beyond this classroom study and informal discussions by practitioners [10,22], we know little about conversational programmers in today's software industry. To what extent do these conversational programmers actually exist in practice? How do conversational programmers perceive programming literacy? How do they interact with software developers? What motivates them to acquire programming skills, if at all?

In this paper, we investigate the prevalence and perceptions of conversational programmers at XYZ, a large multinational software company. We conducted a large-

CHI'16, May 07-12, 2016, San Jose, CA, USA
 ISBN 978-1-4503-3362-7/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2858036.2858323

scale survey targeting professionals in roles outside of software development (e.g., sales and marketing, customer relations, managers, lawyers, user experience researchers, designers, and others) and encouraged participation from respondents who had no formal training in CS. Based on answers from 3151 respondents who never or rarely wrote any code, we found that close to half (42.6%) of them *had* invested in learning programing skills on the job. While many of these respondents were learning programming to support small end-user programming tasks, such as analyzing data (24.5%) or creating prototypes (20.6%), over *half* of our respondents' motivations fit the definition of conversational programmers [6]—they were learning programming to improve their technical conversations with developers and customers (29.6%) and viewed programming literacy as a marketable skill for their overall career (22.7%). Furthermore, we found that many of the conversational programmers who engaged in end-user programming tasks were, in fact, using their artifacts to improve communication with developers.

The main contribution of this work is in providing empirical insights that establish the existence and characteristics of conversational programmers in the context of a large software development company. Our findings complement existing insights about professional developers [21,30] and end-user programmers [3,20,27,32] and raise important questions about how to teach programming so that students not only learn the mechanics of writing code, but also learn techniques for establishing common ground [8] in conversations with developers. We hope that our analyses will be useful for HCI and computing education researchers and practitioners to consider as they chart the future of training a programming literate society.

RELATED WORK

To contextualize our insights about conversational programmers, we draw upon a variety of research from HCI, computing education, and software engineering.

End-user programmers vs. conversational programmers

For several years, there has been a push to teach programming skills to students who are not necessarily going to be professional software developers. For example, numerous efforts have been created to teach programming to students in Biology, Fine Arts, and Business [11,12,36] so that they can not only enhance computational thinking skills [37], but also have skills to design or customize software to solve specific problems [3,27]. Commonly known as end-user programming, this sub-discipline concerns "a set of methods, techniques and tools that allow users of software systems, who are acting as non-professional software developers, at some point to create, modify, or extend a software artifact" [20]. In fact, research shows that the population of end-user programmers is growing and even outnumbers professional programmers many fold [32].

On the other hand, a recent study [6] suggests that not everyone who learns to code is necessarily going to be an end-user programmer or a professional developer. Some learners have the goal of becoming *conversational programmers*: they want to develop programming literacy skills not to write code in the future but mainly to develop conversational skills and communicate better with developers in the future and to improve their marketability in the software industry [6]. Our main goal in this paper was to assess the prevalence of conversational programmers in an industrial setting: are there people who learn programming on the job even when they are not required to write code? What motivates them? While our findings corroborate prior work and show that end-user programming tasks (e.g., analyzing data, prototyping) are still a major reason why professionals learn programming on the job, we also found that conversational programmers are, in fact, prevalent in industry. And, even the end-user programming tasks that our respondents described were often carried out in the context of improving communication about design decisions.

Understanding and lowering barriers to programming

Given the momentum around "programming for all" [10,29,31], numerous studies from HCI and computing education have contributed insights into the struggles of learning programming in class and on the job. Some have focused on lowering the barriers to using programming tools and development environments [16,19] and focus on things such as syntax issues and program flow. Others have even proposed new programming languages, such as *Scratch* [29], that simplify programming concepts for learners of all ages. There is also a long history of designing graphical or visual programming environments (e.g., programming by example [24], and visual programming [33]).

The central goal of these works has been to understand the challenges people have in writing code and in developing tools or approaches to address these challenges. The assumption that the end goal is to write code has influenced most pedagogical strategies and tools that aim to lower the barriers to programming. However, our findings about conversational programmers in industry show that even though they never or rarely wrote code on the job, they still learned programming to improve their technical conversations. We raise several questions around what it means for us to teach programming to this class of students and to design new tools and programming environments that facilitate the conversational aspect of learning programming.

Social aspects of developing software

Another class of research that is relevant to our study is related to the social and collaborative aspects of software development. Over the years, there have been a number of studies on understanding development practices [1,15] and information needs of software developers [18]. Since

software decisions about design and implementation require discussions and coordination, getting everyone to have a shared understanding or establishing "common ground" [8] is important. Common ground is the knowledge and awareness that participants have in common when they are communicating with each other [8]. Previous studies suggest that to establish this common ground in software teams, developers need to be trained not only in programming, but also in communication skills and "people skills" [1,10,15,21].

But, these studies are mainly based on perspectives of software developers, which represent only one side of the partnership in software teams. Our results on conversational programmers (who came from roles in marketing, sales, management, UX/Design, customer relations, and others) suggest that while communication skills are important, they are not perceived to be enough—our study adds insights about how programming knowledge is used as a conversational medium to establish common ground. We also illustrate other strategies that conversational programmers had developed to participate in technical conversations, such as involving experts, engaging in the conversations, and finding information online.

In summary, while we have many insights into the work of end-user programmers, barriers associated with learning programming, and the social and collaborative aspects of developing software, to our knowledge, no study has previously considered the perspectives of conversational programmers in industry.

METHOD

To investigate our research questions about conversational programmers, we decided to take a survey-based approach, as it would allow us to capture perspectives on a large scale across demographics, job roles, and college majors. Our research site was XYZ, a multinational software company with over 80,000 employees. XYZ specializes in over two dozen personal, enterprise, mobile, gaming, and cloud computing applications.

In mid-2015, we sent an online survey to a sample of XYZ employees who were not currently working as software developers but who worked in roles where they were likely to discuss technical concepts with either developers or customers (i.e., they were potentially *conversational programmers*). As per the definition of conversational programmers, the survey instructions encouraged participation from people who did not major in computer science, computer engineering, software engineering, information technology and related fields. We received 3682 responses (~14% response rate[1]). One possible factor that lowered the response rate was that many of the sampled employees still had some formal computer science training since XYZ was a software company, so they did not take our survey.

We developed our survey questions via an iterative approach, starting with a pilot survey sent to 1000 randomly-selected XYZ employees. Our response rate for the pilot survey was 24.5%, which was higher than our actual survey since the pilot did not specifically target people without CS training and many respondents ended up being CS majors. We used feedback from the pilot to refine the wording of questions and to finalize the choice labels for the multiple-choice questions.

The survey instrument

Our survey consisted of 13 multiple-choice and 3 open-ended questions. The first 4 questions were about demographics: job role, years of experience in that role, college major, and gender. The next 4 questions asked about programming background and education:

> Are you comfortable writing code in one or more programming languages? If so, which ones?
>
> Which programming language(s), if any, do you need to know to do your job?
>
> How many courses involving computer programming did you take as part of your formal education (undergraduate and graduate)? Choices: 0, 1, 2-4, 5 or more.
>
> Choose all of the ways (if any) in which you have learned programming to help with your job. Choices: None, books, websites, colleagues, online courses, company-offered training, part-time university courses, other.

In a subsequent open-ended question, we further asked them to explain what motivated them to learn programming on the job.

The next 3 questions asked how often people wrote or communicated about code. The five choices were: {daily, a few times per week, a few times per month, rarely, not at all}

> How often do you write code as part of your current job?
>
> How often do you have technical conversations (e.g., about code, software architecture, programming concepts) with software developers as part of your job?
>
> How often do you have technical conversations (e.g., about code, software architecture, programming concepts) with customers (either internal or external) as part of your job?

In another open-ended question, we used the critical incident technique [4,9] and asked respondents to describe a recent memory of a technical conversation with a developer or a customer that presented a challenging situation and how they tried to resolve that situation.

The next 2 questions were about the perceived marketability of programming skills. Respondents answered each on a 7-point Likert scale from Strongly Disagree to Strongly Agree:

> In a recent study, first-year non-computer-science students indicated that learning programming made them more "marketable" in today's job market, even if they were not going to write code for their job [6]. Based on your own

[1] For confidentiality reasons, we cannot reveal exact job role names, proportions, or numbers of personnel at XYZ.

experiences, to what extent do you agree or disagree with these students' opinions?

To what extent do you agree or disagree with the following statement: If I were to start my education all over again, I would have taken more programming courses.

Finally, we had an open-ended question asking respondents to give advice to future employees in similar roles who would be working closely with software developers.

Data overview and analysis

Our initial 3682 respondents came from a variety of job roles, including those in sales, marketing, management, business operations, legal affairs, design, and customer relations[1]. Respondents had diverse levels of experience in their roles: 29.2% had 5 or fewer years of experience, 41.5% had 6 to 15 years, and 28.8% had more than 15 years of experience[1]. 62.9% identified as male, and 35.3% as female. They came from over 40 different undergraduate majors, including business, economics, communications, management, marketing, psychology, design, and HCI (only 0.08% majored in computer science/engineering/information technology). When we asked respondents about how often they wrote code as part of their current job, an overwhelming majority (86.6%) responded with either "not at all" (71.5%) or "rarely" (15.1%). We filtered our dataset and the rest of the analysis to include only the 3151 respondents who never or rarely wrote code on the job (we also eliminated 38 respondents (1.0%) that did not answer this question). Therefore, our findings represent perspectives of non-CS majors who were not in engineering roles and were not required to write code on the job.

Of those who filled out the survey, there was a high response rate for the three open-ended questions (> 50%). We used an inductive analysis [34] approach to devise a classification scheme for each of the three questions. All three of the authors were involved in iteratively developing a coding scheme. To formally assess the reliability of our coding scheme and to measure agreement between all coders, we computed the Fleiss Kappa on a uniform random sample of 150 responses from each of the three open-ended questions. We iterated on the classification until we found moderate to strong agreement by the three coders (κ=0.73).

EXPERIENCE WITH PROGRAMMING

To understand the context of our respondents' perceptions, we look at their experience with programming through formal education or on-the-job training.

Programming courses and prior experience

First, we were surprised to learn that over half of our respondents (54.2%) had taken at least one programming course as part of their formal training, even though the majority were not CS or engineering majors. Since many non-CS programs are increasingly requiring or strongly encouraging students to learn introductory programming [11,12,36], perhaps this statistic is just demonstrating this growing educational trend.

In designing our survey, we realized that some people may also learn programming on their own or through extracurricular activities. So, we asked respondents to specify what programming languages, if any, they were comfortable writing code in. We found that just over a third of the respondents (34.0%) were comfortable writing code. The most common programming languages cited were HTML, SQL, Visual Basic, CSS, and C#.

Programming on the job

When asked about programming languages used on the job, the majority (84.2%) said that they did not use any programming languages on the job. The other 15.8% of the respondents mentioned using a few different languages, including SQL, C#, HTML, shell scripting, and JavaScript.

We also asked respondents to select all of the ways in which they had learned a programming language to help them with their job (if any). Interestingly, even though most of the respondents never or rarely wrote code, close to half of them (42.6%) had still invested in learning programming on the job. The learning efforts ranged from consulting programming-related websites, to reading books, to learning from colleagues, to enrolling in specialized courses and training (Table 1).

Motivations for learning programming on the job

To probe into why people would invest time and effort in learning programming when they did not need to write code as part of their job, we asked them an open-ended question about what motivated them to learn programming on the job. We received 1759 written responses for this question, constituting a response rate of 55.8%. Through our analysis of the open-ended responses, we came up with the classification scheme shown in Table 2 and categorized each of the responses.

As shown in Table 2, the most common motivation for trying to learn programming on the job was to improve technical conversations with developers and customers (29.6%). Other motivations for learning programming described by respondents can be categorized as traditional end-user programming tasks (e.g., analyzing data, creating prototypes, improving efficiency) that have been discussed extensively in prior work [20,32]. But, in many cases, our respondents were learning to write code for these end-user programming tasks to also improve some aspect of their communication with developers. An additional 22.7% of the respondents learned programming for their personal

Strategy	Percent
Websites	32%
Books	28%
From colleagues	24%
Online courses	18%
Training offered by company	14%
Part-time college/university courses	12%

Table 1. Top resources for learning programming on the job (*note that the overall total is > 100% since some responses fell into multiple categories)

Motivation Category	Percent	Example Response
improve conversations with software developers or customers	29.6%	*I wanted to make sure I could hold my own in discussions with software developers, and I wanted to be able to talk about our products and platform with external customers.*
analyze data	24.5%	*I have learned some scripting languages which has helped me analyze data. Nothing too difficult and the motivation was to get the information to better understand my customer base.*
personal growth and marketability	22.7%	*Kids come out of school these days with such skills I can only compete with if I keep my knowledge fresh and diverse, my skills polished.*
build prototypes or demos	20.3%	*...wanted to illustrate the ideas I had and sometimes showing it is a better way to illustrate things. Easier to communicate with other developers.*
improve efficiency (e.g., automate repetitive tasks)	13.0%	*Wanted to build some simple tools to speed up my job and to understand better how the developer was coding a feature*

Table 2. Classification of respondents' self-reported motivations for learning programming on-the-job (*note that the overall total is > 100% since some responses fell into multiple categories)

growth and to stay relevant and marketable in the software industry. We discuss each of these motivations below.

TECHNICAL CONVERSATIONS WITH SOFTWARE DEVELOPERS AND CUSTOMERS

The top motivation for learning programming on the job (29.6%) was to be able to communicate better with software developers and customers. Our analysis further revealed that almost half of the respondents (47.6%) regularly participated in technical conversations with developers (few times a day, week, or month) and another large number of respondents (41.4%) regularly took part in technical conversations with customers.

Below we discuss some of the reasons why respondents learned programming to help them with their technical conversations. We also describe the context of these conversations and the other strategies that the respondents had developed over the years to tackle these technical conversations.

Programming literacy for improving conversations

Most of the respondents who were making an effort to learn programming to improve technical conversations were not interested in writing code, but rather in learning the common phrases and terminologies to follow along in the conversation at a higher level:

I don't necessarily need an in depth technical knowledge and no programming knowledge is required however it would be useful to have a basic understanding at least to be able to contribute to high level conversation.

I didn't really need it [coding] but felt it would help me with the 'bigger picture.'

In contrast, some respondents wanted to understand conversations at a lower level and understand references to different parts of the software architecture being discussed:

...to really know an area well and be able to communicate with the developers it's great to know what the process of making code changes takes and how the engine is architected. Granted it's large and you'll never be an expert in every area but at least when having discussions and someone mentions terms you know which segments of the code they're referring to.

Some respondents described how they would participate in code reviews to understand or debug particular issues during conversations:

I tend to have most of my technical conversations around bug bashes - the test scenarios, what would constitute a bug, how to prioritize the bugs and what should be considered a blocking bug or not.

Other respondents who did not have training in writing code still found it valuable to understand the basic concepts so that they could better advocate for design or implementation changes in their conversations with developers:

I don't know how to write code, but I often, several times a week, need to look at code and discuss changes with developers. It really helps that over the years I have figured out basic HTML so that I can describe what needs to happen with devs and we can understand each other.

Some respondents wanted to understand programming so that they could have more empathy in conversations with external technical customers, including developers, managers, technical support, data specialists, installation support, and others:

I decided to learn how to do some basic programming so that I could have a better sense of what my customers go through in order to implement a system.

In summary, the conversational programmers in our study found it helpful to learn programming so they could better understand technical vocabularies, customer needs, and developers' decisions and constraints during technical conversations. Since these conversations are important for software design and implementation decisions, we probed into better understanding the context of these conversations and other strategies used by respondents that did not involve learning programming on the job.

Other strategies for improving conversations

For our open-ended question about technical conversations, we received 1848 responses, constituting a response rate of 58.6%. We analyzed each of these open-ended responses and eliminated about 8.3% of the responses that did not specifically answer the question. Another 7.8% of the respondents explicitly stated that technical conversations

Strategies for Technical Conversations	Percent
relying on developers or other experts to do the translation (internal or external to the team)	39.3%
engaging in the conversation (e.g., asking lots of questions, going to the white board, focusing on the business/ customer priorities)	26.0%
finding information on the topic (e.g., by searching online, books, internal documents)	8.9%
being upfront about the level of technical knowledge	4.9%

Table 3. Respondents' strategies for technical conversations with developers and customers.

with customers or software developers did not present any notable challenges because of their experience or past training in programming:

I have been working with XYZ for the last 15 years. That gave me an ample time to understand the architecture and API…which is what mostly [gets] discussed…

The remaining responses described various technical conversations, any challenges that they presented, and how they were resolved. Below, we focus on the remaining responses that explicitly described a strategy (other than programming) that respondents used when having technical conversations. Table 3 shows our classification of the top conversational strategies and we discuss them below.

Relying on technical experts: A common strategy (39.3%) described by respondents was seeking expertise of software developers or other technical experts to translate unfamiliar terminologies. For example, some respondents described cases where they would seek help from experts following a conversation:

I was in a meeting where there was some heavy techno jargon being thrown around, so after the meeting I had a quick call with one of the attendees to put it into English, because part of my job is also explaining these concepts to others in our org and outside the org in a less tech way.

Some respondents described how they always had technical experts on hand (e.g., as part of their teams) and it was routine to defer to them:

[When] we need to get into the weeds technically and go deep, I will ALWAYS defer to my technical specialist or architect on my team. In my estimation [our] technical resources at XYZ are the best in the business and I leverage the heck out of them...

All of the respondents' examples overall supported the finding that the most used strategy for having successful technical conversations was relying on experts. However, some examples clearly indicated that there was also some hesitation in being dependent on other experts (e.g., busy developers) so often:

I always wish I was more technical in my roles as a salesperson...I need to depend on those that are more technical than me more than I would like.

Engaging in the conversation: Another common strategy (26.0%) had to do with making efforts to engage in the technical conversations. For example, many respondents explained that the best strategy for them was to ask the developers to slow down or simplify the concept they were trying to get across:

During an architecture review...an engineer started talking too technical. I didn't understand what he was talking about, so I simply asked him to describe in plain language - which he did…I've generally found that asking engineers to talk in plain language solves the problem.

Many other respondents tried to steer the conversation to make it more customer or business-focused rather than being bogged down in code:

In my position, bringing the conversation up a level or talking at a high level is more important than the details. At least at first. In this situation, I asked to regroup and make sure we all aligned on the high level points that I could understand.

While many respondents were able to participate better by engaging in the conversation, some did talk about tradeoffs and feeling like they were derailing the conversation or slowing it down by asking too many questions. Also, some respondents noted that it was helpful when developers could also understand the larger business and customer contexts and steer the technical details accordingly.

Finding information on the topic: Some respondents (8.9%) also described how they would invest time in searching online or reading up internal documents either to prepare for a technical conversation or to follow up after a conversation:

Technology is changing daily so personally to have productive discussions with my developer colleagues I try [to] keep up with changing technology by reading up books and websites.

In describing this strategy, a key problem noted was that there may not always be enough time to do this as business conversations were often time-critical.

Being upfront about level of technical knowledge: A small number of respondents (4.9%) said that their way of dealing with technical conversations was to be upfront about their level of technical knowledge to set the tone of the conversation:

I openly explained that this is not my domain and that I would call upon colleagues who are better placed than I to elaborate on the technical side of the question.

Despite having developed such strategies for tackling technical conversations, a number of respondents (9.9%) recounted specific situations where they felt like they were not satisfied about their level of participation in some conversations:

I had a question, but didn't know enough about the programming language or the project architecture to ask it intelligently. I had to take shots in the dark and ask lots of follow up questions

In summary, although over half of our respondents could better understand and participate in technical conversations by simply relying on experts or asking the right questions, a large number of them still emphasized the importance of being programming literate to truly understand the context of some technical conversations. Next, we look at how respondents were actually learning and using programming skills on the job to support such conversations.

END-USER PROGRAMMING TASKS TO SUPPORT CONVERSATIONS

As shown in Table 2, the respondents who were learning programming on the job were mainly trying to analyze data, create prototypes, or improve efficiency in some aspect of their job. Although this was not surprising given the extensive prior work on end-user programming [3,20,27,32], we found that an implicit motivation to learn such end-user programming tasks was to have more informed conversations with developers.

Analyzing data

Since business decisions are now becoming more data-driven than ever before, a large percentage of respondents (24.5%) reported that they learned programming for analyzing data. Many of their examples alluded to the need to access data from multiple sources and to generate reports based on data analysis, often to develop a shared context with developers:

I have learned a little bit of programming in order to analyze data (ex: writing [database] scripts). I've also learned a lot about the tools that developers are using on a daily basis…it helps us have a shared framework for talking about work items and the process of building software.

Some respondents also felt that data analysis was needed for steering their team, being "on the same page", or even for guiding certain decisions:

My primary motivation has been driven by a need to analyze data and design reports. My efforts will be very basic -- even somewhat crude -- but it has often proven important to quickly getting all key people aligned on what we're trying to accomplish.

Building prototypes or demos

Another common motivation for learning programming (20.3%) was the ability to create prototypes and demos for customers and developers. As expected, a common reason for creating prototypes was for making it easier for customers to understand a product's interaction and features. But, for many of the responses about prototyping, we saw that respondents placed a lot of value on building prototypes to clarify their own understanding and to use it as a communication aid in conversations with developers:

Since the work I was producing was going to be interactive I also wanted to be able to design prototypes that were interactive. This helped me answer my own questions about how things should work and was a really great tool to communicate with engineers

Another reason for learning to code was to be able to more clearly articulate visions and design ideas to developers:

I have been learning HTML/CSS and JavaScript to be able to create my own interactive prototypes of my designs, and to be able to converse more fluidly with the [design] team that does the front-end code of our products. My motivation to learn programming was to be able to make web pages and build prototypes. Even with the one class [I took] though, I've been able to communicate better with the software engineers on my team.

Improving efficiency

Finally, around 13.0% of the respondents learned programming to improve efficiency, often by automating a repetitive task. But, some respondents explained how in the process of digging into the code, they were also able to better understand the internals and allowed them to have more concrete discussions about implementation decisions:

When a need presents itself, I dig in to learn what I must so I can make informed decisions and communicate those decisions to the teams who implement them or use the resulting systems.

In summary, a large number of the motivations for learning programming revolved around end-user programming tasks, as has been shown in prior work [20]. But, it is interesting to note that in many cases, respondents were using their artifacts as an aid in conversations about design and implementation details and related decisions.

PERSONAL SKILLSET AND CAREER MARKETABILITY

In addition to improving technical conversations and supporting end-user programming tasks, a significant portion of our respondents (22.7%) had learned programming out of curiosity, personal growth, and to stay relevant and marketable in the technical job market.

For example, as one of the respondents described:

I use programming as a tool to help me execute a vision. I learn new things, e.g., programming languages, because they are needed to help move forward to that vision. I do not think in terms of "motivated to learn", rather inspired to create and all that requires, which can be learning new programming languages.

Another common theme was to use programming skills as an addition to their skillset:

I was motivated by seeing a need and being motivated and interested in helping fill that need. Also, I would say that continuing education, addressing gaps in my skillset, and wanting to stay on top of new technologies were also motivators.

Several respondents said that they just felt knowing programming would keep them relevant and marketable. Many of their sentiments corroborated previous findings about students [6] and their perceptions of programming as being a vehicle for broadening career options. In fact, even though our respondents never or rarely wrote code as part of their job, the survey responses showed that the majority (75.7%) agreed to some extent that learning programming would make them more marketable in today's job market for their chosen career path. Similarly, 64.3% agreed to some extent with the statement, "*If I were to start my education all over again, I would have taken more programming courses.*"

Advice for future conversational programmers

Our final open-ended question solicited advice from respondents for future employees in a similar role who would be working with software developers. We received 2050 respondents for this question (a response rate of 65.0%). We classified respondents' advice into five main categories (Table 4).

Learn Programming

We found that almost half of the respondents (48.2%) gave advice to learn programming in some form. We highlight three themes that emerged from these sorts of advice.

First, respondents emphasized learning general programming or computer science concepts to improve communication with developers:

Without a doubt I would advise on learning the basics about computer programing even if you aren't considering developing a fluency or expertise; the basics help you understand the foundational building blocks, or logic, for the work and gives you a small toolset to help communicate more effectively.

Another common advice was to learn a specific programming language(s)—there were a variety of suggestions about which language to learn:

Non-comp-sci graduates must learn at least one of the core programming languages (C#, Java, C++...) and do a mid size project in it. This will setup the groundwork to learn anything else much easier.

At a minimum, people should have a working knowledge of HTML/CSS. While these are not actual programming languages (they are markup languages), they provide a bridge between UX/UI design, for example, and dev.

A large number of respondents (21.9%) gave the specific advice to enroll in formal programming courses—in most cases, the advice was to take at least one programming course during formal education:

Take at least an introductory course in whatever area you are working in. For example, if you work on the [database product] team, understand the basics of RDBMSs. If you work on a cloud service team, understand the basics of client-server, and scale. Just enough to understand how the pieces fit together.

Understand Context of Software Development

In addition to learning programming, another common advice (18.9%) for conversational programmers was to be well-versed in the general principles of software development and architecture:

You don't need to learn to code proficiently, but you need to understand basics about how technology works and how software can unburden people from non-value added work so they can focus on things that only people can do. You need to understand things like what it means to "have an API to call" or what it means when a decision is "algorithmic" or when a task can be automated. You don't know how to automate it, but you are very valuable if you know why something should be automated, and that it can be.

Advice for future conversational programmers in the software industry	Percent
learn to code (general concepts, specific programming languages, taking courses)	48.2 %
understand principles of software development and software architecture (without needing to code)	18.9%
develop communication & interpersonal skills	16.1%
take courses in non-CS subjects (e.g., sociology, design, psychology)	3.3%
understand internal company software frameworks	2.1%

Table 4. Classification of respondents' advice for future employees preparing for similar careers

Another advice was to develop an understanding of the organizational structure and how software development fits into the overall process:

Understand the responsibilities and knowledge level of the individual to whom you're speaking (i.e., are you talking to a CxO, business analyst, IT manager, or direct [software] developer?) and adjust your topic, style, & content accordingly. Understand how these people fit in within an organization - what motivates them, what are they accountable for, what is their desired outcome - and that will help you understand what you need to learn more about in order to be able to support/serve them.

Develop Communication Skills

Another 16.1% of the respondents advised future graduates to improve communication, interpersonal, and non-technical skills so that they could better empathize and work with developers:

Learn to present concepts by being about to partner with technical resources to "translate" technical vision to non-technical audiences. Learn the basics of project management, technical writing, and presentation skills.

Finally, some of our survey respondents (3.3%) also emphasized the benefits of taking non-technical courses to develop communication skills:

Balance the technology courses with the communication courses. Without strong written and spoken communication skills, you only offer 50% of the value required to be successful.

In summary, given the efforts that current conversational programmers were making to participate in technical conversations, it was not surprising that their common advice for future employees was to develop basic skills in programming (e.g., by taking at least one course) and to also foster interpersonal and communication skills.

DISCUSSION

Decades of research, as well as recent high-profile "learn-to-code" initiatives such as *Hour of Code*[2], have framed programming as an empowering tool for *creating* new software artifacts. One common vision for researchers and

[2] https://code.org/learn

educators is to encourage people to learn programming so that they can develop computational thinking [37] skills and become creators and makers. However, our results establish the existence and characteristics of conversational programmers in a large software development context who view programming literacy also as an empowering tool for *communicating* about both software artifacts and broader aspects of the technology business. We now reflect on our main findings and raise key questions for future research directions in HCI and computing education.

Programming knowledge as a shared medium for establishing common ground in conversations

Software development is a highly social activity in that major decisions are made through conversations. Prior works have shown that establishing common ground [8] in conversations can often be difficult for multidisciplinary teams [17]. There are also a number of studies that shed light on how software developers should develop communication skills to facilitate conversations with other team members [1,15,21]. Our study complements these findings by showing the other side of the conversational equation—professionals in roles other than software development who never or rarely write code on the job and come from various educational backgrounds. While many have developed strategies to improve their participation in technical conversations, (e.g., by relying on experts, asking questions, and looking up information on their own), half of them had also invested time in learning programming. The main implication of our findings is that professionals in roles other than software development view programming as a shared medium for achieving common ground, not just for creating artifacts.

While this paper has focused on how people who do not write code on the job and still learn programming, it would be interesting to solicit feedback from software developers about the topics they would recommend their peers learn in order to communicate more effectively with colleagues in different roles. Perhaps we will discover that developers would like to devote more time to learning the skillsets of business, sales, marketing, psychology, and sociology, even though they will not be working full-time in those roles.

A lifelong learning perspective on programming

In addition to the efforts in learning programming on the job, about two-thirds of our respondents said that they would take another programming class if they were to start all over again and about half of our respondents gave the advice that future employees should take at least one programming course. But, what is the ideal programming course for future conversational programmers? Our work raises a number of questions for future research in HCI and computing education about training the next generation of conversational programmers. For example, there have been many initiatives in creating simplified introductory programming courses for non-CS majors [e.g., 14], but the core focus is still on *creation* (students are taught to write programs, debug them, and use various development environments, etc.) While a high number of our respondents were still engaged in end-user programming tasks, often they would use their artifacts to aid conversations. Yet, we rarely explicitly teach for conversations in our introductory programming curricula. Are there other pedagogical strategies that we can use to complement our current intro programming curricula so that we teach students how to use their programming skills as a medium for establishing common ground?

Also, since technology and programming languages rapidly change, it is important for intro programming courses to have a lifelong perspective and teach transferrable concepts. For example, big data analytics and cloud computing are relatively new technologies that did not exist a few years ago and have introduced new programming paradigms. Just over 20% of our respondents gave the advice that one course would give future conversational programmers the necessary raw materials to build skills. But, certainly, many respondents were investing in learning new skills in analyzing data and prototyping on the job (even though they had taken a programming course before).

Role models for learning programming

Given the importance of programming as a way to facilitate workplace conversations, it would be beneficial for popular "learn-to-code"[2,3] campaigns to broaden their pitches to become more inclusive of professions beyond those related to software development. Doing so can diversify the potential student base for computer science and programming at both K-12 and college levels, since students who do not want to become software developers will still see value in understanding code. If this generation of young people can find their own role models in positions such as sales, marketing, management, business operations, legal affairs, design/UX, and customer service promoting the value of learning programming, then they are more likely to feel like programming is a marketable professional skill for them and not just for aspiring future software developers or end-user programmers.

Innovations in programming languages and tools

A lot of research in HCI, software engineering and programming languages has invented new tools and environments (e.g., *Scratch* [29], *Touch Develop* [35], and *AppInventor* [38]) for simplifying the programming experience for beginners. There have also been innovations in visual programming languages that allow users to visually demonstrate or sketch their program flow [2,33]. While many of these efforts have been successful at introducing basic programming concepts, the focus is still on writing code and understanding logic and syntax. What could we do

[3] http://www.nytimes.com/2015/09/16/nyregion/de-blasio-to-announce-10-year-deadline-to-offer-computer-science-to-all-students.html

to enhance future innovation in programming languages and tools such that the conversational aspect could be embedded into the languages?

Perhaps there can be some innovations to consider in the design of new programming environments. For example, we can develop environments that show learners how a particular program (or a part of it) could be used as a conversational tool. In the formal methods community [7], for instance, there are ways to help testers write the desired specification in a high-level language (based on computational logic and without having to write code), which can then be automatically checked against the code written by developers using model checking techniques. Perhaps similar ideas can be adapted to help conversational programmers better understand and communicate with developers, without having to become expert programmers. In addition, crowdsourcing mediums for getting technical help (e.g., *StackOverflow*) that are currently used by developers [23] could also be explored further to help the needs of conversational programmers.

Limitations

We surveyed employees from only one company, so we cannot make claims about the generality of our findings across companies or industry sectors. Also, our sample of employees was not random since we selected based on job roles that likely involved discussing technical concepts with either developers or customers. Those who responded to the survey may be ones with stronger opinions about conversational programmers. Thus, the proportions we report in this paper are not necessarily indicative of the proportions of employees at XYZ with those sentiments.

CONCLUSION

Over the years, research from HCI and computing education has helped us better understand the barriers to learning and teaching programming. However, many of these insights are based on the assumption that learners will eventually use coding skills to become professional developers or domain-specific end-user programmers. Our results suggest that a large number of professionals who learn programming on the job are not using their skills to write code. These learners are *conversational programmers* who want to use programming skills as a medium for establishing common ground in technical conversations with developers and customers or to improve their marketability in the software industry.

Although our study was done at a single software company, as more companies in diverse sectors move toward offering software-based products and services (e.g., automobile manufacturers, health-care providers, financial advisors), we predict that the population of conversational programmers will likely grow across industries in the coming decade. Our study opens up a number of paths for future research to investigate how to better understand, support, and educate these conversational programmers.

REFERENCES

1. Andrew Begel and Beth Simon. 2008. Struggles of new college graduates in their first software development job. *Proceedings of the ACM technical symposium on Computer science education*, ACM, 226–230.
2. Margaret M. Burnett. 1999. "Visual programming," in *Encyclopedia of Electrical and Electronics Engineering*. John G. Webster, Ed. John Wiley and Sons Inc., New York.
3. Margaret M. Burnett and Brad A. Myers. 2014. Future of end-user software engineering: beyond the silos. *Proceedings of the on Future of Software Engineering*, ACM, 201–211.
4. José C. Castillo, H. Rex Hartson, and Deborah Hix. 1998. Remote usability evaluation: can users report their own critical incidents? *CHI 98 Conference Summary on Human Factors in Computing Systems*, ACM, 253–254.
5. Rory Cellan-Jones. 2014. A computing revolution in schools. *BBC News*. http://www.bbc.com/news/technology-29010511
6. Parmit K. Chilana, Celena Alcock, Shruti Dembla, Anson Ho, Ada Hurst, and Philip J. Guo. 2015. Perceptions of Non-CS Majors in Intro Programming: The Rise of the Conversational Programmer. *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing,* 251-259.
7. Edmund M. Clarke, Orna Grumberg, and Doron Peled. 1999. *Model checking*. MIT press.
8. Herbert H. Clark and Susan E. Brennan. 1991. Grounding in communication. *Perspectives on socially shared cognition* 13: 127–149.
9. John. C. Flanagan. 1954. The critical incident technique. *Psychological bulletin* 51, 4: 327–358.
10. Paul Ford. 2015. What is code? *Bloomberg Businessweek*. http://www.bloomberg.com/graphics/2015-paul-ford-what-is-code/
11. Andrea Forte and Mark Guzdial. 2005. Motivation and nonmajors in computer science: identifying discrete audiences for introductory courses. *IEEE Transactions on Education* 48, 2: 248–253.
12. Mark Guzdial. 2003. A media computation course for non-majors. *Proceedings of the ACM technical symposium on Computer science education*, ACM, 104–108.
13. Mark Guzdial. 2004. Programming environments for novices. *Computer science education research* 127–154.
14. Mark Guzdial and Andrea Forte. 2005. Design process for a non-majors computing course. *Proceedings of the ACM technical symposium on Computer science education*, ACM, 361–365.

15. Michael Hewner and Mark Guzdial. 2010. What game developers look for in a new graduate: interviews and surveys at one game company. *Proceedings of the ACM technical symposium on Computer science education*, ACM, 275–279.
16. Caitlin Kelleher and Randy Pausch. 2005. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR)* 37, 2: 83–137.
17. Gary Klein, Paul J. Feltovich, Jeffrey M. Bradshaw, and David D. Woods. 2005. "Common ground and coordination in joint activity," in *Organizational simulation*. W.R. Rouse and K.B. Boff, eds., John Wiley & Sons Inc., New York.
18. Andrew J. Ko, Robert Deline, and Gina Venolia. 2007. Information Needs in Collocated Software Development Teams. *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 344–353.
19. Andrew J. Ko, Brad A. Myers, and HH Aung. 2004. Six Learning Barriers in End-User Programming Systems. *Proceedings of the Symposium on Visual Languages and Human Centric Computing, IEEE,* 199–206.
20. Henry Lieberman, Fabio Paternò, Markus Klann, and Volker Wulf. 2006. *End-user development: An emerging paradigm*. Springer.
21. Paul Luo Li, Andrew J. Ko, and Jiamin Zhu. 2015. What makes a great software engineer? *Proceedings of the 37th International Conference on Software Engineering*, IEEE, 700–710.
22. Matthew Magain. 2013. *How Much Code Should A UX Designer Write?* http://uxmastery.com/how-much-code-should-a-user-experience-designer-write/
23. Lena Mamykina, Bella Manoim, Manas Mittal, George Hripcsak, and Bjoern Hartmann. 2011. Design lessons from the fastest q&a site in the west. *Proceedings of the annual conference on Human factors in computing systems*, ACM, 2857–2866.
24. Brad A. Myers. 1986. Visual programming, programming by example, and program visualization: a taxonomy. *Proceedings of the annual conference on Human factors in computing systems* ACM, 59–66.
25. Brad A. Myers and Andrew J. Ko. The Past, Present and Future of Programming in HCI. *Human-Computer Interaction Consortium (HCIC 2009)*.
26. Brad A. Myers, John F. Pane, and Andrew J. Ko. 2004. Natural programming languages and environments. *Communications of the ACM* 47, 9: 47–52.
27. Bonnie A. Nardi. 1993. *A small matter of programming: perspectives on end user computing*. MIT press.
28. John F. Pane, Brad Myers, and Leah B. Miller. 2002. Using HCI techniques to design a more usable programming system. *Proceedings of Symposia on Human Centric Computing Languages and Environments*, IEEE, 198–206.
29. Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: programming for all. *Communications of the ACM* 52, 11: 60–67.
30. Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. 2012. How do professional developers comprehend software? *Proceedings of the 34th International Conference on Software Engineering*, IEEE, 255–265.
31. Douglas Rushkoff. 2010. *Program or be programmed: Ten commands for a digital age*. O/R Books, New York.
32. Christopher Scaffidi, Mary Shaw, and Brad Myers. 2005. Estimating the numbers of end users and end user programmers. *Symposium on Visual Languages and Human-Centric Computing,* IEEE, 207–214.
33. Nan C. Shu. 1988. *Visual programming*. Van Nostrand Reinhold New York.
34. Anselm L. Strauss and Juliet M. Corbin. 1998. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Sage Publications.
35. Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, and Manuel Fahndrich. 2011. TouchDevelop: programming cloud-connected mobile devices via touchscreen. *Proceedings of the SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, ACM, 49–60.
36. Greg Wilson. 2006. Software carpentry. *Computing in Science & Engineering* 8: 66.
37. Jeannette M. Wing. 2006. Computational thinking. *Communications of the ACM* 49, 3: 33–35.
38. David Wolber. 2011. App inventor and real-world motivation. *Proceedings of the ACM technical symposium on Computer science education*, ACM, 601–606.