Bringing "High-level" Down to Earth: Gaining Clarity in Conversational Programmer Learning Goals

Kathryn Cunningham
Northwestern University
Evanston, Illinois, USA
kcunningham@northwestern.edu

Yike Qiao
Northwestern University
Evanston, Illinois, USA
ikaqiao@u.northwestern.edu

Alex Feng
Northwestern University
Evanston, Illinois, USA
alexfeng@u.northwestern.edu

Eleanor O'Rourke
Northwestern University
Evanston, Illinois, USA
eorourke@northwestern.edu

ABSTRACT

As the number of conversational programmers grows, computing educators are increasingly tasked with a paradox: to teach programming to people who want to communicate effectively about the internals of software, but not write code themselves. Designing instruction for conversational programmers is particularly challenging because their learning goals are not well understood, and few strategies exist for teaching to their needs. To address these gaps, we analyze the research on programming learning goals of conversational programmers from survey and interview studies of this population. We identify a major theme from these learners' goals: they often involve making connections between code's real-world purpose and various internal elements of software. To better understand the knowledge and skills conversational programmers require, we apply the Structure Behavior Function framework to compare their learning goals to those of aspiring professional developers. Finally, we argue that instructional strategies for conversational programmers require a focus on high-level program behavior that is not typically supported in introductory programming courses.

CCS CONCEPTS

• **Social and professional topics → Computing education**.

KEYWORDS

Conversational Programmers, Learning Goals, Instructional Design

ACM Reference Format:
Kathryn Cunningham, Yike Qiao, Alex Feng, and Eleanor O'Rourke. 2022. Bringing "High-level" Down to Earth: Gaining Clarity in Conversational Programmer Learning Goals. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2022), March 3–5, 2022, Providence, RI, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3478431.3499370

1 INTRODUCTION

"Writing programs" is the most frequent learning goal in CS1 courses [2]. However, recent research has found that not all students who want to learn about code aspire to author programs. *Conversational programmers* seek programming instruction with the goal of improving their communication and collaboration with software developers and other technical workers [4, 5, 24]. Conversational programmers can be found in the growing population of non-majors taking computing courses [3], particularly in computing-adjacent majors like Information [7] or Engineering Management [4]. Conversational programming skills are relevant for many job roles, including managers, designers, entrepreneurs, and marketers [24].

Unfortunately, we have evidence that typical introductory programming instruction does not meet conversational programmers' needs. A study of adult learners aiming to improve their technical communication by studying formal (e.g. coursework) and informal (e.g. tutorials and forums) programming learning resources found that this effort often ended in a feeling of failure [24]. A key issue was that the knowledge these learners sought was not what they found. They wanted "conceptual" and "big picture" knowledge about capabilities and applications of programming technologies, but instead were lost in details about syntax and semantics [24].

As the number of software engineers grows [16], the number of professionals needing conversational programming skills will likely grow as well. However, while recent research has highlighted the prevalence of conversational programmers [4, 5, 24], we still have a limited understanding of how to effectively support these learners in reaching their particular goals in the relatively short time they can dedicate to computing education. A key challenge is that researchers have not yet defined the type of knowledge that conversational programmers need, and how this differs from the knowledge that aspiring code writers will need.

In this paper, we address this gap by analyzing existing empirical work on conversational programmers to extract a set of nine distinct learning goals. We find that conversational programmers' goals often reference specific tasks they want to complete, which frequently involve making connections between code's real-world purpose and the internal elements of software. To further define the knowledge that conversational programmers need, and compare this to the knowledge of traditional software developers, we apply the Structure Behavior Function framework to organize their goals. We find that conversational programmers need a more shallow understanding of code that focuses primarily how structures and behaviors contribute to its function. Based on these findings, we argue that instructors should acknowledge that there are multiple valid ways of understanding programs that serve different goals, and that curricula for conversational programmers should focus on the roles of abstract, high-level components of authentic programs.

2 WHO ARE CONVERSATIONAL PROGRAMMERS?

Chilana et al. coined the term "conversational programmer" after exploring the goals of Engineering Management students related to programming [4]. These students cited a desire to improve communication with professional programmers and improve job marketability as motivations for learning to code, despite not aspiring to write much code in the careers. Later papers have profiled conversational programmers at large software companies [5], among professional adults [24], and in non-computer science majors [1, 7]. Common job titles of conversational programmers include manager, designer, entrepreneur, and marketer [24].

This research found that the defining trait of conversational programmers is an aim to understand and contribute to technical conversations [4, 5, 24]. Managers, for instance, mentioned wanting to better understand the feasibility, limitations, and trade-offs of different technologies [5, 24]. Designers brought up being able to more precisely communicate designs to developers [5, 7, 23]. As students, conversational programmers are distinct from other non-major groups. On the spectrum of code proficiency, conversational programmers slot in between non-programmers and end-user programmers [4] (see Figure 1). In contrast to some populations of non-majors, who may not see the applicability or relevance of programming to their future careers, conversational programmers actively seek programming information because they believe it will be beneficial [24].

While prior research on conversational programmers has highlighted their prevalence and provided valuable insights into their goals and professional roles, these studies did not formally define their learning goals to support instruction. In the field of education, learning goals often encompass information about the knowledge and skills learners need to master and the tasks learners want to be able to complete [15]. By formalizing this information, educators can design instruction that is tailored to the specific learning goals of their students. In this work, we aim to build on prior research on conversational programmers by analyzing findings to define formal learning goals for this population.

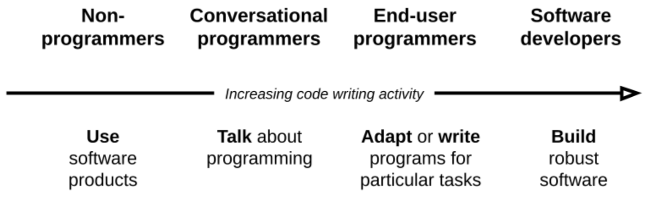


Figure 1: Types of programmers and their defining tasks (adapted from [4]).

3 WHAT DO CONVERSATIONAL PROGRAMMERS WANT TO KNOW?

To identify what conversational programmers want to know and do, we first reviewed empirical research on this population. After identifying relevant studies of professional conversational programmers, non-major students, and adult learners, we analyzed their findings to distill a set of nine distinct learning goals. This approach allowed us to identify recurring themes across these diverse perspectives on conversational programmers.

3.1 Method

3.1.1 Gathering relevant literature. To identify literature about conversational programmers, we performed a two-pronged search. First, we conducted a search of the ACM Digital Library, IEEExplore, and Google Scholar for the term "conversational programmer." Second, we used Google Scholar to identify publications that cited any of the three most highly cited articles from the first search: Chilana et al. 2015 [4], Chilana et al. 2016 [5], and Wang et al. 2017 [24]. We reviewed these 78 articles, and retained those that identified goals of conversational programmers through survey or interview studies. The six articles that met our criteria are listed in Table 1.

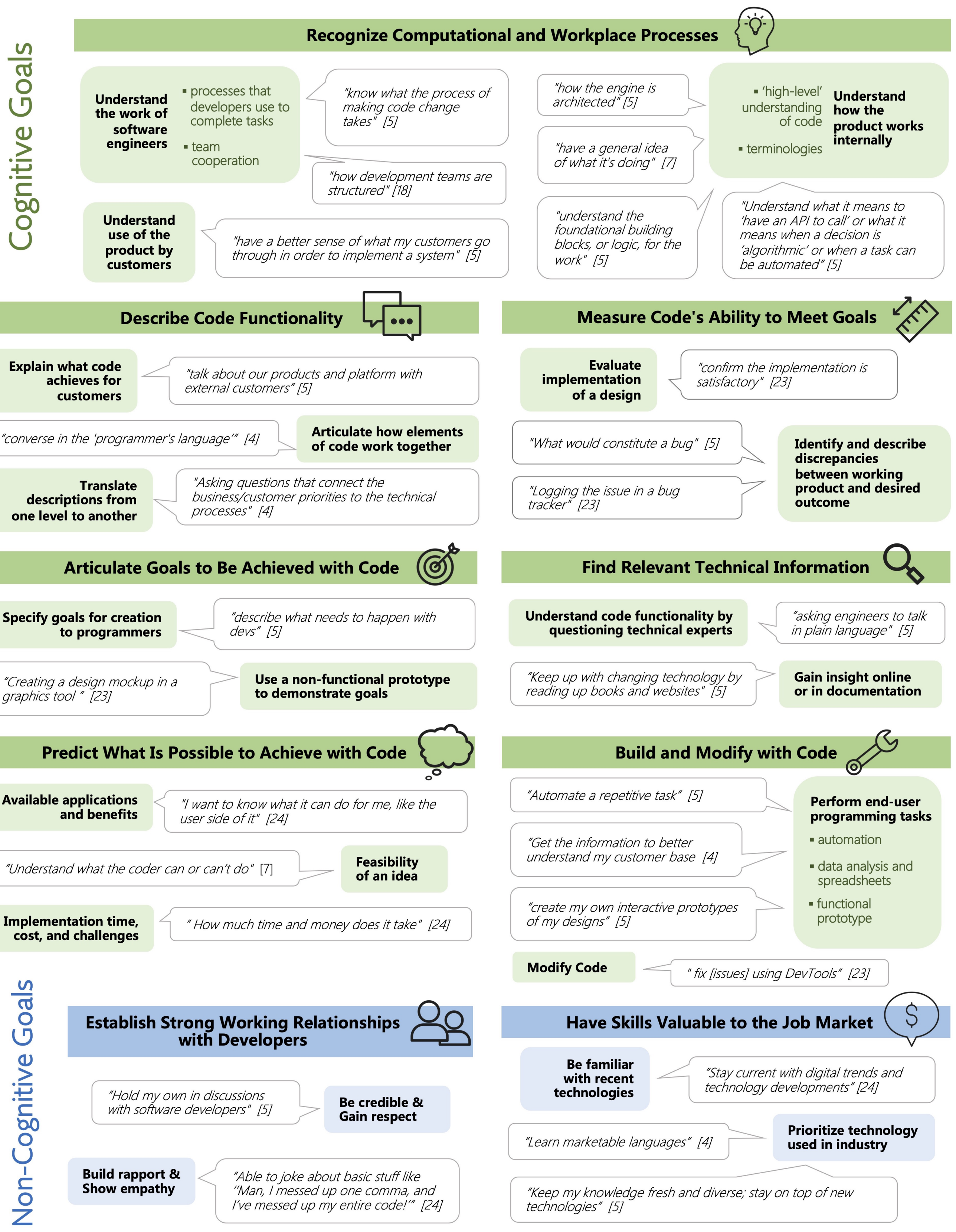
3.1.2 Extracting learning goals. Our process for extracting conversational programming learning goals was adapted from the procedure Rich et al. used to identify computational thinking learning goals for K-8 students [17]. Borrowing their definition, we specify a learning goal as *any explicit statement or implicit endorsement of what a conversational programmer can or should be able to do in relation to computer science.* Learning goals were drawn from participant quotes, survey responses, and summaries provided by article authors. Two researchers individually reviewed each article to identify potential learning goals, and then met to collaboratively decide on final goals. This process produced 98 learning goals.

Two papers had an outsized impact on the learning goal pool: analysis of Chilana et al. 2016 [5] produced 42 learning goals and analysis of Wang et al. 2017 [24] produced 21 learning goals.

3.1.3 Organizing learning goals into clusters. The first three authors collaboratively grouped learning goals to identify clusters. Learning goals were first grouped by shared topics and keywords (e.g., "bugs", "marketable"). Then, grouping continued based on the actions learning goals required (e.g., "describe", "predict", "measure", "build"). Our analysis resulted in nine clusters, shown in Figure 2.

Table 1: Publications that have interviewed or surveyed conversational programmers about desirable outcomes.

Article	Population of conversational programmers
Abdunabi et al.[1]	Non-major students
Chilana et al. [4]	Non-major students
Chilana et al. [5]	Professional conversational programmers
Cunningham et al. [7]	Non-major students
Tanner et al. [23]	Professional conversational programmers
Wang et al. [24]	Adult learners



Recognize Computational and Workplace Processes

Understand the work of software engineers
- processes that developers use to complete tasks
- team cooperation

"know what the process of making code change takes" [5]

"how development teams are structured" [18]

Understand use of the product by customers

"have a better sense of what my customers go through in order to implement a system" [5]

"how the engine is architected" [5]

"have a general idea of what it's doing" [7]

"understand the foundational building blocks, or logic, for the work" [5]

Understand how the product works internally
- 'high-level' understanding of code
- terminologies

"Understand 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" [5]

Describe Code Functionality

Explain what code achieves for customers

"talk about our products and platform with external customers" [5]

"converse in the 'programmer's language'" [4]

Articulate how elements of code work together

Translate descriptions from one level to another

"Asking questions that connect the business/customer priorities to the technical processes" [4]

Measure Code's Ability to Meet Goals

Evaluate implementation of a design

"confirm the implementation is satisfactory" [23]

"What would constitute a bug" [5]

"Logging the issue in a bug tracker" [23]

Identify and describe discrepancies between working product and desired outcome

Articulate Goals to Be Achieved with Code

Specify goals for creation to programmers

"describe what needs to happen with devs" [5]

"Creating a design mockup in a graphics tool" [23]

Use a non-functional prototype to demonstrate goals

Find Relevant Technical Information

Understand code functionality by questioning technical experts

"asking engineers to talk in plain language" [5]

"Keep up with changing technology by reading up books and websites" [5]

Gain insight online or in documentation

Predict What Is Possible to Achieve with Code

Available applications and benefits

"I want to know what it can do for me, like the user side of it" [24]

"Understand what the coder can or can't do" [7]

Feasibility of an idea

Implementation time, cost, and challenges

" How much time and money does it take" [24]

Build and Modify with Code

"Automate a repetitive task" [5]

"Get the information to better understand my customer base [4]

"create my own interactive prototypes of my designs" [5]

Perform end-user programming tasks
- automation
- data analysis and spreadsheets
- functional prototype

Modify Code

" fix [issues] using DevTools" [23]

Establish Strong Working Relationships with Developers

"Hold my own in discussions with software developers" [5]

Be credible & Gain respect

Build rapport & Show empathy

"Able to joke about basic stuff like "Man, I messed up one comma, and I've messed up my entire code!"" [24]

Have Skills Valuable to the Job Market

Be familiar with recent technologies

"Stay current with digital trends and technology developments" [24]

"Learn marketable languages" [4]

Prioritize technology used in industry

"Keep my knowledge fresh and diverse; stay on top of new technologies" [5]

Figure 2: Conversational programmer learning goals, as expressed by current and aspiring conversational programmers.

3.2 Cognitive learning goals

The first seven clusters involve cognitive processes like recalling, summarizing, inferring, and creating.

3.2.1 Recognize computational and workplace processes. Conversational programmers value general familiarity with how coding is done and what components make up programs. Conversational programmers want to understand the procedures their colleagues use when they work on technical tasks, both individually (e.g. making code changes [5]), and as a team. Code's internal function is also of concern to conversational programmers. They want to understand both the programming-related terminology others use, and which parts of code are relevant to current conversation. Conversational programmers also seek a better understanding of "what customers go through" when implementing a technical system [5].

3.2.2 Describe code functionality. Simply understanding code isn't sufficient for conversational programmers—they also need to be able to communicate and explain code at multiple levels. At one level are big-picture conversations about "products and platforms with external customers" [5]. On another level are technical conversations with developers. Connecting "customer priorities to the technical processes" is a task that calls for translation between these two levels [4].

3.2.3 Predict what is possible to achieve with code. Conversational programmers want to understand the benefits and limitations of various technologies. Managers, for instance, regard understanding benefits and costs of technologies as particularly important [24]. Knowing "what the coders can and can't do" is crucial in coordinating realistic deadlines with developers [7], and also in empathizing and adapting when development roadblocks occur [5].

3.2.4 Measure code's ability to meet goals. After a plan or design is implemented, conversational programmers often need to compare actual and designed functionality to assess product standards [23]. When code fails to meet expectations, they hope to be able to communicate discrepancies accuratelywith developers through bug reports [5] and conversations [23].

3.2.5 Articulate goals to be achieved with code. As one might expect from designers, entrepreneurs, and managers, conversational programmers want to be able to articulate ideas and goals in language that developers can understand. Often, a simple mock-up (i.e. an interactive prototype using HTML, CSS, and JavaScript) [5, 23] helps conversational programmers have more "concrete discussions", especially around implementation details [5].

3.2.6 Find relevant technical information. Conversational programmers do not have to be technical experts, but they value the ability to fill personal gaps in knowledge. This involves asking productive questions so engineers can clarify technical concepts in layman's terms [5], and identifying appropriate external resources [5, 24].

3.2.7 Build and modify with code. Occasionally, conversational programmers edit or create programs. UX designers, for instance, often find themselves making minor CSS fixes [23]. Although 71% of conversational programmers at a large software company said they never wrote code [5], some still performed end-user programming tasks such as data analysis or prototype creation [5].

3.3 Non-cognitive learning goals

The remaining two clusters involved goals conversational programmers found valuable interpersonally and in career development.

3.3.1 Establish strong working relationships with developers. Understanding code and coding is more than technical ability—it also supports soft skills that promote relationships. Conversational programmers expressed a desire to build rapport between themselves and developers. One way to do this is to express empathy for programmers' challenges, even with jokes (i.e. regarding coding struggles like misplaced commas) [24]. Conversational programmers also believe that demonstrating knowledge about the process of programming and digital trends could elicit greater "respect" from, and "credibility" with, technical team members [24].

3.3.2 Have skills valuable to the job market. Conversational programmers connected their skill development to professional opportunities. The technical skills they mentioned included programming languages as well as familiarity with the most recent technology. Skills considered more valuable were those that were currently in use in industry. For example, undergraduate conversational programmers preferred learning Java over Processing, because they considered Java more likely to be used in the workplace [4].

4 UNDERSTANDING CONVERSATIONAL PROGRAMMER KNOWLEDGE

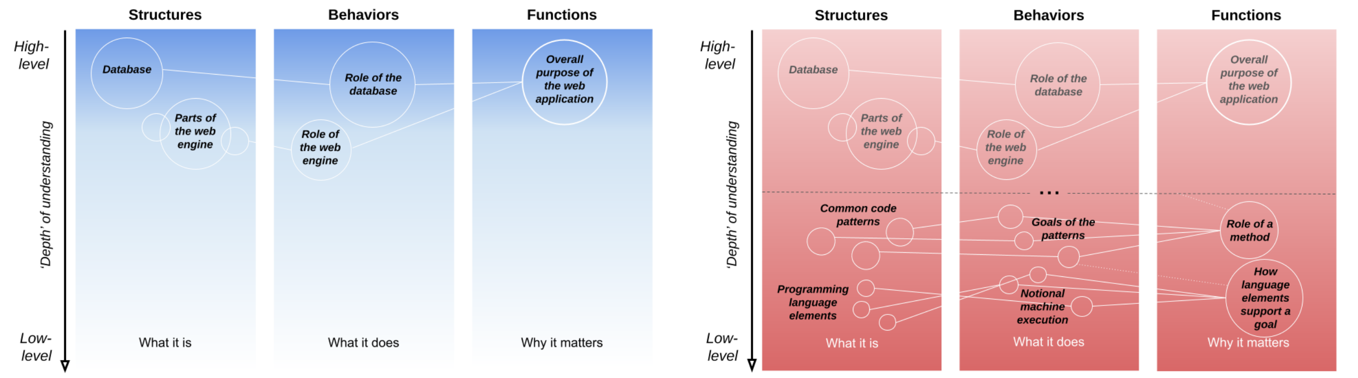
The nine learning goals we identified through our literature analysis highlight that conversational programmers want to engage with code in different ways than traditional software developers. Their goals often involve making connections between code's real-world purpose and the internal and structural elements of software, and communicating this understanding in conversation with others. We also found that the learning goals primarily described tasks that conversational programmers want to complete, rather than the types of programming knowledge they want to attain.

In order to better define this knowledge, and help us distinguish it from the knowledge that software developers or other code writers require, we organize it using the Structure Behavior Function (SBF) framework [11]. The SBF framework is a general framework for knowledge of designed artifacts, used in cognitive science [8], the learning sciences [13], and design science [10].

While knowledge frameworks do exist to describe program comprehension, they typically focus on understanding of low-level details rather than knowledge about code's real-world purpose that is central to conversational programmers' goals. For example, the notional machine [21] describes code's execution line-by line, and the Block Model [18] describes knowledge used to work from individual tokens up to entire program understanding. The SBF framework is ideal for our context because it (a) spotlights function, which appears frequently in conversational programmers' learning goals; (b) is general enough to incorporate a variety of types of knowledge; and (c) is flexible enough to build on existing work in computing education.

4.1 Defining Structure, Behavior, and Function

In this section, we present the components of the SBF framework and show how it can be applied to program understanding.



(a) Conversational programmer knowledge primarily involves understanding how high-level structures and behavior are directly connected to the overall purpose of a piece of code.

(b) Developer knowledge also involves understanding connections to overall purpose, but requires low-level understanding of the programming language elements used to write code as well.

Figure 3: The difference in knowledge needed for conversational programming and software development tasks.

4.1.1 Three types of knowledge. SBF categorizes knowledge about an artifact in three distinct but interacting types:

Structure is the parts of the artifact and their arrangement. It is *what* the artifact contains, or is made of.

Function is the purpose of the artifact. It is the reason the artifact was created and *why* it does what it does.

Behavior is the mechanism by which the artifact works. It is *how* structures achieve their purpose. Often, many behaviors occur to reach a single functional goal.

4.1.2 Knowledge is hierarchical and layered. Cognitive scientists who have applied the SBF framework noted that the functions of an artifact might be considered behaviors when that artifact is a component of a larger artifact [12]. In other words, the functions of sub-components are behaviors of the entire artifact. This layering of components means that there are *higher-level* structures and behaviors that are directly related to an artifact's overall function, and *lower-level* structures and behaviors that are more distantly related the artifact's overall function.

4.1.3 SBF in code. **Structure:** What is code made of? Certainly syntax elements like loops and variables, but there are many more meaningful structures in code. Studies of program comprehension tell us that common code patterns are also units programmers recognize in code [20]. Programs may also contain functions, objects and even larger structures like parts of a design pattern. Programs may be related to other structures like configuration files and databases.

Function: Why is code written? Of course, there is the ultimate reason: the goal or purpose of the entire program. Schulte identified this and three additional types of functions in the Block Model [18], including how sub-goals relate to the program's overall goal and how individual statements contribute to sub-goals.

Behavior: How does code work? The operation of the notional machine is a behavior of syntax statements [21]. The goals associated with programming plans are another type of behavior [19]. As a program grows larger its behavior might include the roles of various sub-elements, such as how a database supports quick access for customer data.

Layers: Programs have many levels of abstraction, and certainly fit the hierarchical model SBF suggests. The structures and behaviors we describe above range from lower-level (e.g. syntax structures and semantics) to higher-level (e.g. pieces of a software system, like a web engine and its roles). Three layers of abstraction are shown in Figure 3b.

4.2 Applying SBF to conversational programmer learning goals

In this section, we describe insights from the conversational programmer learning goals in the language of the SBF framework.

4.2.1 Code's highest-level function is the keystone of conversational programmer knowledge. A striking number of conversational programmer tasks reference code's overall purpose. This purpose is described in words ("Explain what code achieves for customers"), given as directions ("Specify goals for creation to programmers"), and evaluated ("Measure code's ability to meet goals"). Additional tasks involve relating a technical concept to a purpose, such as when conversational programmers reflect on possibilities ("Feasibility of an idea") and help technical and non-technical audiences communicate ("Translate descriptions from one level to another"). Even when performing a code writing task like building a prototype, the desired outcome is to communicate potential functionality [5].

4.2.2 Higher-level structures and behaviors best support conversational programmer goals. When conversational programmers describe what they need to know, they often use terms like "general", "conceptual", and "big-picture". What do those terms mean?

The SBF framework gives us some clue. High-level structures and behaviors most directly explain a program's overall function (shown by their connections in Figure 3). Therefore, knowledge about them best supports the numerous conversational programmer learning goals that focus on function. Conversational programmers' quotes about their learning goals often reference high-level structures and behaviors. Wanting to know "how the engine is architected" [5] suggests knowledge of high-level structures and having "a general idea of what it's doing" [7] suggests high-level behavior knowledge.

4.2.3 Conversational knowledge and writing knowledge differ in SBF depth. Our analysis suggests that the knowledge most valuable to conversational programmers lies in high-level structures and behaviors, since they connect most directly to code's function. In practice, conversational programmers are focused on understanding a particular system or technology and the goals it can achieve. They need knowledge that supports making connections "top-down" from code's function to its behaviors and functions, which is at the higher levels of abstraction (see Figure 3a).

By contrast, software developers are expected to be able to create programs in a variety of languages, towards a variety of goals. This need for widely-applicable programming knowledge explains why introductory courses focus on lower-level structures and behaviors, such as the syntax and semantics of a programming language [2], and build "bottom-up" from there. This focus is understandable because people who write code professionally need knowledge that spans from lower-level to higher-level on the SBF hierarchy to write and debug code effectively (see Figure 3b).

5 HOW TO DESIGN CONVERSATIONAL PROGRAMMER INSTRUCTION

We claim that conversational programmers find knowledge of high-level structures and behaviors most valuable. However, existing instruction does not emphasize this knowledge early on. We describe three goals to guide conversational programmer education.

5.1 We need to define high-level code *structures* and *behaviors*

To effectively teach the high-level knowledge that conversational programmers want, we need to better define the high-level structures and behaviors conversational programmers should be familiar with. Conversational programmers value an understanding of technologies used in practice, so it is particularly important to understand the high-level structures and behaviors of authentic programs or even whole software systems.

There has been a great deal of focus on categorizing and describing low-level code behavior, evidenced by the great number of program visualization tools created by computing education researchers to illustrate notional machine operation [22] or expression evaluation [14]. However, there is less focus on describing the functionality of common sub-structures of software systems in a way that is accessible to novices.

The best approach to identify the relevant high-level structures and behaviors may be empirical. From technical conversations between conversational programmers and others, we can identify mentioned structures and behaviors. We can also select programs and software systems that meet conversational programmers' goals of authenticity, and name their high-level structures and behaviors.

5.2 Scaffolding should allow understanding of high-level code structures and behaviors, *without* requiring low-level understanding

Typical programming instruction starts with the syntax of a programming language and teaches students to build larger and larger programs over time. This bottom-up approach values depth of understanding for every program that students work with. However, conversational programmers have neither the time nor the interest for this approach [24]. They want and need to understand authentic systems, but in a high-level way that clearly connects to function.

For conversational programmers, we propose a more top-down approach: learners should be able to work with higher-level structures and behaviors without needing to "build up" understanding by scrutinizing lower-level behaviors. To make this possible, scaffolding is required. A few existing instructional strategies support this outcome, including those that center higher-order functions [9] or programming plans [7]. These approaches provide additional information about the behavior of the functions or plans, so learners don't have to infer it. Case studies of authentic systems that highlight high-level structures and behaviors could also be valuable.

5.3 Low-level knowledge should be taught for *empathy* rather than mastery

Is low-level programming knowledge irrelevant? We argue that for conversational programmers, low-level structure and behavior knowledge is most useful in the service of understanding the experiences of developers. By understanding the process they undergo to write code, and common challenges during that process, conversational programmers can meet their goals of building empathy for technical co-workers as well as developing credibility.

What does it mean to learn for empathy rather than mastery? The outcomes should match conversational programmers learning goals: to understand "what the process is like to accomplish certain things" [7] and "how the developer [would code] a feature" [5], without the need to code oneself. To increase authenticity to the work of programmers, relevant activities might include more complex components, such as more program files and configuration.

It is interesting to note that highly-scaffolded approaches to code writing that are common for teaching programming to non-majors (by definition) avoid some of the authentic challenges developers face when using industry-level tools, such as syntax issues and import issues. We do not advocate that conversational programmers be thrown in the deep end of code creation, particularly because they have low self-efficacy for programming [4]. However the support we provide should not occlude valuable learning experiences.

6 CONCLUSION

As a result of our analysis of conversational programmer learning goals and application of the SBF framework, we propose an instructional approach that flips the focus of programming learning. By prioritizing the high-level function, behavior, and structure of authentic programs, we can teach the knowledge that conversational programmers want and need.

The example of conversational programmers shows that computing learners with different career endpoints may want *different types of knowledge* about programs. By teaching everyone with approaches that forefront low-level knowledge, we run the risk of demotivating those who have different goals and value different coding knowledge [6, 24]. For all learners, we should understand the type of program knowledge they value and design to make it accessible early on with new instructional strategies.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant # 2030859 to the Computing Research Association for the CIFellows Project. We thank Mark Guzdial and Barbara Ericson for their formative discussions on this work. We thank the Delta Lab community for their valuable feedback.

REFERENCES

[1] Ramadan Abdunabi, Ilham Hbaci, and Heng-Yu Ku. 2019. Towards Enhancing Programming Self-Efficacy Perceptions among Undergraduate Information Systems Students. *Journal of Information Technology Education: Research* 18 (April 2019), 185–206. https://doi.org/10.28945/4308

[2] Brett A. Becker and Thomas Fitzpatrick. 2019. What Do CS1 Syllabi Reveal About Our Expectations of Introductory Programming Students?. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 1011–1017. https://doi.org/10.1145/3287324.3287485

[3] Tracy Camp, W. Richards Adrion, Betsy Bizot, Susan Davidson, Mary Hall, Susanne Hambrusch, Ellen Walker, and Stuart Zweben. 2017. Generation CS: The Growth of Computer Science. *ACM Inroads* 8, 2 (May 2017), 44–50. https://doi.org/10.1145/3084362

[4] Parmit K. Chilana, Celena Alcock, Shruti Dembla, Anson Ho, Ada Hurst, Brett Armstrong, and Philip J. Guo. 2015. Perceptions of non-CS majors in intro programming: The rise of the conversational programmer. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Atlanta, GA, 251–259. https://doi.org/10.1109/VLHCC.2015.7357224

[5] Parmit K. Chilana, Rishabh Singh, and Philip J. Guo. 2016. Understanding Conversational Programmers: A Perspective from the Software Industry. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, New York, NY, USA, 1462–1472. https://doi.org/10.1145/2858036.2858323

[6] Kathryn Cunningham, Rahul Agrawal Bejarano, Mark Guzdial, and Barbara Ericson. 2020. "I'm Not a Computer": How Identity Informs Value and Expectancy During a Programming Activity. In *Proceedings of the 14th International Conference of the Learning Sciences*. International Society of the Learning Sciences (ISLS), 705–708.

[7] Kathryn Cunningham, Barbara J. Ericson, Rahul Agrawal Bejarano, and Mark Guzdial. 2021. Avoiding the Turing Tarpit: Learning Conversational Programming by Starting from Code's Purpose. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3411764.3445571

[8] Johan De Kleer. 1984. How circuits work. *Artificial intelligence* 24, 1-3 (1984), 205–280.

[9] Kathi Fisler, Shriram Krishnamurthi, and Janet Siegmund. 2016. Modernizing Plan-Composition Studies. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (Memphis, Tennessee, USA) *(SIGCSE '16)*. Association for Computing Machinery, New York, NY, USA, 211–216. https://doi.org/10.1145/2839509.2844556

[10] John S. Gero. 1990. Design prototypes: a knowledge representation schema for design. *AI magazine* 11, 4 (1990), 26–26.

[11] Ashok K. Goel, Andrés Gómez de Silva Garza, Nathalie Grué, J. William Murdock, Margaret M. Recker, and T. Govindaraj. 1996. Towards design learning environments — I: Exploring how devices work. , 493–501 pages. https://doi.org/10.1007/3-540-61327-7_148

[12] Ashok K. Goel, Spencer Rugaber, and Swaroop Vattam. 2009. Structure, behavior, and function of complex systems: The structure, behavior, and function modeling language. *Artificial intelligence for engineering design, analysis and manufacturing: AI EDAM* 23, 1 (2009), 23–35.

[13] Cindy Hmelo-Silver. 2004. Comparing expert and novice understanding of a complex system from the perspective of structures, behaviors, and functions. , 127–138 pages. https://doi.org/10.1016/s0364-0213(03)00065-x

[14] Amruth N. Kumar. 2015. The Effectiveness of Visualization for Learning Expression Evaluation. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (Kansas City, Missouri, USA) *(SIGCSE '15)*. Association for Computing Machinery, New York, NY, USA, 362–367. https://doi.org/10.1145/2676723.2677301

[15] Patricia Marsh. 2007. What is Known about Student Learning Outcomes and How does it relate to the Scholarship of Teaching and Learning? *International Journal for the Scholarship of Teaching and Learning* 1, 2 (July 2007). https://doi.org/10.20429/ijsotl.2007.010222

[16] U.S. Bureau of Labor Statistics. 2021. *Software developers, quality assurance analysts, and testers : Occupational outlook handbook*. U.S. Bureau of Labor Statistics. https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm

[17] Kathryn Rich, Carla Strickland, and Diana Franklin. 2017. A Literature Review through the Lens of Computer Science Learning Goals Theorized and Explored in Research. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) *(SIGCSE '17)*. Association for Computing Machinery, New York, NY, USA, 495–500. https://doi.org/10.1145/3017680.3017772

[18] Carsten Schulte. 2008. Block Model: An Educational Model of Program Comprehension as a Tool for a Scholarly Approach to Teaching. In *Proceedings of the Fourth International Workshop on Computing Education Research* (Sydney, Australia) *(ICER '08)*. Association for Computing Machinery, New York, NY, USA, 149–160. https://doi.org/10.1145/1404520.1404535

[19] Elliot Soloway. 1986. Learning to Program = Learning to Construct Mechanisms and Explanations. *Commun. ACM* 29, 9 (Sept. 1986), 850–858. https://doi.org/10.1145/6592.6594

[20] Elliot Soloway and Kate Ehrlich. 1984. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering* SE-10, 5 (Sept. 1984), 595–609.

[21] Juha Sorva. 2013. Notional Machines and Introductory Programming Education. *ACM Trans. Comput. Educ.* 13, 2, Article 8 (July 2013), 31 pages. https://doi.org/10.1145/2483710.2483713

[22] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A Review of Generic Program Visualization Systems for Introductory Programming Education. *ACM Transactions on Computing Education* 13, 4 (2013), 15.1– 15.64. https://doi.org/10.1145/2490822

[23] Kesler Tanner, Naomi Johnson, and James A. Landay. 2019. Poirot: A Web Inspector for Designers. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3290605.3300758

[24] April Y. Wang, Ryan Mitts, Philip J. Guo, and Parmit K. Chilana. 2018. Mismatch of Expectations: How Modern Learning Resources Fail Conversational Programmers. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, Montreal QC Canada, 1–13. https://doi.org/10.1145/3173574.3174085