Accessibility of Collaborative Programming for Blind and Visually Impaired Developers

by

Maulishree Pandey

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Information)
in the University of Michigan
2023

Doctoral Committee:

Associate Professor Steve Oney, Chair
Associate Professor Andrew Begel, Carnegie Mellon University
Assistant Professor Anhong Guo
Associate Professor Sile O'Modhrain

Maulishree Pandey
maupande@umich.edu
ORCID iD: 0009-0005-0543-3088

DEDICATION

To Maa and Papa who let me choose my adventure.
To Choti who gave me perspective on this journey.
And to Murru, who was my pit stop
and helped me cross the finish line.

ACKNOWLEDGMENTS

On October 6, 2016, as a newly enrolled master's student, I emailed Steve Oney to request a meeting. I had heard him share his research at MISC, the lecture series hosted by my school. I was hoping for an opportunity to do research with him alongside my master's. Steve replied a few days later in a tone characteristic of him - kind, respectful, and even apologetic for the delay in responding (he was only six days late, four if you count the weekend). Little did I know then that the meeting would lead me to a PhD. As I approached the end of my master's, he suggested I apply for doctoral programs. I was convinced about finding an industry job. Among the plans I have abandoned, it remains the best one yet! Over the years, Steve has given me room to grow as a researcher. I recall chasing a myriad of ideas in the first semester. He would patiently hear each of them, give me the time to explore, and switch. When I found the topic that became my PhD, he suggested I collaborate with people who are well-versed in the space and maximize my mentorship. When I expressed doubts about my ability, he told me I was ready but he would let me chart my timeline. Looking back, it seems Steve knew reasonably well the endeavors I could embark on, including this dissertation. He has made me a better researcher and taught me to collaborate through humility, patience, and due credit to others. Thank you, Steve! I aspire to be a mentor like you someday.

When I interned with Andrew Begel in the summer of 2021, I was unprepared for the amount of fun, research takeaways, and victories! He helped me bring together all the strands that had shaped my research philosophy and turn the project into a usable product. Andy constantly nudges me to challenge research ideas that are deemed settled. One of my favorite memories was when we exchanged over 30 emails to take apart the data analysis for CodeWalk. Thank you for always making time for me and encouraging me to take risks.

Every time I reach out to Sile O'Modhrain for an impromptu discussion, she makes time! She helped me get started with accessibility research and has helped me bring the journey to a close. She has taught me to tinker with the methods to get at the answers. Through her, I met Joshua Miele, James Coughlan, and Giovanni Fusco, who helped me iron out my first major study. The study remains a special one! I remain indebted to Sile for her time, attention, and humor.

Anhong Guo provided feedback when I needed it, including detailed notes on writing drafts. I have drawn inspiration from his research approach and how he maintains project implementations, new and old. His incredible students have helped me identify extensions to my work.

My research would not have been possible without my participants' time, perspectives, and insights. I look forward to a lifetime of friendship with many of them.

There are several other faculty members and researchers whose support got me here. Tao Dong has taught me to adapt my work to the industry. Always prompt, his guidance has ranged from academic to career advice. Robin Brewer helped me hone my field prelim. Coursework with Joshua Kupetz, Mark Ackerman, and Mark Guzdial formed a strong foundation for my research. Conversations with Cecily Morrison and Denae Ford helped me frame my contributions for the broader audience. I am grateful to Michael Nebeling, Tawanna Dillahunt, Zoya Bylinskii, Valentina Shin, Sun Young Park and Barbara Ericson for their time at different points in my academic journey.

Venkatesh Potluri and I have focused on the same research topics and yet found a way to build on each other's ideas. Our long conversations about science combined with his excellent food suggestions remain special. I have turned to Vaishnav Kameswaran for every problem big and small. He has been a collaborator, mentor, and friend. While he may deny it, he has emerged as a leading research voice in his space. With him and Hrishikesh Rao, there is always space for more mischief. Hrishi has an uncanny ability to listen and give great advice. He is also among the rare few who receive all feedback with humility. Yan Chen and Hariharan Subramonyam welcomed me on their projects when I was starting out. I engineered a snappy system with Tianyuan Cai's precise instructions for research beyond my dissertation. Thank you to Sharvari Bondre and Chris Hong, who did research with me over the semesters. I hope I was able to impart the knowledge and wisdom I have learned from everyone else around me. I appreciate Brooke Sasia and Chetan Keshav for their help with transcription, analysis, and development.

I feel lucky to have found close friendships with Jane Im and Rebecca Krosnick over dinner and squash. I eagerly await their stellar dissertation defenses. Penny Trieu and Minha Noh have given me glimpses of the journey that awaits me. Thank you for being transparent and supportive! Shout out to my peers who made grad school wonderful – Warren Li, Divya Ramesh, Gabriel Grill, Amna Batool, , Xinghui Yan, April Wang, Lei Zhang, Heeryung Choi, Abraham Mhaidli, and Lu Xian. Some of my most cherished memories were created in DoIIIT Studio with Jeff Huang, Licia He, Cindy Lin, and Jasmine Jones. I spent a lot of time tinkering, laser cutting, and putting together odd projects. A doctoral experience is complete when it includes folks who expand your understanding of the world and show you how to improve it. I cannot wait to read Pratiksha Menon's dissertation cover to cover. I draw inspiration from Linda Huber's commitment to making grad school better for everyone.

I improved not only as an instructor but also as a programmer when I GSI-ed for Jackie Cohen. You trusted me with running coding boot camps. It came in handy when I planned bigger summer courses. Thanks to Chris Teplovs for being the most accommodating lecturer during the pandemic. You both made teaching a joy! Special thanks to Amy Eaton for patiently resolving all

my administrative issues!

I now turn to friends and family who are my pillars. I can never express enough gratitude to my parents, Divya Thakur and Arvind Pandey, for letting me be so unapologetically 'me.' Aruja Pandey has taught me to be perceptive. Yet I am nowhere close to her wisdom. I was lucky to be only hours away from Utpal Singh and family, and only a plane journey away from Manisha Rai.

Rashmi Muraleedhar (Murru) has been my source of strength. I turn to her when I need to rant, reflect, and relate. I curious to see where your academia takes you! Pallavi Chogale has given me room to fall and recover physically and emotionally. Her vexation, tips, and long voice notes have been critical to my healing. I continue to turn to Esha Jhaveri when I need a lesson in empathy and kindness. Esha, you have taught me to be better since we were fourteen.

I knew I was cutting it close when Akanksha Gupta (Kancha) said, "mujhe tension ho rahi hai! likh le" (Listen, I am getting stressed. Finish writing!). Everyone needs a friend like her in their corner. Khushi Gupta's (Khuf) gentle advice (along with Kancha's tough love) helped me figure out post-PhD plans. Thank you for the subtle ways in which you protect me. Minal Jain showed faith in all the silly, occasionally creative, ideas I sent her. I continue to tease apart moral dilemmas with her. The trio and Viresh Gehlawat (plus Smokey) are my safe place in the US. I have wanted to move closer to them and I am glad it is finally happening. I did not particularly think of the ups and downs of grad school. Shruti Soni prudly reminded me of how far I had come. Divya Nandwana woke me on my worst mornings and made sure I was writing. You have kept me accountable and going since 2010. I cannot wait to celebrate with you and Soujanya Mandadi. Souj, I still do not know how you manage to find time every time I visit, but I am glad you do. Abhay Pande has been a patient friend, who celebrates my wins even when I relay them to him too late.

I pulled through the final few months because of Anjali Singh. My only regret is we became close friends so late in the Ph.D. You allowed me to grieve; you taught me gratitude. You also introduced me to the ever so funny Harshita Kondisetti and the dashing Coco! Every time Ann Arbor (and California) became lonely, I turned to Namita Nisal. Her generosity carried me through my worst times, including my physical injuries. Rahul Sawn gave me the confidence to climb again. I look forward to our future adventures in the wild west. Richa Mukerjee was my co-working partner in the months leading up to my defense. Thank you for helping me focus when I needed it.

I thank Smith-Kettlewell Eye Research Institute and LightHouse for the Blind and Visually Impaired for their assistance in study planning and recruitment. My research was supported by Google and NSF Award 2007857.

I have thought about this section the longest. It will always remain in the making.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

LIST OF TABLES

TABLE

LIST OF APPENDICES

LIST OF ACRONYMS

ARIA Accessible Rich Internet Applications

AT Assistive Technology

BVI Blind and Visually Impaired

CLI Command Line Interface

GUI Graphical User Interface

HCI Human-Computer Interaction

IDE Integrated Development Environment

JAWS JobAccess With Speech

JSON JavaScript Object Notation

NVDA NonVisual Desktop Access

UI User Interface

WCAG Web Content Accessibility Guidelines

WIMP Windows, Icons, Menus, Pointing Device

WYSIWYG What You See Is What You Get

XML Extensible Markup Language

ABSTRACT

The profession of software engineering is highly collaborative. Developers must perform synchronous activities and are expected to be aware of each other's actions as they make asynchronous contributions to the source code. In a mixed-ability team involving sighted as well as blind and visually impaired (BVI) developers, effective collaboration requires accessibility considerations and improvements across four broad areas. First, the programming environment that comprises programming and collaborative software has to be made accessible. Second, the graphical information has to be adapted to the audio medium to support BVI developers' full participation in collaborative activities. Third, the growing popularity of cross-platform frameworks within product teams and their claims of creating accessible applications has to be questioned as they can have a bearing on BVI developers' workflows and the experiences of BVI end users. Finally, since all collaboration is centered around contributing maintainable source code, we need to consider BVI developers' readability preferences to ensure ease of programming on screen readers. This dissertation reports on four studies that investigated the areas mentioned above.

I first conducted a qualitative study to investigate the logistics of collaborative programming in mixed-ability workplaces. I identified various sociotechnical challenges that impacted communication, help-seeking, and collaboration between BVI and sighted developers. It revealed the extra articulation work BVI developers have to perform to modify the established work practices.

Drawing on the first study and existing research on group work, I implemented CodeWalk, a set of features to improve shared awareness during code walkthroughs. CodeWalk demonstrated that we can reduce the burden of additional work, specifically coordination work, on BVI developers by designing inclusive collaborative tooling.

Next, I conducted an empirical study to examine the accessibility of UI frameworks and libraries, which are growing in popularity across software engineering teams. I report how these enable BVI developers to upskill themselves as front-end and full-stack developers but also lead to breakdowns in code authoring, debugging, and collaboration. The study also yielded insights regarding setup work, which is the work that goes into finding, configuring, and installing accessible frameworks and tools.

I conclude with a study on what constitutes code readability for BVI developers. Code reading is a fundamental asynchronous programming activity critical to long-term software maintenance. I contribute a taxonomy for what is good code formatting on screen readers vs. GUIs to support

better code readability in mixed-ability teams. I also derive recommendations for IDE tooling to reduce the effort that BVI and sighted developers need to expend toward maintaining code quality.

The primary contributions of this dissertation are the various forms of additional work BVI developers have to perform across synchronous and asynchronous collaboration. I offer recommendations for organizing information across GUIs and screen readers for accessible collaboration in mixed-ability contexts.

CHAPTER 1

Introduction

Software engineering is among the most collaborative professions, involving a mix of synchronous and asynchronous activities at different points in the software development lifecycle. Developers have to work closely with other team members to plan the software design and functionality, situate their individual tasks within the larger project, and choose the technical stack and tools for development. They have to review each other's code to provide and receive feedback and to inform their own contributions to the source code. Lastly, they have to write code that is readable by their current colleagues and comprehensible to future team members to ensure ease of software maintenance in the long run. For effective collaboration in these activities, collaborating developers need to be aware of each other's activities to plan their actions [71]. Specifically, software engineering requires focused awareness, a necessary condition for tightly-coupled synchronous activities, as well as peripheral awareness, where people are roughly aware of the coding responsibilities of others in the team [18]. The systemic and ableist biases against people with disabilities presume that developers engaging in collaborative activities are sighted. As a result, the tools and activities have evolved to support sighted developers and have neglected the needs of blind and visually impaired (BVI) developers. Thus, mixed-ability teams which comprise sighted as well as BVI developers, tend to present accessibility challenges to the latter.

There are several reasons to investigate and address the accessibility challenges in collaborative programming. The most important argument for accessibility comes from the social model of disability [155], which states that the society's failure to take into account the needs of people with disabilities prevents them from participating on an equal basis with others. In the case of software engineering, the programming community's inadequate support for BVI developers has limited them from contributing to their full potential [155]. They have to workaround accessibility challenges and advocate for more accessible practices to ensure their full participation in the workplace [46]. Despite organizations attempting to increase the diversity in the workplace and the demand for developers being at an all-time high, BVI developers continue to form only 1.1 — 1.7 % of the developer community [205, 158, 159]. By continually addressing the accessibility

issues in collaborative programming, the field is bound to attract more talent from among BVI developers. The efforts would make the programming community more inclusive and result in better employment outcomes for BVI developers.

1.1 Graphical User Interfaces and Screen Readers

Before describing the broad challenges that constitute the (in)accessibility of collaborative programming, let's discuss how sighted and BVI people interact with computers and mobile devices. Sighted people typically rely on the visual medium, which enables a rich suite of browsing and navigation strategies owing to its two-dimensional nature. On the other hand, audio is the primary medium of interaction for BVI people. It is more serial and ephemeral in its output of information [22]. Below I describe the broad differences in both interaction mediums.

Graphical user interfaces (GUIs) were developed at Xerox PARC. They started gaining popularity when Apple released Macintosh in 1984 and soon became the standard interface for computer applications [227]. GUIs introduced the WIMP interaction technique – windows, icons, menus, and a pointing device, typically a mouse or a touchpad. With the arrival of GUIs, it was far easier to manipulate two-dimensional information (e.g., calendars, spreadsheets, visualizations, etc) by pointing and clicking. The interaction was a significant step up from command line interfaces (CLIs), which displayed information serially, and users had to write complex commands to work with the information [227]. GUIs have also standardized the 'What You See Is What You Get' (WYSIWYG) paradigm, where sighted users saw the results of their interaction in real time.

Screen readers are computer applications that provide spoken feedback to BVI users. In CLI days of operating systems, screen readers accessed the text displayed on the screen and relayed it as audio output [176]. Thus, sighted and BVI users accessed pretty much the same information through different mediums. The screen content changed from text to 2D graphics as GUIs became more dominant. Screen readers had to go from rendering screen text as audio to providing access to graphical information. Jim Thatcher[1] and Jesse Wright developed the first screen reader at IBM to make GUIs accessible [220]. In present times, JAWS (Job Access with Speech) [80] and NVDA (NonVisual Desktop Access) [153] are among the most popular GUI screen readers. All major operating systems also include screen readers as part of their platforms, for example VoiceOver [11] on Mac OS and Orca [222] for Linux and Unix operating systems. However, the finer aspects of visual information (e.g., the relationship between windows, color-coded information, etc) remain cumbersome to access via screen readers. To summarize, BVI and sighted users are not accessing

[1]Dr. Thatcher was among the first Ph.D. holders in Computer Science. He completed his doctorate in 1963 from the University of Michigan. His advisor, Jesse Wright, who was blind, had a significant influence on the development of the IBM Screen Reader/2.

the same information, which makes collaboration challenging.

1.2 Collaborative Programming in Mixed-Ability Teams

To make collaborative programming accessible for blind and visually impaired (BVI) developers, we need to investigate and improve the following four areas for screen reader use (see Figure 1.1): (1) programming tools (e.g., code editors, static analysis tools, etc.) and collaboration tools (e.g., code review tools), (2) collaborative activities such as pair programming, code reviews, etc. (3) the programming technologies or technical stack used by product teams (e.g., UI frameworks) (4) the source code that BVI developers and sighted developers contribute and collaborate over. I provide further details on each of these below:

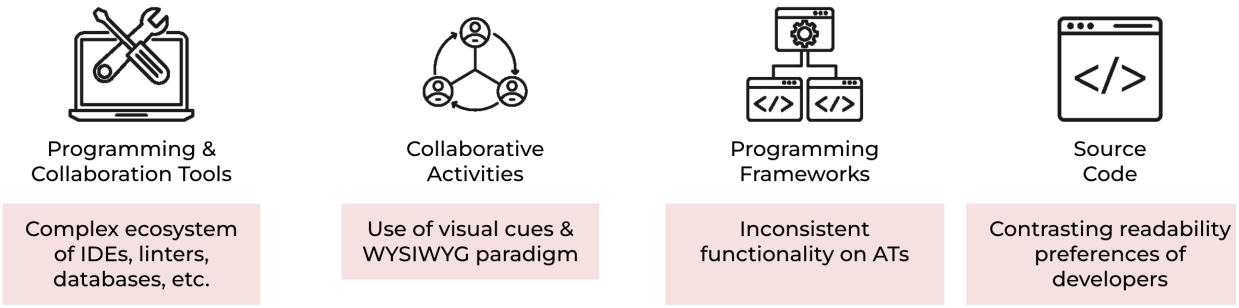


Figure 1.1: Four key aspects of collaborative programming in mixed-ability teams. Red boxes show examples that lead to accessibility breakdowns in each of them

1.2.1 Programming and Collaboration Tools

Much prior accessibility research has looked into improving programming tools such as code editors and debuggers (see §2.3). However, programming is hardly ever limited to these tools. It also relies on collaboration tools such as communication software, version control systems, etc., and can even require internal company tools such as code reviewing platforms, code formatters, etc. Similar to programming tools, collaboration software lack accessibility for all of their features and often interrupt the workflows of BVI developers. In Chapter 3, we report on the range of tools BVI developers use for collaboration. Prior work has shown that BVI people have to perform additional work to address accessibility problems [47]. To ensure productivity, we need to make both programming and collaboration tools accessible, keeping in mind the social contexts of their use.

1.2.2 Collaborative Programming Activities

As mentioned earlier, a shared workspace should make collaborators aware of each others' activities to help plan their actions [71]. Unfortunately, the asymmetry of information on screen readers and GUIs means that developers struggle to form awareness about their collaborators' actions in mixed-ability contexts. In Chapters 3 and 4, we demonstrate how BVI developers have limited access to the shared workspace, which limits them to specific roles during collaboration. We design and develop CodeWalk to show how a shared workspace that relies on visual feedback can be made accessible through additional audio feedback for synchronous and remote code walkthroughs.

1.2.3 Programming Frameworks

Programming frameworks such as React [131] and Flutter [89] have made UI programming easy and scalable. Developers can engineer web and mobile applications for multiple operating systems and devices from a single codebase. These frameworks also offer well-designed UI components, requiring minimal visual modification during UI programming. However, as these frameworks become ubiquitous across software development teams [44], we need to pause to examine their accessibility for BVI developers. We do not know how they impact collaborative aspects such as coauthoring code, debugging, and team presentations. Chapter 5 investigates how UI frameworks affect BVI developers' programming and collaboration workflows.

1.2.4 Source Code

Code reading is one of the most crucial activities in software engineering and, arguably, the most performed one. Developers read code written by their colleagues during synchronous and asynchronous collaboration. Readable code is easier to understand, extend, and maintain, with the lattermost being the costliest and most extensive aspect of any project's lifecycle [40]. In large projects and teams, developers are expected to contribute code that is easy to read [180, 72]. The GUIs of code editors encode visual cues about programming syntax, errors, and comments to improve readability for sighted developers. These details are currently unavailable on screen readers, demonstrating a lapse in WYSIWYG for BVI users. In addition, our current understanding of readability is based on the preferences of sighted developers. In Chapter 6, I examine what makes code readable to BVI developers and how we can expand the definition of readability for the larger programming community.

The four are not disparate problems but overlap with one another and collectively form an ecosystem. My dissertation shows how accessibility challenges within one group can feed into

another. To improve the accessibility of collaborative programming and ensure shared awareness of sighted and BVI developers, we need to consider the groups as a whole.

1.3 Thesis Statement

My thesis statement is that **the interplay between programming and collaboration tools, assistive technologies (ATs), programming practices, and the organizational norms around communication and help-seeking shape the collaborative experiences of BVI developers in mixed-ability contexts; we must design tools to not only improve accessibility but also to minimize the various forms of additional work BVI developers perform to achieve effective collaboration with sighted colleagues.** One way to reduce the burden of extra work on BVI developers is through designing inclusive IDE tooling that makes sighted developers aware of their BVI colleagues' coding and accessibility preferences and vice versa. It is easy to set up and use tooling that is well-integrated into the IDE, reducing the work needed to configure it. When designed to be accessible, it allows BVI developers to use mainstream tools [193]. Lastly, it improves shared workspace awareness for all collaborators [71].

1.4 Contributions

My dissertation extends the prior work in Human-Computer Interaction (HCI), software engineering, accessibility, and collaboration in the following ways:

- I contribute empirical evidence on how common collaborative programming activities are performed in mixed-ability contexts. My findings reinforce that collaboration is a sociotechnical achievement. The social aspects play a significant role in determining the accessible experiences of a BVI developer. For instance, at the outset, the BVI developer may have to adopt the team's choice of programming tools, technology stack, and code styling standards. Through slow and careful social interactions, they educate the team and modify the established work practices to accommodate their access needs.

- I report the various forms of invisible and additional work that characterize collaborative programming in mixed-ability contexts: (1) articulation work with team members, (2) coordination work during synchronous and asynchronous collaboration, (3) setup work to configure one's programming environment. I derive design recommendations on how to reduce each of them, with CodeWalk serving as an end-to-end example of the design criteria and process one can follow to improve the accessibility of collaborative programming.

- My research detaches the source code text from its visual appearance and formatting. I show that while it is vital to translate the information available in visual markup of code, the source code itself is not fully available to screen reader users. Put otherwise, the WYSIWYG paradigm holds for sighted developers, but the screen reader output does not fully map to the on-screen text for BVI developers. I report the readability preferences of BVI developers and contribute an inclusive taxonomy for code readability. Using this taxonomy, we can rethink code styling guidelines and static analysis tools to bridge the difference in GUI text and screen reader output.

1.5 Dissertation Outline

My outline for the rest of this thesis is as follows:

- In Chapter 2, I describe prior research on collaboration, software engineering, HCI, and accessibility. The literature review revealed the gaps in the accessibility of collaborative programming and how I attempt to address them through my dissertation.
- In Chapter 3, I describe the study we conducted to identify the challenges BVI developers face in the various collaborative programming activities of pair-programming, code reviews, software design, and UI development. The study provided insights into the logistics of working in mixed-ability workplaces. It showed that BVI developers have to expend additional efforts in articulating their programming and collaboration preferences to their sighted colleagues, which ultimately results in the co-creation of more accessible solutions and work practices.
- Chapter 4 describes CodeWalk, a set of features added to Microsoft's Live Share VS Code extension to support remote and synchronous code review and refactoring tasks. CodeWalk is an example of how to design accessible IDE tooling to increase shared awareness and reduce the burden of additional work on BVI developers.
- In Chapter 5, I describe the study we conducted to understand the collaborative experiences of BVI developers in UI development as they use UI frameworks and libraries. The study found that while UI frameworks upskill BVI developers and enable them to work more independently, inaccessible UI components affect BVI developers' code authoring, debugging, and testing workflows. Furthermore, inaccessible programming tools lead to differences in how sighted and BVI developers author UI code, ultimately complicating collaboration.
- In Chapter 6, I investigate the code readability preferences of BVI developers and explain how they differ from the preferences of sighted developers. I conclude the chapter with

recommendations for code formatters, code style guides, and programming languages, which have so far prioritized readability for sighted developers.

- Lastly, Chapter 7 summarizes the work so far and situates the dissertation within the existing HCI, accessibility, and software engineering research. We also describe future work that can build on the contributions of the current dissertation.

CHAPTER 2

Background[1]

My thesis builds on the prior research in two primary areas: (1) the accessibility of programming, which has primarily focused on studying and improving the accessibility of individual programming tools rather than investigating them as sociotechnical challenges, (2) the accessibility of group work, which due to systemic and ableist biases, has assumed sighted people to be the default parties to collaboration and has led to tools and activities geared at improving collaboration among sighted developers.

I start by discussing collaborative activities central to software engineering (see §2.1). Next, I discuss the theoretical frameworks on collaboration and awareness and how these have shaped collaboration tools for developers (see §2.2). I then turn to early empirical studies with BVI developers that helped improve the accessibility of programming tasks developers typically perform in their individual capacities (e.g, code navigation, debugging, etc.) (see §2.3). I then describe the growing trend in accessibility research to understand the situated use of technologies (see §2.4). The final section draws on the literature discussed so far to highlight the gaps in accessibility of collaborative programming (see §2.5).

2.1 Activities in Collaborative Programming

The software engineering, CSCW, and HCI communities have largely recognized the importance of collaboration and communication in programming [191, 25, 202, 115]. To coordinate development and maintenance of complex software, the process often begins with planning the *software architecture*, which is "commonly considered to be the structure of a large piece of software, presented

[1]Parts of this chapter are adapted from the Related Work sections of these publications: [1] Pandey, Maulishree, *et al*. Understanding accessibility and collaboration in programming for people with visual impairments. In *Proceedings of the ACM on Human-Computer Interaction 5.CSCW1 (2021): 1-30.* [2] Potluri, Venkatesh, and Pandey, Maulishree *et al*. Codewalk: Facilitating shared awareness in mixed-ability collaborative software development. In *Proceedings of the 24th International ACM SIGACCESS Conference on Computers and Accessibility. 2022.* [3] Pandey, Maulishree, *et al*. Accessibility of UI Frameworks and Libraries for Programmers with Visual Impairments. *In Proceeedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2022).*

as a nested set of box and arrow diagrams" [102]. The architecture communicates the relationships between different components like the database, servers, and the user-interface [84]. It enables team members to develop a common vocabulary of the software and facilitates communication among them.

Some workplaces use *pair programming*—a software development practice where programmers work side-by-side to write and review code [50]. In one of the most common models of pair programming, one programmer is responsible for typing the code. They are known as the *driver* (leader), while their colleague, the *navigator* (follower), gives instructions and feedback on the code being written. Pair programming results in more high-quality code, creative problem-solving, knowledge transfer among team members, and higher work satisfaction for the programmers involved [240, 152].

Programmers also have to concurrently edit the same code in software development [75] and data science [233]. They have to adhere to agreed-upon rules of code writing and styling to maintain the readability and consistency of code [180]. Readability is critical to software maintenance, which tends to form 70 percent of any software project's life cycle [40]. In large software development companies such as Google [114] and Facebook [226], developers engage in *code review* to maintain code's readability and long term quality. The process not only enables code compliance with established code styling guidelines [72] but also helps find defects in code, build awareness about the project, and find alternative solutions to programming problems [16]. I discuss the factors that shape readability in Chapter 6 (see §6).

2.2 Awareness for Sighted People in Remote Collaboration

Groupware is only effective when it supports collaboration across time and space constraints in a shared workspace [106]. Buxton describes a shared workspace in terms of (1) person space [53], (2) task space [53], and (3) reference space [52]. Person space offers a strong sense of copresence with remote collaborators. For instance, teleconferencing platforms combine video, audio, and even chat to convey facial expressions, gestures, and spoken messages. Sharing the task space refers to being copresent in the context of the task itself. In collaborative software development activities, the shared source code forms the task space. Reference space is where the person and task overlap [52], allowing remote participants to gesture and point to reference one another as well as the task at hand. An example is using text highlighting during screen shares to direct collaborators' attention to specific details.

Dourish and Bellotti identified two approaches to present awareness information in shared workspaces — active and passive [71]. Active approaches include role assignment (e.g. owner, reviewer, editor, etc), audit trails, annotations, and messaging. Active mechanisms require explicit

action on collaborators' part (e.g. leaving a comment on the shared artifact like document or source code). Conversely, conveying a collaborator's whereabouts and edits automatically in real-time is a passive approach — collaborators do not have to make explicit efforts to communicate their actions.

Scholars and practitioners have blended different kinds of shared workspaces with active and passive approaches to communicate awareness information in remote synchronous software development. Consider the example of real-time co-editing of source code. A primary concern is that the shared activity should not introduce bugs, preventing the code from successfully compiling [87]. IDE plugins like FASTDash [30] and Syde [98] summarize the real-time activity of collaborators to help them avoid editing conflicts. They also allow developers to leave annotations to inform collaborators of their actions. Systems like AtCoPE [76] and Collabode [87] allow programmers to concurrently edit the source code, enabling collaboration in a shared task space. CodePilot [236] extends the activities supported in the shared task space to allow collaboration across the software development process — editing, testing, debugging, and even version control.

Real-time activities like pair programming and code walkthroughs impose an additional requirement on developers to remain *closely* coordinated [218]. Collaborators are supposed to work in a tightly-coupled manner and have *focused awareness*, as opposed to *peripheral awareness*, where people only have rough ideas of each others' actions [18]. Therefore, real-time collaborative programming activities often rely on explicit role assignment. In pair programming, the leader drives the session and writes or explains the code; the follower offers feedback or heeds the explanations. To collaborate efficiently and maintain mutually recursive awareness of one another (also known as *shared intentionality* [218]), the participating developers must look at the same regions within the source code.

Saros [182], an Eclipse plug-in, displays each programmer's text cursor to communicate their location in the source code and provides a *Follow* mode for programmers to sync their IDE viewports during pair programming and code walkthroughs. D'Angelo and Begel used a novel gaze visualization technique to communicate the lines of code a collaborator was looking at during pair programming [61]. The visualization changed color when both programmers' gaze overlapped, communicating awareness and co-presence passively in shared task space. Their evaluation of the gaze visualization technique revealed that when the collaborators' eye gaze was in sync, they more efficiently spoke about their source code using deictic references (*e.g.* terms such as this, here, that, etc.)

The research discussed above has focused solely on collaborations among sighted developers; these studies do not report anything about the needs of BVI developers. The tools noted above rely heavily on visual information, which leads to significant accessibility problems in mixed-ability contexts [117, 164], including software development [162]. Prior work offers limited insights

about collaboration in mixed-ability programming contexts. To shed light on how sighted and BVI developers achieve collaboration and build shared awareness, I first discuss HCI studies that sought to improve the productivity of BVI developers as individual contributors. I follow this with a discussion of the more recent accessibility literature that investigates BVI people's interactions with technologies within their context of use. The two overlapping research areas illuminate the gaps that we need to address to improve the accessibility of collaborative programming.

2.3 Accessibility and Programming

Prior work in HCI has investigated accessibility challenges related to individual tools. Mealin and Murphy-Hill were the first to touch upon the high-level accessibility challenges in software engineering [130]. They found that programmers faced challenges when using integrated development environments (IDEs), seeking information in IDEs with screen readers, and programming user-interfaces. Subsequent studies have confirmed the lattermost finding [198, 7].

The access challenges in IDEs can be broadly categorized into four groups: (1) *discoverability* of IDE features, (2) *glanceability* of information in various panels, (3) *navigability* of code, and (4) *alertability* of errors and bugs [173]. IDEs and text editors rely heavily on visual aids such as syntax highlighting and indentation to assist in source code navigation, organization, and visual search [8, 173, 187]. IDEs also organize information visually into panels and windows. Since IDEs are designed for GUIs first and only *adapted* for screen readers, BVI developers do not get access to the visual structure of IDEs, making it difficult for them to locate the features and windows efficiently [176]. TV Raman's Emacspeak offers an alternative approach to code editors by making the spoken feedback sufficient for interaction [176].

The challenges in IDEs are exacerbated by lack of accessible information about the IDE features [166]—documentation about programming tools is often designed for sighted developers, relying on visual content such as screenshots [19]. In addition, keyboard shortcuts that programmers with visual impairments rely upon are generally complex [19]. This increases the cost of learning and deters programmers from switching over to new tools. A common workaround is switching to plaintext editors [130, 7] in conjunction with command-line interfaces (CLIs) for installation, debugging, and version control [184]. However, the latter present text in unstructured form without any markup, which poses navigation challenges for screen reader users [184]. BVI developers also seek sighted assistance to address the accessibility issues with programming tool use [7]. However, seeking assistance draws attention to the additional time it takes them to complete programming tasks [8] and can reinforce ableist perceptions about their abilities, as I show in Chapter 3.

Researchers have developed tools to address the above challenges. Researchers have proposed audio-based code navigation to assist with navigability [104, 78, 20, 173]. Similarly, tools like Sod-

Beans [209] and CodeTalk [173] suggest that audio-based tools can significantly reduce debugging time. There is also a push towards developing accessible programming environments that can be integrated with various programming languages [187]. While these solutions have demonstrated promise for visually impaired programmers, they also result in multiple disintegrated solutions for different problems. It also places the onus on the programmer to find these tools, maintain appropriate versions and integrate them into their workflow.

2.3.1 Accessibility of UI Development

When it comes to specific programming activities such as UI development, solutions are mostly targeted at sighted developers and designers to support them in building accessible interfaces. I highlight them for two reasons. First, their underlying interactions and interfaces remain visual and, therefore, of limited use to programmers with visual impairments. For example, Hansen *et al*. created an interactive tool to recommend sufficient color contrast in UI designs [97]. While developers and designers with visual impairments would find utility in such a tool, its reliance on visual elements limits its generalizability to the group. Second, these studies provide valuable insights into the limitations of UI frameworks as they are used by sighted developers [245]. Sighted developers have found that Xamarin [138] and React Native [131] do not expose all the accessibility APIs, making it difficult to create fully accessible mobile applications [128]. Similarly, the web frameworks, Angular, Vue, and React, do not notify sighted developers about accessibility violations [122]. Empirical studies offer insights into the challenges of creating webpages using HTML, CSS, and JavaScript [121, 112, 151]. Programmers with visual impairments have shared that they feel less confident about CSS modifications [130, 111] and seek sighted assistance to verify the layout and CSS edits [121, 151]. Prior work has also revealed that people with visual impairments find it easier to understand the spatial layout on touchscreens compared to computers [170]. On the other hand, most layout editors within IDEs do not offer pixel positions, relative locations, and dimension information. To address these challenges, Borka developed the Developer Toolkit, an NVDA addon, that informs developers of location and dimensions of UI elements [41]. Researchers have also developed multimodal systems to convey spatial layout of webpages in non-visual formats — using tactile print-outs to represent the HTML [121]; organizing tactile beads on a sensing board to create new layouts [192]; using gestures to edit HTML/CSS on tablets with VoiceOver feedback [171]. Potluri *et al.* have discussed the potential of using AI to support color selection, iconography, layout design, etc [169]; their representation in tactile forms (e.g., color wheel diagrams, Braille font charts, etc) have shown promise in teaching web development [111].

The aforementioned studies and systems primarily focus on accessibility challenges with individual tools in isolated contexts. There is a significant gap in the literature with regard to the

challenges programmers face in social contexts. Among the few exceptions are studies that focus on experiences of students with visual impairments in computer science programs [54, 19]. These reported that it is challenging for students to participate in class discussions and access visual materials like slide presentations, diagrams, and notes on whiteboards. The latter challenge continues to persist for programmers with visual impairments [7, 130]. The studies do not discuss the social and personal implications of these challenges for programmers with visual impairments. In addition, the collaborative programming activities that are considered critical for success in the workplace [48] remain understudied. The recent empirical studies suggest that accessibility challenges in professional settings are complicated by workplace dynamics and project management practices [103, 179]. BVI developers often reach out to their sighted colleagues to solve breakdowns in programming tools, especially selecting teammates who understand workflows with assistive technologies [212]. However, as I show in Chapter 3, help-seeking in the workplace can be complicated by the team and organization's attitudes towards accessibility and inclusion.

2.4 Accessibility and the Social

2.4.1 Assistive Technology Use in Social Settings

Accessibility research in HCI is increasingly examining the situated use of ATs and emphasizes considering social contexts when designing them [68]. In prior research, people with disabilities reported that ATs tend to lag behind mainstream products in functionality and aesthetics [194]. They tend to attract unwanted attention to their users due to their design [194, 193] and breakdowns [2, 195], which foregrounds the users' disability [246]. Thus, for people with disabilities, deciding whether to use ATs in social settings is a negotiation between utility, avoiding attention, and feeling self-conscious due to the resulting attention from others in the space. I add to this body of work by studying AT use in a professional context, where such decisions can have additional implications for productivity, perceived competence, and independence.

There are also misconceptions among people without disabilities that ATs make a disabled person "normal" and that their ability is contingent on ATs [194]. Shinohara and Wobbrock therefore recommend designing ATs that enable users to convey their ability and identity [195]. For instance, studies have shown that people with disabilities value their sense of independence [110] and the outward appearance of independence [143]. ATs should then help convey one's independence to others in social settings. This is known as designing ATs for "social accessibility" [195], and is likely to foster sociotechnical access for people with disabilities and enable them to participate in social settings [148]. Shinohara *et al.* suggest three AT design tenets for fostering social access: (1) involving users with and without disabilities in the design process to ground AT design in the

mainstream, (2) considering both functional and social scenarios of AT use, and (3) using design methods that foreground social contexts of use [193].

2.4.2 Help-Seeking and Help-Giving

Seeking assistance in the workplace is an important way for employees to resolve their problems. Gourash defines help-seeking as "any communication about a problem or troublesome event which is directed toward obtaining support, advice, or assistance in times of distress" [91]. The process of seeking help consists of three parts: recognizing the problem, consciously deciding to act on it, and selecting a source for help [59]. Individual attributes like gender, education, race, socio-economic status, and age have been considered when studying the help-seeking process [23]. In the workplace, employees prefer to reach out to experts or senior employees, as they find the help of higher status individuals and experts to be more constructive [147]. Help-seeking from superiors and experts is shaped by awareness of their expertise and ease of access to them [228]. In addition, employees need to trust that help-givers will not judge them for seeking assistance [228]. It is easier to seek help when the problem is shared by many employees, as this attributes the problem to external sources and reduces the risk of judgment [23]. The threat to self-esteem and the inability to reciprocate help can deter people from seeking it [10]. This raises questions about seeking assistance with accessibility challenges, a problem shared only by employees with disabilities.

Help-giving is relatively less studied in research but is considered to be closely intertwined with help-seeking and requires interpersonal interaction among employees. It relies on employees' "sense of citizenship" since they are not formally necessitated to provide help to others [23]. The desire to reciprocate assistance is a key motivator for help-giving in the workplace [90].

Research has attempted to understand when people with disabilities seek help and how it affects them. When seeking assistance as a recourse from malfunctioning ATs, the needs of the people with disabilities are often misunderstood and their autonomy is overridden [239]. For instance, people with visual impairments have reported that sighted people sighted people tend to provide unsolicited help by taking a BVI person's arm to navigate a space. Prior studies have referred to this as unwanted help — assistance provided based on incorrect assumptions about people's abilities and without due understanding of their needs [223, 239]. Seeking assistance also has social costs [234, 244], making the person appear less competent and highlighting their disability. In research with people with visual impairments, participants indicated that they felt the need to reciprocate the help and did not want to burden their friends and family [43]. Instead, they preferred using sighted assistance from crowd workers [157, 43]. This adds more perspective to the question I raised earlier: how do programmers with visual impairments feel about reaching out to sighted people, including their colleagues?

2.4.3 Accessibility in Mixed-Ability Contexts

There is an increasing emphasis on understanding accessibility in mixed-ability contexts [45, 47, 243] and designing technologies that respond to peoples' abilities [241]. This is evident from the various technology-mediated solutions designed to facilitate group work in online photo-sharing [129, 244], learning [82, 133, 141, 142, 224], sports [21], and music creation [156].

Branham and Kane studied how accessibility was achieved and maintained by inhabitants in the context of home spaces [45]. They defined this as *collaborative accessibility*—"taking active roles in co-creating an accessible environment". They found that accessibility was intertwined with personal relationships. Thus, accessibility (and the lack thereof) affected how couples or housemates shared experiences, which in turn impacted the well-being of their relationships. Addressing certain inaccessibility-related challenges could foster kindness and care. In such contexts, technologies should be designed keeping in mind the interdependencies within such relationships [27]. Technologies should also be committed to helping people achieve what matters to them and not be focused solely on accomplishing tasks [29].

It has been shown that people with visual impairments have to perform *invisible work* [207] to address accessibility challenges. While accessibility is created through coordination between multiple technologies and people [49, 110], the onus falls largely on people with visual impairments [234, 62]. For instance, Das *et al.* observed that software updates often break the accessibility of writing tools and ATs. People with visual impairments have to reconfigure the settings and relearn the keyboard shortcuts to continue to collaborate with sighted people on writing projects [62]. Thus, people have to work beyond their professional responsibilities to find accessible solutions [47] and continually advocate for their access needs [231]. The above studies demonstrate the need to develop a situated understanding of technology use to uncover its social implications for solo and group work. My research contributes to this growing body of work.

2.5 Accessibility for Mixed-Ability Programmers in Remote Software Development

Given the limited research on awareness needs of BVI developers, I refer to the empirical insights from the accessibility literature discussed above to identify how sighted and visually impaired developers achieve *real-time*, remote collaboration. A common approach is asking collaborators to describe their actions, but sighted people often forget to verbalize the relevant details, resulting in incomplete collaborator awareness [62], also confirmed in Chapter 3. BVI people hesitate to repeatedly request information to avoid slowing down the pace of the collaboration or imposing on their sighted collaborators [62].

In another workaround, collaborators work on their respective computers with the BVI developer sharing their screen using a video calling application so that the sighted developer can follow them [162]. Collaborators rely on chat features to copy-paste text and share line numbers, etc., to collaborate more accessibly. While this workaround allows a sighted developer to track a BVI developer's location, the information is not reciprocated to the BVI developer. It also places the onus of driving the collaboration session on BVI developers. BVI people occasionally have to relinquish control of their computers to let the sighted colleague control their screen and make real-time changes [215]. Since BVI and sighted computer users navigate interfaces differently [31, 168], the approach causes the screen reader to change focus unexpectedly without feedback to the BVI person, impinging on their agency, and raising privacy concerns [215]. Latency issues also lead to unwieldy drag and click interactions for sighted collaborators.

Another strategy is to use NVDA Remote [225] or JAWS Tandem [188], screen reader addons that transmit announcements instead of simply relaying the video of the collaborator's screen during screen share [34, 215]. Unfortunately, these addons often suffer from long latency, causing the BVI person to receive announcements after 15-30 seconds [215]. Plus, sighted collaborators have to set up the screen reader and the addon with matching configurations at their ends, giving them additional invisible coordination work and adding to the total collaboration time, as I show through my empirical studies. Tools like Sinter address the latency issues and strict configuration requirements of remote screen reading but have not yet been evaluated in collaborative contexts [35].

Das *et al*. designed auditory representations to support *asynchronous* collaborator awareness for activities such as commenting and editing in shared text documents [64]. The evaluation of their design elements revealed that the use of non-speech audio, voice modulation, and contextual presentation could improve awareness of BVI authors. Recently, CollabAlly [119] and Co11ab [63], both Google Docs extensions, have been designed to support *synchronous* collaborator awareness in shared document editing. The extensions use spatial audio and voice fonts to represent the actions of collaborators joining and leaving the document, addition and deletion of comments, and movement of their cursor into or away from the BVI user's paragraph. Co11ab also uses a variant of Follow mode [182] to sync *collaborators' viewports*. Shared document editing differs from pair programming and code walkthroughs in an important way, however. Shared document collaborators need real-time awareness to actively *avoid* each other's cursors in order to prevent overwriting collisions. Software development collaborators, on the other hand, intentionally work on the same lines of code *together*, requiring close and immediate coordination for extended periods of time.

In summary, the workarounds discussed above insufficiently convey collaborator awareness, suffer from delays, place a disproportionate burden on BVI people for driving the collaboration, and compromise their agency. Furthermore, some of the non-visual techniques [64, 119] proposed for collaborative writing do not fulfill the unique needs of collaborative programming, which re-

quires more focused awareness [18].

CHAPTER 3

Understanding Accessibility of Collaborative Programming Activities[1]

3.1 Introduction

The steady increase in lucrative programming job opportunities has the potential to positively impact the aspirations and social mobility of BVI developers. Programming is also considered a *relatively* accessible field in Science, Technology, Engineering and Mathematics (STEM); most programming is text-based, making it easier to write code with assistive technologies (ATs) such as screen readers and braille displays. By contrast, many other STEM fields rely heavily on inaccessible diagrams and equations.

In recent times, programming has moved away from command-line software towards graphical user interface (GUI)-based software like IDEs and text editors. These software have several features that advantage the sighted developers but pose challenges for BVI developers and inhibit collaboration among coworkers [9]. Prior research on Human-Computer Interaction (HCI) has studied the challenges that BVI developers face but much of this work has focused on specific tasks and individual programming tools. Challenges in mixed-ability collaborative contexts remain understudied. This is a gap worth examining because of the social, academic, and professional implications it can have for BVI developers. Most software is built collaboratively; programmers often have to collaborate with other programmers and team members, including designers and project managers [99, 116]. Challenges in collaboration are likely to reinforce some of the ableist perceptions about the abilities of people with visual impairments and limit their opportunities for employment and advancement [60].

In this chapter, we investigate the collaborative experiences of BVI developers with a focus on the following research questions: (1) What are the collaborative activities and associated chal-

[1]This chapter is adapted from the publication: Pandey, Maulishree, *et al*. Understanding accessibility and collaboration in programming for people with visual impairments. In *Proceedings of the ACM on Human-Computer Interaction 5.CSCW1 (2021): 1-30.*

lenges that BVI developers encounter in professional contexts? (2) How do BVI developers address these challenges? (3) What implications do these challenges have for solo and group work? We conducted semi-structured interviews with 22 people with visual impairments who are employed as software developers, data analysts, IT professionals, and researchers. They frequently collaborate with colleagues as part of their jobs. Our findings and the subsequent discussion are relevant to employers and designers who aim to create accessible and inclusive work environments. This work makes several contributions to the Computer-Supported Cooperative Work (CSCW) and HCI literatures:

- An analysis of our interviews with BVI developers, which provides insights into the logistics of working in mixed-ability workplaces. Our findings extend prior work by focusing on sociotechnical challenges such as communication, collaboration, help-seeking, and biases. Our findings also validate many of the challenges that prior work has found with inaccessible individual tools. (see §3.3)
- A discussion to build on the current theorizing of accessibility of group work in HCI and CSCW. We recommend that future research in this area should examine interactions around help, especially provision of help by people with visual impairments. (see §3.4.1.1)
- A discussion on the accessibility of collaborative activities in programming, and design recommendations grounded in our empirical contributions. (see §3.4.1.2)

3.2 Methods

3.2.1 Participants

We conducted semi-structured interviews with 23 people with visual impairments (19 male, 4 female). Participants (P1–P23) were between 24 and 73 years of age. We excluded one participant (P4) from our final analysis because he self-reported his visual impairment as low-vision while the remaining participants identified as nearly or fully blind. As a result, P4 used screen magnification on his digital devices while the other participants used screen readers or a combination of screen readers with braille displays. The screen readers mentioned by the participants included NVDA [153], JAWS [80], Orca [222], and ZoomText [81].

Our participants included software engineers, data analysts, IT professionals, freelancers, and researchers. They were employed in software companies, universities, research organizations, and NGOs. Table 3.1 lists the demographic details of each participant along with the current programming languages they use and the nature of the organization they work in.

Table 3.1: Demographic characteristics of participants in the study on experiences of BVI developers in collaborative programming activities.[2]

#	Age	Gender	Visual Ability	Prog. Experience (in years)	Prog. Languages	Prog. Editors	Organization
P1	29	M	Vision loss from retinitis pigmentosa in early 20s	7	Java	Visual Studio	Freelancer
P2	26	M	Did not share	1-2	HTML, CSS, PHP, Python, Java	Notepad, Eclipse occasionally	NGO
P3	30	M	Blind since birth	11-12	Python, SQL, PHP, JavaScript	Notepad++	Sports Company
P4[3]	45	M	Gradual vision loss from retinitis pigmentosa	20+	.NET, JavaScript, HTML, CSS	Different text editors	Software Startup
P5	24	M	Blind since birth	3-4	Java, Python	IntelliJ, Visual Studio	IT Company
P6	45	M	Blind since 1 year old	10+	Python, HTML, CSS, PHP, Java	Visual Studio, VS Code	Freelancer
P7	32	F	Legally blind with corrected vision 20/200	3	Python, Java, HTML, CSS, JavaScript	VS Code	Healthcare Company
P8	27	M	Blind since 6 years old	4	Python, JavaScript	Visual Studio	IoT Startup
P9	39	M	Blind since birth	20	Python, Go, Perl, SQL	Vim	U.S. State Government ITS
P10	52	M	Lost total vision in an accident at 50	24	Python, JavaScript (Node.js)	VS Code	Telecom Company
P11	39	F	Did not share	2	HTML, CSS, JavaScript	Native Text Editor	U.S. State Government ITS
P12	28	M	Blind since birth	5	C#, Python, Java	Visual Studio	Digital Software Agency

Continued on next page

Table 3.1 – continued from previous page

#	Age	Gender	Visual Ability	Prog. Experience (in years)	Prog. Languages	Prog. Editors	Organization
P13	41	M	Blind since birth	15	HTML, CSS, Java, JavaScript, Python	Eclipse, Notepad++	Software Startup; University
P14	32	M	Blind since birth	19	C#, Android, PHP	Visual Studio	Freelancer
P15	29	M	Blind since birth	13	C, Go, Python, Haskell	Emacs	University; Independent Research Organization
P16	73	M	Vision loss from retinitis pigmentosa in late 30s	30	COBOL	Organization's internal text editor	Retired from Bank
P17	50	M	Vision loss from retinitis pigmentosa in early 20s	26	Visual FoxPro	Visual FoxPro	Healthcare Company
P18	39	M	Blind since birth	4	JavaScript, Python	Emacs	Large International Software Company
P19	30	M	Blind since birth	4-5	Go	Different text editors, avoids IDEs	Big Data Analytics Company
P20	55	F	Did not share	20+, scattered experience	Python, C, ChucK	Notepad++	University
P21	35	M	Blind since birth	16-17	C#, SQL	Visual Studio	Advertising Agency
P22	33	M	Vision loss from macular degeneration in early 20s	10	HTML, PHP, Python, JavaScript, AutoHotkey	Notepad++	University
P23	27	F	Vision loss from retinitis pigmentosa in mid-teens	7	.NET, Java, PHP, HTML, CSS	Eclipse, Visual Studio, Notepad	Large International Software Company

3.2.2 Procedure

We obtained the approval to conduct the study from the Institutional Review Board (IRB) of our university. The eligibility criteria for our study was that participants should at least be 18 years of age and they should self-identify as programmers.

We recruited participants through personal contacts (n=3), snowballing (n=2), and by posting the recruitment call online (n=18). We posted on the program-l mailing list (which mainly comprises BVI developers) [1] and r/blind[4] (a community for people with visual impairments hosted on Reddit). We conducted interviews with programmers from the United States, Europe, Africa, India, and China. In some cases, there are very few professional BVI developers in the entire country, making it relatively easy to identify the participants. Therefore, to preserve participants' anonymity, we are not listing the countries they came from.

We conducted the interviews on participant's preferred platform of choice. These included phone, Skype, Google Hangouts, and WhatsApp. Interviews typically lasted between 45–65 minutes. All interviews were conducted in English as the participants were comfortable with the language. Each interview was audio-recorded for which informed verbal consent was obtained prior to the start of the study. Each participant was compensated with an Amazon gift card worth $15 USD or the equivalent amount in their local currency.

The interview questions focused on participants' programming education, preferred programming tools and software, and experiences in collaborating with other programmers. We captured rich details about the challenges participants faced, how they identified workarounds, and how this shaped their collaboration in mixed-ability contexts. The interviews were transcribed verbatim by a third-party transcription service (approved by the university's IRB) and verified by the first author, providing us with rich narratives about participants' experiences as programmers.

3.2.3 Analysis

Before starting the analysis, we pre-coded [181] the data as we conducted the interviews. We highlighted quotes and sections in printed transcripts. We also wrote analytic memos [181] to identify emerging themes as well as missing details in the data to refine the questions for subsequent interviews. In the first round of coding, we used descriptive codes [181] to identify various programming activities, collaborative activities, challenges faced by participants, and workarounds. We further organized programming activities into three categories: (1) pre-programming stage focusing on installation and integration of various tools (2) programming stage focusing on code

[2]Note: "Did not share" refers to participants not describing their visual ability at any time during the interview. We interpret this as their decision to not foreground their disability in the interviews.

[3]Participant's data excluded from the analysis

[4]https://www.reddit.com/r/Blind/

writing, debugging, and compiling (3) post-programming stage focusing on code sharing. In the second round, we used pattern coding [181] to reorganize the codes from phase 1 into five high-level themes: (1) Group Work (2) Ecosystem of Tools and Assistive Technologies (3) Sighted Assistance (4) Extra Work and (5) Social and Personal Implications.

3.3 Findings

Our findings are organized into three broad sections. We begin by describing the various tools that participants used to perform programming and related activities. Next, we discuss the work practices that participants co-created with their colleagues to achieve the collaborative programming activities. In the final section, we discuss the various social interactions like communication, education, help-seeking, and advocacy that participants performed to negotiate of these practices.

3.3.1 Need to Access Multiple Inaccessible Tools

As we found in our interviews, making "programming tools" (such as integrated development environments, terminals, and debuggers) accessible is necessary but not sufficient. Being a programmer involves much more than writing code [165]. Our participants reported that as part of their jobs, they spend significant time working on project planning, communicating with team members, and coordinating with others to write code. All of these activities were critical to being an effective programmer and required interacting with a variety of tools. In this section, we elaborate on the accessibility challenges participants faced with different software and how it affected collaboration. We specifically cover (1) the kinds of code editors participants choose to use and the factors that go into their decisions, (2) the challenges they can face when setting up their development environment, (3) the tools they use for non-coding tasks, and (4) how tool usage affected their job application and promotion.

3.3.1.1 Choosing an Editor

Although programmers do much more than writing code, writing code (along with compiling, debugging, and analyzing output) is still a crucial job function. We thus begin by focusing on which code editors our participants used and how they chose them.

Many participants reported using an integrated development environment (IDE), which allows users to write code, share code, compile, execute code, debug, and view the output all in the context of one application. Different IDEs are appropriate for different platforms and languages. The IDEs that our participants used include Microsoft Visual Studio, Eclipse, Android Studio, and Microsoft Visual Fox Pro.

Other participants preferred using text editors instead of IDEs. Text editors only allow programmers to write code; compiling, debugging, and executing must be done from a separate application. As a result, text editors typically have a simpler user interface, which can be advantages: *"Your run of the mill IDE is just too much windows and lists to scroll through and things."* (P15). Some participants who primarily programmed in IDEs also used text editors as supplemental buffers where they stored various pieces of code and information. Participants felt it was easier to copy-paste things to a text editor, which was more accessible than other software they had to interact with.

The decision was also influenced by the complexity of the project, often determined by the number of lines of code, the number of code files one has to work with, and number of programmers involved. Many participants felt that it was often faster to *"write a small program, say, 100 to 200 lines program"* (P23) in a text editor. But with projects involving longer programs and multiple files, they preferred using an IDE:

> *I think where it gets taxing is when you have to maintain a project, say you're developing a web application in Java. Then it's so hard to do all the conflict files and just pair the WAR file and everything manually... the IDE does it so easily.* – P23

> *I am somewhat limited by the Vim accessibility [...] it works well for me to be in a small team where I'm the main programmer. I think that it would be more difficult to work in a team with more programmers in a much larger code base where you really have to use an IDE to be able to work efficiently and make sense of it.* – P9

Although the accessibility of IDEs and text editors was a major reason for participants using them, their decision was also shaped by several social factors including the programming tool their team was using. To be consistent with the team, participants had to compromise on accessibility, which necessitated additional work. In P17's case, this involved switching and moving code between two versions of the same IDE. The newer version was more accessible for him. However, he also had to ensure that his code was backwards-compatible and could be compiled in the version that his colleagues were using:

> *Right now I'm using Visual Studio 2017 for everything, and we're trying to get over to Visual Studio 2019. So I have both installed on my computer and sometimes I'll need to bounce into 2019 because it works a little bit better for some accessibility. But I make sure that any of the builds and everything I do really comes from 2017 because we want it to be in the same thing that everybody's using.* – P17

3.3.1.2 Setup Costs

Although participants generally agreed that they did not encounter many issues while using their editor of choice, installation and customization could be difficult:

> *I think for most people that would probably agree that it's like setting up the [programming] environment to start with, takes the time and getting all the tools lined up.*
> – P20

Setting up a programming environment necessitates assessing the compatibility of the software with screen readers and identifying the more accessible installation option between the command line and the installation wizard [66]. The most common way to assess screen reader compatibility and the accessibility was checking if the software documentation referenced screen readers and keyboard shortcuts. In other cases, participants reported emailing the developers of the IDEs to check whether IDEs had been tested with screen readers. They would also post on mailing lists dedicated to programmers which according to many participants, provided highly contextual and niche information about accessibility and usability of IDEs. They preferred these smaller mailing lists over larger programming-related Q&A sites like Stack Overflow because these lists alleviated the burden of explaining what a screen reader is or the multitude of ways applications can be inaccessible. By contrast, members in larger Q&A sites seemed to have a limited understanding of ATs.

> *No one on Stack Overflow is discussing the fact that in order to use Visual Studio Code with JAWS, you have to restart JAWS or you can only have one Visual Studio code window open at a time or you know that there's some weird interaction with the virtual cursor. Like no one's going into that level of, of niche detail on Stack Overflow.*
> – P9

Participants also faced issues in accessing the installation-wizard with the screen reader. A wizard is a GUI with a series of dialog boxes. It can be downloaded and run on the computer for an out-of-the-box installation. Participants reported that sometimes the *"installers aren't accessible whereas the programs themselves are"* (P1). They would seek sighted assistance to help install the programming tool and its packages (e.g., to click inaccessible combo-boxes and pop-ups). They, therefore, preferred to install software through the command line, if this option was available.

Beyond installing software, establishing a development environment sometimes requires referencing third-party software development kits (SDKs). SDKs allow developers to write code that access external resources like proprietary data, functionality, or computing power. They have become increasingly popular with the rise of cloud computing platforms. Many SDKs require authentication and take time to set up. For example, an SDK may require programmers to create an account to register their information before they can move on to the next step:

> *In order to start programming for the Alexa, you need to create an account on their website [...] a very interesting thing was that after you fill the form for instance [...]*

you need to submit the form. [...] And I need to spend like five minutes looking for that button [...] I accidentally scrolled up to the top, and then I saw that on the top it says "Submit" [...] those are some issues that can cost time until you find them. – P1

3.3.1.3 Working Outside of the Code Editor

Besides IDEs and text editors, participants also had to interact with other software, related to project management (e.g. JIRA, Microsoft Teams), file sharing (e.g. Git, SVN, Microsoft Teams), communication (e.g. Slack, Skype, proprietary messaging software), and internal tools (databases, virtual machines, web servers, etc). The information on these software informed their programming activities. Therefore, breakdowns in access in these tools/software had a direct bearing on their ability to carry out their responsibilities.

Participants' choice of programming tool depended on how easily they could switch to applications they were using concurrently. Generally, the teams used software that was designed for Windows and Mac operating systems. As P9 pointed out, using Linux would allow him to program more efficiently. But the screen reader on Linux would reduce the accessibility of other applications he uses in parallel:

I find that Windows is best and accessibility wise, and it does fit best into the work infrastructure [...] it's primarily windows directory, Outlook, Office 365 [...] If I really wanted to try to use Linux, then that would be supported. We have a Unix administration team. And as far as I know, I think they have Windows and Linux desktops. But I look at Linux accessibility every once in a while, and I think that in the GUI with Orca and all that it's just not not far enough along for me to really be competitive. – P9

Participants often had to use software like JIRA to track issues in their projects. They were required to log into the software to retrieve the project features and bugs assigned to them. However, the accessibility challenges in the software necessitated seeking sighted assistance:

I get someone visually and they come over, I say, "Okay, Joe. You told me that there is a ellipses button, that's a status button there. I'm not finding it!" And then he'll stand next to me, I press tab key and he says, "Oh, right now you're highlighting where I can see the box around it." I'm like, "Okay." And he's told me where the status is a second time, "So don't go away. Let me press the space bar to activate it." And it came up, or showed me a list of blanks. And sometimes he'll tell me, "Oh yeah, good. Now I can see your list of links. Just click on whatever the action was that they want me to do." And I'll say, "Well, it didn't tell me anything. It didn't tell me a list came up." "Oh. Now press the arrow keys!" – P17

The above quote illustrates how P17 works with a sighted colleague, Joe to overcome the accessibility challenges with JIRA. P17's screen reader, JAWS, is unable to read some of the JIRA buttons that are otherwise visible to sighted people. Thus, P17 is unable to identify the right button to click on to bring up his list of to-dos. He tabs through the buttons and stops when Joe mentions that the right button is highlighted. Our participant asks Joe to not leave until he has clicked on the button as there is no feedback when the list i.e. the result of the button click, comes up on the screen. Once the list is visible to Joe, he duly informs P17 who then copies the whole screen into a text file. Thus, we see how completing a seemingly simple task like clicking a button and determining its result necessitates work i.e. a series of interactions without which it is impossible for them to complete their job as a programmer.

Likewise many participants also reported the use of software such as Slack, Skype, and Microsoft Teams to communicate with team members, share snippets of code and details of the software project. As P11 elaborates, interaction with such software provides information that is useful to programming although it takes a lot more time for participants, compared to their sighted colleagues, to identify the exact thread that is pertinent to them:

> *One of the things that may be challenging is, I know, it's not necessarily related to the programming aspect, but I know a lot of information technology services will utilize chat platforms like Slack or Teams or something like that [...] When you try to go to the thread, have to go through tons of threads and read each one so you find the one you're looking for, which can be kind of daunting and frustrating for us. Because while we see our colleagues doing it in two minutes, it may take an hour.* – P11

As with the project management software, participants had to invest additional time and work to get around the accessibility constraints of this software. Participants mentioned that ideally they would prefer managing the challenges independently but the presence of deadlines, time required in implementing various solutions, and the increasing frustration of not finding *"a way around"* (P5) necessitated seeking sighted assistance. The other alternatives were contacting the customer support of the companies releasing the software and the IT support within their own organization.

3.3.2 Emergent Collaboration Practices with Team Members

Our participants shared with us details of activities where they collaborated with other members of the team: (1) code writing and styling, (2) code reviews, (3) pair programming, (4) software design, and (5) UI development. Our analysis revealed that they worked with their colleagues to modify the established work practices around these activities, resulting in practices that were more accessible.

3.3.2.1 Code Writing and Styling

When programming as part of a team, programmers often have to follow code styling rules or code standards. These are generally rules regarding the visual presentation of code so that it is more readable, navigable, and sections of code are easily identifiable. For example, Google's JavaScript style guide [238] specifies rules regarding use of braces, indentation, declaration of variables, addition of comments, and more.

Participants reported that they learned the code styling rules when they started collaborating with sighted programmers. For instance, indenting code blocks enables sighted programmers to easily identify relevant sections of code when scrolling past them [134]. However, since indenting did not serve any visual purpose for our participants, they did not consider putting additional spaces in the way they wrote code:

> *So when I started out, I was mostly doing this with Braille and speech was sort of secondary, which meant that I learned from very early on, I got into habits that were better on a Braille display. You know, you don't put spaces around equal signs because they don't matter and you could fit two more characters on your Braille display if you don't put the spaces in. So I never did that. I would always put brace on the same line because again, that's one less line that you have to scroll with Braille. So kinda stylistically I learned some things that I have since discovered are not mainstream and most people don't do.* – P3

The quote highlights that participants who used braille displays developed code writing habits to make the best use of the limited space available on the displays (typically 20–80 characters). Thus, their manner of writing code was at odds with that of their sighted colleagues. When possible, they preferred removing characters like extra whitespaces, braces, trailing punctuations in the code they received from their sighted colleagues. The problem of indentations, created through tabs and spaces, persisted even on screen readers since the characters were announced:

> *The fact that it always says four spaces, eight spaces, 12 spaces, it actually slows me down when skimming through code and I disable it.* – P12

On screen readers, participants also spoke about lack of nuanced information. Capitalization of variable names could lead to illegible pronunciation on screen readers:

> *It shouldn't all be, '`thisismyname`'. The variable is called '`thisIsMyName`'. It shouldn't all be in lower case! It shouldn't all be upper case!* – P17

> *The difference between Pascal and camel case is that the first word has or doesn't have a caps letter. Words 2, 3, 4, always start with a caps letter. The difference is for the*

> *first word. The screen reader does not read this, you have to read it by hands. You can't make the difference between the two casings and the difference is important.* – P12

The above quotes illustrate the challenges for participants with poorly named variables. While poor capitalization and naming is frowned upon by sighted programmers too, they can make sense of it visually. However, this information is invisible to screen readers.

In general, participants reported that they found it easier to navigate, search, and edit their own code as compared to other person's code. Working with another person's code was more challenging due to the combination of (1) inaccessibility of programming software, as described in the previous subsection (2) limitations of the access technologies in providing important information, described above (3) how their team members wrote code, described above. Participants had therefore developed code writing strategies that served them well when reading the code via a screen reader:

> *I was using the comments and the separated dashes and kind of titling certain things within a comment [...] when you hear a line being read as dash dash dash dash dash, then that's how somebody would know up here comes my next comment. This and kind of just as much description as possible, and getting people on the same page to code in the same way that you are sharing with.* – P11

> *So whenever I'm collaborating, say we're working on the same piece of code, or I'm supposed to continue working on some code that someone else had initially worked on, I always ask them to comment when they make changes. So that it's easier for me to find which piece of code they exactly changed last. So actually when I was working in Mumbai I had made this thing that everyone would put their initials followed by the time of when they were changing a particular code block in comments above the code block and then mark begin. And then after they're done changing N number of lines at the end they would again put a comment and say end of changing this.* – P23

Thus, strategies like unique commenting style and descriptive comments helped participants identify important sections in the code efficiently. Participants also shared these strategies with their colleagues, either informally or in code-reviews (next section), who were often willing to follow them. For the benefit of sighted programmers, they would follow visually-focused styling rules. This demonstrates that participants and their colleagues would collaborate, resulting in a new set of code styling rules more suited for mixed-ability programming contexts.

3.3.2.2 Code Reviews

Some participants reported that their teams had formal code reviews. Some of the more complex practices related to code writing were developed through code reviews. A few participants shared how code reviews made sure everyone on the team wrote (1) shorter code segments that made navigation easier on ATs (2) documented the code, which reduced the task of information-seeking and made searching the codebase more efficient (3) reduced redundancy in code, which again positively affected code searching:

> *[...] we adopt a practice in general that's not specific to accessibility, but everything in our code is just completely modularized. If you have more than 30 lines of code in a function, everyone's like refactor this put it into helper file[...] So, we really don't have very long code segments that you have to navigate through. Mostly, just sometimes, in one file there'll be several functions. But, even that, we try to modularize the files so there not huge files with too much in them* – P18

> *[...] So part of the process that helps me is that the code base is very well organized. All the different components are very well separated and where there is common functionality, the functionalities are separated out in its own package as well. So from there on it's basically a matter of knowing what you're looking for* – P19

The formalization of rules also ensured that participants did not have to reach out to their colleagues separately and ask them to modify their code writing practices. It also led to a standard set of practices—participants knew what was expected of them in terms of styling and writing and they knew what to anticipate from their colleagues:

> *If you're sharing the code base, one of the ways to get around it, too, which I didn't mention, is to maybe have everybody code in the same way.* – P11

> *By then doing that, which is sticking to standard practices for programming, then it's beneficial for all.* – P17

Participants spoke positively about the code-review activity if the software used to facilitate it was accessible. This allowed them to perform efficiently without asking others for help. P18 compared his experience at his current organization with that at his previous organization. In the current workplace, his team used a web-based code-review system that was accessible with screen readers. He explained he did not have to ask for accommodations and he was able to participate in the activity like his other colleagues. In his previous workplace, a sighted employee was hired specifically to assist him with the inaccessible code-review system. This not only affected his

collaboration experience but also impacted his productivity. It would take him a *"couple hours a day"* (P18) just to share his comments on improving the code. This reemphasizes the importance of looking beyond the accessibility of programming tools for collaboration in mixed-ability contexts.

3.3.2.3 Working Together and Pair Programming

Many participants reported that their teams practiced pair programming. As explained in the related work, in pair programming one programmer is responsible for typing the code, known as the *driver*, while their colleague, the *navigator* gives instructions and feedback on the code being written. Our participants expressed that while they were able to perform as the driver, they could not easily reverse the roles and give directions as the navigator:

> *So I can be the person writing the code and someone advise me but obviously the reverse isn't so easy [...] So usually I'm the person writing the code because obviously I can't look over the shoulder [...] I would like to have to be able to do the same thing, but it's not essential, me being the person looking over the other person's shoulder.* – P19

The above quote shows how participants contributions in pair-programming are limited due to lack of access. They cannot provide critique and recommendations on the code. The complications arise due to (1) lack of access to colleagues' computers (2) social and legal limitations with regard to installation of ATs (3) colleagues being unable to describe the code structure and errors in a manner that is easily understandable to the participants. Next, we describe each of these in detail.

Participants mentioned that ATs were generally not installed on their colleagues' computers. Thus, the code on their colleagues' computers was inaccessible to them during real-time collaboration. This not only made synchronous programming challenging but it also prevented participants from providing help to their colleagues in real-time, an important aspect of pair-programming. A few participants preferred their colleagues share the code via email in textual form instead of describing the code and errors to them. This way they could access the code on their computer that had the necessary ATs setup:

> *So he sent the text of the rule and the text for the error. And then I just looked through the code and found what I believed the syntax error [...] So, I found as long as I can get things into a textual representation that works pretty well. [...] So I guess the thing to say is that my co-workers either need to provide things in text form or they need to come and either sit with me or like I need to be the one driving the computer that's used to find the problem. It doesn't work well, for me to stand behind them while they're operating a computer that I can't access and they are sort of halfway trying to describe what's going on on their screen* – P9

P9's quote reveals the breakdown in real-time collaboration and the workaround he has to adopt to provide help to his colleagues. Another alternative was adopting a multi-step process, which also to a degree defeats the purpose of pair-programming, and is time consuming: the code is uploaded to the shared repository, from where it gets pulled and setup on the computer, and then the changes are reviewed. This process is exacerbated by a large code base, where working side-by-side with the colleague is preferred as the changes can be discussed in real-time:

> *The magnification tools that I use [...] makes it very difficult when I'm like pair programming or have just rung a sighted colleague to help me troubleshoot an issue with something [...] Other time they haven't had much trouble because I'll just set ZoomText off while they're doing what they need to do with it then turn it back on while I'm doing what I need to do with it [...] So its kind of a double edged sword because it's either I see what I need to see or they see what they need to see. There hasn't been a perfect solution to that. Because if it's not that [...] I have to upload our code to source control we're using at that time, we're going to pull it down and compile it... all that takes time, especially if you're talking about a large code base* – P7

P7 used ZoomText, an AT that combines magnification and screen reading technology. She shared her workaround for achieving real-time collaboration by reducing the magnification of text on her computer. This allowed her sighted colleagues to read and navigate the code on her computer but prevented her from understanding the changes they were making in real-time. Thus, it provided intermittent access to the participant and her sighted colleague. P7 went on to talk about a recent feature that made screen-sharing and, as a result, pair-programming with colleagues easier:

> *I believe if I'm not mistaken, dual monitors support was added like maybe a year ago. So I feel like in a sense they're moving kind of slowly compared to where the rest of technology is, as well [...] it gives you the option to operate one screen magnified. So my screen could be magnified for me [...] with that same image on another screen in regular size. So if I'm working with someone who's sighted and we have two monitors, then that makes it a little bit easier* – P7

Thus, the collaboration still happened on her computer but it enabled more synchronous work. We also learn how the slow introduction of features to ATs, compared to mainstream technologies, impacts the collaborative experiences of BVI developers. She and a few other participants said that sometimes they would install ATs on their colleagues' computers provided they were willing to install it:

> *... also now that they allow for a free trial version, that's roughly 45 minutes in duration. That means that if I have a co worker that doesn't have it on their computer,*

if they're willing to install it, then they can put it on their computer for us to troubleshoot a bug or something and then they can uninstall it and not have to deal with that permanently. So that's something that's helpful and useful. Believe it or not. – P7

I have a professional license for that (JAWS) [...] that just allows me to be able to install that on any of my work computers with one license and as long as no other visual impaired person or other people actually use that software package for their own use [...] Now, I do have JAWS installed on about three or four other servers, with the purpose of being able to remote into that server and have NBL to access that server remotely. – P17

This shows our participants have to switch and share computers in the workplace to collaborate effectively. To access colleagues' computers, they have to reinstall the AT. But sharing of ATs, specifically screen reader software, is complicated by the limited availability of licenses and policies around who could use the AT as well as due consent of colleagues. The onus of establishing access is generally on the participants and not their colleagues. It also may take up considerable setup time, which the participant has to invest again.

When it was essential for both the participant and their colleague to be working on their respective computers together, they preferred using a communication software to do screen-share:

I'll pull the file up that they're doing as well, and they say what they're looking at. "Oh, I'm going down to this class, the class need this. I'm coming down to the section of code that starts with that." And then by us having the instant message window open, they can paste in the line of their code that they're talking about, they're jumping down to, or they can tell me the line number. If we actually have the same version of the file, I can jump down to the line number and such. – P17

This can be thought of as switching to a collaboration style akin to *remote* pair-programming. In this style of collaboration, both programmers could work on their respective computers. The screen readers did not interfere with their discussions. The screen share allowed the sighted colleague to view where the participant was in the codebase. By announcing the specifics of class and functions, the sighted colleague informed the participant of their whereabouts in the codebase. Both programmers could paste-in specifics of the code in the chat window. The participant could use this to copy-paste and search the codebase more efficiently.

Thus, in most cases, participants needed to access the information on their computers as well as the information on their colleagues' for achieving collaborative tasks. Only in instances where they were the *'subject matter experts'* (P17), they were able to collaborate without necessarily accessing their computers. In these cases, participants knew *'what the details are for the questions that they*

(sighted colleagues) need answers to' (P17). This section provides us with a new understanding of access in the workplace - not limited to one's own computer. Participants and their colleagues had to collaboratively establish workarounds to achieve what sighted programmers are able to carry out relatively easily by virtue of being able to view each others' screens.

3.3.2.4 Software Design

Participants reported facing challenges in accessing and creating diagrams that represent that software architecture. Participants reported that their teams used online tools (LucidChart [125], Microsoft Visio [139], draw.io [124]) and whiteboarding to prepare the diagrams. The visual nature of the task prevented them from participating in fully in the task:

> *[...] design people would get together and sit around a work table and chat about how the concept was to be developed. They would try different solutions and come back to meetings with more plans and more plans in the project design. I was just never involved in those. And that always struck me because I am a trained designer. But in my work as a blind person, I was not capable of doing that because it was all visually focused. They would draw visual diagrams, visual methods, communicating to each other about how flowcharts would go and how the program process was to work. And because of that visual practice, I was excluded from working efficiently with it.* – P16

P16 described how his colleagues, the systems designers, were able to discuss the software architecture in detail and develop the component diagrams. The discussions were useful in planning the project. He was not included in these discussions despite having been trained as a designer (architecture), prior to switching to a career in programming. He was therefore capable of understanding the designs but was not given the opportunity to participate. He further spoke about how it impacted his understanding of the software he developed and troubleshooted:

> *I wouldn't know how the computers connected together to be more efficient, mainframe computers would be connected together. So I couldn't contribute there [...] the way the others worked was anyone could have managed issues of the sequencing of the design process [...] they would troubleshoot any aspect that broke down and pull it in [...] So I didn't ever have a system, a sequence of jobs that needed fixing. It was always just a single program that was part of a process that had a breakdown in it.* – P16

Unable to access the diagrams, P16 only had a limited understanding of how the different components interacted in the software process. Without a complete overview of the project, he was unable to offer suggestions about rewriting and improving the components. On the contrary, his

colleagues had a complete understanding of the entire process. They were able to manage and develop multiple aspects of the software. Likewise, he also spoke of the visual nature of the *"progress reports of a program"*. His colleagues could view the reports to measure the impact of their contributions while he had to acquire the same information from his boss. He was also unable to take on more active roles and had to *"differ to the team leader for the jobs to be done"*— the jobs that were accessible. Thus, his contribution was pigeonholed to code-writing and troubleshooting. It is important to mention that P16 is a retired software developer and he was recounting his past work experiences in the 1970's and 1980's. It is likely that his experiences may not generalize to that of the programmers today. However, it highlights how the lack of access to the high-level software design can result in the programmer with the visual impairment being assigned fewer responsibilities in comparison to the sighted programmers.

Many participants mentioned that their colleagues *"couldn't translate those diagrams into words"* (P16) i.e. describe all of the necessary details of the software architecture. In addition, the expectation was that the documentation would be prepared visually and not rely on descriptions:

> *the other part of the job is before I program the code, I should design the system I'm making in some visual design format that the rest of the team can look at during the meeting [...] So, on my annual review my boss was like, well we need more design meetings to show what you're working on and stuff like that. [...] I told him, I can't make component diagrams, like other people do [...] Usually, I just write out a description and I'll write out the parameters to different functions in a JSON format [...] I'll copy and paste it into the shared Line system that we use [...] I've only done that on minor things* – P18

For minor projects, P18 would write out descriptions of the system design along with the functions that need to be implemented. He would share the *descriptive design* with his team to seek their feedback in design meetings. We also learn these designs are a way to inform the team about one's ongoing project, presents them as the project owner, and highlights their contributions in the workplace. This reemphasizes P16's point about not being able to convey the value of his work. In P18's case, his manager wanted him to be involved in this step but preferred that P18 create visual diagrams. This resulted in him either avoiding the step or let someone else prepare the design. P18 stressed to his manager that he could not *"really do much except write a description of what I'm doing"*:

> *My manager said maybe I can write up a description of what all the components do and how they all work. Then, he can sit with me and help me make a component diagram, which he says should be pretty simple and straightforward. But, I think it*

helps a lot of people be able to visually look at a system and see how all the parts are interacting [...] especially [...] in software engineering there's a lot of people [...] English is their second language. – P18

P18 acknowledged that having such diagrams was useful for sighted colleagues, especially in diverse teams where English may not be the first language of many employees. Thus, the workaround of describing the system was likely to not be useful for the rest of the team. But he also felt that the expectation of delivering visual component diagrams was his *"number one challenge in programming"*. In P18's case, his manager offered to work with him to achieve the mandatory task of preparing system diagrams. The co-creation of the system diagram was going to reduce the extra time and emotional stress that P18 would have to go through he were to work on it alone. At the same time, the nature of this collaboration presents him as the primary contributor since he is writing the descriptions that drives the preparation of the diagram:

If he can help make the little diagram based on what I wrote in the text and we can just talk about it for a minute and he can help me, that saves me a lot of trouble. That's just a relief! As long as it gets done and works, it's fine. I'm not one of those people who cares about I need to independently do everything myself. [...] As long as someone helps me get it done and I'm doing the majority of my own work, that's fine with me – P18

3.3.2.5 UI Development

Another collaboration activity that came up frequently in our interviews was graphical user interface (GUI) development. Generally, the development of the interface is preceded by a discussion phase where designers and programmers come to a common understanding about the form, functionality, and interactions of the GUI. Often these discussions happen over visual artifacts like wireframes and design documents, which developers review as they write code. These artifacts contain details on colors, sizes, placements of GUI elements on the screen. Alternatively, the discussions are informal in nature, with the developers being informed of the general layout and interactivity of the GUI by either the designer or the manager. In this case, the design guidelines are high-level and strict rules and guides are not provided:

[...] our HR has given us a task to create a web page, and you know, create as you want. Like use your efforts and use your imaginary power and design according to your imaginary power. – P2

Generally, the decisions regarding the granularity of design documentation depended on the nature and practices of the workplace. For instance, one participant reported that her previous organization was fairly small and therefore, developers were also responsible for the design:

So the company I had been working for was a small company and they didn't really have the concept of teams, everyone was an individual contributor [...] So when you have an independent project that you are working on, you don't just code you also have to design the interface.[...] it was communicated in text [...] these are the forms and these are the controls that we need on the forms [...] nothing in detail like, this should be 10 pixels away from this sort of thing, no. It was just a very high-level document. – P23

Participants sought sighted assistance during the development process when working with such documentation, evidently because of the visual nature of the documentation. We noted two challenges for our participants. First, our participants could not verify aesthetics and placement of UI elements visually as sighted programmers did. Second, when inspecting the visual output with screen readers, it would announce the UI elements linearly i.e. in the order in which they appeared in the code. Thus, the UI element could be present on the screen but not necessarily be visible:

Sometimes some of them will overlap with each other... And though I could hear two different buttons but it could be the buttons are on top of each other. And the sighted person is only able to see one. – P23

Similarly, participants felt they could not be sure if something was off the margins of the screen. Third, in the context of web-based applications, a sighted programmer can use web-inspector tools in the browser to make temporary changes and inspect how this modifies the interface. However, the web-inspector tools were not accessible to the participants. For example, P11 shared how the screen reader would not announce the URLs on HTML page or not inform her about text-styling i.e. whether it was italicized, bold, underlined, etc. To verify these information, she had to refer back to the HTML and CSS code in the text editor. She would have to search and navigate to right section of the code to get the necessary information and make the changes. This reveals the additional steps that she has to perform as compared to a sighted front-end programmer. Last, participants shared that it was difficult to calculate measurements for width, height, margins, paddings, and placements of UI elements. P6 shared one of the ways was to calculate start and end position of each element on the web page when designing the layout:

[...] the best thing that you could come up with this is tell them to use the coordinate system [...] And basically what it is you count the number of pixels [...] if you want a div on a page that's 100 pixels wide and a 100 pixels tall, [...] a reasonable point would be for 10 pixels from the left edge 10 pixels from the top edge [...] That gives you a good placement on where you could put your other stuff on the page. But it failed at a lot of points because then people told their stuff wasn't lined up, right [...]

you still need have to have somebody spot check it and it can still turn off really weird [...] – P6

But as P6 explained, it still required spot-checking from someone sighted. It was also a mentally-intense process that could not be scaled for more complex websites. Screen readers with the right add-ons could potentially announce the measurements in percentages but this again required mental calculations on the part of the participants. P6 described that he was developing a user-friendly NVDA add-on for the visually impaired programming community to support his peers in UI development.

The access challenges also depended on the kind of UI participants were developing. Many participants shared that one of the good things about mobile UI development was that they could verify the output and interactions by installing the mobile app on their phones:

I feel like that's one place where the touch screen made things a lot easier [...] it became possible to really explore the layout of a GUI and know exactly where things are. [...] I can get an idea how big the button is relative to the window and the screen and I can get an idea where the edges of the buttons are. I think that's quite nice. You slide your finger across a touch screen and the moment you encounter the button, you hear it's name [...] It makes it very easy to explore graphical layouts. – P15

Thus, relative to UI development for the web, developing mobile interfaces was more accessible. Touchscreen interfaces alleviated the issues pertaining to verifying visual feedback. However, participants also shared how web-development was primarily text-based. They had to write code in HTML, CSS, and JavaScript. When developing mobile or desktop UI, one often had to do it IDEs. These come with features that facilitate quick UI design, for example Layout Editor in Android Studio[5] and Windows Forms Designer in Visual Studio[6]. Sighted programmers can use these to drag and drop UI elements to quickly prepare the visual design for desktop and mobile applications. However, our participants could not do the same because it entailed significant mouse work. Without being able to access information on the position of the cursor and UI elements, participants could not envision how the layout was shaping up:

The designers for user interfaces are not accessible. The most accessible designer for Windows applications for example, was ironically in visual studio 2005 I think, which told you where a control was as in pixel locations, but it also told you whether items overlapped. From that point on, no tool tells you this. You have to calculate pixel positions manually, which is not the funny. [...] I'd like UI designer which would be

[5]https://developer.android.com/studio/write/layout-editor

[6]https://docs.microsoft.com/en-us/visualstudio/designers/windows-forms-designer-overview?view=vs-2019

accessible [...] You actually have to write the XML by hand. It doesn't read your controls, it doesn't review pixel presidency, it doesn't read you anything in the UI designer, so you have to actually write the whole design – P12

Some of the above challenges were alleviated when participants were provided more detailed design documents. This enabled participants to work faster. They could look up the measurement details in the documentation and program accordingly. However, this also required participants to share instructions on how to make the documentation accessible and doing so for all the UI elements:

the specifications wouldn't be accurate enough to have numerical values where to put something [...] For instance, some designs that I would receive would have [...] dividers between different buttons. I wouldn't be able to see them. And if they weren't specified in text [...] I wouldn't be able to see them [...] they would follow my instructions on what would make my job faster. [...] having specifications within text rather than just relying on me looking at the designs and implementing them just by using my sight. Ya. – P1

Participants also reported spending time with the designers to understand the layout of the UI. They felt that generally their colleagues struggled with explaining things verbally, a detail we reported in the context of system architecture diagrams too. Thus, participants felt the onus was on them to ask the right questions, at least in the early days of their collaboration. Over time, the effort required to frame appropriate questions reduced:

When I put the question very precise one [...] They answer and they are eager to answer. But if I ask for example, can you give me an idea of the layout, why it is too general, and they used to say maybe much more than I need or maybe they miss some parts. It's to me, just to try to at the beginning, to ask very, very precise questions [...] It's not always easy because it's not always easy to know which are the elements that they want to just put on the page so I just try and to refine step by step. – P13

Accessibility challenges in UI development also shaped a few participants' decisions to pursue programming that would require them to deal with the front-end as little as possible:

It's easier that there are far fewer accessibility concerns with back end and as far as its employment goes, they have a need for it. – P9

Participants who had specialized in front-end programming felt they faced significant challenges finding employment. Participants shared several instances of employers doubting their programming abilities and the credibility of their education:

I was more than qualified for some of these jobs, like web designer, or web developer one at a university. I went to this interview and it was a panel interview with the manager of the group and the whole entire team. [...] Well, in the interview, the manager of this group actually asked me how many web design classes were you exempted from? – P6

3.3.3 Social and Personal Implications

In the previous sections, we have discussed the challenges participants faced in collaboration and how they managed these challenges. In this section, we report the impact of accessibility challenges on participants' advocacy and help-seeking.

Most participants, independent of the country they resided and worked in, hesitated to ask their employers to provide them with commercially available ATs like JAWS, ZoomText, and braille displays. Participants felt their employers would perceive it as an expensive request and felt uncomfortable asking their *"boss to spend so much money"* (P12). A few participants felt their request would be seen as *"excuses"* (P23) for accessibility challenges. Given the challenges in finding employment, participants preferred to not emphasize lack of access as it may be misinterpreted as lack of programming ability. Thus, participants preferred switching to free and open-source alternatives or using their personal licenses instead of asking for accommodations that they are more comfortable with and that might be more effective.

Participants reported that they preferred explaining their access needs through one-on-one and informal conversations. They gave small demonstrations on how they used ATs to *"show people [rather] than to tell them"* (P15) about potential breakdowns. They felt such interactions were better at familiarizing colleagues with ATs, their workflow, and changing misperceptions about their programming ability. It also made the colleagues more open towards their preferred strategies:

I know some people will like to just mention things or have a sit down meeting and do everything at once. I think it makes people more anxious and it makes it more daunting and unrelatable for them. So what I try to do is try to teach as I go because I find that if you break things up, it doesn't seem as daunting and then the more they get to know what your style is, little by little, it happens naturally, you know – P11

I think once they understand your workflow as a blind person, it's easy to show them stuff and to get them to understand how you work – P15

While informally educating their colleagues made them more amenable to changing their work practices, it was also a slow process. Participants shared that explaining certain visual concepts

verbally was *"tough"* (P19) and they had to *"remember the virtue of trying to be patient with people"* (P11). Participants also felt that not everyone was open to changing their ideas or perceptions about them, in which case, they had to advocate more strongly for themselves and work around or avoid such colleagues:

> *You always get some people who have prejudices. Our manager is actually not very good, he's got quite a few interesting ideas. So we clash a few times [...] So it all comes down to then just communication again. And as I said, I do find people are sometimes prejudiced and it's bit stressful because if people disregard your contribution, it's not pleasant. And it sometimes happens and you kind of have take a step back and you say, "Hey, don't disregard me, I know what I'm doing" and sometimes people listen, sometimes they don't! I try to avoid the people that doesn't listen and work with the people that do.* – P19

Participants shared that advocacy and collaboration was easier when their team had employed a person with a disability previously. In this case, employees had some experience of working in a mixed-ability context and some of the collaboration practices were already in place. Thus, participants did not have to put in additional work in educating their colleagues or requesting access. They found that sighted colleagues were more comfortable working around the established practices to cater to them:

> *I think I sort of had an easy time getting into the workplace because they already had experience with a blind employee. They would want me to write documentation and they would say, "Write this documentation. When you're done, send it to this person. She will add some screenshots and some diagrams." It worked out quite well.* – P15

Participants also spoke about their experiences in organizations that had policies in place with regard to inclusion and accessibility. Participants felt more comfortable in requesting accessible alternatives and voicing their concerns:

> *[...] at [Company X][7] I didn't have to do that because you have a big team and then they already know what is inclusion, and what is accessibility. It was a place where I could say that, "Okay this is not something accessible to me so why don't you help me with this or why don't you delegate it to someone else?"* – P23

P23 spoke more positively about her experiences as an employee in her current organization. She felt she could not make similar requests in the previous organization, also her first employer. The intersection of disability and status made her position more precarious in the previous organization.

[7] We substituted P23's organization's name to preserve anonymity. It is a large international software company.

Thus, she preferred working harder in addressing the accessibility challenges and avoided seeking sighted assistance.

Participants' perceptions about the workplace also impacted how they sought help. This was to a large degree shaped by social and technical factors. For instance, in small teams, participants would work with the same group of people on all projects. Therefore, participants were familiar with everyone and felt comfortable reaching out to their *"supportive group of coworkers"* (P21). Many participants felt positively when the internal tools like code review systems and internal websites were accessible:

> *[Company Y][8] has a whole accessibility team. They're mostly located in retail accessibility, but they provide advice and consulting for all the other teams. Generally, it seems like they make an effort to make all their websites and internal tools accessible as best they can.* – P18

Accessible internal tools enhanced participants' work experience in three ways. First, they enabled participants to work more efficiently. Second, participants had to only occasionally seek help and that too with *"minor things like clicking the combo box"* (P18). They did not have to worry about incurring social debt by wasting their colleagues' time. Such quick and infrequent acts did not necessarily draw attention to participants' disability. It was instead understood as a shortcoming of the software. Third, it suggested to them that the organization was committed to providing an accessible work environment. Presence of an accessibility team meant that there was recourse from more serious challenges in internal software and the organization was also likely to fix them. This would allow participants to work independently in future and not require seeking sighted assistance.

On the contrary, in the face of lack of accessible tools, participants' had to seek assistance on a more regular basis. Participants felt the act of seeking assistance did not emphasize inaccess as much as it drew attention to their disability. Participants also felt they could not easily reciprocate the help due to lack of ATs on colleagues' computers, as discussed in section 3.3.2.3. Participants also worried about their colleagues feeling obligated to help them. They did not want their colleagues to feel that it was *"one of their responsibilities is to help"* (P17) them. To avoid this, participants would try to reach out to different colleagues every time. A few participants shared they would spend time on finding someone who they felt they would easily be able to communicate with and they would be more willing to spend time in answering their questions:

> *There used to be lady in the cube just right across mine. She was very nice! She left a while ago. And there is nobody very close by that I feel really comfortable [...]*

[8]We substituted P18's current organization's name to preserve anonymity. It is a large international software company.

sometimes I try rebooting and nothing really seems to happen. So once or twice I have had to just ask people, "can you tell me what's happening with my screen" and that kinda thing. [...] But I really dislike having to do that. – P10

The above quotes show how help and decisions around help-seeking are shaped by socio-technical considerations. Participants' experiences are shaped largely by their team – as shown by the contradicting experiences of P18 and P23 who had worked in the same organization but within different teams. This demonstrates the degree to which participant's experiences are socially situated.

Advocacy and help-seeking in the context of programming complicated participants' sense of independence. Participants felt they could *"influence how things are done"* (P19) and push for more accessible alternatives and advise their team on developing more accessible software. This contributed to their sense of independence. At the same time, seeking assistance for challenges, however little and infrequent, impinged on their sense of independence. This resulted in *relative accessibility* of programming that contributed to a *relative sense of independence*:

You kinda run into this weird thing of partly empowering because computers are everywhere and everybody uses them and you are one of the people that knows more about them than most, and you can make them do what you want, and you can administer them, and you can program on them, and its really, really fun! But at the same time, you are far less able in a lot of ways because you can't access the same diagrams and tools and diagnostics, all the other things that any other sighted person, or any sighted person would be able to [...] So its a weird mix of more independent because I can do more on computers than a lot but less so, because at the same time I can't do as much. – P3

3.4 Discussion

Our findings show that BVI developers use a complex ecosystem of tools. This ecosystem includes software related to programming, project management, communication and internal corporate tools. Each of these is critical to the core task of programming and often must be used concurrently. The accessibility challenges in the ecosystem affect collaboration and help-seeking practices between programmers in mixed-ability contexts. BVI developers and their sighted colleagues co-create new work practices in order to collaborate effectively. The practices are also shaped by characteristics of the team, advocacy, and additional work on the part of BVI developers.

Based on our analysis, we have framed our discussion around (1) accessibility of group work, focusing on real-time collaboration (2) implications for collaborative programming.

3.4.1 Accessibility of Group Work

3.4.1.1 The burden of additional work

As we found in our study, several programming-related workflows (including pair programming, UI development, and system design) rely on visual artifacts and as a result, were inaccessible to participants. Nonetheless, our participants found unique workarounds to circumvent the challenges. For example, in synchronous programming, participants and their colleagues used communication software to do remote screen share and inform each other of their whereabouts in the codebase by announcing line numbers. This enabled them to work on their respective computers and access each other's programming contribution without having to change their AT settings. Finding such workarounds is *invisible work* [207]; it is necessary but falls outside the purview of formal definitions of work for our participants. By highlighting this work, we bring to fore the otherwise invisible work done by people with visual impairments in creating and maintaining access [27]. The invisible work is not limited to finding workarounds to circumvent inaccess. It includes other activities, also not included within formal definitions of work, such as information-seeking on mailing lists, identifying the right colleague to seek assistance from, and proving to potential employers about one's ability to interact with the ecosystem of tools.

Furthermore, to perform their roles in the various programming workflows, especially in the context of collaborative tasks, participants had to articulate their own ways of working in the first place. They used informal demonstrations and one-on-one meetings with team members and managers to communicate their strategies. Through these interactions, participants conveyed their preferred methods for pair programming, code styling, communication tools, and more. Generally, the articulation for access needs happened outside the context of programming related tasks, and as characterized by one of the participants, was often a slow a process. Again, this goes to show that access is not inherent in the workplace or in programming workflows. It is the *articulation work* i.e. the work to make work work performed by people with visual impairments that leads to creation of access and modifies the established arrangements around work practices [58]. Our findings show that the nature of articulation work was contingent on the workplace and participants' perceptions of the workplace. For instance, participants were more at ease advocating their needs and had to do less articulation in workplaces that had previously hired people with visual impairments or seemed to prioritize inclusiveness and accessibility. Participants in less accommodating workplaces had to perform emotional labor as they tried to be patient in explaining their workflows to their colleagues.

3.4.1.2 Fostering better interactions around help-seeking

We saw instances of people seeking help from colleagues to circumvent challenges with technologies and activities that relied on visual artifacts. Similar to prior work in the workplace contexts

[47, 62], we found that the nature of the relationship between our participants and their colleagues affected when and how our participants sought help. For instance, in smaller teams, participants shared a good professional relationship with most colleagues and felt comfortable seeking assistance for accessibility challenges. Consistent with the prior work, our participants expressed concerns about incurring social debt [43, 234, 244]. They were concerned about the impact help-seeking would have on their sense of independence [239]. However, in our study we also observed that decisions around seeking assistance were based on participants' perceptions of the accessibility of the work environment. For example, participants were more at ease in seeking assistance from colleagues when they felt their workplace made efforts to provide accessible internal tools and accessibility support. In such cases, the act of help-seeking was minor, quick, and infrequent. It was not likely to foreground the person's disability or result in colleagues spending too much time in assisting the person with visual impairment. When seeking help for minor challenges, participants also had to perform less work in explaining the issue to their sighted colleague. This was also evident from participants' preference for using mailing lists to seek information about accessibility of programming tools. They preferred seeking help on mailing lists about breakdowns with programming tools instead of large programming websites like Stack Overflow. Here, the desire avoid the work associated with explaining concepts like accessibility and screen readers, which were inherently understood by people on accessibility mailing lists guided participants' decisions.

Beyond help-seeking, there can be challenges to *giving* help as a programmer with a visual impairment. We recognize that help-giving—working through others problems and brainstorming solutions—could be an important way for our participants to establish competence through everyday actions and interactions [83], thereby conveying their abilities [86, 110]. This would also enhance interactions between people with visual impairments and their colleagues [223]. However, help-giving—particularly giving real-time feedback to colleagues on their code—can be infeasible due to the unavailability of ATs on colleagues' computers.

We recommend that in designing to facilitate group work in mixed-ability contexts, designers use the Design for Social Accessibility (DSA) framework. This framework emphasizes to designers that ATs should be a vehicle to convey the end-user's ability and identity in social settings [195]. This is done by considering both functional and social factors of AT use [193]. We argue that designers should specifically use methods to foreground interactions around help in professional contexts, especially help-giving by people with visual impairments. For instance, we noted how participants worked around the problem of lack of ATs to assist their colleagues. They would install the trial version of ATs on colleagues' computers or use their licence to install multiple versions on different computers. While these workarounds allowed synchronous assistance, they necessitated extra work on the part of the BVI developers. Additionally, it required colleagues' consent and was further complicated by legal limits on installations. Therefore, in design of ATs and collaborative

tools, we recommend considering the time and work required by these workarounds as well as their impact on real-time help-giving and collaboration.

3.4.2 Implications for Collaborative Programming

Prior studies on accessibility of programming have been limited in their scope. They have studied the experiences of BVI developers removed from group-work settings, which require carrying out of multiple collaborative activities. Our empirical contributions serve as a generative site for thinking about accessibility in collaborative programming. We discuss some of the design implications in this section, situating them in the perspectives recommended for designing for disability [193, 27, 28].

In mixed-ability contexts, programming is a socio-technical achievement. BVI developers carry out a series of social and technical interactions to address the accessibility challenges—creative code writing strategies, articulating their workflows, advocating for their access needs, and more. Designers should consider and foster these interactions and build on the workarounds that BVI developers have identified [28]. When creating accessible solutions for programming, designers should take into account the various factors that shape the choice of programming tools—project complexity, requirements of the workplace, its concurrent use with other tools in the ecosystem. We also strongly recommend examining the *setup process*. This requires ensuring the accessibility of various activities in the installation process such as account creation, assessing the tool's accessibility, and finding the appropriate installer.

Current code styling standards are largely intended to improve code navigation and readability for sighted programmers. However, in our study we report on the emergence of a new set of practices that were beneficial for our participants as well as their sighted colleagues. Some of the rules like writing modular code and frequent documentation were useful to everyone. On the other hand, visually focused practices (e.g., indenting code segments, using inline spaces, or placing braces on different lines) did not necessarily help BVI developers but they adopted them in their collaboration with sighted programmers. We also noted that participants' strategies (such as adding descriptive comments, using camel case for names, and long variable names) were incorporated by their colleagues. We therefore recommend that code styling standards, especially when shared online by large software companies like Google,[9] should also advocate for adoption of strategies preferred by BVI developers and thereby present a more inclusive document. This would inform sighted programmers about the code writing preferences of BVI developers and reduce the work of communication on the part of BVI developers. It would also improve the efficiency of tasks associated with code reading and writing on computers with and without ATs in mixed-ability

[9] http://google.github.io/styleguide/

contexts. Additionally, in larger companies, an inclusive set of standards can lead to more efficient collaboration in code reviewing, and therefore, a more positive experience for BVI developers.

In UI development, participants had to expend mental effort in calculating the pixel position of elements when design documents lacked this information or were high-level. Participants reported that ATs lacked relevant information and UI development tools were largely inaccessible. They had to seek sighted assistance frequently to verify the placement and aesthetics as they were developing the UI. As in the context of homes [45], repeated assistance with things like spot-checking may be minor but can add up. Participants preferred help that was minor and infrequent and allowed them to independently carry out the majority of the work. Lack of inaccessible tools also had implications for participants' employment opportunities and careers. It prevented participants from contributing to front-end development and made them choose other sub-domains within programming like backend programming or data management. This speaks to the relative accessibility of programming—it is more accessible than other STEM fields but domains within it still remain relatively inaccessible. This again motivates thinking about accessibility of UI development tools using the DSA framework to convey programmers' ability and competence at developing UIs [193]. For instance, one participant explained that few IDEs had relatively accessible UI tools but these were replaced with inaccessible options in future versions. Another participant was working on developing an NVDA add-on to support his peers. Such tools can serve as starting points to brainstorm about improving the accessibility of front-end development tools. They also provide opportunities to engage BVI developers in the design process as "designing bodies" [28].

Crowd-supported solutions like VizWiz [32], BeMyEyes[10], and AIRA[11] are recommended alternatives to sighted assistance from personal and professional networks. Past research has shown that people with visual impairments prefer using these networks because they offer quicker and contextual help without leading to social costs [43, 129]. Thus, assistance for certain programming activities like spot-checking, assessing the UI, and accessibility challenges in setting up the programming environment can be outsourced to these services. Given their familiarity with ATs, they may be better suited at providing assistance than the IT support. They are also likely to reduce the extra work that BVI developers have to perform in explaining the accessibility challenges to sighted people. However, usage of these services is also likely to be regulated by an employing organization's policies around intellectual property. There is risk of disclosure of internal ideas and artifacts. This warrants thinking about formal integration of these services in the workplace to support BVI developers.

[10]https://www.bemyeyes.com/
[11]https://aira.io/

3.4.3 Limitations

Despite our best efforts to have a more balanced gender representation, most of our participants were men (18 of 22). This was possibly due to two reasons. First, most of our participants were recruited online (17 of 22) and online communities are predominantly male [230]. Second, the field of software engineering and programming is heavily skewed towards men [107] and disabled people also marginalized on the basis of their gender face further barriers to participation in computing fields [39].

Our participants hail from various countries. We are aware that cultural and legal differences persist in the workplace of different countries and this is likely to shape our participants' experiences. To contrast and compare the findings, we would need a larger sample of participants from each of the countries. Another limitation of our study is that the findings are informed mainly by self-reported data with BVI developers. We do not have the perspectives of their sighted colleagues. We, therefore, cannot speak to how sighted programmers feel about changing the work practices.

3.5 Conclusion

Work at the intersection of accessibility, HCI, and programming tends to examine people with visual impairments and their interaction with a single category of tools [7, 8, 166, 173]. However, our study suggests that, in a collaborative environment, BVI developers use an ecosystem of tools to accomplish their tasks. They have to access internal resources such as databases and virtual machines, acquire the information on responsibilities assigned to them from project management software like JIRA and Microsoft Teams, and use communication software to coordinate collaborative programming activities. In this light, we echo the findings of Das *et al.*, who also find that people with visual impairments use multiple ATs and word processors in work environments to collaboratively write with their colleagues [62]. Similarly, our study highlights the need for access studies in HCI to be broader in their examination of programmers' interactions with tools to collaborate with their colleagues.

CHAPTER 4

CodeWalk: Facilitating Shared Awareness in Mixed-Ability Collaborative Software Development[1]

4.1 Introduction

Synchronous software engineering activities like pair programming and code walkthroughs are useful for developers to share knowledge, improve and refactor the source code, and debug the code together. Developers have to remain *closely synced* to achieve effective collaboration and communication in these activities. If a developer moves to a new location in the code *i.e.*, line, function or file, their collaborator should follow them immediately; real-time edits should become apparent right away to enable quick feedback. When collocated, sighted developers work together on one system to observe and discuss the source code without expending additional effort to stay on the same page. However, referencing a collaborator's screen is inaccessible for blind or visually impaired (BVI) developers, often requiring them to drive the collaboration on their computers [162].

This screen inaccessibility is magnified in remote synchronous collaboration. Developers typically either use screen shares or integrated development environments (IDEs) with integrated collaboration support (e.g. VS Code, JetBrains, Floobits, CodeTogether, etc.) to work synchronously (see Figure 4.1). These approaches assume that everyone can see their screens [162, 215]. However, BVI developers cannot access the screen share video or the visual awareness cues in IDEs through assistive technologies such as screen readers. They have to constantly request that their sighted colleagues speak code locations, such as line numbers, functions, file names, etc., out loud, in order to stay in sync. Much like collocated collaboration, BVI developers end up driving the activity. Sometimes, they even hand off their computer's control to sighted colleagues in refactoring and debugging tasks, which reduces their own agency.

[1]This chapter is adapted from the publication: Potluri, Venkatesh, and Pandey, Maulishree *et al*. Codewalk: Facilitating shared awareness in mixed-ability collaborative software development. In *Proceedings of the 24th International ACM SIGACCESS Conference on Computers and Accessibility. 2022.*

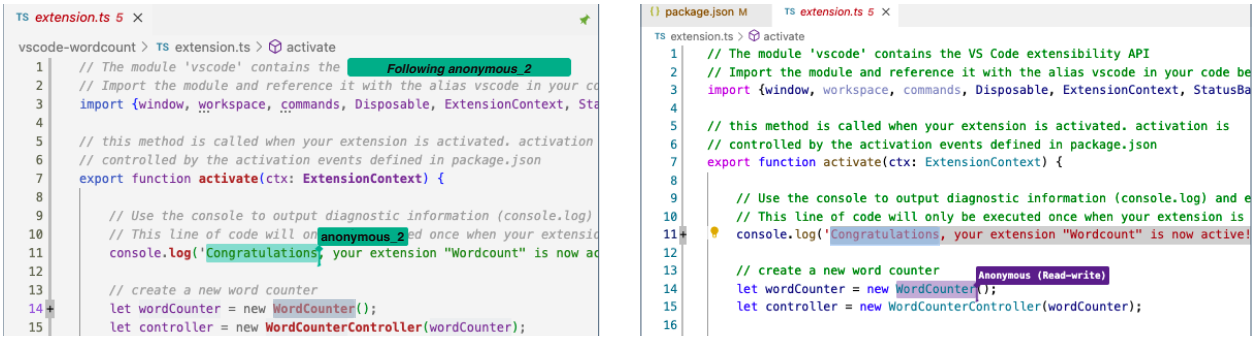


Figure 4.1: VS Code is an IDE that offers integrated collaboration support through its Live Share extension. Live Share enables developers to work together on source code through document sharing and co-editing in their respective IDEs. It represents collaborators' location and selection in the source code through colorful cursors.

That task of providing accessible awareness information lies at the heart of facilitating effective remote collaborations in mixed-ability contexts [94, 109, 162]. Research has begun to explore making shared workspaces accessible to BVI users [64, 119]. However, these solutions are intended for general-purpose document co-editing; they do not cater to the unique needs of software engineering tasks like pair programming and code walkthroughs.

Prior to our work, no programming environment with *accessible*, *remote*, synchronous co-editing support was publicly available. We have created and released such an environment, in accordance with four design criteria (see §4.2), which include maintaining the agency of BVI developers and reducing their burden to drive the collaboration.

In this paper, we present *CodeWalk*, a set of features added to Microsoft's Live Share VS Code extension[2] (available to all Live Share users since November 2021), with support for remote, synchronous code review and refactoring tasks. Our design derives from an investigation of relevant literature in remote collaboration and a set of formative design activities led by a BVI developer and researcher on the team. During our design process, we compared techniques for capturing a collaborator's navigation, editing, and referential (i.e., pointing at or highlighting parts of the code) activities and presenting them to BVI users using a combination of sound effects and speech (see §4.4.1). We evaluated CodeWalk in a within-subjects controlled experiment involving 10 BVI professional developers (see §4.5). Our results show that CodeWalk increased study participants' awareness of their collaborator's actions and reduced the coordination overhead required to sync on code locations. Participants strongly preferred CodeWalk over the baseline — the unextended version of VS Code with Live Share which provides awareness cues visually (see §4.6).

Our work is an end-to-end demonstration of how to improve the accessibility of an IDE's remote collaboration features. We make the following contributions to the HCI, accessibility, and software

[2]https://docs.microsoft.com/en-us/visualstudio/liveshare/use/enable-accessibility-features-visual-studio-code

engineering design communities.

1. A design for supporting tightly-coupled synchronous programming activities (see §4.3 and §4.4).

2. CodeWalk, an implementation of a set of features added to Microsoft's Live Share VS Code extension that makes synchronous programming tasks accessible to BVI developers (see §4.4).

3. Validation of our design's capability to increase shared awareness and facilitate efficient synchronization during remote collaboration, meeting our design criteria (see §4.5 and §4.6).

The COVID-19 pandemic exacerbated the need for collaborative programming environments that can enable BVI developers, one of the largest physical disability groups of software developers [205], to participate in remote work at par with their sighted peers. CodeWalk addresses this timely need and provides a foundation for future software engineering tools to facilitate accessible collaborations.

4.2 Design Criteria

Based on this literature review (see §2.5), we synthesized the following design criteria for CodeWalk. We annotate each criterion with citations to the literature that inspired them.

- D1. *to minimize the cognitive load on the BVI developer* [64] (e.g., maintaining accessible workspace awareness [63, 119] while minimizing conflict with collaborators' conversations) during synchronous programming activities

- D2. *to maintain agency of BVI developers* [196] in mixed-ability collaboration [162, 62]

- D3. *to reduce the burden on BVI developers* [215] of driving the collaboration session to accessibly collaborate [162]

- D4. *to support tightly-coupled collaboration* between all collaborators [218]

4.3 Design

In this section, we describe the formative design activities we conducted that led to the design of CodeWalk.

Table 4.1: Descriptions of code walkthroughs that informed CodeWalk's design. Each walkthrough occurred between a pair of sighted and/or BVI developers along with a sighted observer watching a shared screen or listening to a BVI developer's screen reader.

ID	Leader	Follower	Sighted Observer Task
CW1	Sighted	BVI	Watched sighted developer's screen
CW2	BVI	Sighted	Watched sighted developer's screen
CW3	Sighted	BVI	Listened to BVI developer's screen reader
CW4	BVI	Sighted	Watched BVI developer's screen

4.3.1 Formative Design Activity 1: Choosing a Baseline IDE

Our first design activity was to choose a good baseline IDE to build upon, one that was already accessible by BVI developers and had facilities for collaborative co-editing support. Several popular IDEs that offer collaborative co-editing support, such as JetBrains CodeWithMe [108], Sublime [93] and Atom [85], are unfortunately difficult to use by BVI developers. A few require BVI developers to perform additional setup steps and others even lack accessibility support for screen reader users to be able to perform basic code editing [213].

By contrast, we found Microsoft's Visual Studio Code IDE (VS Code) to be both accessible and easily extensible. Its accessible command palette makes it easy for screen reader users to find commands. In addition, we learned that the VS Code team speaks with the BVI developer community regularly to improve its accessibility [135, 136]. VS Code supports collaborative work through its Live Share extension. Similar to Saros [182], Live Share supports synchronous collaboration through a Follow mode feature, which draws the leader's cursor in all of the followers' IDEs and keeps it in sync as the leader moves around the document. Live Share also supports co-editing, keeping a shared view of the source code in sync between the connected parties. Though there is little information on the accessibility of Live Share's features for BVI developers, there are enough features to make it a good choice for our project's baseline IDE.

4.3.2 Formative Design Activity 2: Code Walkthroughs

To assess the accessibility of VS Code with Live Share for teams with BVI and sighted developers, three of the authors (one of whom is also a BVI developer) conducted four code walkthroughs (see Table 4.1). Two walkthroughs were led by a sighted researcher and two were led by the BVI researcher-developer. We tried to cover all combinations of abilities in a pair along with each taking on a leader or follower role. All of the walkthroughs involved mixed ability teams (CW1, CW2, CW3, and CW4). In all of the walkthroughs, a third sighted researcher observed the shared screen, however in code walkthrough CW3, the sighted observer simply tried to listen to and comprehend

the (slowed down) screen reader audio used by the BVI developer.

Each code walkthrough looked at different example source code from VS Code's library of extensions. Each example extension consisted of multiple files written in TypeScript, several files of which were read through during each walkthrough. Each walkthrough was 60 to 90 minutes long.

In addition to recording the code walkthrough sessions, the sighted observer took detailed notes during each code walkthrough, noting down accessibility breakdowns and workarounds. They minimized their interruptions, limiting questions to clarifications of leader and follow actions to locate one another in each file and to help work around any non-obvious accessibility barriers. In total, the sighted observer took six pages of notes. Immediately after each code walkthrough, both the leader and the follower memoed, reflecting on their experiences [181]. As a group, the entire research team rewatched the sessions from the recording and discussed the memos and notes in their weekly meetings.

We observed that conversations between leader and follower primarily focused on code discussions and clarification questions during breakdowns in accessibility. When in Live Share's Follow mode, the IDE drew each developer's cursor and synced their viewports on everyone's screen. Unfortunately, since this information was only visual, the BVI follower was not aware of any of it and was frequently lost. Consequently, the sighted leader had to speak their location out loud to the BVI follower to help facilitate tightly-coupled collaboration (Design Criterion D4). This active approach was error-prone because the sighted leader sometimes forgot to mention their location, especially when they were navigating quickly around the source code. The BVI developer initiated another workaround, asking clarification questions to sync with the leader. This often put the burden on the BVI developer (Design Criterion D3) to request enough accessible information to follow along in the code walkthrough.

When the BVI developer led the code walkthrough, they never got lost. However, they often became unsure whether the Follow mode had really synced the pairs' viewports, and had to ask their sighted follower to confirm they could see the expected code in their window. Finally, the sighted collaborator often talked at the same time as the BVI developer was trying to listen to their screen reader. This made it difficult for the BVI developer to listen to either audio stream. The research team discussed that some of the audio overlaps could be avoided if the sighted developer knew when the BVI developer's screen reader was speaking. But, revealing the use of AT is a sensitive issue for many screen reader users, thus we decided to designed CodeWalk's features to judiciously and carefully make use of audio effects and speech to reduce the cognitive load (Design Criterion D1) experienced by the BVI developer.

Table 4.2: Mixed ability code walkthrough scenarios that informed the design requirements for CodeWalk. Each scenario was inspired by at least one code walkthrough. Sighted+ and BVI+ indicates more than one developer. Following or watching "on the side" splits the VS Code editor and puts one in Follow mode.

Scenario	Leader	Follower	Walkthrough	Activity
1	Sighted	BVI	CW1, CW3	Follower joins collaboration session hosted by leader.
2	Sighted	BVI	CW1, CW3	Follower tethers cursor to leader.
3	Sighted	BVI	CW1	Follow leader "on the side" without tethering.
4	Sighted	BVI	CW1	Follower restarts tethering after watching leader "on the side"
5	Sighted	BVI	CW1, CW3	Follower tells leader they are lost.
6	Sighted	BVI	CW1, CW3	Follower takes notes during collaborative session.
7	Sighted	BVI	CW1, CW3	Follower fails to notice what command the leader just used.
8	Sighted	BVI	CW1, CW3	Follower asks leader about the command they just used.
9	BVI	Sighted	CW2, CW4	Leader invites follower to join collaboration session.
10	BVI	Sighted	CW2, CW4	Leader jumps to follower's cursor, answers the follower's question, and jumps back.
11	BVI	Sighted	CW2	Leader asks follower to show them something.
12	BVI	Sighted	CW2	Leader asks follower a question to test if they are lost.
13	BVI	Sighted	CW2, CW3	Follower asks for help using a "I need help" command.
14	BVI	Sighted, BVI	CW2, CW3, CW4	Leader gets follower's cursor location from VS Code.
15	BVI	Sighted+, BVI+	CW4	Leader gets approximate location of multiple followers from VS Code.

4.3.3 Formative Design Activity 3: Synthesizing Code Walkthrough Scenarios

Inspired by our literature review and our code walkthroughs, we created 15 scenarios comprising short events that occurred (or we wished had occurred if the IDE were more accessible) across our code walkthroughs. In addition, we considered both sighted and BVI developers as leaders, but skipped scenarios involving solely sighted developers. Some scenarios explore possible communication mechanisms between leaders and followers (e.g., non-verbal, notifying the leader, notifying all collaborators, or not notifying at all and syncing up after the session). All of these scenarios are listed in Table 4.2.

Here is an illustration of Scenarios 1 and 2, which expose some inaccessible features of the baseline IDE and the design features we explored to address them. Blake, a sighted developer, wants to refactor a piece of code. He asks Mia, a BVI developer and his colleague for advice. They set up an audio call to verbally discuss the code as they view the code in a Live Share collaboration session hosted by Blake. Blake shares the session link with Mia, who joins the session and is

presented with Blake's code in her IDE. Mia invokes the Follow mode command to stay in sync with Blake's viewport. As Blake navigates in the IDE, Mia's IDE shows a copy of Blake's cursor in a distinct color, which unfortunately is not accessible to Mia. Though the viewport changes, Mia's cursor remains untethered from Blake's. Therefore, Mia has to occasionally interrupt Blake to ask him to speak his line numbers and keywords out loud so that she can navigate there herself and use her screen reader to read the code that Blake is referring to.

This scenario exposes the limitations and asymmetry of current IDEs in supporting tightly-coupled collaboration and shared awareness (Design Criterion D4) among sighted and BVI developers. To address the asymmetry, CodeWalk automatically tethers a BVI developer's cursor to the leader's (section 4.4.1), so that their cursors move in unison whenever the leader initiates the navigation action. However, this only happens in Follow mode. Now, Mia should normally have no doubts about being in sync with Blake, but can still detach (i.e., turn off Follow mode) from the leader if she wants to explore the code on her own. The feature can also be useful in Scenarios 9 through 13 where she leads the collaboration. She does not have to worry about the correct code segment being displayed in Blake's IDE.

Furthermore, to reduce the burden on BVI developers (Design Criterion D3), to preserve their agency (Design Criterion D2), and minimize cognitive load, we designed several features in CodeWalk to convey a collaborator's location, navigation, and edit actions accessibly to BVI developers using a passive, automated approach. We describe the detailed implementation of these features next.

4.4 CodeWalk

CodeWalk is a set of features released with Microsoft's Live Share VS Code extension that supports accessible, remote, synchronous code review and refactoring activities. We describe the cursor tethering and audible feedback features that power its capabilities and discuss some of the implementation details that we found needed careful design.

4.4.1 Features

4.4.1.1 Cursor Tethering

Live Share's Follow mode yokes each collaborator's editor viewport together, a passive visual mechanism that is not useful to BVI developers. CodeWalk facilitates tightly-coupled collaboration (Design Criterion D4) by tethering BVI collaborators' cursors with the host of a Live Share session. In designing this feature, we explored several options to toggle tethering along with various levels

Table 4.3: Use of audio cues to convey awareness indicators in Follow mode (unless specified otherwise in the row)

Awareness Information	Non-Speech Indicator	Non-Speech Indicator Frequency	Speech Indicator	Speech Indicator Frequency	Built-in Visual Indicator
Viewport scrolls	Click wheel sound	Every scroll event	None	None	Screen scrolls
Scroll direction	Falling or rising tone depending on direction	When scrolling stops	None	None	Can be inferred from the scrolling viewport
Current Viewport	None	None	"Lines X to Y on screen"	When scrolling stops	Visible on screen
Cursor moves by single line to line N	Keyboard click	Every cursor move	"Line N"	1.5 seconds after cursor moves end	Cursor moves on screen
Cursor moves multiple lines to line N	Falling or rising tone depending on move direction	Every event	"Line N"	1.5 seconds after cursor moves end	Cursor moves on screen
Cursor moves by multiple lines to line N	Falling or rising tone depending on move direction	Every event	"Line N"	Every event	Cursor moves on screen
Selection	Depends on selection (keyboard/mouse)	Every event	"Selection on line N"	1.5 seconds after selection is made	Selection visible on screen
Edits on follower's line	Keyboard type	For every character typed	None	None	Cursor moves on screen; edits visible on screen
Edits on follower's line (Follow mode off)	Keyboard type	For every character typed	"<collaborator>is editing the same line as you"	As long as edits continue on the same line	Cursor moves on screen; edits visible on screen
Edits within 5 lines of follower (Follow mode off)	Proximity sound	For every character typed	"<collaborator>is editing nearby"	As long as edits continue on the same line	Cursor moves on screen; edits visible on screen
Follow status	Pull and push sound	When follower starts and stops following leader	"You are now following <collaborator>"	Every event	None

of autonomy, ranging from always tethering cursors to only tethering cursors when the user toggles Follow mode.

Always tethering the BVI developer's cursor to their collaborator minimizes their cognitive load (Design Criterion D1), but reduces their agency (violating Design Criterion D2), as the sighted colleague would have total control over their BVI colleague's cursor. BVI and sighted developers navigate code and interfaces differently [174, 8, 168]; they may want to read a part of the code that their sighted collaborator is talking about by character or by word, a kind of fine-grained navigation a non-screen reader user has no idea about. To support this need, we support temporarily untethering the BVI follower's cursor whenever they move it around, giving them control to move the cursor to the code they want to read. After 10 seconds of inactivity, CodeWalk retethers the cursors.

4.4.1.2 Conveying Collaborator Actions via Audio

CodeWalk uses a combination of sounds and speech to passively communicate a tethered collaborator's location and their navigation and edit actions, reducing the burden on BVI developers to ask about them (Design Criterion D3). We explored several sound designs, drawing inspiration from audio cues used by popular accessible navigation apps and operating systems. We experimented with futuristic artificial sound effects as well as skeumorphic sounds of keyboard clicks and scroll wheels. We felt that since BVI developers were already familiar with the sounds of standard computer hardware, the skeuomorphic sound effects would be the best one to convey navigation actions. Navigation distance and direction lacked obvious skeumorphic analogs, so we designed a set of artificial rising and falling tone sound effects to be played a short time after navigation activity ends. If the user clicks the mouse somewhere else in the codebase, CodeWalk plays an artificial "teleportation" sound instead of a mouse click to make it more obvious that something drastic has happened to the cursor location, which may invalidate the mental model the BVI developer has of the region where they thought the cursor was located.

In designing using speech and sound effects, our primary focus is to minimize cognitive load relative to the frequency and specificity of the information to be conveyed. We draw inspiration from accessible data visualization and programming efforts [201, 211, 127] and use speech to announce highly specific information like line numbers, which is needed less frequently. We use sound effects to convey less specific information, such as the actions performed and navigation direction - these actions occur at a much higher frequency during collaboration. The use of speech and sound effects has shown to improve awareness between collaborators [132]. Sound effects minimize cognitive load on BVI developers (Design Criterion D1) because they do not require conscious interpretation and can be heard even when a screen reader is actively speaking. However, they do not give enough information about a collaborator's location. To address this, after navigation activity has stopped for 1.5 seconds, CodeWalk uses computer-generated speech to tell the BVI developer what line of code (and file name if it changed) they are now on. Most sound effects are around 200 ms (though one is longer, at 550 ms). Similarly, speech announcements are kept short and precise. The complete business logic for CodeWalk's sound effects and speech can be seen in Table 4.3.

Sighted collaborators commonly use visual reference space gestures such as cursor location, text highlighting, and mouse waving to refer to code [61], gestures largely unavailable to BVI developers [164, 117]. CodeWalk supports selection awareness by speaking the portion of code highlighted by a collaborator. This simplifies the process of understanding what a collaborator wants to talk about and reduces the burden on BVI developers to ask sighted colleagues to verbally announce their selections. An example illustrating these sound effects and speech can be seen in Figure 4.2.

Figure 4.2: Image shows BVI developer's code editor as she follows a sighted leader. CodeWalk tethers the cursors of collaborators in Follow mode. When the sighted leader uses arrow keys to navigate, CodeWalk plays skeuomorphic keyboard sounds for each line moved. When they stop navigation at line 19, CodeWalk plays an artificial falling tone to indicate downward movement followed by line number announcement. Similarly, when they highlight a word, CodeWalk announces the selection.

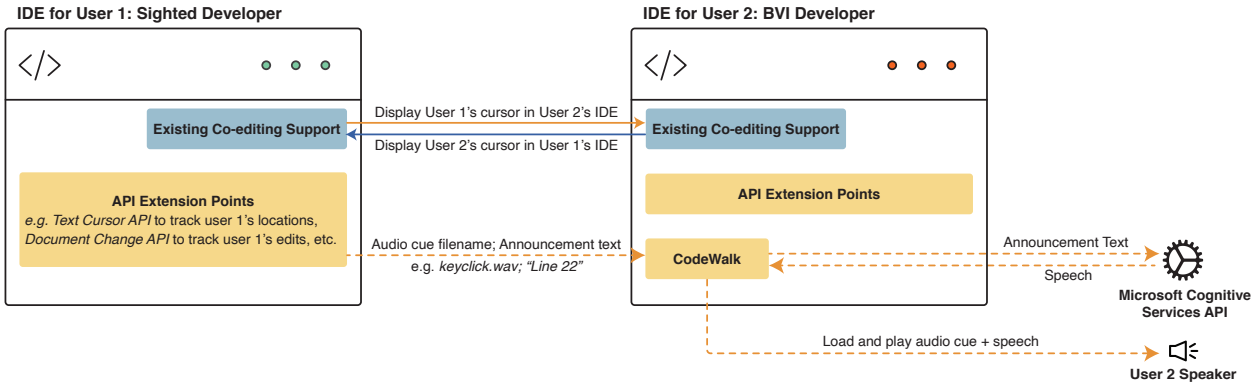


Figure 4.3: System Architecture Diagram for CodeWalk

When a collaborator edits the code with their keyboard, sharper, shorter key click sound effects are played. If the collaborators are untethered (i.e., Follow mode is off), then they may both be editing the document simultaneously. The baseline VS Code Live Share gives no indicator that collaborators' edits may collide, other than drawing the two cursors near one another. This, of course, is inaccessible to BVI developers. In CodeWalk, whenever the collaborators are editing within 5 lines of one another, CodeWalk speaks a warning, "your collaborator is editing nearby." If the collaborator is on the same line, the warning repeats, "your collaborator is editing the same line as you," which should hopefully cause the collaborators to stop what they are doing and negotiate their next actions together, verbally.

4.4.2 System Implementation

The basic architecture of CodeWalk can be seen in Figure 4.3. Each developer runs an instance of the VS Code IDE, extended by CodeWalk. CodeWalk extends four extensibility points provided by VS Code and Live Share (i.e., programming APIs enabling third-party developers to enhance specific features of the IDE), which we illustrate in the following scenario walkthrough.

Mia, a BVI developer (User 2 in Figure 4.3), installs CodeWalk along with Blake, a sighted colleague (User 1 in Figure 4.3). Blake and Mia enter into a joint collaboration session facilitated by VS Code Live Share's existing co-editing support. Various extension points are triggered as they collaborate. The first triggers when Blake changes his cursor location. It sends the new location to Mia's IDE along with a tag explaining what action caused it. CodeWalk then runs through its business logic (described in Table 4.3) to determine the kind of audio feedback to play (sounds and/or speech) and queues them for playback on Mia's computer. Each of Blake's navigation actions may trigger a tuple of one or two sounds along with a spoken message, each separated by a delay. Typically, the first sound is skeuomorphic (*i.e.*, for key clicks, mouse clicks, or the scroll wheel). It is followed by a 1.5-second delay, a falling or rising tone (to indicate navigation direction), and an spoken announcement of the new line number. The 1.5 second delay avoids spamming Mia with additional sounds and speech if Blake pauses momentarily during his actions (e.g. pausing to adjust mouse wheel when scrolling through a file). As there are no cross-platform APIs for asking screen readers to generate custom announcements, we generate CodeWalk's spoken announcements using the Microsoft Azure Cognitive Services Text-to-Speech (TTS) API. If too many sounds are requested to be played in a row, queued sounds and speech may be delayed. If they are delayed over one second, it is considered out of date and CodeWalk ejects it from the playback queue. Additionally, CodeWalk categorizes sounds into notifications and warnings. Events associated with the former are interruptible, meaning if a second event comes in before the first one is done playing, it will cancel the first and start the second right away. Warning sounds are uninterruptible. They are reserved only for edit actions to prevent co-editors from overwriting one another's changes.

The second extension point tethers the co-editors' cursors together. When Mia follows Blake, her cursor will move automatically wherever Blake's cursor goes. When tethering is turned on, all of Blake's edits will always happen on the same line as Mia's, so we suppress any spoken warnings. When tethering is turned off, edit sounds and announcements only play when Blake is editing within five lines of Mia, else the sheer quantity of sounds would overwhelm her.

A third extension point tracks and conveys selection actions between the co-editors. Mia hears a verbal announcement of the selection whenever Blake selects some text in his editor, as long as she is tethered to Blake.

The final extension point queues sounds to be played whenever Mia toggles Follow mode on or off. Similar to what happens in Zoom or Microsoft Teams, a sound is played whenever a co-editor joins or leaves the collaboration session, followed by an announcement of the co-editor's name and their cursor location.

4.5 Evaluation Study

We conducted a within-subjects study to understand and compare the effectiveness of VS Code Live Share with CodeWalk features against our *baseline*, plain VS Code Live Share [137]. Our study aimed at answering the following research questions: (1) How well does CodeWalk improve coordination during remote synchronous collaboration between sighted and BVI developers? (2) How does it affect the communication between developers about the source code? (3) How does it shape BVI developers' perceptions of their collaborative experience?

4.5.1 Participants

Eligible participants had to be 18 years or older, identify as blind or visually impaired, be comfortable with using screen readers, have at least a year of programming experience in one of the following languages: C, C++, C#, Python, JavaScript, TypeScript, or Java (*i.e.*, the programming languages into which we translated our tasks), have collaborated on code, and be able to communicate about code in spoken English. We recruited participants by posting on social media platforms and mailing lists (e.g., program-l) that primarily comprised BVI developers.

Our study accepted 10 BVI developers (P1–P10). Nine participants identified as male; one as female. Participants were between 21 and 47 years old (average age 33.6; median age 31.5). Table 4.4 summarizes the details of participants' demographics, country of residence, and current job title. Each participant was compensated with USD $100 (or its equivalent in local currency) for their participation in the study.

4.5.2 Tasks

We employed a 2x2 within-subjects experimental design. Each participant, in collaboration with a sighted confederate (one of the authors and the study coordinator), performed a series of tasks without CodeWalk (the baseline condition) and another set of tasks with CodeWalk (experimental condition). Like prior HCI studies [38, 105, 149], the confederate was instructed to strictly and consistently follow the study protocol with all participants, which is known to lead to more generalizable results [101].

We developed two sets of tasks (henceforth, set A and set B) for the study. We randomized the order of task sets and the conditions across the participants. Each set comprised three tasks, resulting in a total of six tasks. We designed the tasks to range from easy to difficult within each set, enabling participants to ease into the programming environment. In both sets, the first task was based on editing a string; the second task required editing code central to program execution; the third task required refactoring a specified set of lines into a function. The research team conducted

Table 4.4: Demographic characteristics of CodeWalk participants and their study session details

#	Gender	Age	Country	Profession	Condition 1	Condition 2	Prog. Language
P1	M	34	India	Senior Software Engineer	CodeWalk (Set B)	Baseline (Set A)	JavaScript
P2	M	24	India	Software Development Engineer	CodeWalk (Set A)	Baseline (Set B)	Python
P3	M	27	India	Technology Analyst	CodeWalk (Set B)	Baseline (Set A)	JavaScript
P4	M	47	USA	Software Engineering Manager	Baseline (Set B)	CodeWalk (Set A)	JavaScript
P5	F	29	USA	Data and Applied Scientist	CodeWalk (Set A)	Baseline (Set B)	Python
P6	M	44	USA	Senior Program Manager	Baseline (Set B)	CodeWalk (Set A)	C#
P7	M	21	USA	Software Engineering Intern	Baseline (Set A)	CodeWalk (Set B)	Python
P8	M	46	Sweden	Software Developer	Baseline (Set B)	CodeWalk (Set A)	C#
P9	M	35	USA	Senior Software Developer	Baseline (Set A)	CodeWalk (Set B)	Java
P10	M	29	Netherlands	Freelance Developer	CodeWalk (Set A)	Baseline (Set B)	Python

multiple rounds of discussion to ensure that both task sets were of equivalent levels of difficulty.

The confederate *led* the code walkthrough and asked the participant to *follow* them during the tasks. They asked the participant to recommend changes and solutions to complete each task. Participants could explore, edit, or verify the actions of the confederate as they wished. This made the collaboration feel more natural.

All tasks were based on Hangman,[3] a common text-based game. The tasks were representative of software development activities and required participants to perform code reviews, bug fixing, and code refactoring. We downloaded publicly available source code for Hangman in C#, Java, Python, and JavaScript so that participants could perform the tasks in their preferred programming language. For internal validity, we selected code samples with similar lengths and modified their source code to have similar file names, function names, variable names, and code structure. All code samples included (1) a main code file representing the game's logic, (2) a text file containing 851 words to play the game, and (3) a text file listing both sets of tasks in the order determined for the participant. The C# code sample included an additional file that represented the game's UI; this file was referenced for the string editing task. The code samples in JavaScript included

[3]https://en.wikipedia.org/wiki/Hangman_(game)

HTML/CSS files which were not required for the study. Table 4.4 lists the order of conditions and task sets for each participant, along with the programming language used in their study session. Appendix A provides the main source code files, tasks, and words used in the study.

We conducted a pilot study with one BVI developer (not including in the main study) to ensure that each task set was possible to complete within 20 minutes and the total study time did not exceed 90 minutes. Based on their feedback, we found we needed only to improve two things: our instructions on how to connect remotely and the description of the extensions' features in both conditions.

4.5.3 Procedure

We conducted the studies remotely over Microsoft Teams or Zoom, as per the participant's preference. Participants were not required to turn on their cameras for the study. Before the main tasks, the study coordinator explained the key features of the IDE used in the study, the baseline condition, and CodeWalk. We asked participants to share their screen without including the system audio so that the confederate would not hear the participant's screen reader or CodeWalk audio output. We used the video conferencing tool's recording feature to capture the conversation between the participant and the study coordinator, which we referred to during our analysis.

To facilitate switching between study conditions, we created a Windows 10 virtual machine (VM) with two different versions of VS Code — one version with the baseline condition and another augmented with CodeWalk. Both versions had the same features, keyboard shortcuts, and UI settings. We installed JAWS (version 2020) and NVDA (version 2021) on the remote VM. We also set up Code Factory Eloquence [56], a popular text-to-speech (TTS) synthesizer used to customize screen reader voice and speech. Before each study session, we set NVDA as the default screen reader.

Participants connected to the VM using Microsoft Remote Desktop software. Upon login, we informed participants that they could modify the screen reader settings. Only P1 and P4 used JAWS; the others performed the tasks with modified settings for NVDA. We also turned on screen recording within the VM to record the screen reader speech. Due to technical glitches with screen recording software, we were unable to record the screen reader usage for P4 and missed a portion of the screen reader usage for P1 and P2.

Participants were instructed to switch to the IDE window for the first condition (see Table 4.4 for the order) and invite the study coordinator (referred to as the 'confederate' in this paragraph) to the collaboration session. The confederate was under strict instructions to hide the participant's shared screen to not look at their IDE contents. Participants could ask questions about the IDE features or share their comments about the baseline and CodeWalk during the study. We believed

this approach allowed the collaboration and the conversation to proceed more naturally. After twenty minutes, the participant and the confederate switched to the other experimental condition to perform the next set of tasks.

After each condition, participants verbally responded to a 12 statement Likert-scale questionnaire (see Table 4.6). The questionnaire was adapted from existing scales [189, 61] and assessed participants' opinions regarding awareness and collaboration. Participants had to indicate on a five point scale whether they strongly disagreed (1) or strongly agreed (5) with the statements. The study concluded with an informal interview about participants' experience with CodeWalk and a short questionnaire about their personal and programming background.

4.5.4 Data Analysis

The confederate wrote analytic memos [181] after each study session to reflect on how each condition shaped their awareness and collaboration. One researcher reviewed all the video recordings and conversation transcripts to highlight the timestamps of sync operations, analyzed using a Poisson regression (see §4.6.1). Section 4.6.2 discusses how we adapted an existing list of codes from [61] to analyze the conversation between the confederate and the participants. Section 4.6.3 details our analysis of participants' responses to Likert-scale questionnaire . Lastly, two authors used descriptive coding [140] to analyze the interviews and organized the codes into themes around collaboration and feedback (see §4.6.3).

4.6 Study Results

4.6.1 How well did CodeWalk improve coordination during collaboration?

To analyze how well the participants could follow the confederate, we compared the number of times they attempted to *sync their location* with the confederate's location. We operationalized location syncing as attempts, including successful attempts, by participants to move their cursor to the confederate's location using one of the following: (1) moving from one file to another (2) going from one line to another (3) using the find tool to search for a specific word to navigate to its location (4) toggling the tether command if unsure of the tether status of cursors. Participants synced their locations to read the code that the confederate was referring to and to follow them during the collaboration. We hypothesized that participants would require fewer sync operations in CodeWalk because they would feel *less lost* compared to the baseline. We analyzed the screen share recording and participants' screen reader speech to calculate the total number of sync attempts.

The median value for the number of times participants tried syncing in CodeWalk was 1, com-

Table 4.5: Reference codes used to analyze the conversation between the CodeWalk participants and the sighted confederate

Code	*Description*
Deictic	When a participant or the confederate uses a deictic reference such as this or here, e.g., *"Let's start with this task."*
Anaphora	When a participant or the confederate refers to a past action or location, e.g., *"Can you go back?"*
Abstract	When a participant or the confederate uses a broad category to refer to an object, e.g., *"We need to understand the function."*
Reading	When a participant or the confederate loudly reads a portion of the code, generally done when approximate location of the collaborator is known, e.g., *"Press Enter to leave the game!"*
Typing	When a participant or the confederate is referring to the text being typed, e.g., *"Let me confirm what you wrote."*
Specific	When a participant or the confederate uses a specific name to describe an object, e.g., *"Let's go to didGuessCorrect()."*
Line number	When a participant or the confederate uses a specific line number, e.g., *"I am on line 31."*

pared to the median value of 8 in baseline, a huge drop. Figure 4.4a a visualizes the number of sync attempts for each participant in both conditions. As recommended for integral data with possibility of rare occurrences, we fit a Poisson regression [242]. We found that participants made *significantly fewer attempts* to sync locations in CodeWalk condition ($p = .000875 < 0.01$). The result indicates that CodeWalk enabled the participants and the confederate to stay closely coordinated during collaboration. Note, P7's outlier value in the CodeWalk condition. The followup interview and his screen share recording revealed that he had not realized that his cursor was tethered to the confederate's. He interpreted the sounds and speech in CodeWalk as locations he should move to, resulting in similar behavior across both conditions.

4.6.2 How did CodeWalk affect communication about the source code?

Since CodeWalk conveyed information on a co-editor's location and actions, we hypothesized that CodeWalk would enable the participants and the confederate to converse about code using more abstract and deictic references compared to the baseline. We also hypothesized that they would use more line numbers and specific names in the baseline condition compared to CodeWalk. Both our hypotheses were informed by D'Angelo and Begel [61].

To analyze how the confederate and the participant referred to locations in the code, we adapted and extended the list of *referents* from D'Angelo and Begel [61] by making two additions (anaphora and reading). Table 4.5 shows the codes of all 7 referents along with their definitions and examples. Each time a participant or the confederate made a reference to the code, we recorded the referent and its category. We calculated the total number of referents uttered by the confederate and the participant in each study session. Thus, we ended with 14 total referent counts (7 for the confederate; 7 for the participant) in each condition per session. We normalized the data in each condition by calculating the percentage of referents in each category. We did not include P2's data because he experienced significant lags in screen reader speech causing the confederate and P2 to verbalize and read code aloud for a large portion of the study (an unlikely scenario for collaboration in outside the study). We visualize the fraction of referents in each category for each condition in Figure 4.4b. The figure shows that *specific* referents were used most heavily during the tasks in both conditions. We also note a greater usage of deictic and abstract referents in the CodeWalk condition compared to the baseline. We carried out a one-way ANOVA to compare the percentage of referents in both conditions. The analysis revealed no significant difference in the percentage values of any category except the abstract referents ($p = .037 < 0.05$), which were greater in the CodeWalk condition.

4.6.3 How did participants perceive their collaboration experience with CodeWalk?

4.6.3.1 Responses to Likert-Scale Questionnaire

Table 4.6 lists the statements in the Likert questionnaire along with the p value of participants' responses in both conditions. All statements had an equal or higher median value in the CodeWalk condition. A higher median indicates more agreement among the participants, implying a better overall experience with CodeWalk.

A one-tailed Wilcoxon signed-rank test indicated that the value for responses to ten out of twelve statements were significantly higher in the CodeWalk condition ($p < 0.05$). Their statement codes are followed by an asterisk in Table 4.6. The significantly different values confirm that participants *felt* more aware of the confederate's locations and actions with CodeWalk. Furthermore, participants felt that the shared awareness was reciprocated by the confederate when using CodeWalk *i.e.*, the participants felt that the confederate was also aware of their actions (S4 in Table 4.6). This indicates greater shared intentionality with CodeWalk.

Two statements (S9 and S12) were not significantly different across conditions. These focused on participants' perceptions of the confederate's communication style and effectiveness in collaboration during the tasks. Since the confederate remained unchanged in both conditions, participants

may have felt that their communication style remained consistent across conditions. Participants' responses to S9 and S12 may have also been subject to *demand characteristics*, cues that shape participants' desire to form a positive impression on the experimenter [150]. Participants may have wanted to appear polite in their responses about the confederate's communication and collaboration abilities.

Table 4.6: Statements in the Likert-scale questionnaire used during CodeWalk's evaluation study. Rightmost column indicates p values for participants' responses in both conditions. All statements had equal or higher median value in the CodeWalk condition. * beside the statement code indicates $p < 0.05$.

#	Statement	p value
S1	I was keenly aware of everything in my environment.	.0009*
S2	I was conscious of what is going on around me.	.0035*
S3	I was aware of what my teammate did and how it happened.	.0029*
S4	I was aware that my teammate is aware of my actions.	.0197*
S5	I am aware of how well we performed together in the team.	.0118*
S6	I felt like my teammate and I were on the same page most of the time.	.0118*
S7	I could tell what my teammate was thinking about/looking at/talking about most of the time.	.0328*
S8	I felt like we shared common subgoals as we worked on the task.	.0294*
S9	My teammate communicated clearly during the task.	.1284
S10	I communicated clearly with my teammate during this task.	.0169*
S11	It was fun to work with my teammate on this task.	.0294*
S12	My teammate worked effectively with me to accomplish the task.	.1284

4.6.3.2 Interview Results

Video analysis revealed that the participants felt aware of being in the confederate's vicinity. They would highlight code or read code aloud to direct the confederate's attention. In addition, we observed that the confederate could easily keep track of the participant's cursor with CodeWalk's tethering feature. The participant's cursor was always visible in the confederate's viewport, and if the participant moved out of the viewport to read code, the confederate would scroll to keep track. On the other hand, participants reported that they *"leaned on the communication"* with the confederate *"pretty heavily"* (P7) in the baseline condition. They had to either wait for the confederate to verbally announce their location using a line number or function name or request the location information to sync cursors, also indicated by Figure 4.4a.

We noted instances where participants used CodeWalk's tethering feature to direct the confed-

erate. For example, P5 asked the confederate to take her *"to the line again"* to revisit the source code. The confederate moved their cursors to the location P5 had specified; the sounds confirmed arrival for P5. Later on in the interview, P5 shared that the *"auto move [of cursors] was really useful"*. Similarly, P6 directed the confederate to move their cursor to various lines during the refactoring task. After each move, he would explore the code at the destination line, make recommendations for improving the code, and then instruct the confederate to take him to the next location.

Participants used CodeWalk's sound effects extensively to maintain awareness of the confederate's actions. For instance, after the confederate finished typing, P9 mentioned, *"Yeah, I can tell you are done 'cause the typing noise stopped"*, and then went on to verify the changes made by the confederate. Participants used the speech announcements to keep track of location changes. Many participants phrased this as being aware that *"things were happening"* (P8). Even on the occasions when the confederate moved quickly, leading to a succession of sounds, participants felt that *"at least [CodeWalk] conveyed a sense of movement"* (P4). The increased awareness seemed to positively shape the participants' feelings about collaboration and assuaged their worries about feeling lost: *"Because I could just snap to wherever you were, I wasn't worried about wandering off" — P4*.

Furthermore, participants liked the design choice of primarily using audio cues to convey the confederate's actions and relying on speech sparingly. They shared that the audio cues *"packed a lot of info"* (P7) without seeming verbose. In addition, participants did not seem to mind when the audio cues played simultaneously with the screen reader speech, but they indicated a preference for shorter sounds. Most participants were able to quickly map the skeuomorphic audio cues to their awareness indicators. It took a few participants longer to associate the non-skeuomorphic audio cues with their intended meaning of direction changes. However, they acknowledged that they had not *"used it [CodeWalk] enough"* (P6) to remember the sounds and believed that *"some more sessions"* (P1) would enable them to map all the audio cues to their respective meanings.

Every participant told us that they would like to use CodeWalk to collaborate with their teammates. P5 mentioned that using CodeWalk in code reviews would enable her to be on the *"same page without lagging behind."* P7 shared that CodeWalk would be *"absolutely instrumental"* in his pair programming assignments, and he would *"install it immediately"* if it were released. P9 felt that it would allow him to mentor junior developers by letting them *drive* collaboration sessions: *"When I'm collaborating, I'm the one driving and I share my screen and they look at it. It's just easier that way [...] I would be much more likely with an extension like this to let them drive more often."* Participants also appreciated that CodeWalk was built for VS Code, a mainstream and accessible IDE that sighted *"people might have"* (P5). Upon its launch, they could use it without asking their colleagues to switch to a new IDE. These quotes suggest that CodeWalk can

enable BVI developers to participate in collaborative activities without requiring them to manually manage the sessions on their own.

4.6.4 Threats to Validity

Our study employed a sighted research team member as a confederate for all study sessions. Employing a single confederate across all participants is common in HCI [38, 105, 149] and is recommended for maintaining the internal validity of the experiment [101]. Despite following the study protocol strictly, the confederate may have gained experience and improved as a communicator with each session. Thus, it is likely that later participants' collaboration experience may have been better than the former, resulting in fewer differences in metrics between conditions. We believe the within-subjects design choice would have addressed any learning effects on the part of the confederate.

Due to their research experience in accessibility, the confederate may better understand participants' awareness needs than sighted people unused to collaborating with BVI developers. Participants commented that the confederate was *"a very good communicator"* (P7), also confirmed by the lack of significant difference in responses to S9 on the Likert-scale questionnaire (see Table 4.6). The confederate's communication may have suppressed differences in referent counts between conditions. Therefore, in real-world conditions with more typical collaborators, CodeWalk may show even more improvement in communication metrics over the baseline.

We deployed CodeWalk on a cloud-based virtual machine (VM) to simplify the installation for our participants. Using screen readers through remote VM may have increased latency. Some participants reported lags in screen reader playback which may have impacted their experience with the extension and shaped their feedback. The latency issues are unlikely to occur in real-world conditions, since the extension would be installed on the user's own home system. Thus, we expect the experience of CodeWalk to be better upon its release.

4.7 Discussion

Overall, we find CodeWalk successfully translates and conveys reference space gestures from sighted developers to their BVI colleagues, extending Buxton's model [52] for effective remote collaboration to mixed-ability collaborations. In this section, we summarize our findings, consider the role of interdependence in our design, and relate our results to the two projects that are most similar to ours. We then reflect on our own research practices and propose future work.

4.7.1 Summary of Findings

Our study results show that significantly fewer attempts were needed by our BVI participants to sync locations with CodeWalk than in the baseline condition. This suggests that the coordination burden (Design Criterion D3), which often requires explicit communication of awareness cues between collaborators, is reduced through CodeWalk's sound effects and speech. Automating the transmission of code location and navigation actions helps to ensure that the sighted colleague also benefits from a reduced coordination burden, since they need not remember to convey those actions verbally either. The participants' increased use of abstract referents to code locations showed a corresponding decrease in the number of more specific referents (using line numbers and function names). This suggests a reduction in cognitive load (Design Criterion D1) on the part of the BVI developer. It also shows an increased sense of shared awareness and shared intentionality between the participants, which ensures that the BVI developer had the capability to contribute equitably according to their ability rather than be sidelined by inaccessible collaboration tools.

From the Likert scale statements, we learned that participants felt that CodeWalk improved their awareness of their environment, their teammate, and their teammate's actions. They also felt that they were more likely to be on the same mental page most of the time and were able to work effectively together. BVI developers were more likely to highlight text on the screen using their keyboard, confident in the knowledge that their sighted colleagues would be able to notice it and react to it. BVI participants also could tell from the sound effects when their colleagues were navigating or when they had stopped, concluding that they were now free to engage in conversation and explore the source code. Not only did this increase improved their communication, it also minimized the cognitive load (Design Criterion D1) of trying to intuit what their sighted colleague might be doing without any audible feedback.

CodeWalk's cursor tethering was designed to support tight coupling (criterion D4) between participants at the task level, so that when one navigated through the code or edited some text, the other would immediately be made aware and be able to respond. Some participants responded by directing the confederate to move to additional locations, showing increased agency (Design Criterion D2), looking around the code with their own screen reader, and then driving the confederate to the next code location.

CodeWalk's skeumorphic sounds (e.g., key clicks and scroll wheel sounds) were straightforward for the participants to understand with no training. However, some sound effects, e.g., rising tone and falling tone, used to convey directionality of movement, were not immediately obvious to the listeners. While participants got better at distinguishing these during their study session, others may need more time to get better at this.[4]

[4]See Cat_ToBI (http://prosodia.upf.edu/cat_tobi/en/ear_training/listening.html) to practice distinguishing rising and falling tones from one another.

CodeWalk's spoken sentences were necessary to orient the BVI participants after their sighted colleagues navigated to new areas in the code. Sometimes, however, these spoken words collided with the participant's own screen reader speech. An early version of CodeWalk played its sounds and speech by extending the NVDA screen reader, which enabled us to detect overlapping speech utterances and cancel one of them. However, to ensure CodeWalk worked with multiple screen readers on multiple platforms (including Mac and Linux), we used Microsoft's Azure Cognitive Services to generate speech and platform-specific sound APIs to play it. It is possible to address the overlapping audio, however due to time constraints, we were unable to program CodeWalk to cancel our audio while the screen reader was talking. We encourage screen reader and operating system manufacturers to offer extensible platform-agnostic APIs for integrated systems like CodeWalk with screen readers.

One interesting form of spoken collision remains. Sighted colleagues receive no indicators when BVI users are listening to their screen readers, and thus do not realize to stop talking to the BVI user to avoid overlapping with the screen reader or CodeWalk. Participants expressed a need for avoiding *"double-speak"* (P8) between the collaborator and speech announcements by their screen readers and CodeWalk. We plan to explore designs of a visual indicator to non-screen reader users of CodeWalk to let them know when screen reader speech is active for any of their collaborators. This feature should require BVI users to opt-in before it is turned on because BVI users' opinions of whether to reveal their use of AT to colleagues varies by culture [120] and may have significant workplace consequences [26].

4.7.2 Accessible Co-Editing

Lee et al. [118]'s CollabAlly tool developed similar sound effect and speech-based feedback for BVI writers in common co-editing environments (e.g. Google Docs). CollabAlly found success in identifying collaborators' ongoing work and comments in the document, enabling collaborator awareness to avoid overwrites synchronously and asynchronously. Our work examined the awareness needs found in synchronous tasks of code walkthroughs and reviews and found the timeliness of push-based notifications vital to enable BVI collaborators to stay in sync with their sighted colleagues for extended periods of time. Many coding tasks fluidly switch between asynchronous and synchronous modes leaving unanswered how to best support users' cognitive load by conveying awareness information simultaneously using pull and push-based modalities.

Das et al. [64, 63]'s Co11ab work supporting mixed ability co-editing stops short of handling collaborators editing near one another. Prior to CodeWalk, these kinds of close edits would preferentially disadvantage the BVI collaborator as their sighted colleagues could see the impending collisions and take their own steps to avoid them. CodeWalk's use of non-interruptible warning

messages as colleagues get too close served to encourage all parties to communicate using alternate, more accessible, channels (e.g. a concurrent audio call) in order to appropriately synchronize their edits and avoid conflicts.

4.7.3 Interdependence

Bennett et al.'s reframing of the goals of assistive technology as *interdependence* instead of *independence* ring true in CodeWalk's scenarios [27]. Collaboration between colleagues of mixed abilities encourages each to play to their own strengths, while requiring that each cede some of their own power and control to cooperate effectively with others. Working together in a code walkthrough or code review, a BVI developer who might have special expertise in accessibility can disseminate that knowledge to sighted non-specialists *in situ* and create a better result for their customers. As shown by Pandey et al. [162], long-term mixed ability collaborators establish mutual reliance by learning how to work together by paying attention, responding to, and adapting to one another's task-related behaviors, habits, and needs. CodeWalk's sound effects and speech events make a colleague's navigation and edit work visible to BVI collaborators, enabling them to be used by a BVI collaborator as an essential assistive technology for remote collaborative work. Finally, CodeWalk challenges the established hierarchy of sighted participants controlling the task, enabling BVI developers to *lead* code walkthroughs and reviews instead of meekly defaulting to follow.

4.7.4 Researcher Reflections

This work improves our own practice to communicate accessibly and to advocate for our own accessibility when participating in collaborative software development activities. For example, the BVI member of the research team now always asks sighted collaborators to verbalize code locations. A sighted member realized that he needed to remember to stop speaking every so often to allow his BVI collaborator to "read" the code for themselves using their screen reader. The study coordinator recognized that each BVI developer's access and communication needs are different and that expressing these needs can be tricky when collaborating with someone for the first time. They have become mindful of attuning their communication to the preferences of their BVI collaborators. Finally, as we collaboratively author this paper using the Overleaf Latex editor, we yearn for it to make use of auditory feedback in order to fully include our BVI co-author in our writing efforts.

4.7.5 Future Work

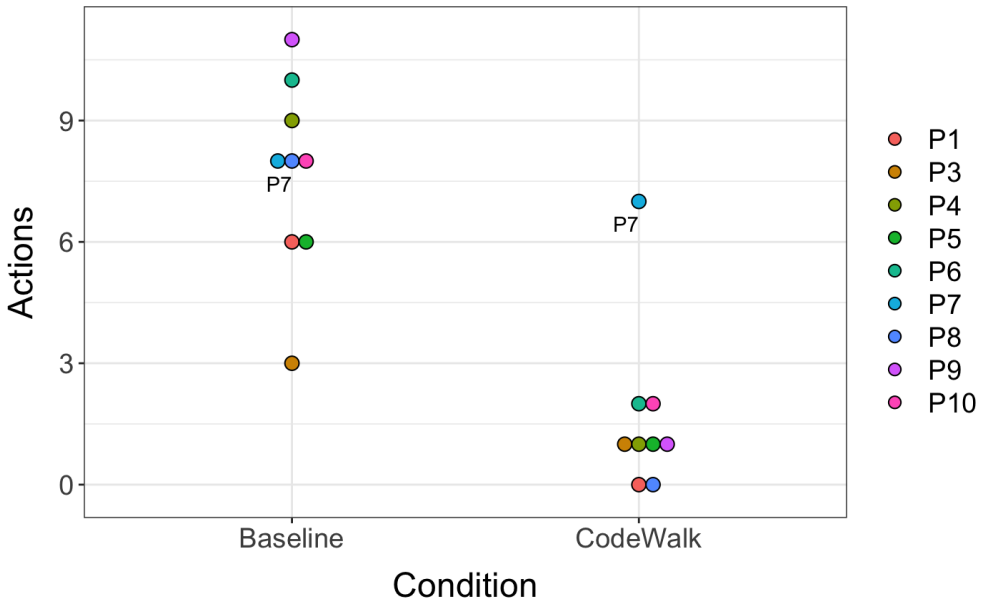
In the future, we would like to explore how to combine the lessons learned from CodeWalk, CollabAlly, and Co11ab in supporting mixed ability remote collaboration, whether it be for document co-editing, software co-editing, or additional collaborative software development tasks. In particular, future designs should explore ways that BVI collaborators can most effectively and equitably lead interactions, in one-to-one and one-to-many scenarios, including collaborations involving two or more BVI developers.

Many participants said we should ensure that CodeWalk was accessible to deaf-blind programmers and usable with Braille displays. They felt that its reliance on the audio medium could exclude deaf-blind programmers. In future, we will extend our design to support communication of awareness information through tactile media.

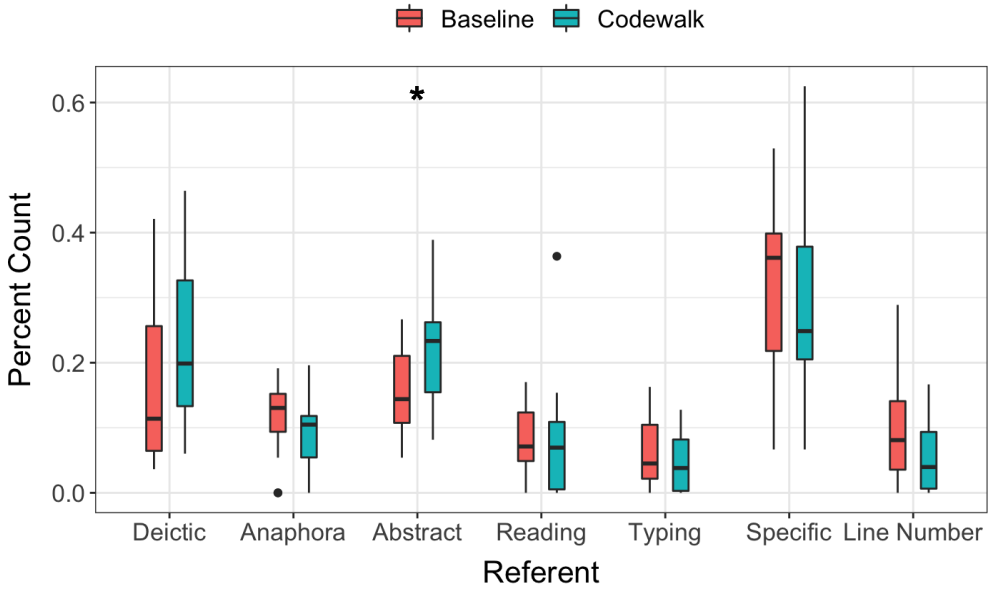
We recommend that application design standards, such as ATAG (Authoring Tool Accessibility Guidelines) and WCAG (Web Content Accessibility Guidelines) be extended to support mixed ability teams and provide non-visual information about collaborators' location, navigation, and edit operations. This could increase the use of accessibility practices in the design of collaborative authoring tools.

4.8 Conclusion

Existing tools to facilitate tightly-coupled software development tasks rely on visual cues and create accessibility barriers to equitably collaboration for BVI developers. To address this accessibility gap, we designed, developed and evaluated CodeWalk, a set of features added to Microsoft's VS Code Live Share extension that makes a collaborator's location in a code file and their actions accessible through cursor tethering, as well as sound effects and speech. CodeWalk's features improved coordination between BVI developers and their sighted peers while reducing the explicit effort that BVI developers need to put to stay coordinated. We hope CodeWalk can serve as an exemplar for IDE manufacturers to make their environments more accessible to blind and visually impaired software developers.



(a) Number of times participants attempted to sync locations with confederate in each condition.



(b) Percentage of referents of each type uttered in each condition. A * is shown above referent types that are significantly different across conditions.

Figure 4.4: Results from video and conversation analysis of CodeWalk's Evaluation Study

CHAPTER 5

Accessibility of UI Frameworks and Libraries[1]

5.1 Introduction

UI frameworks and libraries have become increasingly popular for web and mobile development [44]. They help developers by offering native and custom UI components that enable the creation of complex interfaces [100]. Several frameworks and libraries, such as Flutter [89], React Native [131], and Cordova [221], also enable cross-platform development, allowing product teams to reach a wider number of platforms and end-users while developing in a single codebase. Many frameworks also claim to be accessible out-of-the-box, suggesting that the resulting UI would be accessible for people with disabilities. Given their widespread use and the advantages they offer, UI frameworks and libraries can have an outsized effect on the accessibility of UI programming and the web. They underscore the need to understand the accessibility of UI development for BVI developers as they use these UI frameworks and libraries. The consistent growth of UI developer job roles [205, 158, 159] also highlights the need to understand and improve the accessibility of the field to make it more inclusive.

Prior research in Human-Computer Interaction (HCI) and software engineering has studied the accessibility challenges in UI development [162, 130]. However, their focus was mainly on understanding the accessibility issues with IDEs and the need for sighted assistance in development. This chapter takes a deep dive into the challenges in UI development and collaboration due to use of UI frameworks and libraries. Specifically, we ask the following research questions: (1) What are the motivations for BVI developers to use UI frameworks and libraries? (2) How do these frameworks and libraries shape their programming experiences and collaboration with sighted developers?

We report findings from a two-part mixed-methods study. First, we performed content analysis of 96 publicly archived mailing list posts on UI development; we followed this with 18 semi-structured interviews with BVI developers who have explored or used UI frameworks and libraries

[1]This chapter is adapted from the publication: Pandey, Maulishree, *et al*. Accessibility of UI Frameworks and Libraries for Programmers with Visual Impairments. *In Proceeedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2022).*

as part of coursework and professional responsibilities. Drawing on our analysis, we contribute the following:

- Evidence that accessibility challenges are difficult to isolate to programming tools or UI frameworks and libraries. We need to consider the interplay between programming tools, assistive technologies, operating systems, and UI frameworks to improve accessibility (see §5.3.2).
- An understanding of how accessibility challenges hindered code writing, testing, and demonstrations for BVI developers. (see §5.3.3)
- Design recommendations regarding documentation and supporting help-seeking for BVI developers. (see §5.4)

Our findings contribute to HCI, accessibility research, and software engineering research. They are especially important to people designing visual programming tools and languages.

5.2 Methods

We adopted a mixed-methods approach and conducted two studies to understand the UI development experiences of BVI developers.

5.2.1 Study 1: Analyzing Archived Posts on UI Development

We scraped the archived posts dated from January 2018 to December 2021 from the program-l mailing list (program-l@freelists.org) — an active and free discussion group for BVI developers to ask questions and share resources. The archive for the mailing list is publicly available [1] and dates back to November, 2004. Our choice of the four-year time period was guided by the goal to capture conversations before the start of the COVID-19 pandemic and to target the most recent technologies.

The posts and replies are archived as separate web pages in chronological order. We scraped a total of 11,915 web pages (average 248.23 emails per month). We combined the original posts and their replies into threads and saved them as text files for analysis, resulting in 2,607 files.

The first author went through the subject lines to identify threads most likely related to UI development. We identified a total of 726 threads on the topic. Next, we randomly sampled 150 threads over three rounds (50 per round). The approach allowed us to perform qualitative analysis in intervals and reach thematic saturation [144]. When coding, if the content of the thread seemed unrelated to GUI development, we removed it from our analysis. In total, we analyzed 96 threads;

the breakdown after eliminating unrelated threads was 33, 31, 32 threads in round 1, round 2, and round 3 respectively. The final list of threads was organized alphabetically and indexed to quote from in the present paper. We describe our analysis of the email threads in section 5.2.3.

5.2.2 Study 2: Semi-Structured Interviews

Our thematic analysis gave us a breadth of understanding about the accessibility challenges in UI development when using frameworks and libraries. To gain a more in-depth understanding of their use and impact on collaboration, we decided to conduct interviews. The first author conducted semi-structured interviews with 18 BVI developers. Eligible participants had to be at least 18 years old and either explored or possess experience in using UI frameworks and libraries to build web or mobile applications. We recruited participants through snowball sampling (n = 2), posting on the program-l mailing list (n = 13), and posting on the r/blind community on Reddit (n = 3).

Participants were 19 to 46 years old (median age 26.5; average age 28.16). Only one participant (P17) identified as female; the remaining participants identified as male. P2 and P5 identified as programmers with low-vision and used screen magnifiers and zooming respectively. P3, P4, P9, and P12 shared having retinitis pigmentosa; P14 shared having macular degeneration. The onset of visual impairment differed among these participants. The remaining participants reported having little to no usable vision since birth. Besides P2 and P5, each participant used a combination of screen readers. JAWS [80] and NVDA [153] were the most popular screen readers among our participants. P3, P4, and P9 reported using VoiceOver [11] along with other screen readers. Table 5.1 summarizes participants' demographics and programming experience.

Our interviews lasted between 40 and 75 minutes and were conducted remotely over participants' preferred video conferencing platform. Participants verbally consented to audio recording the interviews. We asked participants about the frameworks they have explored or currently use, challenges they encounter during programming, their experience with documentation and tutorials, and their motivations for learning UI development. The interviews concluded with a short questionnaire about participants' demographics and programming background (Table 5.1). We compensated each participant with a $30 USD gift card or its equivalent in local currency. Each participant interview was transcribed in English for analysis, described in the next section.

5.2.3 Analysis

Two members of the research team analyzed the first round of email threads using open-coding to identify initial themes, followed by inductive coding [181] for all of the threads. We developed a total of 41 codes, which were clustered into 7 higher level themes. The members also wrote analytical memos [181] during the coding process to analyze emerging themes and identify gaps

in the data. We performed weekly reviews as a research team to discuss the findings and prepare questions that would be relevant to follow-up on in interviews.

We were unable to transcribe the first two interviews due to the poor quality of the audio recordings. We relied on our notes for those interviews. The remaining interview transcripts were first open-coded by two team members, followed by organizing the data into codes from Study 1 and creation of 1 additional high-level theme. After coding the transcripts, we did a final reorganization of the codes, resulting in six high-level themes, which included codes on challenges in UI development, lack of documentation, considerations behind choosing UI frameworks, etc.

5.3 Findings

We present the key results from our analysis, focusing on how the (in)accessibility of UI frameworks and libraries shapes the programming processes and experiences of developers with visual impairments. Quotes are slightly edited for clarity. Quotes from archived mailing list threads (study 1) contain thread IDs (T#) and quotes from interviews (study 2) include participant IDs (P#).

5.3.1 Motivations for Using UI Frameworks and Libraries

We found that BVI developers were motivated by different reasons to pursue UI development. Employment opportunities were a common reason among interview participants (n = 9) to learn UI development. Participants shared that being familiar with UI development improved their chances of being hired, even though their preferred job roles were back-end development. Other participants were intrinsically motivated; they (n = 3) shared that they had always been interested in UI development. P7 shared that he had always considered himself *"as more of a designer"*. Furthermore, learning UI development established conversational fluency with front-end developers and designers:

> P16: *"Sometimes I should check something with the front end guys. And it's crucial for me to know how web development works in a big picture. [...] how HTTP works, what are HTTP methods - GET, POST, how RESTful API works and so on."*

Many interview participants (n = 5) explicitly stated that they preferred using UI frameworks and libraries rather than writing code from scratch. UI frameworks offered a *relatively* independent way of creating the front-end. For example, when a member inquired about the possibility of developing *"good-looking web interfaces as a blind person,"* members on the mailing list strongly recommended frameworks such as DOJO [77] and Bootstrap [217]:

> T15: *"It [Bootstrap] is highly idiomatic and easily calculatable with ratio of columns and rows. Its built-in components are already good-looking enough, and you can easily customise with skins or simple CSS touches. Its developers also consider and even dictate best practices for accessibility."*

Frameworks and libraries also provided helpful visual scaffolding for developers and the designers they worked with. Designers had to develop visuals using the existing components instead of requesting developers to create custom components. As P3 explained, he could directly *"use the SDK"* in his code:

> P3: *the mockups were based on the components that already exists [...] whoever builds like the visual design part has to align with the standards of the SDK. It's not like they are inventing a UI.*

Frameworks and libraries also helped differentiate between UI design and development responsibilities. Participants shared that they did not have to *"worry about colors, contrast, stuff like that"* (P3). The visual details were either considered by the framework designers or were specified by the in-house design team. Thus, BVI developers could focus on the functionality of the UI:

> T15: *my boss brought our company's graphic designer into my department to help. He has taken my super-simple UI and turned it into something my company could show off. So there definitely is a certain art to it and vision is not the issue.*

Developers also spoke of their unique expertise in making UIs accessible for end-users with visual impairments. They sought assistance to make the UI components usable for sighted end-users and coached sighted developers on how to make the components accessible for screen reader users :

> T3: *as screen reader users, we are the experts [...] You always want someone with a pair of eye-balls to check out the colors. You also want someone that has a decoration talent to help identify where each color combination should go on the site*

Thus, many interview participants considered the job roles to be interdependent. According to them, each team member, with their skills and competence with assistive technologies, improved the accessibility and user experience of the interface.

When choosing which framework to learn, we noted a strong preference for frameworks that were popular. For instance, P8 shared that he was *"currently working in winUI because it is the hottest technology"*. Similarly, when advising a developer about selecting a framework among Angular, Vue, and React, the mailing list members recommended the lattermost since it *"still has the lead in terms of jobs"* (T82).

In summary, the use of frameworks and libraries afforded higher levels of independence, delineated between design and development responsibilities, enabled creation of good looking UIs, and improved employment opportunities for BVI developers. However, as we explain next, the limited accessibility of front-end frameworks and programming tools could hinder developers' collaboration and performance in the workplace.

5.3.2 Accessibility Challenges

Accessibility barriers played a decisive role in our participants' programming experiences. We first discuss their experiences with software critical to UI development, such as IDEs, emulators, and browser developer tools, followed by the challenges with UI frameworks and libraries.

5.3.2.1 Inaccessible Programming Tools

Consistent with prior work, we confirmed that GUI builders in most IDEs were not accessible with screen readers [162]. Sighted developers can use them to *drag and drop* the UI components and create the layout quickly. Since mouse interactions are not accessible to people with visual impairments, they often had to *"hand write everything for the UI"* which took *"a lot of time"* (P15). In section 5.3.3.1, we describe how the different approaches to UI design affected collaboration between BVI developers and their sighted colleagues.

We recorded instances of mailing list members searching for accessible GUI builders (n = 3) so that they do not have to type the entire UI code. For instance, one thread enquired about accessible interface builders for C++. While the discussion led to the discovery of an accessible extension, it only offered a limited set of widgets:

> T38: *The name of this extension is Nitisa. This is a Visual Studio extension and can be designed for C. But it doesn't use Visual studio as a Toolbox. I would love to have a GUI designer that can use Visual studio Toolbox.*

Developers logged issues on GitHub and directly reached out to development teams to improve the accessibility of GUI builders and IDEs. Some product teams acknowledged accessibility issues and proposed fixes, which developers viewed positively. However, the improvements could also be slow to come through, with the updates sometimes removed from the mainstream tool:

> T73: *you would want to have Git installed, so you can point to the accessibility branch and run WXGlade once you have switched to that branch.*

The quote above is from a thread where it was pointed out that to use wxGlade, the GUI builder for wxPython, one needed to check out the *accessibility* branch instead of working off of the main

branch. In a similar vein, participants shared that the updates to the IDEs could negatively affect the accessibility features and they had to either revert to an older version or await future releases.

Emulators provided by major IDEs like Android Studio were often inaccessible with screen readers. Developers had to run the application on their personal devices, which was time-consuming in the initial stages of the project. For freelancers, the lack of accessible emulators limited the number of devices they could test their application on. They had to either ask friends and family for their devices or hope that the UI they had developed would be displayed correctly on devices with varying screen resolutions and dimensions:

> P15: *I have to test it on different people's phone if they allow me to get the result. So it takes a lot of time. It really takes a lot of time!*

The problem was amplified for macOS and iOS. Apple's policy requires testing the app with their device. However, the exclusive availability of JAWS and NVDA on Windows and the poor accessibility of IDEs with VoiceOver, Apple's screen reader, made Windows the preferred programming environment for the developers in our studies. Without an accessible emulator and availability of a device, they could not develop UIs for Apple devices:

> T92: *you need a mac in order to test your app on an ios device. This is quite frustrating [...] I could install a mac virtual machine, however then I have to deal with learning to use the OS and navigating my way around xcode. Has anyone found a way to develop apps for iOS that is accessible? Or is there an accessible iOS emulator that is good?*

The participants from Iran (P15 and P16) shared that they had to contend with an extra layer of inaccessibility. IDEs offered by Google and Apple, including the devices by the latter, were not usable in Iran because of the government sanctions imposed on the country.

Besides IDEs and GUI builders, accessibility issues with browser developer tools were mentioned most frequently in the email threads (n = 6). Poor accessibility would hinder developers from navigating and searching the DOM. To work around this, they had to either try different browser and screen reader combinations or get sighted assistance:

> T79: *I couldn't track down/find [graphical elements] in the original examples initially, but my sighted brother managed to find them sort of hidden in the DOM for me*

The different combinations of programming tools, browsers, and screen readers led to a long tail of individualized accessibility issues. The differences in programming environments made it difficult to provide instrumental help to address the accessibility problems. For example, one email thread shared tips and tricks to save Google Chrome's console logs due to poor accessibility of its Developer Tools. However, differences in keyboard layouts and browser versions made it difficult for members to apply the solutions effectively:

T23: *"I have the option to save the logs on google chrome. I think you are not running latest beta version of google chrome. Perhaps you try updating google chrome on your windows machine"*

5.3.2.2 Inaccessible UI Components

To be able to use a framework efficiently, the UI components must be accessible. To assess a framework's accessibility, participants shared that they often browsed the official documentation to find a mention of accessibility. This served as a hint for whether the development team had given any thought to accessibility. However, positive search results did not necessarily guarantee accessible UI components:

P12: *The page of that component library claimed itself to be 'out of the box accessible' and they [participant's team] blindly imported everything the modals, the accordions, the buttons, each and everything [...] And we found very disappointing results [...] the buttons looked like buttons but were announced like menus to the screen reader*

We noted a general consensus that no framework or library was completely accessible. Thus, the decision to use a framework or a library was based on competing factors such as availability of documentation, cross-platform support, and effort needed to improve the accessibility:

T72: *Try XOJO. It is a Windows based cross-plattform development tool using Basic language to develop apps for both Windows and iOS/Mac. It is not fully accessible but I can live with them.*

The mailing list members shared components they had made accessible and compliant through trial and error so that others could refer to them. In Section 5.3.3.2, we describe the impact of inaccessible components on programming processes such as debugging and testing.

5.3.2.3 Inaccessible Layout Managers

Layout managers—tools that automatically group and arrange UI components according to developer-specified constraints—considerably improved the UI development experience. Our participants shared that they often relied on these when tasked with creating the UI and preferred libraries such as PyQT and wxPython that offered a *relative* way of organizing the UI controls:

P10: *If you don't do a layout manager, you need to explicitly say everything. [...] You need to pass the coordinates [...] But, again, that doesn't make any sense to me because I don't know which coordinate to give because I am not seeing it.*

Layout managers also enabled the participants to edit UIs more easily since the dimensions were updated automatically when changes were introduced. As a result, participants felt more confident and competent working with these frameworks:

> P7: *[With wxGlade] I can have a reasonable degree of confidence that those controls are where I say they are*

However, not all frameworks and libraries offer layout managers. For instance, P7 shared that he has not *"found a similar thing"* that allows him to develop front-end for web applications with the *"same convenience"* as wxGlade does for desktop applications. Furthermore, layout managers can also be offered through IDEs or third party tools, which may not be accessible.

5.3.3 Impact on Programming Processes and Performance

The accessibility challenges mentioned above affected the workflows, collaboration, and performance of BVI developers, which we describe below.

5.3.3.1 Writing UI Code

As mentioned earlier, BVI developers either try to find accessible GUI builders—which are rare—or manually code the UI. Developers expressed concerns about the number of lines required to create UI components when typing the code in comparison to using GUI builders:

> T6: *If you design items [...] using the XML editor in Android Studio, as the graphical way of [...] is still inaccessible, you define every component in 4 lines if we don't count the wrappers. With Swing, you have a few lines more: you have to create a container too and add both to the frame which you created previously.*

Inaccessible GUI builders could also complicate collaboration with sighted colleagues. It prevented them from creating *"clean looking resource file"* (T30) that their sighted colleagues could review quickly. They also felt that the additional lines of code made readability and navigation difficult with screen readers, especially when editing the UI. They had to redo the calculations if dimensions or positions were changed. In contrast, the resource file containing the UI code was automatically adjusted for sighted developers as they manipulated the measures with the GUI builder. Similarly, identifying and updating the location of visual parameters was difficult given the nested nature of the source code:

> T2: *I just found myself overwhelmed by the number of options and layouts with very little idea how to make sure they do what I want. I lose track once I am about two levels deep into the user interface element structure.*

Some GUI builders also produce incomprehensible code. One GUI builder, for example, produced generic variable names for UI controls. It was difficult for the developers to map these names to UI controls' position and functionality:

> T56: *Putting 2 buttons on a WPF designer surface, then tabbing around, forces the screen reader to say 'grid', 'button', 'button', 'window'. What button is what one?*

P9 shared that he had instructed his team to provide meaningful names to the UI controls to make collaboration on UI code easier. After laying out the controls, his sighted colleagues would edit the variable names in the resource file. Participants also shared that sighted developers did not realize that if they dropped the elements in random order, it did not change the UI visually but disorganized the accessibility tree. Accessibility trees are based on the DOM tree and expose a semantic version of the UI to screen readers via platform-specific APIs [145]. If the UI elements are not in the correct order or misrepresented, then it affects screen reader navigation and interaction. For BVI developers, this hindered their ability to debug and test. P9 mentioned that he had told his sighted colleagues to be mindful of the *"tab order"* when using the GUI builder:

> P9: *if we have the correct tab order, you start in the upper left corner and you go through the controls and the labels and grids. But if the tab order is out of order, you can jump between [imitates screen reader]. That makes it very hard to manage.*

5.3.3.2 Debugging and Testing

A major consequence of poor accessibility was the difficulty in debugging and testing one's output. Furthermore, the broken accessibility of certain components prevented developers from reproducing the bugs of their sighted colleagues. For P16, it hindered his collaboration with front-end developers:

> P16: *When I want to reproduce a bug [...] some parts of this web UI is not very accessible [...] For example, when I press enter in a web element, it does not work [...] I found out that if I press* `insert + space` *to go from a browse mode to focus mode in my screen reader [...] it will work.*

As mentioned earlier, even the software and frameworks that enjoyed the consensus of being *largely accessible*, presented some issues. The scarcity of documentation on accessibility of UI components meant that BVI developers often had to just *"dive in and try"* (T8) to assess the severity of issues across frameworks and libraries:

> P14: *I produced a Qt 5 interface that I cannot interact with [...] after a long, long time of research, I learned about some basic things that can adjust the code to make it accessible to the screen reader.*

Given the general unavailability of documentation on the accessibility of UI components, mailing list members reached out to one another for documentation and resources and gathered reviews on a framework's accessibility. They would mention the framework they were using and the specifics of their programming environment. Others on the list would share their experiences with the framework in their specific environments and even offer to test the source code or the specific UI components at their end:

> T28: *wx uses native controls, so I don't see why they shouldn't work on the mac or Linux. When I get home I'll run one of my in progress wx apps on my iMac and I can give you definitive* information.

Sharing debugging and accessibility experiences allowed the developers to work around the lack of documentation and identify the platform and screen reader combinations on which their UIs would work. However, this kind of sharing and support was not possible for programmers working on proprietary and private codebases.

The time and effort needed to test and fix the accessibility of UI components could range from adding ARIA attributes [146] to the markup to using scripting tools like Web Accessibilizer [237] for fixing issues at scale to even writing code that uses separate UI components for different platforms to offer a consistent user experience with screen readers:

> P6: *what I ultimately had to do was add logic into the program that if you're running it on windows, it uses one version of the tree control and if you're running it on anything else, it uses a different version*

5.3.3.3 Social and Personal Implications

As prior work has found, BVI developers were often tasked with educating their colleagues about accessibility issues and advocating for accessible solutions [162]. Participants were often also tasked with explaining accessibility issues to their sighted colleagues. For example, P6 had to demonstrate the trade-offs of a cross-platform framework across Linux, Mac, and Windows and explain how UI components behaved differently with various screen readers:

> P6: *I would show him here's how it sounds on windows, here's how it sounds on Mac, here's how it sounds on Linux. Here's the information that one of the tree controls is giving you in one environment versus the other, and this is why this is a problem*

Participants also described having to advocate for accessible solutions within their team. Often the decision to use a particular framework or programming tool was taken by the team collectively. If they chose things with poor accessibility, it could severely impact the productivity of BVI developers. For instance, P8 had to convince his team to use Xamarin and Visual Studio for the

Android application they were building; the poor accessibility of Android Studio would keep him from giving his *"full efforts"*:

> P8: *I explained to them that if we develop using Xamarin, we will be able to do it in less time.*

The decision to switch to Xamarin came with trade-offs for P8. He said Xamarin did not provide access to all the Android APIs. He had to rewrite code to wrap some of the libraries on his own. We recorded concerns about the poor support for native libraries, including accessibility APIs, on the mailing list threads (n = 5) as well.

Inaccessible UIs also prevented our participants (n = 2) from using the UI and experiencing the user workflows independently. For instance, P10 had joined as a back-end developer on an existing project. He could not *"go back and make the"* UI accessible in one go. Unable to use the application fully, he could not build sufficient context about the project. He shared that he had to attend multiple meetings with the design team and his manager to understand the UI design and functionality expected from controls he could not access.

Both P10 and P16 shared that they had pushed for making their UIs accessible, not only to make themselves more productive but also for other screen reader users. However, it was difficult to implement accessibility in legacy UIs, an issue also raised in several email threads (n = 5). Furthermore, workplace dynamics complicated the implementation of accessibility. P10 mentioned that the changes had to be approved by senior management, who may consider the trade-offs between his productivity as a developer and the time it would take to improve the accessibility. P16 shared that his position as the only blind person in the organization and as a new member of the team foregrounded his request. Insisting upon accessibility could suggest to the team that he was not able to do his job as well as other developers.

Participants (n = 3) shared that poor accessibility of the UI presented challenges during demonstrations. In meetings involving stakeholders and clients, it could also suggest poor quality of work by the team:

> P16: *The problem is that when you want to give a demo to a client and there is accessibility issues, it slows you down [...] and they might think that you are not capable enough to do these things*

P16 further added that in remote client meetings during the COVID-19 pandemic, poor demonstrations could disclose his disability and reinforce ableist perceptions about his ability and competence as a programmer. Therefore, when presenting the UI to an external audience, participants generally had a sighted team member *"click on buttons for fill these forms for me"* (P16) while they handled the technical narration. The approach allowed them to present and highlight their

contributions. P3 also shared that he occasionally recorded his screen while operating the UI to capture the workflow and do *"non-live demo"* and independent presentation (P3).

These instances highlight that accessibility issues in UI development could affect responsibilities beyond software engineering tasks, which developers are expected to perform in professional settings.

5.4 Discussion

5.4.1 Accessibility of the Programming Environment

Much of the focus of HCI and software engineering research has been on improving the accessibility of programming tools [166, 187, 173] and programming activities such as debugging [173, 209], navigation [14, 20], and UI development [171, 111]. While these efforts are needed, our findings show that accessibility issues cannot be isolated to any particular programming tool or activity. They result from the interactions between the various software that make the programming environment — IDEs, browser developer tools, UI frameworks and libraries, operating systems, and screen readers. The combinations of these result in myriad configurations, which leads to a long-tail of individualized accessibility issues. The situation is exacerbated by the lack of (official) documentation and online resources that discuss accessibility. In the case of UI development, they complicate the processes of code writing, debugging, and ensuring accessibility with screen readers. They also impact collaboration between BVI developers and their sighted colleagues since they use different approaches to UI development.

While sighted developers can turn to large forums like Stack Overflow, the recourse for BVI developers is to reach out to one another and report the accessibility problems to the developer teams. However, we show that the differences in programming environments also make it difficult for BVI developers to give and receive instrumental help. We recommend researchers and designers consider the accessibility of the entire programming environment instead of considering accessibility improvements to any particular software. We also highlight the need to design platforms that can support information-seeking and help-seeking for BVI developers for accessibility challenges. We can draw on the archives of various online communities such as the program-l mailing list to create a wiki that documents preferred programming tools, UI frameworks and libraries, accessibility breakdowns to watch for, and their workarounds.

5.4.2 Meeting the Promises of UI Frameworks and Libraries

Our findings show that UI frameworks have the potential to allow for *relatively* independent UI creation with reduced need for sighted assistance. Familiarity with popular UI frameworks and libraries also made BVI developers eligible for growing employment opportunities in the field. However, the choice of the framework was moderated by the availability of accessible UI components and accessible programming tools, native *look and feel*, and cross-platform support. Our analysis revealed that many frameworks listed themselves as out-of-the-box accessible and cross-platform. However, BVI developers often found that both promises were only partially met. The behavior of the components depended on the interaction between the programming environment and screen readers, thereby interrupting the process of debugging, testing, and demonstrations for BVI developers. Since the visuals and the performance of the UI components remained consistent for sighted developers, they seldom realized the impact of using these frameworks and libraries on their colleagues. Thus, BVI developers had to either convince their team to switch to more accessible alternatives or work with the choices made by their colleagues. We recommend that official documentation of the UI frameworks and libraries should prioritize accessibility and mention screen reader compatibility. The approach would also benefit sighted programmers by making them aware of the accessibility issues and fixes required for the UIs to work consistently with different screen readers and platforms.

5.4.3 Limitations and Future Work

Despite our efforts to have a balanced gender representation, our interview study's sample was heavily skewed towards men. We believe this was due to the software engineering field and the online communities we recruited from being male-dominated. In future work, we aim to understand the accessibility challenges and experiences of gender-based minorities.

The programming experiences of our participants were likely shaped by the workplace norms and laws specific to their country and culture. While we highlight the access issues resulting from government sanctions on our Iranian participants, the interview study's sample size did not permit an analysis of differences due to participants' resident country.

Our participants and mailing list members had a variety of vision-related disabilities. Due to the small sample size and since visual ability varies on a spectrum, we did not analyze how the visual impairment's nature and onset correlated with our participants' programming experiences. Our findings and recommendations are intended for people designing programming tools and visual languages for screen reader users. We will interview developers who use screen magnifiers to expand our results to other assistive technologies in future work.

The period of this research overlapped with the COVID-19 pandemic. Only one interview

participant (P16) shared how the pandemic affected his remote work experience. While none of the sampled threads mentioned the pandemic directly, an analysis correlating with the pandemic dates could surface accessibility challenges due to remote collaboration.

5.5 Conclusion

We conducted mixed-methods qualitative research to understand the experiences of BVI developers with UI frameworks and libraries. We show that the promises of cross-platform support and out-of-the-box accessibility are only partially met for BVI developers. Our findings highlight that accessibility barriers in UI frameworks and libraries interrupt critical programming processes and affect collaboration. We recommend prioritizing accessibility in the official documentation of UI frameworks and libraries. We also urge HCI researchers and practitioners to consider supporting the information and help-seeking needs of BVI developers.

Table 5.1: Programming experience & UI framework use as reported by participants in study on accessibility of UI frameworks.

ID	Age	Country	Prog. Education	Current Job Title	Prog. Experience	Prog. Languages & Frameworks
P1	23	USA	Bachelor of Computer Science	Software Developer	4 years	Java, C#
P2	26	USA	Bachelor of Computer Science	Software Developer	3 years	Java, PHP, Node.js
P3	30	US	Bachelor of Computer Science	Full Stack Developer	6 years	Java, TypeScript
P4	39	UK	Master's in Machine Learning	Computer Science Teacher	9-10 years	Python, Java, Swift
P5	30	Switzerland	PhD in Computer Science	Software Engineer	10 years	C++, Python, C
P6	22	USA	Bachelor of Computer Science	Incoming Software Engineer	7-8 years	C#, C++, Python, JavaScript
P7	19	USA	Self-Taught	Accessibility Specialist	5 years	Python (wxGlade)
P8	27	India	Master's in Computer Applications	Accessibility SME & Tech Lead	7-10 years	Java, C# (Xamarin), C, C++
P9	46	Sweden	Self-Taught	Software Engineer	30 years	C# (WinForms), .NET
P10	23	India	Bachelor of Computer Engineering	Software Engineer	3 years	Python (PyQT), C#
P11	27	Bahrain	Bachelor of Computer Science	Applying to Prog. Jobs	6 years	Python (wxPython), Java, Angular
P12	28	India	BTech in Electronics	Accessibility Consultant	6-7 years	Java, React, Swift, Kotlin
P13	22	Pakistan	Self-Taught	Student	2 years	C# (WinForms), HTML/CSS
P14	35	Hong Kong	Self-Taught	Research Assistant	10 years	HTML, Python (PyQT, Flask)
P15	35	Iran	Self-Taught	Freelance Software Engineer	13-14 years	JavaScript, Java, C# (Xamarin)
P16	26	Iran	Self-Taught	Junior Back-end Java Developer	3 years	Java, HTML/CSS
P17	24	Egypt	Self-Taught	Student (Preparing for Master's)	1-2 years	Python (PySimpleGUI), HTML/CSS
P18	25	India	Self-Taught	DevOps Engineer	3 years	ReactJS, Python (wxPython), Flutter

CHAPTER 6

Towards Inclusive Source Code Readability

6.1 Introduction

Reading code is one of the most fundamental and important activities in software development. *Readability* is a subjective measurement of how easy it is to go through any given code. More readable code is easier to comprehend and maintain in the long term. Typically, software maintenance forms 70 percent of any project's life cycle [40], making it the most intensive aspect of software development projects. Elshoff and Marcotty recommended adding another phase to the software lifecycle just to make the source code more readable [73]. They suggested the phase should require developers to apply consistent formatting, leave good comments, and remove unused code blocks. Software companies enforce adherence to coding standards [180] and use code reviews to ensure code quality [72]. Companies like Google and AirBnb have even made their coding standards public to ensure consistent and readable code contributions from the larger programming community [88, 6]. Others have recommended ensuring the readability of documentation to aid developers in making readable edits to codebases [96, 4]. Some also propose teaching students to write readable code as part of standard programming coursework [65].

The focus on readability has led to the development of rich visual design and functionality in code editors. For instance, indentation is long known to improve readability among sighted developers [197]. Code editors like Sublime Text, IntelliJ, etc display vertical lines to visually match indentation levels. IDEs such as VS Code offer mini-maps, which are zoomed out representations of the code structure. Developers can quickly navigate to different code blocks by identifying their shape and relative position in the map.

However, our current understanding of readability is based on the opinions and preferences of sighted developers [70, 154]. Blind and visually impaired (BVI) programmers use assistive technologies (ATs) such as screen readers. These lack the visual expressiveness and information density of graphical user interfaces (GUIs). The serial and ephemeral nature of screen reader output [22] leads to different browsing [33] and skimming [5] strategies among BVI people in

comparison to sighted people. The differences in screen readers and GUIs suggest that BVI developers may have different readability preferences, which need to be investigated to improve the accessibility of programming. In this paper, we focus on code readability for BVI developers. Specifically, we pose the following research questions:

1. What are the similarities and differences in code readability preferences between BVI and sighted developers and why?

2. What implications do these differences have for programming tools such as static analyzers and code editors, code styling guidelines, and programming languages?

We conducted an exploratory qualitative study with 16 BVI developers to answer our research questions. As part of the study, we asked participants to review 15 rules related to code readability (see Table 6.1). We presented two functionally equivalent but differently formatted versions of the same code snippet for each rule. One version's presentation was informed by PEP8, the official Python styling convention; the second version's formatting was informed by accessibility research. We asked participants to select the option they preferred for each rule. We asked follow-up questions to understand their preferences and concluded the study with a short semi-structured interview to elicit their experiences and workflows with code styling during collaborative activities such as code reviews.

Our research leads to a more inclusive understanding of *code readability* and makes the following contributions to the fields of Human-Computer Interaction, accessibility, and software engineering research:

- A taxonomy for what is good code formatting on screen readers vs. GUIs to support better code readability

- Empirical data to explain how various factors shape code readability on screen readers

- Design recommendations for code editors and programming languages

6.2 Background

Buse and Weimer defined code readability as "a human judgement of how easy a text is to understand" [51]. Readability is known to improve program comprehension but is distinct from overall understandability of code. For instance, readable code may still be difficult to understand due to unfamiliar APIs, poor documentation, and complexity of source code [186]. Sighted developers do not read code linearly. They are far more likely to *skim* the source code to locate regions of

interest where they do more *focused reading* [199]. In this section, we first draw on empirical studies at the overlap of accessibility and programming to explain what we know about code reading, comprehension, and navigation on screen readers. Then we summarize the factors that shape code readability for sighted developers.

6.2.1 Code Reading on Screen Readers

The primary focus of existing HCI and accessibility studies has been on code navigation and comprehension. However, a close review of these papers reveals a few insights about readability.

6.2.1.1 Linear Navigation

Prior research suggests that BVI developers want to avoid going through the codebase line by line but are forced to do so to get an overview of the code structure [8]. Francioni and Smith developed JavaSpeak to enable BVI developers to acquire details about the code structure and semantics more efficiently [79]. JavaSpeak spoke the code with different intonations to communicate structure. The researchers also suggested using prosodic elements like speaking rate, pitch, phrasing, etc to communicate semantic characteristics about code [79], a recommendation seconded by Stefik [208]. Screen readers like JAWS [80] and NVDA [153] use prosody to indicate the capitalization, which may come in handy during programming.

Stefik suggested using audio cues to inform BVI developers about the "scoping relationships between pieces of syntax" to communicate the information syntax highlighting provides [208]. An example of Stefik's suggestion would be the work by Hutchinson and Metatla [104]. They designed 12 audio cues to represent different programming constructs, such as the sound of door opening for `if` blocks and door closing for `else` blocks [104]. The ideas was that developers could use the audio cues to skip listening to the entire statement and move through the codebase more efficiently [104]. However, BVI participants in the study reported wanting more practice with the audio cues to map them accurately to the constructs. Evidence suggests that skeuomorphic audio cues can help reduce the learning curve [172].

Studies suggest that BVI developers avoided indenting code altogether unless collaborating with sighted developers [8, 162]. It makes linear code reading very verbose by announcing all the whitespaces. BVI developers are known to develop custom scripts to minimize the indentation announcement [8]. For similar reasons, they prefer to not receive all punctuation announcement [20]. One way to address verbosity is by outputting the semantic meaning of a code statement but that can make editing the syntax challenging in real-world projects [208]. Thus, researchers have used the approach only for making source code more understandable to novice BVI developers [185, 210]. Lastly, recent evidence shows that poor identifier or variable names affect code

reading and debugging on screen readers [162, 160] but we lack perspective on their casing, length, and naming choices.

6.2.1.2 Non-Linear Navigation Enables Skimming

Sighted people can use an array of methods for non-linear navigation: scroll, point and click, use keyboard shortcuts, utilize IDE features like tree views and mini maps, and keyword search. BVI developers only have a subset of these options available to them to make sweeping jumps through the code [172]. Keyword search is reportedly one of the most common methods for code navigation [8, 175]. BVI developers have reported maintaining a document to easily look up variable and function names [7]. However, search can be time consuming and frustrating when multiple results pop up for the same keyword [8]. BVI developers have to review code statements multiple times to verify they are at the right line [8]. As a workaround, they may leave comments to bookmark interesting locations in the code [8].

Another common strategy is to jump between function signatures [13, 15]. Audio-based plugins are especially helpful in non-linear navigation. StructJumper provided a hierarchical tree view of the source code's nested structure to facilitate skimming and non-linear navigation [20]. Its evaluation showed that efficient navigation meant BVI developers did not have to remember too much of the code during code reading [20]. The success of hierarchical trees was extended to support navigation of larger software projects with several files [167, 203].

6.2.2 Factors Affecting Code Readability for Sighted Developers

Prior research suggests that readability depends on the following: (1) use of spacing to make blocks visually distinguishable and easily identifiable using indentation, vertical line breaks, and whitespaces (2) identifier names and their naming style (camel case vs. snake case), (3) line length [154] for source code and comments, and (4) text formatting. We discuss these below; table 6.1 summarizes the factors and their sub-factors.

6.2.2.1 Spacing

Indentation is one of the most widely used approaches for modifying code layout. Early evidence suggested that as program complexity increased, indentation improved program comprehension [197, 55]. Subsequent studies investigated the optimal amount of indentation that aided in readability without increasing typing effort. For instance, Miara *et al.* suggested using 2–4 spaces to indent code blocks in Pascal, with 2 spaces offering most readability across developers' experience levels. Furthermore, they found that an overly indented code made scanning difficult. Indentation also had diminishing returns in heavily nested code or when entities were separated by

blank lines [55]. While developers' opinions remain undecided between 2 vs. 4 spaces [24, 70], the latter gives the visual appearance of a tab character and may lead to inconsistent use of tabs and spaces during collaboration, causing breakdowns in programming languages such as Python.

Another way to improve source code navigation is through *segmenting*, *i.e.*, putting blanks lines between code blocks that are functionally not similar [51, 183, 216, 235]. While it has not been found to have a significant effect on program comprehension and recall [123] and developer opinion seems split on the topic [70], coding standards recommend the use of vertical space to delineate code blocks [229]. Furthermore, the approach is an alternative to more explicit form of coding such as marking the beginnings and ends of code blocks with explicit statements or comments, which makes the code longer and difficult to read [214].

Coding standards also recommend using whitespaces around operators to improve legibility at line level [229]. While they have not been reported to significantly improve readability [183], they are considered good coding practice [51].

6.2.2.2 Identifiers

Meaningful identifier names (e.g., variable names or function names) have been found to improve readability [216] whereas poor naming practices can increase developers' cognitive load [74]. Developers may not follow good naming practices due to differing opinions on what constitutes a good name [216], with novice developers more likely to use poor naming choices [178].

When it comes to identifiers, the word boundary style also matters. Sharif and Maletic investigated the effect of camel case and snake case on identifier names [190]. They found that participants took 13.5% longer to recognize camel case identifiers [190]. On the other hand, Binkley *et al.* [37] found that regardless of developer experience, camel casing led to higher accuracy for source code manipulation in Java and C. Their follow-up study found that beginners recalled camel cased identifiers better whereas experts recalled better with snaked case. However, there was no statistically significant difference in visual effort needed for both styles [36]. Furthermore, regardless of the word boundary, longer names took more time to be recognized [37].

6.2.2.3 Line Length

Readability also depends on line length. Long lines of code are more difficult to understand, much like long sentences. Most coding standards recommend limiting lines to 79 characters [229]. It allows sighted developers to open multiple editor windows side by side and avoid horizontally scrolling [70]. Some researchers have even recommended that programming languages should favor constructs that allow developers to write shorter lines of code, for example using pre and post increments (*e.g.* `i++`) instead of addition operations (*e.g.* `i = i + 1`) [51].

Coding standards such as PEP8 typically recommend a shorter line length of 72 characters for more free flowing text such as comments and docstrings, which are strings used to document functions and classes [229]. Comments are especially useful in large non-modular code [219]. Developers are encouraged to use comments sparingly and write them in simple language [204], while ideally writing code where the intent is apparent without the need for additional explanations [92].

6.2.2.4 Text Formatting

Readability is shaped by *legibility* of the displayed text, which comprises layout (discussed above) and text formatting characteristics. Good legibility is related to readers' spatial visual abilities [247]. Depending on one's visual acuity, one needs to modify formatting attributes such as font type, contrast, font size, etc. to maximize the legibility of readable text [247]. For instance, Baecker applied the principles of graphic design to C programs [17]. He relied on different font types, proportional character spacing, and color contrast to improve the parsing of complex statements and special symbols by 25%, as measured by performance on a comprehension test [17]. Similarly, Raymond explored the use of typography to enhance readability [177]. Code editors set the formatting characteristics to reasonable defaults and these can be personalized by sighted developers to their liking. Among the factors discussed above, visual formatting is least relevant to BVI developers.

Modern code editors offer syntax highlighting and auto indentation to help sighted developers in identifying areas of interest. Static analysis tools such as code linters flag departures from coding standards such as line length violations or poor indentation without having to run the code. Together, these features facilitate skimming and focused reading for sighted developers. But we know little about how BVI developers identify areas of interest and what helps them in focused reading. Our study attempts to address that gap.

6.3 Study Design

We conducted an exploratory qualitative study with 16 BVI developers to understand their preferences and perspectives on factors that impacted code readability.

6.3.1 Procedure and Stimulus

We obtained IRB approval from the university for our study. We recruited our participants through snowball sampling and online forums such as program-l, a mailing list primarily comprising BVI developers [1]. The eligibility criteria for participation were that developers should be 18 years or older, possess at least one year of experience programming with screen readers, and be able to

Table 6.1: Readability factors we considered in our study. #O1 and #O2 indicate the number of participants who chose option 1 and option 2 respectively for any factor/sub-factor combination. #O3 indicates participants who had no preference or proposed a third alternative. Last column is a sum of O1 – O3 and equals the total number of participants in our study

Factor	Sub-Factor	Code Type	Option 1 (O1)	# (O1)	Option 2 (O2)	# (O2)	# (O3)	Total
Spacing	Indentation	Nested Data Structures	Separate parentheses and key-value pairs	12	Match key-value pairs and parentheses	4	0	16
		Docstrings	Indent docstring arguments	4	Do not indent docstring arguments	9	3	16
	Segmenting	-	Use 2 blank lines to separate entities	4	Use single blank line to separate entities	12	0	16
	Whitespaces	Math Operators	Surround operators with whitespaces	10	Avoid whitespaces	3	3	16
		Slice Operators	Surround operators with whitespaces	10	Avoid whitespaces	5	1	16
Identifiers	Word Boundaries	-	Use snake case	2	Use camel case	10	4	16
	Length	-	Long variable names	13	Short variable names	0	3	16
	Intent of Use	-	Use consistent prefixes	2	Use consistent suffixes	12	2	16
Line Length	-	Function Calls	Render arguments on separate lines	8	Render arguments on same line	6	2	16
	-	Function Signatures	Render arguments on separate lines	10	Render arguments on same line	5	1	16
	-	Call Chains	Treat dot operator as a delimiter	14	Do not treat dot operator as a delimiter	1	1	16
	-	Binary Operations	Place line break before the operator	7	Place line break after the operator	4	5	16
	-	Comments	Split comments across lines	3	Do not split comments	12	1	16
	-	Imports	Place imports on different lines	7	Place imports same line	6	3	16
String Quotes	Quote Character	-	Use single quote	2	Use double quotes	12	2	16

communicate about code in spoken English. Since programming languages differ significantly in their code styling guidelines, we chose Python and JavaScript to examine BVI developers' code styling preferences. Our choice was informed by the immense and consistent popularity of both programming languages in the developer community [159, 206].

We circulated a questionnaire to screen participants who met our eligibility criteria. The questionnaire asked respondents to self-report their programming experience in Python and JavaScript on a scale of 1 to 5; 1 meant no experience and 5 meant expertise in the language. We selected respondents who reported an experience of 3 or higher. We received a total of 20 responses and conducted the study with 16 respondents. Since all recruited participants reported higher comfort with Python, we conducted the study entirely in Python. The questionnaire also collected details about participants' demographics, assistive technology use, and job role (see Table 6.2).

During the study, we presented participants with a markdown file that listed 15 code formatting rules based on the factors identified from existing research around source code readability (see section 6.2.2). For each rule, we provided two functionally equivalent Python code snippets, inspired by Santos and Gerosa's study design [70]. One version conformed to PEP8 standards [229] (*e.g.* indented code block, snake case for identifiers); the other option was either formatted based on the evidence from accessibility research (e.g., unindented code to minimize verbosity) [8] or the alternative considered in studies with sighted developers (*e.g.* camel case for identifiers) [190]. We randomized the order of rules and the order of options before each study session to mitigate learning effects across participants. Table 6.1 summarizes the rules and their breakdown across factors that affect readability. A sample markdown file is shown in Appendix for reference.

We asked participants to open the markdown in a code editor of their choice. Participants were told to read each rule and its options as they would naturally go through any code. For each rule, we asked them to share which option they preferred and why. The research coordinator asked follow up questions about how the options affected readability, navigation, and verbosity on screen readers. Participants had the choice of creating alternatives if they did not like either of the two options presented in the markdown. The study concluded with a semi-structured interview to elicit their perspectives about differences in code styling preferences with sighted developers and the workflows they followed to improve code readability during collaboration. We compensated each participant with a USD $60 gift card (or its equivalent in local currency) for their participation.

6.3.2 Participants

14 participants identified as men; 2 identified as women. Participants were between 18 – 38 years old. They were employed as backend developers, full stack developers, tech lead positions, or were pursuing a higher education degree in computer science or a related field. All participants relied

on screen readers to interact with digital devices; three participants reported using braille displays in the screening questionnaire. Specifically for the study, 14 participants used NVDA and 2 used JAWS (see Table 6.2).

6.3.3 Analysis

We transcribed the data collected from each study session. We first organized participants' comments on code snippets and their responses to the semi-structured interview into five themes: (1) readability, (2) ease of navigation, (3) typing effort, (4) collaboration, (5) programming tool and screen reader settings. The initial themes were identified through research team's weekly discussions and the analytical memos we wrote after every 3—4 study sessions [181]. Next, we used inductive coding [140] to develop sub-themes within each of them, followed by merging of certain themes. In the end, we ended with three high-level themes that form the results section of our paper.

6.4 Findings

In this section, we delve into the impact of these factors on two kinds of code reading: (1) focused reading and (2) skimming. Table 6.1 shows distribution of participants' preferences for the factors and their respective options. Participants' quotes are lightly edited for clarity. MP:rephrase

6.4.1 Impact of Line Length on Readability

We open our findings section by discussing how line length and type of code (e.g., function calls, library imports, comments, etc) shaped participants' code styling preferences.

6.4.1.1 Line Length

Participants preferred lengthy function calls, signatures, and chained statements to be split across multiple lines instead of single line (e.g., Option #1 in Listing 1). PEP8 recommends limiting line length to 79 characters unless teams prefer otherwise [229]. The character limit enables sighted developers to open files side by side without horizontally scrolling to read the overflowing text. While our finding is in agreement with PEP8's guideline, our participants' choices were driven by reasons of code comprehension. Screen readers are programmed to read out all the content on the line when the cursor reaches it. Participants shared a long and complex line of code, such as a function chain (e.g., Option #2 in Listing 1), was difficult to process when read out in one go. To avoid the continuous audio stream, they used the control-right and control-left arrows to read one

```
# Option 1: Treat dot operator as a delimiter
def example(session):
    result = (
        session.query(models.Customer.id)
        .filter(models.Customer.account_id == account_id)
        .order_by(models.Customer.id.asc())
        .all()
    )

# Option 2: Do not treat dot operator as a delimiter
def example(session):
    result = (session.query(models.Customer.id).filter(models.Customer.
    account_id == account_id).order_by(models.Customer.id.asc()).all())
```

Listing 1: Options presented to participants for call chains

word at a time. However, that proved to be too slow a reading pace. On the contrary, when code was split across multiple lines, the screen reader read smaller chunks. These were not only easier to process but also gave more control to participants. They could choose which chunk to pause on or skim past without listening to it entirely:

> *"So like if they are in the same line like, my mental process cannot process anything. So like this one, if it is split into multiple lines, I just read a part of the content bit by bit [reads Option #1 of Listing 1]. So this line is not too long so after reading it [...] And after processing, I can just move to the next line."* — P15

If the line included complex variable names, participants had to navigate through each character to verify the contents. Here again, chunking helped! Participants could get to complex-sounding arguments quickly by first down-arrowing to the chunk they were interested in and then using right and left arrows to verify the characters:

> *"I want to read this character by character. Probably I'll be a little bit more faster because I'm right in the starting of the line, and I don't need to find that word. Immediately I can start reading, right? From the first character. "* — P11

We noted that participants' preferences were mediated by the likelihood of code reuse. A few participants pointed out that function signatures could be kept on one line despite its length since one is unlikely to change it. P15 mentioned that keeping function definition on one line enabled him to *" just copy the line and paste it"*, which he could then populate with the arguments to invoke the function. Typing or copy-pasting the function call in multiple places helped memorize the function definition. The ability to easily recall the code meant they could skip past the signature, which in turn made them prioritize formatting choices that facilitated efficient navigation.

A few participants said that in addition to splitting a lengthy line, they also preferred using named arguments. P8 described his work as a game developer involved function overloads that had up to 15 similar sounding arguments, such as the X, Y, and Z coordinates to map the three dimensional sound. In such cases, splitting the code across lines was not enough to remember the order of arguments. Furthermore, IntelliSense, the code editor feature that displays documentation upon mouse hovers, was not fully accessible to BVI developers:

> *"You [sighted developers] all have a lot of cool stuff where you can highlight something with a mouse [...] That's not something we get as blind programmers. I think it's getting better now 'cause you can do it in VS Code. You can kind of highlight an argument and I think you can press F12, and it will tell you what it goes with. But still it's not the most intuitive thing [...] But I love named arguments, I really adore them!"* — P8

We also found tension between participants' desire to reduce navigation and splitting the code. For instance, a few participants proposed a third option of keeping 2—3 arguments per line instead of one argument on each line. It meant less typing effort compared to the formatted option as well fewer down arrow presses. P12 shared that the Eclipse IDE offered a way to wrap lines in a manner which is accessible to both screen reader users and GUI users:

> *"So in Eclipse, sometimes I've seen [...] a few of the function names, which have a lot of arguments, so they get intended in a way that they fit on the screen. So you might be having one argument in front of the function name, and then here we'll have a couple of arguments in the second line, then another three arguments in the third line that way. So yes, it provides better readability and better scalability."* — P12

A couple arguments on each line were short enough to process while one navigated downwards without adding vertical length to the code. Others shared that they would prefer a single argument on each line despite it requiring more arrow presses. This not only ensured consistency but also reduced the burden of having to remember that some lines could have multiple arguments, ultimately preventing the loss of information if one skimmed the code too fast. Participants felt that longer but consistent formatting positively shaped code comprehension when they revisited the code after a hiatus.

A few participants also recommended refactoring the code and making it more modular instead of longer function chains, emphasizing participants' desire for non-linear navigation. A more modular code enables developers to jump across functions, also reported by Albusays *et al*. [8]

6.4.1.2 Type of Code

Length of line interacted with type of code in determining participants' preferences. We included examples to account for different types of code statements: (1) function signatures or definitions (2) function calls (3) function chains (4) comments (5) import statements. Majority of the participants preferred separation for the former three (as discussed in the previous section) whereas the preferences were more divided for the latter two, shaped by the need for consistency, efficient navigation, and less typing.

Participants mentioned that comments were typically written in English without special syntax or characters. They were easier to comprehend even when they exceeded the recommended character length, with our example being 109 characters long:

> *"It's [comments] not that much sensitive that I need to read character by character. Whereas, if it is a code, syntax, right? That I need to read character by character. So that makes sense to logically break."* — P11

The preference is in contrast with PEP8's recommendation, which suggests limiting comments to 72 characters for ease of visual consumption [229]. Participants also mentioned that ideally comments should be written in plain English because its purpose is to explain the code. However, if a comment was fairly descriptive and listed *"2 or 3 different steps"* (P2), they would consider breaking them down.

Although we did not include an example, we followed up with participants about their views on inline comments. PEP8 recommends using inline comments sparingly as they can distract from code reading [229]. Only select participants said they relied on inline comments and limited them to *"two to five words"* (P6). Most participants preferred comments to be on their own line because it tended to interfere with code reading in two ways. First, when participants tried to jump to the end of the code, their screen reader focus got placed at the end of the comment instead. They had to use control-left arrow to go backwards from the comment until they reached the end of the code itself, wasting time in in-line navigation. Second, they were likely to completely miss the comment when skimming the code quickly.

Much like comments, we noted difference in opinions with regard to import statements due to three reasons (see Listing 2). First, the participants made a distinction between standard libraries and third-party libraries. Our example included standard Python libraries and a few participants said they were likely to *"group them together"* (P8) to *"get over them quicker with the down arrow"* (P1). On the other hand, third party libraries needed to be placed on their own individual lines because one was likely to import a submodule or rename the module:

> *"They can just import a specific module into the namespace. So then, you do* `from this import this`*, or, you know,* `import this as this`*"* — P2

```
# Option 1: Place imports on different lines
import os
import sys
import random
import json

# Option 2: Place imports on the same line
import os, sys, random, json
```

Listing 2: Options presented to participants for import statements

Second, editing concerns affected choices. A few participants felt that import statements were only typed once, mostly read once at the beginning of the code, and were unlikely to be modified again. Therefore, one could place multiple imports on a single line without compromising readability. Others felt that because imports were typed precisely once, they should in fact be separated out, ultimately affording more convenience if any library had to be removed or replaced:

> *"if it is one library per line, so that if you just want to remove one of the library, it is more easier."* — P15

Third, participants' programming experience with other languages had a bearing on their opinions. For instance, P12 recalled that JAVA only permitted placing imports on separate lines. He chose the same option to stay consistent in our study despite describing the practice as a *"headache"* (P12).

6.4.2 Impact of Programming Environment on Readability

We now elaborate on the effect of screen reader settings such as punctuation settings and synthesizer choice on identifiers and quotation characters. We also describe how screen reader settings interact with code editor features.

```
# Option 1: Long names
radioButtonHeight = "20"

# Option 2: Short names
radioBtnHt = "20"
```

Listing 3: Options presented to participants for identifier length

The perceived *verbosity* of code had a bearing on participants' styling preferences. For instance, certain naming choices required listening to more audio output and slowed down participants.

Consider the options we presented to evaluate preferences for identifier length (see Listing 3). Sighted developers are likely to read both options as "radio button height". On the contrary, for our participants, the second option was announced as "radio B T N H T" – a more verbose output despite being fewer characters to type:

> *"So would you believe that even though option 2 is shorter, it's actually longer on the screen reader. Yeah, it's more syllables. Ain't that crazy! Because 'radioButtonHeight' is 5. But it's more characters, whereas 'radioBtnHt' [...] is actually 8."* — P8

Participants shared that a verbose name was harder to remember. Furthermore, they may confuse the output with similar sounding alphabets while skimming. The name may also be mispronounced by differences in capitalization or due to synthesizer choice:

> *"`API` is capital A, capital P, capital I. It's not a word but people try to use it as a word, so what they do is 'capital A, small P, small I' (Api). Then it will not read as API, that's when I get confused."* — P11

> *"I had one or two instances, where it will just call out something else. For example, my screen reader will often call out 'capital A, capital S' (AS) as American Samoa."* — P12

A funny instance of screen reader mispronunciation was when function signature arguments were rendered on separate lines (see Rule #3.0.2 in Appendix). The signature's closing parentheses and colon '`):`' ended up on a separate line. P12 chuckled when it was announced as "sad face". Such differences made the seemingly shorter option more verbose, harder to remember, and could introduce errors in the code. To avoid these issues, participants had to slow down their navigation and clarify the spelling by reading the variable *"character by character"* (P11).

We had included examples of names that encoded the context of use in either the suffix (e.g, `foregroundColorMenu` or the prefix `menuForegroundColor`) of the identifier. Majority of the participants preferred context to be announced first (e.g, `menuForegroundColor`, `footerForegroundColor`) to reduce verbosity associated with long names during code skimming. A few participants pointed out that they would prefer `foregroundColorMenu` only if the code also contained counterparts such as `backgroundColorMenu`. They felt it would be more useful to glean the global relationship between identifier categories before learning about the specific UI elements they were responsible for. Participants' comments are reminiscent of Hungarian notation [200] and suggest a preference for quick navigation with lower verbosity.

Verbosity was also determined by the screen reader's punctuation setting. As shown in Table 6.2, 8 participants had set their punctuation settings to *all*, 5 had set it to *most*, and 3 had

set it to *some*. *All* announced every punctuation character but meant greater verbosity, which interfered with reading and processing. On the other hand, *most* or *some* was likely to skip over important characters. The setting had a strong bearing on whether to use camel case or snake case. PEP8 recommends snake case *i.e.*, underscores to separate words in variable names (e.g., `primary_address_apartment`) [229]. However, if participants' screen reader punctuation setting was set to *most* or *all*, it was announced as "primary line address line apartment" on NVDA (JAWS announces underscore "underline"). On the other hand, when punctuation was set to *some*, the announcements were same for both options (`primary address apartment`) but participants had to go through the identifier to ensure the presence of the underscore character. The verification once again meant the slower character-level navigation that interrupted skimming. Therefore, participants spoke of using camel case even if their colleagues preferred snake case:

> *"I prefer Option 1 (camel case) because, A, it's shorter and it reads fine, and B, I am used to Go, and that's part of their style [...] I like CamelCase for my Python variables. I've convinced my colleagues not to judge me for it"* — P7

```
# Option 1: Place line break after the operator
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)

# Option 2: Place line break before the operator
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

Listing 4: Options presented to participants to understand line break preferences

The choice of punctuation setting could also skip information relevant for code comprehension. For instance, we asked participants where they would like to insert line breaks in long lines — split the line after the operator or before the operator (see Listing 4). Operators placed at the beginning of the line were announced regardless of one's punctuation setting. However, a less granular punctuation setting did not announce operators placed at the end of the line:

> *"If you put the dot at the end, it will not announce,* `filter dot`*. It will just announce* `filter`*. Because for JAWS, it's a full stop."* — P11 (punctuation set to *most*)

> *"It doesn't read the dash on* `dividends - qualified_dividends` *- So it's not reading the dashes but that's my punctuation settings. That's my own fault."* — P8 (punctuation set to *some*)

The above quotes reveal how the dot and the subtraction (announced as dash) operators are treated as if they are being used in a text document and not in a coding environment. P10 reasoned that characters like dot and dash are *"used for many purposes"*. For instance, he shared that not putting whitespaces around the dash operator also mutes its announcement, possibly because it implies a range (e.g., 15-10). Taking into account all of these scenarios is difficult and screen reader developers might have felt that *"not announcing them would make sense"* (P10) in *some* and *most* settings.

Lastly, single quote ('tick' on NVDA; 'apostrophe' on JAWS) was only announced when punctuation was set to *all*; double quote ('quote' on both NVDA and JAWS) was announced for *most* and *all* settings. We noted a strong preference for double quotes among participants because it required less disambiguation and was more likely to be announced. For instance, in the docstring example (Rule #1.1.2 in Appendix), the screen reader did not announce the single quotes to participants who had not set their punctuation to *all*. They had to do character-level navigation to verify whether the line was indeed blank or it had characters relevant to code reading:

> *"I was sure something is there, but I couldn't read and I tried to go back. Then I understood there is an apostrophe, like single quotes [...] If it is not saying blank, there is something but it is not readable [to the screen reader]"* — P11

6.4.3 Impact of Navigation on Readability

Our study showed that there are 5 levels of navigation that allowed participants to skim and read the code in detail: (1) character-level, (2) word-level, (3) line-level, (4) entity-level, (5) editor's search feature. Prior work has investigated and designed tools to improve non-linear navigation *i.e.*, the latter two [20, 167, 203]. We are the first study to describe how the first three shaped readability.

6.4.3.1 Character-level navigation

The previous sections discussed how the lack of punctuation information or too much verbosity meant participants had to parse each character of a line to verify details such as spelling and use of special characters. Presence of whitespaces further slowed down participants by increasing the total characters they had to navigate. For the very reason, majority of our participants preferred tabs over spaces to indent code blocks in Python. They could *"go over a tab with just one press"* (P1)

while spaces were four characters. Besides, whitespaces introduced verbosity at character-level navigation:

> *"I usually don't put spaces, because I think that kind of makes it more time consuming. It's going to keep saying 'space space and space' whatever."* — P2

However, participants agreed that whitespaces facilitated better word-level navigation (discussed next). A few participants mentioned that presence of whitespaces prevented over-editing or accidentally deleting characters by acting like buffer. Furthermore, whitespaces around mathematical operators improved the readability for their sighted colleagues, which they prioritized by either using whitespaces while authoring code or reformatting the code using a code formatter according to coding standards.

```
# Option 1: Use whitespaces
ham[lower : upper + offset]

# Option 2: Avoid whitespaces
ham[lower:upper+offset]
```

Listing 5: Options presented to understand use of whitespaces

6.4.3.2 Word-level navigation

Participants shared that presence of whitespaces tended to improve word navigation by acting as *"word boundaries"* (P1). Some participants also shared that statements comprising slice operations were *"read slowly because of the spaces"* (P12) (see Listing 5). However, whitespaces could cause tensions with one's punctuation setting. For instance, with whitespaces present and the punctuation set to *some*, the screen reader did not announce the colon character. Thus, the operation performed in the statement was not communicated. But without the whitespaces, the colon ended up acting as the word boundary and was output by the screen reader, enabling participants to understand the operation without having to resort to character-level navigation:

> *"With my [punctuation] setting, if there's no space between the colon and the words, it is reading the colon as well the plus sign. So it's reading the entire thing properly. But in the first one, it's not announcing the colon symbol in 'some' setting, and it's treating it as a pause. 'Lower', then a pause, then 'upper'."* — P10

We noticed similar tension when snake case was used for variable names. Underscores acted as word boundaries and allowed participants to jump across individual words despite increasing

typing effort and verbosity (when punctuation was set to 'most' or 'all'). P1 shared how he had started preferring camel case once he discovered an NVDA addon that enabled navigation just like snake case did:

> *"3–4 years ago someone made an NVDA addon called WordNav, which stops control arrows even in camel case. So in a word, it's not like you navigate with control-right/left arrows. With this addon, it stops after the first and second word even though they are not separated by anything"* — P1

In conclusion, while whitespaces made character-level navigation and typing slower, they improved word-level navigation by acting as boundaries between words. Punctuation and special characters could also act as boundaries but it depended on one's punctuation setting.

6.4.3.3 Line-level navigation

We have already discussed how participants were able to pause at will when lengthy code lines were split. By down arrowing through code chunks, they were able to process the code better and avoid word-level or even character-level navigation. In this section, we discuss how the use of indentation and line breaks shaped overall navigation and code skimming.

NVDA allows four options for indentation reporting: (1) none, (2) tones where higher pitch implies greater indentation (3) speech (e.g., "twelve space" or "four tab"), (3) both speech and tones[1]. 5 participants did not use any indent reporting whereas 13 participants had it turned on (see Table 6.2). We noted that the choice of setting influenced participants' presentation choices for nested dictionaries but not so much for docstrings. Typically, participants used indentation reporting to *"visualize where the things are, how far in they are"* (P7). Thus, option 1 was more preferable for Rule #1.1.1. They could down arrow to key-value pairs at the same nested levels and navigate past heavily nested items using the audio cues:

> *If the indentation is consistent, I could just skip past, like let's say there's a list in here. If I don't need that, I can just skip past that to the next block.* — P2

Participants who did not use indentation reporting were divided in their preferences. They compared the effort it took to write well-indented code with the improvements to readability. Usually, they wrote the code without indentation and formatted it later for the benefit of sighted developers. They felt the lack of announcements led to *"a lot of confusion when dealing with more nested structures"* (P14) but keeping it turned on interfered with other aspects of their work such

[1]Only P11 and P14 used JAWS in our study. Both did not use indent reporting. They and a few other participants mentioned that JAWS does not indent reporting.

as emails, document writing, etc. However, even without indent announcement, a few people preferred option 1 for nested data structures. The placement of parentheses on its own line clearly indicated the beginnings and ends of a nested level. P14 said she left small inline comments after each closing brace to serve as checkpoints. These helped her keep track of nested structures and helped her skim faster. Furthermore, the key bindings in code editors helped participants to jump quickly to opening or closing parentheses. Some participants used addons like IndentNav, which allowed skipping to statements that shared the same nesting level. In Python, it could be used to jump across entities, conditionals, and loops:

> *"NVDA has this add on called IndentNav, which basically just lets me navigate past code blocks. So sometimes when I'm skimming and if a block does something and I know what it does, I don't need to go in there, I'll just skip past the indentation. Go to the next block or whatever, skip past the loop and stuff like that. "*

Majority of the participants preferred no indentation in multiline docstrings irrespective of indent reporting. Since docstrings were similar in nature to comments, they were likely to be read only a handful of times. They preferred going through them quickly to get to the main body of the code. Even while writing docstrings, participants preferred spending as little time as possible in formatting the text compared to other aspects of source code. P14 mentioned using the autoDocstring plugin for VS Code, which provided placeholders for populating details about a class or function. The plugin not only ensured correct formatting but also saved her writing time.

6.5 Discussion

Prior research has focused on communicating the information encoded in visual markup to code such as syntax highlighting, code structure, etc, through audio cues and plugins respectively. Our research detaches the source code text from its visual appearance and formatting. Our findings reveal that while it is vital to translate the information available in visual markup of code, the source code itself is not fully available to screen reader users. Put otherwise, the WYSIWYG paradigm holds for sighted developers, but the screen reader output does not fully map to the on-screen text for BVI developers. In this section, we update the table from our related work to move towards an inclusive taxonomy for code readability (see Table 6.3). We also make recommendations for programming tools and code styling guidelines.

6.5.1 Moving Towards an Inclusive Taxonomy for Code Readability

6.5.1.1 Spacing

Our finding contradicts past finding on nested code structures [8]. We find that participants used indentation for navigation, which was further improved through the use of screen reader addons. We further find that separating parentheses (Option 1 of Rule # 1.1.1) instead of grouping multiple parentheses together (Option 2 of Rule # 1.1.1) is more useful in jumping nested code blocks. Lastly, we find that tabs are better than spaces for indentation because they minimize character-level navigation.

When it comes to vertical spacing or segmentation, PEP8 recommends keeping 2 blank lines between entities. While a few participants preferred 2 blank lines to delineate between code blocks, most preferred a single line to reduce linear navigation. We believe the choice of amount of vertical spacing can be left up to BVI developers. Code editors could provide shortcuts to reduce blank lines if they detect screen readers to facilitate efficient line-level navigation.

Lack of whitespaces around mathematical operators makes the code less readable for sighted developers. For BVI developers, the statement may get read without discernible pauses without whitespaces. Lack of whitespaces places the screen reader cursor at the end of the line. Thus, surrounding operators with whitespaces is useful for both groups albeit for different reasons. Code editors could provide mechanisms to reformat selected group of statements to reduce the typing effort for BVI developers, which they described as the primary that deters them from using whitespaces while authoring code.

6.5.1.2 Line Length

Splitting long lines (e.g., function chains, function signatures, etc.) helps both sighted and BVI developers. Sighted developers do not need to horizontally scroll; BVI developers have to process smaller chunks as they read the code. It also improves their navigation experience. They need not listen to the entire line before moving to the following line. It is worth pointing out that sighted developers can toggle on word wrapping, which prevents horizontal scrolling. However, word wrapping produces no effect on BVI developers. In fact, the feature is disabled in IDEs like VS Code if it detects the screen reader []. Either IDEs should enable an equivalent audio wrapping for screen readers, or they offer settings to enforce code splitting consistently.

6.5.1.3 Identifiers

Prior research is divided on the usage of snake versus camel case for sighted developers [190, 37]. PEP8 recommends snake case for variables. But an overwhelming majority of our participants

preferred camel case over snake case for verbosity reasons. We also show that developers ought to consider the syllable count when shortening variable names (*e.g.* `button` instead of `btn`; `checkbox` instead of `chkBx`) to avoid verbose output on screen readers. Lastly, developers are encouraged to create meaningful variable names by encoding the intent of use. In such cases, sighted developers should consider where to place the word representing the context (e.g, `menuColorForeground` vs. `foregroundColorMenu`) to ensure ease of remembrance and code skimming. These decisions can depend on the categories of variables (e.g., foreground colors, background colors, etc), the total number of variables in the code, and the number of words composing the identifier name.

6.5.1.4 Granularity of Navigation

We add to the prior empirical studies on code navigation with screen readers [8, 20, 175]. We find that high verbosity and ambiguous announcement of special characters forces people to perform word level and character level navigation. These are slower forms of navigation, which impacts the activity of code reading. Ideally, the lexicon and the layout of the code should be such that it can be understood by linear level navigation. This means that lines should be chunked such that they are easy to process while reading and easy to recall while navigating backwards. The findings have implications for programming language design. For instance, using complex and verbose keywords can force people to stop skimming and look at the line more closely.

6.5.1.5 Programming Environment and Screen Reader Settings

The manner in which code is written interacts with screen reader settings and affects output. Consider the example where we asked participants whether they prefer inserting line breaks before or after the binary operator. We found that developers were likely to miss operators at the end of lines when skimming too fast or if the punctuation setting was set to *most* or *some*. Similarly, screen readers did not announce single quotes in less granular punctuation settings; using double quotes to quote string variables and docstrings was better. Lastly, collaborators may capitalize names differently (e.g., `API` vs. `Api`), changing the pronunciation entirely on screen readers. These differences do not affect sighted developers – a quote character is read and interpreted as a quote, missing operators are easy to catch, and `API` and `Api` are visually processed the same way. While BVI developers pick up on the code styling preferences of sighted developers easily, sighted developers do not reciprocate similar awareness. We recommend incorporating the readability preferences of BVI developers in code styling guidelines, such as PEP8 [229]. For instance, the above examples can be used to educate the larger programming community about how screen readers may announce different code snippets.

Syntax highlighting helps sighted developers identify regions of interest [199]. Researchers have attempted to use audio cues to communicate the visual cues available to sighted developers in code editors. However, audio cues take time to memorize [104]. We find that audio cues for indent reporting also interfere with tasks of emailing, documeting editing, etc for BVI developers. Our findings also show that the type of code matters. Everyone preferred splitting call chains, the opinion was divided on breaking function signatures, and long imports and comments were least less likely to affect readability. These findings help us decide what programming constructs should be highlighted using audio.

Lastly, we find that code editor plugins and screen reader addons can greatly reduce typing effort while improving readability. For instance, P14 was among the few participants who did not mind indenting docstrings because she used the Autodocstring plugin. Similarly, participants who used addons like IndentNav could achieve more efficient non-linear navigation. IDEs like VS Code were more popular because of their accessibility features and ability to apply consistent indentation, parentheses, and quoting. Such plugins and features improved readability *as* one wrote the code and *not* after the fact by requiring the use of code formatters. We recommend that practitioners and teams building code editors should explore ways to extend the programming environment. The work can be abstracted out at several levels. For examples, JAWS currently does not support indent reporting but IDEs could offer indent reporting through their plugins. IDEs could also enable quick toggles between styles that work well for collaboration and styles that are more effective at an individual level.

Table 6.2: Participants' Demographic details and environment (code editor and screen reader) they used in the code readability study. The first column lists gender and age in brackets (e.g., P1 is 32 years old and identifies as a man)

#	Job Role	Prog. Experience	Region	Screen Reader	Punctuation Setting	Indent Reporting	Code Editor
P1 (32M)	Backend Developer	10–14 years	Europe	NVDA	All	None	Notepad++
P2 (18M)	Student	5–9 years	Canada	NVDA	All	Speech	Notepad2
P3 (20M)	Student	1–4 years	India	NVDA	Most	Speech	Notepad
P4 (23M)	Game Developer	1–4 years	Pakistan	NVDA	All	Speech + Tones	VS Code
P5 (32M)	Backend Developer	20–24 years	UK	NVDA	All	None	VS Code
P6 (26M)	Backend Developer	1–4 years	India	NVDA	Most	Tones	Notepad
P7 (34M)	Backend Developer	10–14 years	South Africa	NVDA	Most	Speech	Notepad++
P8 (38M)	Game Developer	20–24 years	USA	NVDA	Some	Tones	VS Code
P9 (28M)	Full Stack Developer	10–14 years	India	NVDA	All	Speech	VS Code
P10 (18M)	Student	5–9 years	India	NVDA	Some	Speech	VS Code
P11 (24M)	Backend Developer	5–9 years	India	JAWS	Most	N/A	VS Code
P12 (29M)	Tech Lead	10–14 years	Canada	NVDA	Some	None	Notepad++
P13 (25F)	Student	1–4 years	Germany	NVDA	All	Tones	Notepad++
P14 (31F)	Data Scientist	10–14 years	USA	JAWS	Most	N/A	VS Code
P15 (37M)	Researcher, Hobbyist Programmer	15–19 years	China	NVDA	All	Speech	Notepad++
P16 (21M)	Student	1–4 years	Germany	NVDA	All	Tones	Notepad

Factor	Sub-Factor	Code Type	GUI			Screen Readers			
			Recommended Practice	Skimming	Focused Reading	Recommended Practice	Non-Linear Skimming	Linear Skimming	Focused Reading
Spacing	Indentation	Nested Data Structures	Follow consistent indenting	Yes	No	Follow consistent indenting	Yes (with addons)	Yes (with indent reporting)	No
		Docstrings	Indent docstring arguments	Yes	No	Do not indent docstring arguments	N/A	Yes	No
	Separate Closing Parentheses	-	Separate parentheses and key-value pairs	N/A	N/A	Separate parentheses and key-value pairs	Yes	Yes	Yes
	Segmenting	-	Use 2 blank lines to separate entities	Yes	No	Use single blank line to separate entities	N/A	Yes	No
	Whitespaces	Slice and Math Operators	Surround operators with whitespaces	N/A	Yes	Surround operators with whitespaces	N/A	Yes	Yes
Identifiers	Word Boundaries	-	Use snake case	N/A	Same effect	Use camel case	N/A	N/A	Yes
	Length	-	Short variable names	N/A	Yes	Consider syllable count	N/A	Yes	Yes
	Intent of Use	-	Convey intent in either prefix or suffix	N/A	Yes	Use consistent suffixes	N/A	Yes	Yes
Line Length	-	Function Calls, Signatures, Chains	Render arguments on separate lines	Yes	Yes	Render arguments on separate lines	N/A	Yes	Yes
	-	Binary Operations	Place line break before the operator	Yes	Yes	Place line break before the operator or same line	N/A	Yes	Yes
	-	Comments	Wrap comments	Yes	Yes	Do not split comments	N/A	Yes	Yes
	-	Imports	Place imports on different lines	Yes	Yes	Either is fine	N/A	Yes	Yes
String Quotes	Quote Character	-	Use quotes consistently	N/A	N/A	Use double quotes	N/A	Yes	Yes

Table 6.3: Taxonomy for Code Readability on GUIs and Screen Readers

6.5.2 Limitations and Future Work

We studied participants' preferences for Python, which strictly enforces indentation for code execution unlike other languages. Python is also known to be closer to English, with some calling it executable pseudocode [69]. Its seemingly natural language design and indentation enforcement may have led to preferences that may not generalize to other programming languages. In future work, we would contrast our results with languages closer to C-style syntax that have been reported to present barriers to novice programmers [210].

The remote nature of our study prevented us from observing code reading on braille displays. Only three participants reported using braille displays but they did not use them during the study. In future work, we would examine the factors that constitute code readability on braille displays and pin-matrix tactile displays that even display 2D graphics [42].

Despite our efforts, our study sample was heavily skewed towards men, likely due to the lack of equitable gender representation in the software engineering field [158, 159]. Its effect is amplified for BVI women and non-binary developers, who are also marginalized due to ableism and accessibility barriers.

6.6 Conclusion

Code editors and IDEs provide features such as syntax highlighting, vertical rulers, etc., to support code skimming and focused reading among sighted developers. However, we do not know what constitutes good code readability for BVI developers. We conducted an exploratory qualitative study with 16 BVI developers. We presented them with two differently formatted options for 15 functionally equivalent Python code snippets and asked them to choose the option that improved code readability for them. The snippets were created to investigate the effect of indentation, line length, identifier names, and quotation characters. We found similarities and differences in how these factors shaped the readability of BVI and sighted developers. Based on the findings, we contribute an inclusive taxonomy for code readability that considers code reading on GUIs and screen readers.

CHAPTER 7

Discussion

My thesis stated that the **interplay between programming and collaboration tools, assistive technologies (ATs), programming practices, and the organizational norms around communication and help-seeking shape the collaborative experiences of BVI developers in mixed-ability contexts; we must design tools to not only improve accessibility but also to minimize the various forms of additional work BVI developers perform to achieve effective collaboration with sighted colleagues.** In this section, I return to my overall findings to show the different forms the work takes and discuss approaches to reducing them, proving my thesis.

7.1 The Work Behind Collaborator Awareness

7.1.1 Articulation Work

Chapter 3 revealed the importance of articulation work in mixed-ability workplaces. We found that BVI developers have to explain their access needs to their colleagues to modify the existing arrangements around collaboration. Articulation is a slow and repetitive process, especially with colleagues who are new or unfamiliar with accessibility. However, articulation work can be lowered by building more awareness among sighted developers. Chapter 5 recommends centering accessibility in the official documentation of programming tools and frameworks as one of the ways to educate sighted developers. Most software and frameworks dedicate only a single page to accessibility in their documentation, which sighted people tend to miss in their regular use of the documentation. Surfacing accessibility details in other pages, such as the compatibility with ATs and performance of UI components on different screen readers, would enable mixed-ability engineering teams to choose more accessible technology stacks. The popular pages are likely to be read by majority of developers. Furthermore, a greater emphasis on accessibility also places it on par with other critical programming topics such as performance, security, and UX.

I have started initial investigations in this area of work by blending accessibility mentions in high-traffic pages such as the onboarding tutorial of a UI framework. The results are promising

and suggest that we can build awareness about the needs of BVI developers and users through repeated but subtle references to accessibility [161]. In subsequent studies, I would investigate how we can flag the more specific accessibility issues through developer tooling such as code linters to educate developers in the context of their programming environment (see section 7.4)

7.1.2 Coordination Work

Chapter 4 showed the necessity of coordination work during tightly-coupled synchronous activities such as code walkthroughs. An intervention such as CodeWalk can reduce the need for explicit coordination among collaborators. Others have made similar arguments in the context of collaborative document writing. Lee *et al.* developed CollabA11y, a Google Docs extension to make collaborator awareness accessible during collaboration. They also suggested prompting sighted collaborators to summarize their edits to the document to communicate overall changes. Das *et al.*'s Co11ab examined the back and forth between individual and collaborators' edits. Chapter 4 has discussed the differences between CodeWalk, Co11ab, and CollabA11y. In the next paragraph, I comment on the similarities between the three and suggest treating them as an ecosystem during the design process.

The growing research around the accessibility of collaborator awareness in different contexts prompts thinking about the learnability of various audio cues, speech announcements, and spatial audio mapping for the end-users. While the information metaphors for WIMP GUIs are fairly standardized at this point (windows, menus, and icons), the same cannot be said for screen readers and audio interfaces. With standardization, the user has to learn, unlearn, and relearn the meaning of auditory information for each context. Chapter 6 has shown that BVI developers are hesitant to use readily available settings such as indent reporting because it presents a learning curve. We need to identify ways to not only minimize the learning curve of auditory output within the context of programming but also in applications that get used in parallel. For example, the audio cues mapped to scrolling can remain consistent in IDEs, text documents, and communication software; the keyboard shortcuts for Follow mode can be the same in code walkthroughs and collaborative writing; voice fonts associated with a sighted developer can stay consistent across text documents, code walkthroughs. Through concerted research and design efforts, we can reduce coordination work across the suite of activities that developers perform in mixed-ability contexts.

7.1.3 Setup Work

Chapters 3 and 5 also illustrate the work behind finding, installing, and configuring accessible tools and solutions, known as the setup work. Shinohara *et al.* advised grounding the design of accessibility tools in the mainstream [193]. Doing so prevents foregrounding the disability of BVI

collaborators and also helps reduce setup work. There are two ways to reduce costs associated with setup. The first is akin to the launch of CodeWalk. We released CodeWalk as a set of features within VS Code LiveShare, thereby integrating it into a popular collaborative tool for code walkthroughs. It ensured that none of the developers had to install another software to participate in the collaboration.

The second approach is to keep the code editors extensible through plugins and add-ons. Examples of IDE plugins include StructJumper [20] and SODBeans [209]; Co11ab [64] and CollabA11y [118] are examples of plugins for collaborative writing. However, it is critical to ensure that these plugins are maintained and updated to work well with screen readers since prior work, including Chapter 4, has reported that accessibility features tend to lag behind the features intended for sighted users [194].

7.2 Translating WYSIWYG Paradigm to Audio Medium

The level of indentation, font color, or font size a sighted user sets in a GUI editor is exactly what she gets in the output! But when the aforementioned visual information is translated to the audio medium, it does not follow the WYSIWYG paradigm [63, 119]. However, through my dissertation, I argue that the textual content itself may not be fully available on the screen reader owing to the settings of the assistive technology, programming environment, and the larger operating system. Thus, the screen reader output may not contain all the text available readily to the sighted person. For example, the screen reader may announce `diff = 15-10` as `diff equals fifteen ten`. The lack of spaces around the dash operator may suggest that `15-10` is a range and not a subtraction operation.

Das *et al.* argued that we "must attend to not only the availability of information but the effort and time required to access that information" [63]. I recommend taking a step back. **We must first attend to the completeness and accuracy of the available information**. We need to bear in mind that BVI users forego complete textual content by choice to avoid dealing with too much verbosity. I argue that designers and developers ought to consider the following three questions to ensure the accessibility of information on GUIs and screen readers:

1. **What kind of lexicon and characters would the user deal with during the activity?** For example, writing a text document in English is far less likely to deal with complex symbols. Code authoring would involve colons, dunders (double underscores), asterisks, etc. A document involving algebraic equations may comprise even more complex symbols such as square roots, exponents, etc. Chapter 6 showed that screen readers are not great at differentiating between these activities on their own. They currently allow users to create configurations for different applications [153], which can still require significant setup work given the

number of programming and collaborative tools a BVI developer interacts with on a daily basis. Furthermore, Chapter 6 showed that not all participants are willing to create configuration profiles. The alternative is to go in change punctuation and indentation settings each time but that is inefficient given the number of context switching between source code, text documents, emails, and colleagues' messages. Researchers and practitioners can consider ways to offer some of the critical information via the tools intended for the activity instead of relying on the screen reader to provide that information or expecting the user to acquire it on their own.

2. **What is the severity of missing or inaccurate information across activities?** Missing the dot operator (period character), mathematical operators (`+`, `-`, `\%`, etc), or semi colons during code reading can affect debugging and can cause breakdowns in the code. But missing the dot operator or semi colons in collaborative editing will not crash the document. On the contrary, they are perfectly acceptable as characters that should serve to pause the speech during document authoring. Developers are often performing a mix of activities during collaboration. For instance, Chapter 3 reported that collaborators exchanged code snippets via email and Slack. During such communication, it is important to utilize the AT settings specific to both the code editor as well as communication software. Visual interfaces typically permit such customization easily. A sighted person can represent a code snippet using different fonts (e.g., Courier New) or tag it as a block of code in Slack, thereby delineating it from the rest of the email or chat message. But the effect of such customization is unavailable on screen readers. The interconnected nature of activities highlights the need to design for the ecosystem, as opposed to one individual tool at a time.

3. **Who are the collaborators involved in the activity?** Collaborative programming requires writing code that meets the standards adopted by the group whereas one can set standards to boost individual productivity during personal projects. Similarly, participants' comments from Chapter 5 show that legacy code written by sighted developers was difficult to adapt to their readability preferences. One can also imagine that projects driven by BVI developers may not meet the readability needs of sighted developers. It is imperative to be able to switch between various standards easily as well as have the option to customize and set one's own standards.

The above questions can help product designers and developers identify the granularity at which they should translate textual content from the GUI to the screen reader and what aspects of the text should be communicated via pauses, audio cues, or speech itself. I once again emphasize the delineation between the content and its visual markup. Past research has largely focused on

communicating the latter through prosodic elements, concurrent speech, audio cues, and voice fonts [63, 78, 119, 172].

7.3 Long Term Implications for Research Space

The dissertation opened with the four broad areas that need to be considered to improve accessibility of collaborative programming as a whole. Across the four studies, I argue that the research community in HCI, accessibility, and software engineering needs to take a two-pronged approach toward improving accessibility in the long run. First, the software that cater to the four areas should continue to be improved. Chapter 2 lists the prior efforts to make programming tools more accessible; the dissertation recommends new ways to improve collaborative tools and their interplay with the software ecosystem.

Second, we need to redistribute the work behind accessibility by educating and informing sighted collaborators about the access needs of BVI developers. As stated in chapters 3 and 5, research in this domain can look at building more awareness not only among developers but also in organizations. Currently, much of the effort is targeted at educating sighted developers about access needs of BVI end users and not so much the access needs of BVI collaborators. We see concerted efforts across universities in the USA to teach accessibility as part of computer science and design courses [3, 113, 232]. Faculty members are recognizing that integrating inclusive guidelines and accessibility principles as part of coursework minimizes the chances of future generations of developers engineering glaringly inaccessible applications [57, 126]. Other examples include tools to educate sighted developers about UI themes for low-vision and colorblind users [97]. However, neither the coursework nor the alternative resources impart lessons on how to be a good collaborator in a mixed-ability team. Furthermore, the lack of mention of BVI developers reinforces the notion that sighted developers are the sole group of engineers responsible for creating accessible applications for everyone. Going forward, the research community needs to look at teaching sighted developers on how to adapt and advocate on behalf of their BVI colleagues. The next section describes the extensions to the dissertation, bearing in mind the long term implications for accessibility of collaborative programming.

7.4 Future Work

7.4.1 Inclusive Static Analysis Tools

Static analysis tools, like code linters, flag poor code writing practices and ensure code quality in the long run. Many developers integrate linters in the project life cycle from the start [95],

enabling good code writing practices to become a part of the development process early on, also known as the 'shift left' approach. In future work, I want to investigate how we can design these tools to flag code writing that can comprise the readability preferences of BVI developers (see §6). Furthermore, there are efforts to highlight glaring GUI accessibility issues using code linters. For instance, Android Lint [67] provides warnings for the following: (1) missing labels for UI elements, (2) smaller touch target size, (3) poor color contrast (see Figure 7.1). Besides the aforementioned issues, XCode's Accessibility Inspector [12][1] offers additional support to check if the UI controls would be announced in the right order on screen readers. Chapter 5 showed that sighted developers may not be aware of the impact of these accessibility issues on the workflows of their BVI colleagues. Thus, we should also use static analysis tools to call attention to poor GUI accessibility.

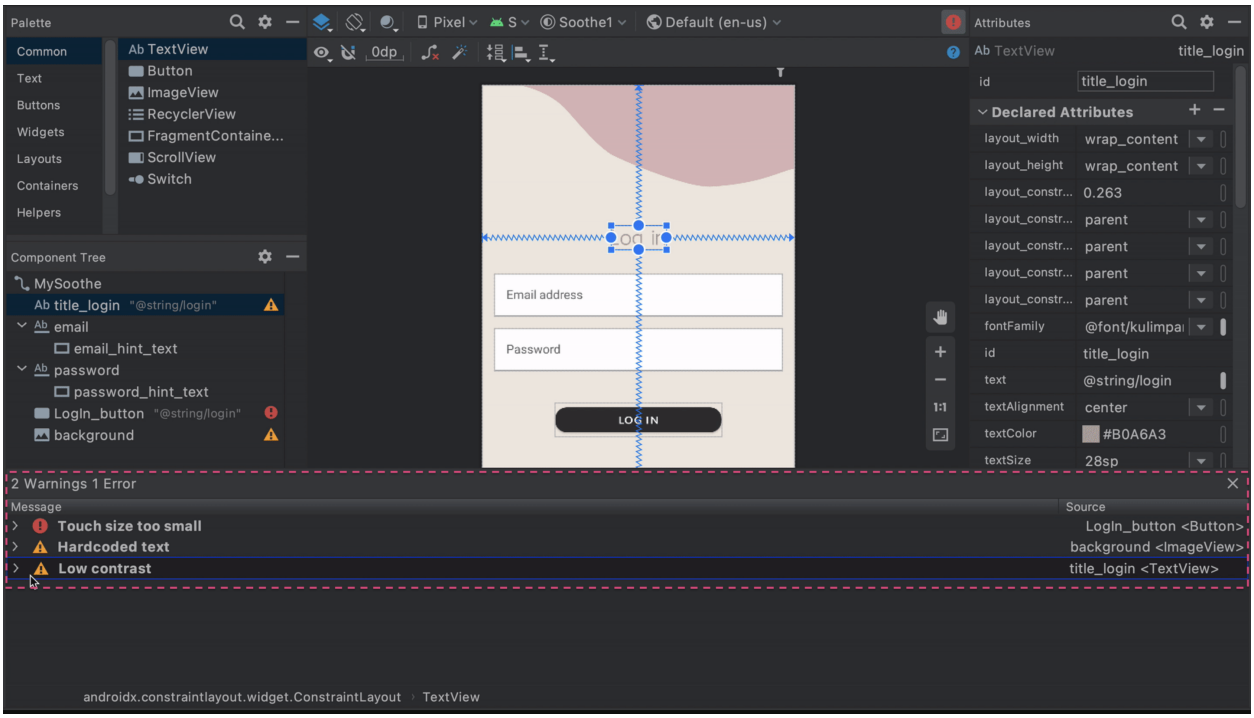


Figure 7.1: Accessibility warnings presented by Android Lint in Android Studio

The design space would require considerations on how to make the feedback accessible and easy to follow for both BVI and sighted developers respectively. Errors and warning notifications in static analysis tools are visual (e.g., yellow squiggly lines to flag unused variables, red squiggly lines for undeclared variables, etc). A few IDEs like VS Code provide keyboard shortcuts to jump between errors but these do not permit the spontaneous fixes that sighted developers can do as they are reading the code. On the other hand, sighted developers may not prioritize accessibility

[1]XCode's Accessibility Inspector is not a static analysis tool; developers need to run the tool

warnings if it is presented as part of the larger list of problems. We could consider highlighting the errors in the layout editor instead of the problems window.

7.4.2 Using Generative AI Tools

We can also look to AI tools like GitHub CoPilot to generate solution code that complies with inclusive code styling guidelines. Currently, developers specify natural language instructions to AI agents to generate code that meets their instructions. In future work, I would investigate the kind of input that developers give to AI tools and how these can be adjusted to yield code that is in line with the readability preferences of BVI as well as sighted developers. For instance, suggestions from BVI developers can be used to improve identifier naming choices for screen reader use.

In UI programming, BVI developers can be asked to rate the generated code to identify accessible components, which can be shared back with sighted developers to improve the accessibility of the framework as a whole. One can also generate code samples to teach accessibility practices to sighted developers, which is difficult to produce at scale for instructional purposes [232]. Lastly, we can consider ways in which legacy code and applications resulting from them can be made accessible for BVI developers and users respectively. For instance, we can generate documentation and address the gaps in available documentation to reduce the information seeking efforts for BVI developers [212].

7.4.3 Braille Displays

My studies do not particularly investigate the usage of Braille displays. The most popular displays typically comprise 32 cells, 40 cells, or 80-cells and display only one line of code at a time. Participants' comments suggest that they used Braille displays to read code with complex characters and identifiers at an individual level and not during collaboration. Furthermore, indentation characters like tabs and spaces used up braille cells in code with nested data structures. In future work, we would examine the factors that constitute code readability on braille displays and pin-matrix tactile displays, which can display 2D tactile graphics [42]. These displays are becoming less expensive. Specifically, tactile pin-matrix displays are gaining popularity for 2D visualizations [42]. Along with tactile drawing tools, these displays have the potential to positively impact activities like UI development and software architecture design [42, 163].

7.5 Personal Reflection

My research has improved my own practices during collaboration. For instance, I ask people about their preferred tools to ensure that the group choice works well for everyone. I have also acquired

insights into authoring more accessible code and documents, thereby influencing my participation in activities outside of software development. For instance, when sharing documents, I opt for text-based formats like Word documents over PDFs to avoid issues with inaccurate or missing tags in PDFs. Additionally, I try to produce documentation that is easy to navigate and skim.

As a sighted accessibility researcher in a team of sighted peers, I recognize that my strength lies in conducting empirical studies and devising strategies to enhance accessibility awareness more broadly. Therefore, my reporting of study findings also serves as means to educate sighted individuals about accessibility challenges and preferences of BVI developers. On the other hand, in a mixed-ability research team, I apply myself more to development projects aimed at improving collaborative software. In such teams, my perspective as a sighted collaborator who uses GUIs and the perspective of the BVI collaborator who uses screen readers come together to identify ways in which existing systems can be adapted or extended for screen reader use.

CHAPTER 8

Conclusion

My thesis is that the interplay between programming and collaboration tools, assistive technologies (ATs), programming practices, and the organizational norms around communication and help-seeking shape the collaborative experiences of BVI developers in mixed-ability contexts; we must design tools to not only improve accessibility but also to minimize the various forms of additional work BVI developers perform to achieve effective collaboration with sighted colleagues. This dissertation has contributed rich, empirical insights that describe the challenges in collaboration and the nature of additional work performed by BVI developers to work around these challenges. Through one of the studies, I show that we can successfully design tools to minimize the additional work during tightly-coupled synchronous code walkthroughs. I offer recommendations on how to extend the design process to other common collaboration activities, including asynchronous activities such as software maintenance and code reviews, thereby proving my thesis.

APPENDIX A

Tasks and Source Code for CodeWalk Evaluation Study

This appendix lists the task sets and source code used in the evaluaton of CodeWalk. They are listed by the four programming languages we used in the study: (1) JavaScript, (2) Python, (3) C#, (4) Java.

A.1 Source Code

All code samples included (1) a main code file representing the game's logic, (2) a text file listing both sets of tasks in the order determined for the participant (see A.2) (3) a text file called `words.txt` containing 851 words in a list to play the game (see Appendix A.2.5). The C# code sample included an additional file (TextVisualize.cs) that represented the game's UI; this file was referenced for the string editing task.

A.1.1 JavaScript

```
var words = [
  "the big bang",
  "the pillow feels soft",
  "odd one out",
  "time to go home",
  "that was easy",
  "hangman is cool",
  "zoologist",
  "quadruplets",
  "the sky is blue"
]

let answer = '';
```

```
let maxWrong = 6;
let mistakes = 0;
let guessed = [];
let wordStatus = null;

function randomWord() {
  answer = words[Math.floor(Math.random() * words.length)];
  console.log(answer);
}

function generateButtons() {
  let buttonsHTML = 'abcdefghijklmnopqrstuvwxyz'.split('').map(letter =>
    `
      <button
        class="btn btn-lg btn-primary m-2"
        id='` + letter + `'
        onClick="handleGuess('` + letter + `')"
      >
        ` + letter + `
      </button>
    `).join('');

  document.getElementById('keyboard').innerHTML = buttonsHTML;
}

function handleGuess(chosenLetter) {
  guessed.indexOf(chosenLetter) === -1 ? guessed.push(chosenLetter) : null;
  document.getElementById(chosenLetter).setAttribute('disabled', true);
  console.log(guessed);
  if (answer.indexOf(chosenLetter) >= 0) {
    guessedWord();
    checkIfGameWon();
  } else if (answer.indexOf(chosenLetter) === -1) {
    mistakes++;
    updateMistakes();
    checkIfGameLost();
    updateHangmanPicture();
  }
}

function updateHangmanPicture() {
  document.getElementById('hangmanPic').src = './images/' + mistakes + '.jpg';
}
```

```
57
58 function checkIfGameWon() {
59   if (wordStatus === answer) {
60     document.getElementById('keyboard').innerHTML = 'You Won!!!';
61   }
62 }
63
64 function checkIfGameLost() {
65   if (mistakes === maxWrong) {
66     document.getElementById('wordSpotlight').innerHTML = 'The answer was: ' +
     answer;
67     document.getElementById('keyboard').innerHTML = 'You Lost!!!';
68   }
69 }
70
71 function guessedWord() {
72   wordStatus = answer.split('').map(letter => (guessed.indexOf(letter) >= 0 ?
      letter : " _ ")).join('');
73   console.log(wordStatus);
74   document.getElementById('wordSpotlight').innerHTML = wordStatus;
75 }
76
77 function updateMistakes() {
78   document.getElementById('mistakes').innerHTML = mistakes;
79 }
80
81 function reset() {
82   mistakes = 0;
83   guessed = [];
84   document.getElementById('hangmanPic').src = './images/0.jpg';
85
86   randomWord();
87   guessedWord();
88   updateMistakes();
89   generateButtons();
90 }
91
92 document.getElementById('maxWrong').innerHTML = maxWrong;
93
94 randomWord();
95 generateButtons();
96 guessedWord();
```

A.1.2 Python

```
import pygame
import random

pygame.init()
winHeight = 480
winWidth = 700
win=pygame.display.set_mode((winWidth,winHeight))

#-------------------------------------#
# initialize global variables/constants #
#-------------------------------------#
BLACK = (0,0, 0)
WHITE = (255,255,255)
RED = (255,0, 0)
GREEN = (0,255,0)
BLUE = (0,0,255)
LIGHT_BLUE = (102,255,255)

buttonFont = pygame.font.SysFont("arial", 20)
guessFont = pygame.font.SysFont("monospace", 24)
resultFont = pygame.font.SysFont('arial', 45)
word = ''
buttons = []
guessed = []
hangmanPics = [pygame.image.load('hangman0.png'), pygame.image.load('hangman1.
    png'), pygame.image.load('hangman2.png'), pygame.image.load('hangman3.png'
    ), pygame.image.load('hangman4.png'), pygame.image.load('hangman5.png'),
    pygame.image.load('hangman6.png')]

limbs = 0


def redrawGameWindow():
    global guessed
    global hangmanPics
    global limbs
    win.fill(GREEN)
    # Buttons
    for i in range(len(buttons)):
        if buttons[i][4]:
            pygame.draw.circle(win, BLACK, (buttons[i][1], buttons[i][2]),
    buttons[i][3])
```

```
            pygame.draw.circle(win, buttons[i][0], (buttons[i][1], buttons[i
    ][2]), buttons[i][3] - 2
                                )
            label = buttonFont.render(chr(buttons[i][5]), 1, BLACK)
            win.blit(label, (buttons[i][1] - (label.get_width() / 2), buttons[
    i][2] - (label.get_height() / 2)))

    spacedWord = ''
    for character in word:
        spacedWord += '_ '
        for item in guessed:
            if character.upper() == item:
                spacedWord = spacedWord[:-2]
                spacedWord += character.upper() + ' '
    label1 = guessFont.render(spacedWord, 1, BLACK)
    rect = label1.get_rect()
    length = rect[2]

    win.blit(label1,(winWidth/2 - length/2, 400))

    pic = hangmanPics[limbs]
    win.blit(pic, (winWidth/2 - pic.get_width()/2 + 20, 150))
    pygame.display.update()


def didGuessCorrect(guess):
    global word
    if guess.lower() not in word.lower():
        return True


def buttonHit(x, y):
    for i in range(len(buttons)):
        if x < buttons[i][1] + 20 and x > buttons[i][1] - 20:
            if y < buttons[i][2] + 20 and y > buttons[i][2] - 20:
                return buttons[i][5]
    return None


def end(winner=False):
    global limbs
    lostTxt = 'You lost, press any key to play again...'
    winTxt = 'WINNER!, press any key to play...'
```

```
    redrawGameWindow()
    pygame.time.delay(1000)
    win.fill(GREEN)

    if winner == True:
        label = resultFont.render(winTxt, 1, BLACK)
    else:
        label = resultFont.render(lostTxt, 1, BLACK)

    wordTxt = resultFont.render(word.upper(), 1, BLACK)
    wordWas = resultFont.render('The phrase was: ', 1, BLACK)

    win.blit(wordTxt, (winWidth/2 - wordTxt.get_width()/2, 295))
    win.blit(wordWas, (winWidth/2 - wordWas.get_width()/2, 245))
    win.blit(label, (winWidth / 2 - label.get_width() / 2, 140))
    pygame.display.update()
    again = True
    while again:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
            if event.type == pygame.KEYDOWN:
                again = False
    reset()


def reset():
    global limbs
    global guessed
    global buttons
    global word
    for i in range(len(buttons)):
        buttons[i][4] = True

    limbs = 0
    guessed = []
    file = open('words.txt')
    f = file.readlines()
    i = random.randrange(0, len(f) - 1)
    word = f[i][:-1]

#MAINLINE

```

```
123
124 # Setup buttons
125 increase = round(winWidth / 13)
126 for i in range(26):
127     if i < 13:
128         y = 40
129         x = 25 + (increase * i)
130     else:
131         x = 25 + (increase * (i - 13))
132         y = 85
133     buttons.append([LIGHT_BLUE, x, y, 20, True, 65 + i])
134
135 file = open('words.txt')
136 f = file.readlines()
137 i = random.randrange(0, len(f) - 1)
138 word = f[i][:-1]
139 inPlay = True
140
141 while inPlay:
142     redrawGameWindow()
143     pygame.time.delay(10)
144
145     for event in pygame.event.get():
146         if event.type == pygame.QUIT:
147             inPlay = False
148         if event.type == pygame.KEYDOWN:
149             if event.key == pygame.K_ESCAPE:
150                 inPlay = True
151         if event.type == pygame.MOUSEBUTTONDOWN:
152             clickPos = pygame.mouse.get_pos()
153             letter = buttonHit(clickPos[0], clickPos[1])
154             if letter != None:
155                 guessed.append(chr(letter))
156                 buttons[letter - 65][4] = False
157                 if didGuessCorrect(chr(letter)):
158                     if limbs != 5:
159                         limbs += 1
160                     else:
161                         end()
162                 else:
163                     spacedWord = ''
164                     for character in word:
165                         spacedWord += '_ '
```

```
166                         for item in guessed:
167                             if character.upper() == item:
168                                 spacedWord = spacedWord[:-2]
169                                 spacedWord += character.upper() + ' '
170                     if spacedWord.count('_') == 0:
171                         end(True)
172 
173 pygame.quit()
174 # always quit pygame when done!
```

A.1.3 C#

A.1.4 Program.cs

```
using System;
using System.Configuration;
using System.Collections.Generic;
using System.Text.RegularExpressions;
using Hangman.View;
using System.IO;
using System.Reflection;
using System.Diagnostics;

namespace Hangman {
    class Program {

        static void Main(string[] args)
        {
            string path = Path.Combine(Path.GetDirectoryName(Assembly.
   GetExecutingAssembly().Location), @"words.txt");
            string[] wordList = File.ReadAllLines(path);
            var randomIndex = new Random();
            var word = wordList[randomIndex.Next(wordList.Length)];
            var visualizer = new TextVisualizer();
            var game = new Game(visualizer, word);
        }
    }

    public class Game
    {
        private List<char> wordToGuess;
        private List<char> wordGuessed;
        private List<char> incorrectGuesses;

        private IVisualizer _gameVisualizer;
        private bool _isRunning = false;

        public Game(IVisualizer visualizer, string word)
        {
            _gameVisualizer = visualizer;
            wordToGuess = new List<char>();
            wordGuessed = new List<char>();
            incorrectGuesses = new List<char>();
```

```
39
40              wordToGuess.AddRange(word);
41              for (var x = 0; x < wordToGuess.Count; x++)
42                  wordGuessed.Add('_');
43
44              _gameVisualizer.WelcomeScreen();
45              if (!_isRunning)
46                  GameLoop();
47          }
48
49          private void GameLoop()
50          {
51              _isRunning = true;
52
53              while (wordGuessed.Contains('_'))
54              {
55                  // Game screen.
56                  Console.Clear();
57                  _gameVisualizer.RefreshGameScreen(wordGuessed,
    incorrectGuesses);
58
59                  // Request the users next guess.
60                  _gameVisualizer.RequestGuess();
61                  var playerGuess = Console.ReadLine().ToUpper();
62
63                  if (ValidateGuess(playerGuess))
64                  {
65                      var guess = Convert.ToChar(playerGuess);
66
67                      if (!wordGuessed.Contains(guess) && !incorrectGuesses.
    Contains(guess))
68                      {
69                          if (wordToGuess.Contains(guess))
70                          {
71                              // Handle guess.
72                              for (var x = 0; x < wordToGuess.Count; x++)
73                              {
74                                  if (wordToGuess[x] == guess)
75                                      wordGuessed[x] = guess;
76                              }
77                          }
78                          else
79                          {
```

```
                            // Handle guess.
                            if (incorrectGuesses.Count >= 6)
                                _gameVisualizer.LoseScreen(wordToGuess);
                        }
                    }
                    else
                    {
                        _gameVisualizer.AlreadyGuessed();
                    }
                }
                else
                {
                    _gameVisualizer.InvalidGuess();
                }
            }

            // The player must have won.
            _gameVisualizer.WinScreen(wordToGuess, incorrectGuesses.Count);
        }

        private static bool ValidateGuess(string guess)
        {
            // Must be alphabetical, and a single character.
            return (guess.Length == 1) && Regex.IsMatch(guess, @"^[a-zA-Z]+$")
    ;
        }
     }
}
```

A.1.5 TextVisualizer.cs

```
using System;
using System.Collections.Generic;

namespace Hangman.View
{
    public class TextVisualizer : IVisualizer
    {

        public void WelcomeScreen()
        {
            Console.Clear();
            Console.WriteLine("Welcome to Hangman!");
            Console.WriteLine("Press ENTER to start the game...");
```

```
            Console.ReadLine();
        }

        public void RefreshGameScreen(IEnumerable<char> word, IEnumerable<char
    > incorrectGuesses)
        {
            Console.Write(Environment.NewLine + "Word: ");
            foreach (var letter in word)
                Console.Write(letter + " ");

            Console.Write(Environment.NewLine + "Incorrect Guesses: ");
            foreach (var letter in incorrectGuesses)
                Console.Write(letter + ", ");

            Console.WriteLine();
        }

        public void LoseScreen(IEnumerable<char> word)
        {
            Console.Clear();
            string lostText = "You Lost! Press ENTER to leave the game";
            Console.WriteLine(lostText);
            Console.WriteLine(Environment.NewLine);
            Console.ReadLine();
        }
        public void WinScreen(IEnumerable<char> word, int
    incorrectGuessesCount)
        {
            Console.Clear();
            string winText = "Winner! Hit ENTER to leave the game";
            Console.WriteLine(winText);
            Console.WriteLine(Environment.NewLine);
            Console.ReadLine();
        }

        public void RequestGuess()
        {
            Console.WriteLine(Environment.NewLine + "Enter a guess");
        }

        public void AlreadyGuessed()
        {
            Console.WriteLine(Environment.NewLine + "You have already guessed
```

```
    that letter!");
55          }
56
57          public void InvalidGuess()
58          {
59              Console.WriteLine(Environment.NewLine + "Invalid Guess. Guesses
    must be a single alphabetical letter.");
60          }
61      }
62  }
```

A.1.6 Java

```java
package Hangman;

import java.util.Scanner;
import java.util.Random;

public class Built {
    public static void main(String[] args){
        Scanner scanner = new Scanner(System.in);

        BufferedReader reader = new BufferedReader(new FileReader("words.txt")
    );
        String str;
        List<String> list = new ArrayList<String>();
        while((str = reader.readLine()) != null){
            list.add(str);
        }
        String[] words = list.toArray(new String[0]);

        boolean weArePlaying = true;
        while(weArePlaying){
            System.out.println("Lets Start Playing Hangman ver 0.1");

            Random random = new Random();
            int wordListLength = words.length
            int randomNumber = random.nextInt(wordListLength);
            char randomWordToGuess[] = words[randomNumber].toCharArray();

            int ammountOfGuesses = randomWordToGuess.length;
            char playerGuess[] = new char[ammountOfGuesses]; // "_ _ _ _ _ _ _
    _"

            for(int i=0; i<playerGuess.length; i++){ // Assign empty dashes at
    start "_ _ _ _ _ _ _ _"
                playerGuess[i] =  '_';
            }

            boolean wordIsGuessed = false;
            int tries = 0;

            while(!wordIsGuessed && tries != ammountOfGuesses){
                System.out.println("Current Guesses: ");
                print(playerGuess);
```

```
40              System.out.printf("You have %d ammount of tries left.\n",
    ammountOfGuesses-tries);
41              System.out.println("Enter a single character: ");
42              char input = scanner.nextLine().charAt(0);
43              tries++;
44
45              if(input == '-'){
46                  wordIsGuessed = true;
47                  weArePlaying = false;
48              } else{
49                  for(int i=0; i<randomWordToGuess.length; i++){
50                      if(randomWordToGuess[i] == input){
51                          playerGuess[i] = input;
52                      }
53                  }
54
55                  if(isTheWordGuessed(playerGuess)){
56                      wordIsGuessed = true;
57                      String winText = "Winner! Enter 'y' to play again";
58                      System.out.println("Congratulations");
59                  }
60              }
61          } /* End of wordIsGuessed */
62          if(!wordIsGuessed){
63              String lostText = "You lost! Enter 'y' to play again; Enter 'n
    ' to stop!";
64              System.out.println("You ran out of guesses.");
65          }
66
67          System.out.println("Would you like to play again? (y/n) ");
68          String choice = scanner.nextLine();
69          if(choice.equals("n")){
70              weArePlaying = true;
71          }
72
73      }/*End of We Are Playing*/
74
75      System.out.println("Game Over!");
76  }
77
78  public static void print(char array[]){
79      for(int i=0; i<array.length; i++){ // Assign empty dashes at start "_
    _ _ _ _ _ _ _"
```

```
80              System.out.print(array[i] + " ");
81          }
82          System.out.println();
83      }
84
85      public static boolean isTheWordGuessed(char[] array){
86          boolean condition = false;
87          for(int i=0; i<array.length; i++){
88              if(array[i] == '_'){
89                  condition = true;
90              }
91          }
92          return condition;
93      }
94  }
```

A.2 Tasks in CodeWalk Evaulation

This contains the task sets used evaluation study. They are presented by programming language.

A.2.1 JavaScript

A.2.1.1 Task Set A

1. How to add another word to the list of words? Can we add ‘zigzag’ to the list?
2. How can we ensure that the game is reset properly?
3. How do you think we can refactor the code on lines 93-96?

A.2.1.2 Task Set B

1. Should we update winTxt in checkIfGameWon() and make it similar to lostTxt in checkIfGameLost()?
2. Where should we update the value of mistakes variable in handleGuess()?
3. How can we refactor the code on lines 79-85?

A.2.2 Python

A.2.2.1 Task Set A

1. How to add another word to the list of words? Can we add ’zigzag’ to the list?

2. How should we modify didGuessCorrect()?

3. How do you think we can refactor the code on lines 135-138?

A.2.2.2 Task Set B

1. Should we update winTxt in end() and make it similar to lostTxt?

2. How can we ensure that the game quits when the escape key is pressed?

3. How do you think we can refactor the code on lines 164-169?

A.2.3 C#

A.2.3.1 Task Set A

1. How to add another word to the list of words? Can we add 'zigzag' to the list?

2. How can we ensure incorrectGuesses is updated?

3. How do you think we can refactor the code on lines 15-18 in Program.cs into a function?

A.2.3.2 Task Set B

1. Should we update winTxt in WinScreen() and make it similar to lostTxt in LoseScreen() in TextVisualizer.cs?

2. How can we ensure that the game quits when the player loses the game?

3. How do you think we can refactor the code on lines 72-76, after the first "Handle Guess" comment in Program.cs into a function?

A.2.4 Java

A.2.4.1 Task Set A

1. How to add another word to the list of words? Can we add 'zigzag' to the list?

2. How should we modify isTheWordGuessed()?

3. How do you think we can refactor the code on lines 10-16?

A.2.4.2 Task Set A

1. Should we update the string stored in winTxt to make it similar to lostTxt?
2. How can we ensure that the game quits when character 'n' is entered by the user?
3. How do you think we can refactor the code on lines 22-25?

A.2.5 words.txt

able
about
account
acid
across
act
addition
adjustment
advertisement
after
again
against
agreement
air
all
almost
among
amount
amusement
and
angle
angry
animal
answer
ant
any
apparatus
apple
approval
arch
argument
arm
army
art
as
at
attack
attempt
attention
attraction
authority
automatic
awake
baby
back
bad
bag
balance
ball
band
base
basin
basket
bath
be
beautiful
because
bed
bee
before
behaviour
belief
bell
bent
berry
between
bird
birth
bit
bite
bitter
black
blade
blood
blow
blue
board
boat
body
boiling
bone
book
boot
bottle
box
boy
brain
brake
branch
brass
bread
breath
brick
bridge
bright
broken
brother
brown
brush
bucket
building
bulb
burn
burst
business
but
butter
button
by
cake
camera
canvas
card
care
carriage
cart
cat
cause
certain
chain
chalk
chance
change
cheap
cheese
chemical
chest
chief
chin
church
circle
clean
clear
clock
cloth
cloud
coal
coat
cold
collar
colour
comb
come
comfort
committee
common
company
comparison
competition
complete
complex
condition
connection
conscious
control
cook
copper
copy
cord
cork
cotton
cough
country
cover
cow
crack
credit
crime
cruel
crush
cry
cup
cup
current
curtain
curve
cushion
damage
danger
dark
daughter
day
dead
dear
death
debt
decision
deep
degree
delicate
dependent
design
desire
destruction
detail
development
different
digestion
direction
dirty
discovery
discussion
disease
disgust
distance
distribution
division
do
dog
door
doubt
down
drain
drawer
dress
drink
driving
drop
dry
dust
ear
early
earth
east
edge
education
effect
egg
elastic
electric
end
engine
enough
equal
error
even
event
ever
every
example
exchange
existence
expansion
experience
expert
eye
face
fact
fall
false
family
far
farm
fat
father
fear
feather
feeble
feeling
female
fertile
fiction
field
fight
finger
fire
first
fish
fixed
flag
flame
flat
flight
floor
flower
fly
fold
food
foolish
foot
for
force
fork
form
forward
fowl
frame
free
frequent
friend
from
front
fruit
full
future
garden
general
get
girl
give
glass
glove
go
goat
gold
good
government
grain
grass
great
green
grey
grip
group
growth
guide
gun
hair
hammer
hand
hanging
happy

harbour
hard
harmony
hat
hate
have
he
head
healthy
hear
hearing
heart
heat
help
high
history
hole
hollow
hook
hope
horn
horse
hospital
hour
house
how
humour
I
ice
idea
if
ill
important
impulse
in
increase
industry
ink
insect
instrument
insurance
interest
invention
iron
island
jelly
jewel
join
journey
judge
jump
keep
kettle
key
kick
kind
kiss
knee
knife
knot
knowledge
land
language
last
late
laugh
law
lead
leaf
learning
leather
left
leg
let
letter
level
library
lift
light
like
limit
line
linen
lip
liquid
list
little
living
lock
long
look
loose
loss
loud
love
low
machine
make
male
man
manager
map
mark
market
married
mass
match
material
may
meal
measure
meat
medical
meeting
memory
metal
middle
military
milk
mind
mine
minute
mist
mixed
money
monkey
month
moon
morning
mother
motion
mountain
mouth
move
much
muscle
music
nail
name
narrow
nation
natural
near
necessary
neck
need
needle
nerve
net
new
news
night
no
noise
normal
north
nose
not
note
now
number
nut
observation
of
off
offer
office
oil
old
on
only
open
operation
opinion
opposite
or
orange
order
organization
ornament
other
out
oven
over
owner
page
pain
paint
paper
parallel
parcel
part
past
paste
payment
peace
pen
pencil
person
physical
picture
pig
pin
pipe
place
plane
plant
plate
play
please
pleasure
plough
pocket
point
poison
polish
political
poor
porter
position
possible
pot
potato
powder
power
present
price
print
prison
private
probable
process
produce
profit
property
prose
protest
public
pull
pump
punishment
purpose
push
put
quality
question
quick
quiet
quite
rail
rain
range
rat
rate
ray
reaction
reading
ready
reason
receipt
record
red
regret
regular
relation
religion
representative
request
respect
responsible
rest
reward
rhythm
rice
right
ring
river
road
rod
roll
roof
room
root
rough
round
rub
rule
run
sad
safe
sail
salt
same
sand
say
scale
school
science
scissors
screw
sea
seat
second
secret
secretary
see
seed
seem
selection
self
send
sense
separate
serious
servant
sex
shade
shake
shame
sharp
sheep
shelf

ship
shirt
shock
shoe
short
shut
side
sign
silk
silver
simple
sister
size
skin
skirt
sky
sleep
slip
slope
slow
small
smash
smell
smile
smoke
smooth
snake
sneeze
snow
so
soap
society
sock
soft
solid
some
son
song
sort
sound
soup
south
space
spade
special
sponge
spoon
spring
square
stage
stamp
star
start
statement
station
steam
steel
stem
step
stick
sticky
stiff
still
stitch
stocking
stomach
stone
stop
store
story
straight
strange
street
stretch
strong
structure
substance
such
sudden
sugar
suggestion
summer
sun
support
surprise
sweet
swim
system
table
tail
take
talk
tall
taste
tax
teaching
tendency
test
than
that
the
then
theory
there
thick
thin
thing
this
thought
thread
throat
through
through
thumb
thunder
ticket
tight
till
time
tin
tired
to
toe
together
tomorrow
tongue
tooth
top
touch
town
trade
train
transport
tray
tree
trick
trouble
trousers
true
turn
twist
umbrella
under
unit
up
use
value
verse
very
vessel
view
violent
voice
waiting
walk
wall
war
warm
wash
waste
watch
water
wave
wax
way
weather
week
weight
well
west
wet
wheel
when
where
while
whip
whistle
white
who
why
wide
will
wind
window
wine
wing
winter
wire
wise
with
woman
wood
wool
word
work
worm
wound
writing
wrong
year
yellow
yes
yesterday
you
young

APPENDIX B

Stimulus for Code Readability Study

B.1 Readability Rules

This section contains the code snippets and factors we included in the markdown to understand BVI developers' readability preferences. Next section shows a sample markdown presented to one of the participants.

````
# Code Formatting Rules

## 1. Spacing

### 1.1 Indentation

#### 1.1.1 Nested Data Structures

Option 1: Keep parentheses and key-value pairs on separate lines
```
{
  "menu": {
    "id": "file",
    "value": "File",
    "popup": {
      "menuitem": [
        {
          "value": "New",
          "onclick": "CreateNewDoc()"
        },
        {
````
````

```
          "value": "Open",
          "onclick": "OpenDoc()"
        },
        {
          "value": "Close",
          "onclick": "CloseDoc()"
        }
      ]
    }
  }
}
```

Option 2: Match key-value pairs and parentheses

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]}
}}
```

1.1.2 Multiline docstrings

Option 1: Doctring is not indented

```
def add_binary(a, b):
    '''
    Returns the sum of two decimal numbers in binary digits.

    Parameters:
    a (int): A decimal integer
```

```
    b (int): Another decimal integer

    Returns: binary_sum (str): Binary string of the sum of a and b
    '''
    binary_sum = bin(a+b)[2:]
    return binary_sum
```

Option 2: Doctring is indented
```
def add_binary(a, b):
    '''
    Returns the sum of two decimal numbers in binary digits.

        Parameters:
                a (int): A decimal integer
                b (int): Another decimal integer

        Returns:
                binary_sum (str): Binary string of the sum of a and b
    '''
    binary_sum = bin(a+b)[2:]
    return binary_sum
```

1.2 Segmenting

1.2.1 Line breaks in source code

Option 1: Use double empty lines to separate functions, conditionals, and classes
```
def factorial(num):
    fact = 1
    for i in range(1, num+1):
```

```
        fact = fact * i
    return fact


if condition:
    print("This condition was TRUE")


class Point:
    x: int
    y: int
```

Option 2: Use single empty lines between functions, conditionals, and classes

```
def factorial(num):
    fact = 1
    for i in range(1, num+1):
        fact = fact * i
    return fact

if condition:
    print("This condition was TRUE")

class Point:
    x: int
    y: int
```

1.3 Whitespaces

1.3.1 Whitespaces in operators

Option 1: Avoid whitespaces before and after operators

```
b = config.base**5.2
submitted+=1
hypot2 = x*x+y*y
```

Option 2: Surround operators with whitespaces
```
b = config.base ** 5.2
submitted += 1
hypot2 = x*x + y*y
```

1.3.2 Whitespace in slice operators

Option 1: Use whitespaces
```
ham[lower : upper + offset]
```

Option 2: Avoid whitespaces
```
ham[lower:upper+offset]
```

2. Identifiers

2.1 Naming style for variables

Option 1: Use snake case
```
primary_address_apartment = ""
```

Option 2: Use camel case

```
primaryAddressApartment = ""
```

2.2 Length preference for variable names

Option 1: Long names
```
radioButtonHeight = "20"
```

Option 2: Short names
```
radioBtnHt = "20"
```

2.3 Consistency in variable names

Option 1: Use consistent prefixes
```
foregroundColorMenu = ""
foregroundColorBody = ""
foregroundColorFooter = ""
```

Option 2: Use consistent suffixes
```
menuForegroundColor = ""
bodyForegroundColor = ""
footerForegroundColor = ""
```

3. Line Length

3.0.1 Formatting function calls

Option 1: Render arguments on the same line
```
ImportantClass.important_method(exc, limit, lookup_lines, capture_locals,
extra_argument)
```

Option 2: Render arguments on separate lines
```
ImportantClass.important_method(
    exc,
    limit,
    lookup_lines,
    capture_locals,
    extra_argument
)
```

3.0.2 Formatting function signatures

Option 1: Render arguments on separate lines
```
# Applies `variables` to the `template` and writes to `file`
def very_important_function(
    template: str,
    *variables,
    file: os.PathLike,
    engine: str,
    header: bool = True,
    debug: bool = False,
):
    with open(file, 'w') as f:
    ...
```

```

Option 2: Render arguments on the same line
```
Applies `variables` to the `template` and writes to `file`
def very_important_function(template: str, *variables, file: os.PathLike,
engine: str, header: bool = True, debug: bool = False):
    with open(file, 'w') as f:
    ...
```

### 3.0.3 Call chains

Option 1: Treat dot operator as a delimiter
```
def example(session):
    result = (
        session.query(models.Customer.id)
        .filter(models.Customer.account_id == account_id)
        .order_by(models.Customer.id.asc())
        .all()
)
```

Option 2: Do not treat dot operator as a delimiter
```
def example(session):
    result = (session.query(models.Customer.id).filter(models.Customer.
    account_id == account_id).order_by(models.Customer.id.asc()).all())
```

### 3.0.4 Line breaks with binary operators

Option 1: Place line break after the operator
```

```
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

Option 2: Place operator after the line break
```
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

3.0.5 Comments

Option 1: Wrap comments across lines
```
from collections import defaultdict

def get_top_cities(prices):
    top_cities = defaultdict(int)

    # Count number of times the city was searched for each price range,
    # get the top 3 cities, and add to dictionary
    return dict(top_cities)
```

Option 2: Do not wrap comments
```
from collections import defaultdict
```

```
def get_top_cities(prices):
    top_cities = defaultdict(int)

    # Count number of times the city was searched for each price range, get
    the top 3 cities, and add to dictionary
    return dict(top_cities)
```

3.0.6 Imports

Option 1: Place imports on different lines
```
import os
import sys
import random
import json
```

Option 2: Place imports on the same line
```
import os, sys, random, json
```

4. String Quotes

4.1 Use of quotation marks in docstrings

Option 1: Use single quotation marks
```
def square(n):
    '''Takes in a number n, returns the square of n'''
    return n**2
```

````
Option 2: Use double quotation marks
```
def square(n):
    """Takes in a number n, returns the square of n"""
    return n**2
```
````

B.2 Sample Markdown

This shows a markdown with order of rules and options randomized. The markdown was given to one of the participants as part of the study.

````
# Code Formatting Rules

## 1. Length preference for variable names

Option 1: Long names
```
radioButtonHeight = "20"
```

Option 2: Short names
```
radioBtnHt = "20"
```

## 2. Consistency in variable names

Option 1: Use consistent prefixes
```
foregroundColorMenu = ""
foregroundColorBody = ""
foregroundColorFooter = ""
```

Option 2: Use consistent suffixes
````

```
menuForegroundColor = ""
bodyForegroundColor = ""
footerForegroundColor = ""
```

3. Whitespace in slice operators

Option 1: Use whitespaces
```
ham[lower : upper + offset]
```

Option 2: Avoid whitespaces
```
ham[lower:upper+offset]
```

4. Line breaks with binary operators

Option 1: Place operator after the line break
```
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

Option 2: Place line break after the operator
```
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

```

## 5. Formatting function signatures

Option 1: Render arguments on the same line
```
Applies `variables` to the `template` and writes to `file`
def very_important_function(template: str, *variables, file: os.PathLike,
engine: str, header: bool = True, debug: bool = False):
    with open(file, 'w') as f:
    ...
```

Option 2: Render arguments on separate lines
```
Applies `variables` to the `template` and writes to `file`
def very_important_function(
    template: str,
    *variables,
    file: os.PathLike,
    engine: str,
    header: bool = True,
    debug: bool = False,
):
    with open(file, 'w') as f:
    ...
```

## 6. Naming style for variables

Option 1: Use snake case
```
primary_address_apartment = ""
```

Option 2: Use camel case
```

```
primaryAddressApartment = ""
```

7. Imports

Option 1: Place imports on the same line
```
import os, sys, random, json
```

Option 2: Place imports on different lines
```
import os
import sys
import random
import json
```

8. Use of quotation marks in docstrings

Option 1: Use double quotation marks
```
def square(n):
    """Takes in a number n, returns the square of n"""
    return n**2
```

Option 2: Use single quotation marks
```
def square(n):
    '''Takes in a number n, returns the square of n'''
    return n**2
```

9. Line breaks in source code

Option 1: Use double empty lines to separate functions, conditionals, and classes

```
def factorial(num):
    fact = 1
    for i in range(1, num+1):
        fact = fact * i
    return fact


if condition:
    print("This condition was TRUE")


class Point:
    x: int
    y: int
```

Option 2: Use single empty lines between functions, conditionals, and classes

```
def factorial(num):
    fact = 1
    for i in range(1, num+1):
        fact = fact * i
    return fact

if condition:
    print("This condition was TRUE")

class Point:
    x: int
    y: int
```

10. Formatting function calls

Option 1: Render arguments on the same line
```
ImportantClass.important_method(exc, limit, lookup_lines, capture_locals,
extra_argument)
```

Option 2: Render arguments on separate lines
```
ImportantClass.important_method(
    exc,
    limit,
    lookup_lines,
    capture_locals,
    extra_argument
)
```

11. Splitting parentheses

Option 1: Keep parentheses and key-value pairs on separate lines
```
{
  "menu": {
    "id": "file",
    "value": "File",
    "popup": {
      "menuitem": [
        {
          "value": "New",
          "onclick": "CreateNewDoc()"
        },
        {
          "value": "Open",
```

```
          "onclick": "OpenDoc()"
        },
        {
          "value": "Close",
          "onclick": "CloseDoc()"
        }
      ]
    }
  }
}
```

Option 2: Match key-value pairs and parentheses
```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]}
}}
```

12. Multiline docstrings

Option 1: Doctring is indented
```
def add_binary(a, b):
    '''
    Returns the sum of two decimal numbers in binary digits.

        Parameters:
                a (int): A decimal integer
```

```
                b (int): Another decimal integer

        Returns:
                binary_sum (str): Binary string of the sum of a and b
    '''
    binary_sum = bin(a+b)[2:]
    return binary_sum
```

Option 2: Doctring is not indented
```
def add_binary(a, b):
    '''
    Returns the sum of two decimal numbers in binary digits.

    Parameters:
    a (int): A decimal integer
    b (int): Another decimal integer

    Returns: binary_sum (str): Binary string of the sum of a and b
    '''
    binary_sum = bin(a+b)[2:]
    return binary_sum
```

13. Comments

Option 1: Do not wrap comments
```
from collections import defaultdict

def get_top_cities(prices):
    top_cities = defaultdict(int)

    # Count number of times the city was searched for each price range,
    get the top 3 cities, and add to dictionary
```

```
    return dict(top_cities)
```

Option 2: Wrap comments across lines
```
from collections import defaultdict

def get_top_cities(prices):
    top_cities = defaultdict(int)

    # Count number of times the city was searched for each price range,
    # get the top 3 cities, and add to dictionary
    return dict(top_cities)
```

14. Call chains

Option 1: Treat dot operator as a delimiter
```
def example(session):
    result = (
        session.query(models.Customer.id)
        .filter(models.Customer.account_id == account_id)
        .order_by(models.Customer.id.asc())
        .all()
    )
```

Option 2: Do not treat dot operator as a delimiter
```
def example(session):
    result = (session.query(models.Customer.id).filter(models.Customer.
account_id == account_id).order_by(models.Customer.id.asc()).all())
```

15. Whitespaces in operators

Option 1: Surround operators with whitespaces
```
b = config.base ** 5.2
submitted += 1
hypot2 = x*x + y*y
```

Option 2: Avoid whitespaces before and after operators
```
b = config.base**5.2
submitted+=1
hypot2 = x*x+y*y
```

BIBLIOGRAPHY

[1] Program-l: V.I. Programmers Discussion List, 2023.

[2] Ali Abdolrahmani, William Easley, Michele Williams, Stacy Branham, and Amy Hurst. Embracing errors: Examining how context of use impacts blind individuals' acceptance of navigation aid errors. In Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, CHI '17, page 4158–4169, New York, NY, USA, 2017. Association for Computing Machinery.

[3] Teach Access. Teach access. bridging the gap between accessibility and education, 2023.

[4] Krishan K Aggarwal, Yogesh Singh, and Jitender Kumar Chhabra. An integrated measure of software maintainability. In Annual Reliability and Maintainability Symposium. 2002 Proceedings (Cat. No. 02CH37318), pages 235–241. IEEE, 2002.

[5] Faisal Ahmed, Yevgen Borodin, Andrii Soviak, Muhammad Islam, IV Ramakrishnan, and Terri Hedgpeth. Accessible skimming: faster screen reading of web pages. In Proceedings of the 25th annual ACM symposium on User interface software and technology, pages 367–378, 2012.

[6] AirBnb. AirBnb react/JSX style guide, 2022.

[7] Khaled Albusays and Stephanie Ludi. Eliciting programming challenges faced by developers with visual impairments: Exploratory study. pages 82–85, 2016.

[8] Khaled Albusays, Stephanie Ludi, and Matt Huenerfauth. Interviews and observation of blind software developers at work to understand code navigation challenges. In Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility, pages 91–100, 2017.

[9] Steve Alexander. Blind programmers face an uncertain future. ComputerWorld, 32(44):86–87, 1998.

[10] Paul R Amato and Julie Saunders. The perceived dimensions of help-seeking episodes. Social Psychology Quarterly, pages 130–138, 1985.

[11] Apple. Accessibility - Vision - Apple. NV Access, 2022.

[12] Apple. Testing for accessibility on OS X, 2023.

[13] Ameer Armaly and Collin McMillan. An empirical study of blindness and program comprehension. In Proceedings of the 38th International Conference on Software Engineering Companion, pages 683–685, 2016.

[14] Ameer Armaly, Paige Rodeghero, and Collin McMillan. Audiohighlight: Code skimming for blind programmers. Proceedings - 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, pages 206–216, 2018.

[15] Ameer Armaly, Paige Rodeghero, and Collin McMillan. A comparison of program comprehension strategies by blind and sighted programmers. In Proceedings of the 40th International Conference on Software Engineering, pages 788–788, 2018.

[16] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In 2013 35th International Conference on Software Engineering (ICSE), pages 712–721. IEEE, 2013.

[17] Ronald Baecker. Enhancing program readability and comprehensibility with tools for program visualization. In Proceedings.[1989] 11th International Conference on Software Engineering, pages 356–357. IEEE Computer Society, 1988.

[18] Ronald M Baecker, Dimitrios Nastos, Ilona R Posner, and Kelly L Mawby. The user-centered iterative design of collaborative writing software. In Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems, pages 399–405, 1993.

[19] Catherine M Baker, Cynthia L Bennett, and Richard E Ladner. Educational Experiences of Blind Programmers. 2019.

[20] Catherine M Baker, Lauren R Milne, and Richard E Ladner. StructJumper: A Tool to Help Blind Programmers Navigate and Understand the Structure of Code. 2015.

[21] Mark S Baldwin, Sen H Hirano, Jennifer Mankoff, and Gillian R Hayes. Design in the public square: Supporting assistive technology design through public mixed-ability cooperation. Proceedings of the ACM on Human-Computer Interaction, 3(CSCW):1–22, 2019.

[22] Mark S Baldwin, Jennifer Mankoff, Bonnie Nardi, and Gillian Hayes. An activity centered approach to nonvisual computer interaction. ACM Transactions on Computer-Human Interaction (TOCHI), 27(2):1–27, 2020.

[23] Peter Bamberger. Employee help-seeking: Antecedents, consequences and new insights for future research. Research in personnel and human resources management, 28(1):49–98, 2009.

[24] Jennifer Bauer, Janet Siegmund, Norman Peitek, Johannes C Hofmeister, and Sven Apel. Indentation: simply a matter of style or support for program comprehension? In 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), pages 154–164. IEEE, 2019.

[25] Andrew Begel. Effecting change: Coordination in large-scale software development. In Proceedings of the 2008 international workshop on Cooperative and human aspects of software engineering, pages 17–20, 2008.

[26] Florian Beijers. How to get a developer job when you're blind: Advice from a blind developer who works alongside a sighted team, 2019.

[27] Cynthia L. Bennett, Erin Brady, and Stacy M. Branham. Interdependence as a Frame for Assistive Technology Research and Design. In Proceedings of the 20th International ACM SIGACCESS Conference on Computers and Accessibility - ASSETS '18, pages 161–173, New York, New York, USA, 2018. ACM Press.

[28] Cynthia L Bennett and Daniela K Rosner. The promise of empathy: Design, disability, and knowing the" other". In Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, pages 1–13, 2019.

[29] Cynthia L Bennett, Daniela K Rosner, and Alex S Taylor. The care work of access. In Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems, pages 1–15, 2020.

[30] Jacob T Biehl, Mary Czerwinski, Greg Smith, and George G Robertson. Fastdash: a visual dashboard for fostering awareness in software teams. In Proceedings of the SIGCHI conference on Human factors in computing systems, pages 1313–1322, San Jose, CA, 2007. ACM.

[31] Jeffrey P. Bigham, Anna C. Cavender, Jeremy T. Brudvik, Jacob O. Wobbrock, and Richard E. Ladner. Webinsitu: A comparative analysis of blind and sighted browsing behavior. In Proceedings of the 9th International ACM SIGACCESS Conference on Computers and Accessibility, Assets '07, page 51–58, New York, NY, USA, 2007. Association for Computing Machinery.

[32] Jeffrey P Bigham, Chandrika Jayant, Hanjie Ji, Greg Little, Andrew Miller, Robert C Miller, Robin Miller, Aubrey Tatarowicz, Brandyn White, Samual White, et al. Vizwiz: nearly real-time answers to visual questions. In Proceedings of the 23nd annual ACM symposium on User interface software and technology, pages 333–342, 2010.

[33] Jeffrey P Bigham, Irene Lin, and Saiph Savage. The effects of" not knowing what you don't know" on web accessibility for blind web users. In Proceedings of the 19th international ACM SIGACCESS conference on computers and accessibility, pages 101–109, 2017.

[34] Syed Masum Billah, Vikas Ashok, Donald E Porter, and IV Ramakrishnan. Ubiquitous accessibility for people with visual impairments: Are we there yet? In Proceedings of the 2017 CHI conference on human factors in computing systems, pages 5862–5868, Denver, CO, 2017. ACM.

[35] Syed Masum Billah, Donald E Porter, and IV Ramakrishnan. Sinter: Low-bandwidth remote access for the visually-impaired. In Proceedings of the Eleventh European Conference on Computer Systems, pages 1–16, London, UK, 2016. ACM.

[36] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I Maletic, Christopher Morrell, and Bonita Sharif. The impact of identifier style on effort and comprehension. Empirical software engineering, 18:219–276, 2013.

[37] Dave Binkley, Marcia Davis, Dawn Lawrie, and Christopher Morrell. To camelcase or under_score. In 2009 IEEE 17th International Conference on Program Comprehension, pages 158–167. IEEE, 2009.

[38] Jeremy Birnholtz, Nanyi Bi, and Susan Fussell. Do you see that I see? Effects of perceived visibility on awareness checking behavior. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pages 1765–1774, Austin, TX, 2012. ACM.

[39] Brianna Blaser, Cynthia Bennett, Richard E. Ladner, Sheryl E. Burgstahler, and Jennifer Mankoff. Perspectives of women with disabilities in computing, page 159–182. Cambridge University Press, 2019.

[40] Barry Boehm and Victor R Basili. Defect reduction top 10 list. Computer, 34(1):135–137, 2001.

[41] Andy Borka. Developer Toolkit, 2019.

[42] Jens Bornschein, Denise Bornschein, and Gerhard Weber. Blind pictionary: Drawing application for blind users. In Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems, pages 1–4, 2018.

[43] Erin L. Brady, Yu Zhong, Meredith Ringel Morris, and Jeffrey P. Bigham. Investigating the appropriateness of social network question asking as a resource for blind users. In Proceedings of the 2013 conference on Computer supported cooperative work - CSCW '13, page 1225, New York, New York, USA, 2013. ACM Press.

[44] Jet Brains. The state of developer ecosystem 2021, 2021.

[45] Stacy M. Branham and Shaun K. Kane. Collaborative accessibility: How blind and sighted companions co-create accessible home spaces. In Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, CHI '15, pages 2373–2382, New York, NY, USA, 2015. ACM.

[46] Stacy M. Branham and Shaun K. Kane. The invisible work of accessibility: How blind employees manage accessibility in mixed-ability workplaces. In Proceedings of the 17th International ACM SIGACCESS Conference on Computers & Accessibility, ASSETS '15, page 163–171, New York, NY, USA, 2015. Association for Computing Machinery.

[47] Stacy M. Branham and Shaun K. Kane. The Invisible Work of Accessibility. In Proceedings of the 17th International ACM SIGACCESS Conference on Computers & Accessibility - ASSETS '15, pages 163–171, New York, New York, USA, 2015. ACM Press.

[48] Eric Brechner. Things they would not teach me of in college: What Microsoft developers learn later. In Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 134–136, 2003.

[49] Robin N Brewer and Vaishnav Kameswaran. Understanding trust, transportation, and accessibility through ridesharing. In Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, pages 1–11, 2019.

[50] Sallyann Bryant, Pablo Romero, and Benedict du Boulay. Pair programming and the mysterious role of the navigator. International Journal of Human-Computer Studies, 66(7):519–529, 2008.

[51] Raymond PL Buse and Westley R Weimer. Learning a metric for code readability. IEEE Transactions on software engineering, 36(4):546–558, 2009.

[52] William Buxton. Mediaspace - meaningspace - meetingspace. In S. Harrison, editor, Media Space: 20+ Years of Mediated Life, pages 217–231. Springer, London, UK, 2009.

[53] William A. S. Buxton. Telepresence: Integrating shared task and person spaces. In Proceedings of the Conference on Graphics Interface '92, page 123–129, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[54] Mary Elaine Califf, Mary Goodwin, and Jake Brownell. Helping Him See: Guiding a Visually Impaired Student through the Computer Science Curriculum. 2008.

[55] Mitchell H Clifton. A technique for making structured programs more readable. ACM Sigplan Notices, 13(4):58–63, 1978.

[56] Code Factory. Eloquence for windows, 2021.

[57] Robert F Cohen, Alexander V Fairley, David Gerry, and Gustavo R Lima. Accessibility in introductory computer science. ACM SIGCSE Bulletin, 37(1):17–21, 2005.

[58] Juliet M Corbin and Anselm L Strauss. The articulation of work through interaction. The sociological quarterly, 34(1):71–83, 1993.

[59] Nicola Cornally and Geraldine McCarthy. Help-seeking behaviour: A concept analysis. International journal of nursing practice, 17(3):280–288, 2011.

[60] D Crary. Employer bias thwarts many blind workers. Associated Press, 2008.

[61] Sarah D'Angelo and Andrew Begel. Improving Communication Between Pair Programmers Using Shared Gaze Awareness, page 6245–6290. Association for Computing Machinery, New York, NY, USA, 2017.

[62] Maitraye Das, Darren Gergle, and Anne Marie Piper. "It doesn't win you friends": Understanding Accessibility in Collaborative Writing for People with Vision Impairments. Proceedings of the ACM on Human-Computer Interaction, 3(CSCW):1–26, 2019.

[63] Maitraye Das, Thomas B. McHugh, Anne Marie Piper, and Darren Gergle. Co11ab: Augmenting accessibility in synchronous collaborative writing for people with vision impairments. In Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems, CHI '22, pages 1–18, New York, NY, USA, 2022. Association for Computing Machinery.

[64] Maitraye Das, Anne Marie Piper, and Darren Gergle. Design and evaluation of accessible collaborative writing techniques for people with vision impairments. ACM Transactions on Computer-Human Interaction, 29(2):1–42, 2022.

[65] Lionel E Deimel Jr. The uses of program reading. ACM SIGCSE Bulletin, 17(2):5–14, 1985.

[66] Google Developers. Wizard (software).

[67] Google Developers. Improve your code with lint checks, 2023.

[68] Guy Dewsbury, Karen Clarke, Dave Randall, Mark Rouncefield, and Ian Sommerville. The anti-social model of disability. Disability & society, 19(2):145–158, 2004.

[69] Charles Dierbach. Python as a first programming language. Journal of Computing Sciences in Colleges, 29(3):73–73, 2014.

[70] Rodrigo Magalhães dos Santos and Marco Aurélio Gerosa. Impacts of coding practices on readability. In Proceedings of the 26th Conference on Program Comprehension, pages 277–285, 2018.

[71] Paul Dourish and Victoria Bellotti. Awareness and coordination in shared workspaces. In Proceedings of the 1992 ACM conference on Computer-supported cooperative work, pages 107–114, Toronto, ON, Canada, 1992. ACM.

[72] Carolyn D Egelman, Emerson Murphy-Hill, Elizabeth Kammer, Maggie Morrow Hodges, Collin Green, Ciera Jaspan, and James Lin. Pushback: Characterizing and detecting negative interpersonal interactions in code review. 2020.

[73] James L Elshoff and Michael Marcotty. Improving computer program readability to aid modification. Communications of the ACM, 25(8):512–521, 1982.

[74] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. The effect of poor source code lexicon and readability on developers' cognitive load. In Proceedings of the 26th Conference on Program Comprehension, pages 286–296, 2018.

[75] Hongfei Fan, Jiayao Gao, Hongming Zhu, Qin Liu, Yang Shi, and Chengzheng Sun. Balancing conflict prevention and concurrent work in real-time collaborative programming. In Proceedings of the 12th Chinese Conference on Computer Supported Cooperative Work and Social Computing, pages 217–220, 2017.

[76] Hongfei Fan, Chengzheng Sun, and Haifeng Shen. Atcope: Any-time collaborative programming environment for seamless integration of real-time and non-real-time teamwork in software development. In Proceedings of the 17th ACM International Conference on Supporting Group Work, GROUP '12, page 107–116, New York, NY, USA, 2012. Association for Computing Machinery.

[77] Open JS Foundation. Dojo. Open JS Foundation, 2022.

[78] Joan M Francioni and Ann C Smith. Computer science accessibility for students with visual disabilities. In Proceedings of the 33rd SIGCSE technical symposium on Computer science education, pages 91–95, 2002.

[79] Joan M Francioni and Ann C Smith. Computer science accessibility for students with visual disabilities. In Proceedings of the 33rd SIGCSE technical symposium on Computer science education, pages 91–95, 2002.

[80] Freedom Scientific. JAWS for Windows. Vispero, 2022.

[81] Freedom Scientific. ZoomText. Vispero, 2022.

[82] Vinitha Gadiraju. Brailleblocks: Braille toys for cross-ability collaboration. In The 21st International ACM SIGACCESS Conference on Computers and Accessibility, pages 688–690, 2019.

[83] H. Garfinkel. Studies in Ethnomethodology. Polity Press Bks. Wiley, 1991.

[84] David Garlan and Mary Shaw. An introduction to software architecture. In Advances in software engineering and knowledge engineering, pages 1–39. World Scientific, 1993.

[85] Github. Teletype for atom, 2022.

[86] Erving Goffman et al. The presentation of self in everyday life. Harmondsworth London, 1978.

[87] Max Goldman, Greg Little, and Robert C. Miller. Real-time collaborative coding in a web ide. In Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology, UIST '11, page 155–164, New York, NY, USA, 2011. Association for Computing Machinery.

[88] Google. Eslint shareable config for the google javascript style guide, 2022.

[89] Google. Flutter. Google, Mountain View, CA, 2022.

[90] Alvin W Gouldner. The norm of reciprocity: A preliminary statement. American sociological review, pages 161–178, 1960.

[91] Nancy Gourash. Help-seeking: A review of the literature. American journal of community psychology, 6(5):413, 1978.

[92] Robert Green and Henry Ledgard. Coding guidelines: Finding the art in the science. Communications of the ACM, 54(12):57–63, 2011.

[93] Geoff Greer and Matt Kaniaris. Floobits real-time collaboration plugin for sublime text 2 and 3, 2020.

[94] Carl Gutwin and Saul Greenberg. A Descriptive Framework of Workspace Awareness for Real-Time Groupware. Computer Supported Cooperative Work (CSCW), 11(3):411–446, September 2002.

[95] Sarra Habchi, Xavier Blanc, and Romain Rouvoy. On adopting linters to deal with performance concerns in android apps. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pages 6–16, 2018.

[96] Nuzhat J Haneef. Software documentation and readability: a proposed process improvement. ACM SIGSOFT Software Engineering Notes, 23(3):75–77, 1998.

[97] Fredrik Hansen, Josef Jan Krivan, and Frode Eika Sandnes. Still not readable? an interactive tool for recommending color pairs with sufficient contrast based on existing visual designs. In The 21st International ACM SIGACCESS Conference on Computers and Accessibility, pages 636–638, 2019.

[98] Lile Hattori and Michele Lanza. Syde: A tool for collaborative software development. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2, pages 235–238, Cape Town, South Africa, 2010. ACM/IEEE.

[99] Rajesh Hegde and Prasun Dewan. Connecting programming environments to support ad-hoc collaboration. In 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, pages 178–187. IEEE, 2008.

[100] Henning Heitkötter, Sebastian Hanschke, and Tim A Majchrzak. Evaluating cross-platform development approaches for mobile applications. In International Conference on Web Information Systems and Technologies, pages 120–138. Springer, 2012.

[101] Scott Highhouse. Designing experiments that generalize. Organizational Research Methods, 12(3):554–566, 2009.

[102] Ric Holt. Software architecture as a shared mental model. Proceedings of the ASERC Workhop on Software Architecture, University of Alberta, page 64, 2002.

[103] Earl W. Huff, Kwajo Boateng, Makayla Moster, Paige Rodeghero, and Julian Brinkley. Examining the work experience of programmers with visual impairments. In 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 707–711, Online, 2020. IEEE.

[104] Joe Hutchinson and Oussama Metatla. An initial investigation into non-visual code structure overview through speech, non-speech and spearcons. In Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems, pages 1–6, 2018.

[105] Jennifer Hyde, Sara Kiesler, Jessica K Hodgins, and Elizabeth J Carter. Conversing with children: Cartoon and video people elicit similar conversational behaviors. In Proceedings of the SIGCHI conference on human factors in computing systems, pages 1787–1796, Toronto, ON, Canada, 2014. ACM.

[106] Hiroshi Ishii and Naomi Miyake. Toward an open shared workspace: computer and video fusion approach of teamworkstation. Communications of the ACM, 34(12):37–50, 1991.

[107] D. Izquierdo, N. Huesman, A. Serebrenik, and G. Robles. Openstack gender diversity report. IEEE Software, 36(1):28–33, 2019.

[108] JetBrains. Meet code with me (eap) — a tool for collaborative development by jetbrains, 2020.

[109] Sasa Junuzovic, Prasun Dewan, and Yong Rui. Read, write, and navigation awareness in realistic multi-view collaborations. In 2007 International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2007), pages 494–503, New York, NY, 2007. IEEE.

[110] Vaishnav Kameswaran, Jatin Gupta, Joyojeet Pal, Sile O'Modhrain, Tiffany C Veinot, Robin Brewer, Aakanksha Parameshwar, and Jacki O'Neill. 'we can go anywhere' understanding independence through a case study of ride-hailing use by people with visual impairments in metropolitan india. Proceedings of the ACM on Human-Computer Interaction, 2(CSCW):1–24, 2018.

[111] Claire Kearney-Volpe, Chancey Fleet, Keita Ohshiro, Veronica Alfaro Arias, and Amy Hurst. Making the elusive more tangible: remote tools & techniques for teaching web development to screen reader users. In Proceedings of the 18th International Web for All Conference, pages 1–14, 2021.

[112] Claire Kearney-Volpe and Amy Hurst. Accessible web development: Opportunities to improve the education and practice of web development with a screen reader. ACM Transactions on Accessible Computing (TACCESS), 14(2):1–32, 2021.

[113] Claire Kearney-Volpe, Devorah Kletenik, Kate Sonka, Deborah Sturm, and Amy Hurst. Evaluating instructor strategy and student learning through digital accessibility course enhancements. In Proceedings of the 21st International ACM SIGACCESS Conference on Computers and Accessibility, ASSETS '19, page 377–388, New York, NY, USA, 2019. Association for Computing Machinery.

[114] Niall Kennedy. Google mondrian: web-based code review and storage, Dec 2006.

[115] Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. The emerging role of data scientists on software development teams. In 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pages 96–107. IEEE, 2016.

[116] Tien Fabrianti Kusumasari, Iping Supriana, Kridanto Surendro, and Husni Sastramihardja. Collaboration model of software development. In Proceedings of the 2011 International Conference on Electrical Engineering and Informatics, pages 1–6. IEEE, 2011.

[117] Richard E Ladner and Kyle Rector. Making your presentation accessible. Interactions, 24(4):56–59, 2017.

[118] Cheuk Yin Phipson Lee, Zhuohao Zhang, Jaylin Herskovitz, JooYoung Seo, and Anhong Guo. CollabAlly: Accessible Collaboration Awareness in Document Editing, pages 1–4. Association for Computing Machinery, New York, NY, USA, 2021.

[119] Cheuk Yin Phipson Lee, Zhuohao Zhang, Jaylin Herskovitz, JooYoung Seo, and Anhong Guo. Collabally: Accessible collaboration awareness in document editing. In Proceedings

of the 2022 CHI Conference on Human Factors in Computing Systems, CHI '22, pages 1–17, New York, NY, USA, 2022. Association for Computing Machinery.

[120] Franklin Mingzhe Li, Di Laura Chen, Mingming Fan, and Khai N. Truong. "i choose assistive devices that save my face": A study on perceptions of accessibility and assistive technology use conducted in china. In Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems, CHI '21, New York, NY, USA, 2021. Association for Computing Machinery.

[121] Jingyi Li, Son Kim, Joshua A Miele, Maneesh Agrawala, and Sean Follmer. Editing spatial layouts through tactile templates for people with visual impairments. In Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, pages 1–11, 2019.

[122] Michael Longley and Yasmine N Elglaly. Accessibility support in web frameworks. In The 23rd International ACM SIGACCESS Conference on Computers and Accessibility, pages 1–4, 2021.

[123] Tom Love. An experimental investigation of the effect of program structure on program understanding. ACM SIGSOFT Software Engineering Notes, 2(2):105–113, 1977.

[124] JGraph Ltd. Diagrams for confluence and jira, Dec 2022.

[125] Lucidchart. Intelligent diagramming, 2022.

[126] Stephanie Ludi. Access for everyone: introducing accessibility issues to students in internet programming courses. In 32nd Annual Frontiers in Education, volume 3, pages S1C–S1C, New Jersey, NJ, USA, 2002. IEEE.

[127] Stephanie Ludi, Jamie Simpson, and Wil Merchant. Exploration of the use of auditory cues in code comprehension and navigation for individuals with visual impairments in a visual programming environment. In Proceedings of the 18th International ACM SIGACCESS Conference on Computers and Accessibility, ASSETS '16, page 279–280, New York, NY, USA, 2016. Association for Computing Machinery.

[128] Sergio Mascetti, Mattia Ducci, Niccoló Cantù, Paolo Pecis, and Dragan Ahmetovic. Developing accessible mobile applications with cross-platform development frameworks. In Proceedings of the 23rd International ACM SIGACCESS Conference on Computers and Accessibility, pages 1–5, 2021.

[129] Reeti Mathur and Erin Brady. Mixed-ability collaboration for accessible photo sharing. In Proceedings of the 20th International ACM SIGACCESS Conference on Computers and Accessibility, pages 370–372, 2018.

[130] Sean Mealin and Emerson Murphy-Hill. An exploratory study of blind software developers. In Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC, pages 71–74, 2012.

[131] Meta. React Native. Meta, Palo Alto, CA, 2022.

[132] Oussama Metatla, Nick Bryan-Kinns, and Tony Stockman. "i hear you": Understanding awareness information exchange in an audio-only workspace. In Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, CHI '18, page 1–13, New York, NY, USA, 2018. Association for Computing Machinery.

[133] Oussama Metatla, Alison Oldfield, Taimur Ahmed, Antonis Vafeas, and Sunny Miglani. Voice user interfaces in schools: Co-designing for inclusion with visually-impaired and sighted pupils. In Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, pages 1–15, 2019.

[134] Richard J. Miara, Joyce A. Musselman, Juan A. Navarro, and Ben Shneiderman. Program indentation and comprehensibility. Commun. ACM, 26(11):861–867, November 1983.

[135] Microsoft. Accessibility in visual studio code, 2020.

[136] Microsoft. Microsoft/vscode-a11y - gitter, 2020.

[137] Microsoft. Visual studio live share, 2022.

[138] Microsoft. Xamarin. Microsoft, Redmond, WA, 2022.

[139] Microsoft. Flowchart maker and diagramming software: Microsoft visio, 2023.

[140] Matthew Miles, A. Michael Huberman, and Michael Saldaña. Qualitative Data Analysis: A Methods Sourcebook. Sage Publications, Thousand Oaks, CA, 2013.

[141] Jonas Moll and Eva-Lotta Sallnäs Pysander. A haptic tool for group work on geometrical concepts engaging blind and sighted pupils. ACM Transactions on Accessible Computing (TACCESS), 4(4):1–37, 2013.

[142] Jonas Moll, Kerstin Severinson-Eklundh, and Eva-Lotta Sallnas. Group work about geometrical concepts among blind and sighted pupils using haptic interfaces. In 2007 2nd Joint EuroHaptics Conference and Symposium on Haptic Interfaces for Virtual Environments and Teleoperator Systems, pages 330–335, 2007.

[143] Cecily Morrison, Edward Cutrell, Anupama Dhareshwar, Kevin Doherty, Anja Thieme, and Alex Taylor. Imagining artificial intelligence applications with people with visual disabilities using tactile ideation. In Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility, pages 81–90, 2017.

[144] Janice M Morse. The significance of saturation, 1995.

[145] Mozilla. Accessibility Tree - MDN Web Docs Glossary: Definitions of Web-related terms, 2023.

[146] Mozilla. Aria - Accessibility: MDN, 2023.

[147] Arie Nadler, Shmuel Ellis, and Iris Bar. To seek or not to seek: The relationship between help seeking and job performance evaluations as moderated by task-relevant expertise. Journal of Applied Social Psychology, 33(1):91–109, 2003.

[148] Mala D Naraine and Peter H Lindsay. Social inclusion of employees who are blind or low vision. Disability & Society, 26(4):389–403, 2011.

[149] Katja Neureiter, Martin Murer, Verena Fuchsberger, and Manfred Tscheligi. Hand and eyes: How eye contact is linked to gestures in video conferencing. In CHI'13 Extended Abstracts on Human Factors in Computing Systems, pages 127–132. ACM, Paris, France, 2013.

[150] Austin Lee Nichols and Jon K Maner. The good-subject effect: Investigating participant demand characteristics. The Journal of general psychology, 135(2):151–166, 2008.

[151] Kirk Norman, Yevgeniy Arber, and Ravi Kuber. How accessible is the process of web interface design? In Proceedings of the 15th International ACM SIGACCESS Conference on Computers and Accessibility, pages 1–2, 2013.

[152] John T Nosek. The case for collaborative programming. Communications of the ACM, 41(3):105–108, 1998.

[153] NV Access. Nonvisual Desktop Access. NV Access, 2022.

[154] Delano Oliveira, Reydne Santos, Fernanda Madeiral, Hidehiko Masuhara, and Fernando Castor. A systematic literature review on the impact of formatting elements on code legibility. Journal of Systems and Software, page 111728, 2023.

[155] Mike Oliver. The social model of disability: Thirty years on. Disability & society, 28(7):1024–1026, 2013.

[156] Shotaro Omori and Ikuko Eguchi Yairi. Collaborative music application for visually impaired people with tangible objects on table. In Proceedings of the 15th International ACM SIGACCESS Conference on Computers and Accessibility, pages 1–2, 2013.

[157] Steve Oney, Alan Lundgard, Rebecca Krosnick, Michael Nebeling, and Walter S Lasecki. Arboretum and arbility: Improving web accessibility through a shared browsing architecture. In Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology, pages 937–949, 2018.

[158] Stack Overflow. Stack overflow developer survey 2021, 2021.

[159] Stack Overflow. Stack Overflow Developer Survey 2022, 2022.

[160] Maulishree Pandey, Sharvari Bondre, Sile O'Modhrain, and Steve Oney. Accessibility of ui frameworks and libraries for programmers with visual impairments. In 2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pages 1–10. IEEE, 2022.

[161] Maulishree Pandey and Tao Dong. Blending accessibility in ui framework documentation to build awareness. In To appear in Proceedings of the 25th International ACM SIGACCESS Conference on Computers and Accessibility, 2023.

[162] Maulishree Pandey, Vaishnav Kameswaran, Hrishikesh V Rao, Sile O'Modhrain, and Steve Oney. Understanding accessibility and collaboration in programming for people with visual impairments. Proceedings of the ACM on Human-Computer Interaction, 5(CSCW1):1–30, 2021.

[163] Maulishree Pandey, Hariharan Subramonyam, Brooke Sasia, Steve Oney, and Sile O'Modhrain. Explore, Create, Annotate: Designing Digital Drawing Tools with Visually Impaired People, page 1–12. Association for Computing Machinery, New York, NY, USA, 2020.

[164] Yi-Hao Peng, JiWoong Jang, Jeffrey P Bigham, and Amy Pavel. Say it all: Feedback for improving non-visual presentation accessibility. In Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems, CHI '21, New York, NY, USA, 2021. Association for Computing Machinery.

[165] Dewayne E Perry, Nancy A. Staudenmayer, and Lawrence G Votta. People, organizations, and process improvement. IEEE Software, 11(4):36–45, 1994.

[166] Vanessa Petrausch and Claudia Loitsch. Accessibility Analysis of the Eclipse IDE for Users with Visual Impairment. 2017.

[167] Vanessa Petrausch and Claudia Loitsch. Accessibility analysis of the eclipse ide for users with visual impairment. In Harnessing the Power of Technology to Improve Lives, pages 922–929. IOS Press, 2017.

[168] Venkatesh Potluri, Tad Grindeland, Jon E. Froehlich, and Jennifer Mankoff. Examining visual semantic understanding in blind and low-vision technology users. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '21, New York, NY, USA, 2021. Association for Computing Machinery.

[169] Venkatesh Potluri, Tadashi Grindeland, Jon E Froehlich, and Jennifer Mankoff. Ai-assisted ui design for blind and low-vision creators. In the ASSETS'19 Workshop: AI Fairness for People with Disabilities, 2019.

[170] Venkatesh Potluri, Tadashi E Grindeland, Jon E Froehlich, and Jennifer Mankoff. Examining visual semantic understanding in blind and low-vision technology users. In Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems, pages 1–14, 2021.

[171] Venkatesh Potluri, Liang He, Christine Chen, Jon E Froehlich, and Jennifer Mankoff. A multi-modal approach for blind and visually impaired developers to edit webpage designs. In The 21st International ACM SIGACCESS Conference on Computers and Accessibility, pages 612–614, 2019.

[172] Venkatesh Potluri, Maulishree Pandey, Andrew Begel, Michael Barnett, and Scott Reitherman. Codewalk: Facilitating shared awareness in mixed-ability collaborative software development. In Proceedings of the 24th International ACM SIGACCESS Conference on Computers and Accessibility, pages 1–16, 2022.

[173] Venkatesh Potluri, Priyan Vaithilingam, Suresh Iyengar, Y Vidya, Manohar Swaminathan, and Gopal Srinivasa. Codetalk: Improving programming environment accessibility for visually impaired developers. In Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, pages 1–11, 2018.

[174] Venkatesh Potluri, Priyan Vaithilingam, Suresh Iyengar, Y. Vidya, Manohar Swaminathan, and Gopal Srinivasa. Codetalk: Improving programming environment accessibility for visually impaired developers. In Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, CHI '18, page 1–11, New York, NY, USA, 2018. Association for Computing Machinery.

[175] Venkatesh Potluri, Priyan Vaithilingam, Suresh Iyengar, Y Vidya, Manohar Swaminathan, and Gopal Srinivasa. Codetalk: Improving programming environment accessibility for visually impaired developers. In Proceedings of the 2018 chi conference on human factors in computing systems, pages 1–11, 2018.

[176] T. V. Raman. Emacspeak—direct speech access. Assets '96, page 32–36, New York, NY, USA, 1996. Association for Computing Machinery.

[177] Darrell R Raymond. Reading source code. In CASCON, volume 91, pages 3–16, 1991.

[178] Phillip A Relf. Tool assisted identifier naming for improved software readability: an empirical study. In 2005 International Symposium on Empirical Software Engineering, 2005., pages 10–pp. IEEE, 2005.

[179] Gema Rodríguez-Pérez, Reza Nadri, and Meiyappan Nagappan. Perceived diversity in software engineering: a systematic literature review. Empirical Software Engineering, 26(5):1–38, 2021.

[180] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: a case study at google. In Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, pages 181–190, 2018.

[181] Johnny Saldaña. The coding manual for qualitative researchers. Sage, 2015.

[182] Stephan Salinger, Christopher Oezbek, Karl Beecher, and Julia Schenk. Saros: an eclipse plug-in for distributed party programming. In Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering, pages 48–55, Cape Town, South Africa, 2010. ACM/IEEE.

[183] Isabel Braga Sampaio and Luís Barbosa. Software readability practices and the importance of their teaching. In 2016 7th International Conference on Information and Communication Systems (ICICS), pages 304–309. IEEE, 2016.

[184] Harini Sampath, Alice Merrick, and Andrew Macvean. Accessibility of command line interfaces. In Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems, pages 1–10, 2021.

[185] Jaime Sánchez and Fernando Aguayo. Blind learners programming through audio. In CHI'05 extended abstracts on Human factors in computing systems, pages 1769–1772, 2005.

[186] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. Automatically assessing code understandability: How far are we? In 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 417–427. IEEE, 2017.

[187] Emmanuel Schanzer, Sina Bahram, and Shriram Krishnamurthi. Accessible AST-Based Programming for Visually-Impaired Programmers. In Proceedings of the 50th ACM Technical Symposium on Computer Science Education - SIGCSE '19, pages 773–779, New York, New York, USA, 2019. ACM Press.

[188] Freedom Scientific. JAWS Tandem Quick Start Guide, 2022.

[189] Chirag Shah and Gary Marchionini. Awareness in collaborative information seeking. Journal of the American Society for Information Science and Technology, 61(10):1970–1986, 2010.

[190] Bonita Sharif and Jonathan I Maletic. An eye tracking study on camelcase and under_score identifier styles. In 2010 IEEE 18th International Conference on Program Comprehension, pages 196–205. IEEE, 2010.

[191] Helen Sharp, Robert Biddle, Phil Gray, Lynn Miller, and Jeff Patton. Agile development: Opportunity or fad? In CHI '06 Extended Abstracts on Human Factors in Computing Systems, CHI EA '06, page 32–35, New York, NY, USA, 2006. Association for Computing Machinery.

[192] Ashrith Shetty, Ebrima Jarjue, and Huaishu Peng. Tangible web layout design for blind and visually impaired people: An initial investigation. In Adjunct Publication of the 33rd Annual ACM Symposium on User Interface Software and Technology, pages 37–39, 2020.

[193] Kristen Shinohara, Cynthia L. Bennett, Wanda Pratt, and Jacob O. Wobbrock. Tenets for social accessibility: Towards humanizing disabled people in design. ACM Trans. Access. Comput., 11(1):6:1–6:31, March 2018.

[194] Kristen Shinohara and Jacob O Wobbrock. In the shadow of misperception: assistive technology use and social interactions. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pages 705–714, 2011.

[195] Kristen Shinohara and Jacob O. Wobbrock. Self-conscious or self-confident? a diary study conceptualizing the social accessibility of assistive technology. ACM Trans. Access. Comput., 8(2):5:1–5:31, January 2016.

[196] Kristen Shinohara and Jacob O Wobbrock. Self-conscious or self-confident? a diary study conceptualizing the social accessibility of assistive technology. ACM Transactions on Accessible Computing (TACCESS), 8(2):1–31, 2016.

[197] Ben Shneiderman and Don McKay. Experimental investigations of computer program debugging and modification. In Proceedings of the Human Factors Society Annual Meeting, volume 20, pages 557–563. SAGE Publications Sage CA: Los Angeles, CA, 1976.

[198] Robert M Siegfried. Visual Programming and the Blind : The Challenge and the Opportunity. Science Education, pages 275–278, 2006.

[199] Janet Siegmund, Norman Peitek, Chris Parnin, Sven Apel, Johannes Hofmeister, Christian Kästner, Andrew Begel, Anja Bethmann, and André Brechmann. Measuring neural efficiency of program comprehension. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pages 140–150, 2017.

[200] Charles Simonyi. Hungarian notation. MSDN Library, November, 1999.

[201] Alexa Siu, Gene S-H Kim, Sile O'Modhrain, and Sean Follmer. Supporting accessible data visualization through audio data narratives. In Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems, CHI '22, New York, NY, USA, 2022. Association for Computing Machinery.

[202] Darja Šmite, Nils Brede Moe, and Richard Torkar. Pitfalls in remote team coordination: Lessons learned from a case study. In International Conference on Product Focused Software Process Improvement, pages 345–359. Springer, 2008.

[203] Ann C Smith, Justin S Cook, Joan M Francioni, Asif Hossain, Mohd Anwar, and M Fayezur Rahman. Nonvisual tool for navigating hierarchical structures. ACM SIGACCESS Accessibility and Computing, (77-78):133–139, 2003.

[204] Diomidis Spinellis. Reading, writing, and code: The key to writing readable code is developing good coding style. Queue, 1(7):84–89, 2003.

[205] Stack Overflow. Stack Overflow Developer Survey 2020, 2020.

[206] Stack Overflow. Stack overflow developer survey 2023, 2023.

[207] Susan Leigh Star and Anselm Strauss. Layers of Silence, Arenas of Voice: The Ecology of Visible and Invisible Work. 1999.

[208] A Stefik. On the design of program execution environments for non-sighted computer programmers. Undefined, 2008.

[209] Andreas Stefik, Andrew Haywood, Shahzada Mansoor, Brock Dunda, and Daniel Garcia. Sodbeans. In 2009 IEEE 17th International Conference on Program Comprehension, pages 293–294. IEEE, 2009.

[210] Andreas Stefik and Susanna Siebert. An empirical investigation into programming language syntax. ACM Transactions on Computing Education (TOCE), 13(4):1–40, 2013.

[211] Andreas M. Stefik, Christopher Hundhausen, and Derrick Smith. On the design of an educational infrastructure for the blind and visually impaired in computer science. In Proceedings of the 42nd ACM Technical Symposium on Computer Science Education, SIGCSE '11, page 571–576, New York, NY, USA, 2011. Association for Computing Machinery.

[212] Kevin M. Storer, Harini Sampath, and M. Alice Merrick. "it's just everything outside of the ide that's the problem": Information seeking by software developers with visual impairments. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '21, New York, NY, USA, May 2021. Association for Computing Machinery.

[213] Sublime users. A request for the implementation of accessibility. issue #3392. sublimehq/-sublime_text, 2020.

[214] Floyd Sykes, Raymond T Tillman, and Ben Shneiderman. The effect of scope delimiters on program comprehension. Software: Practice and Experience, 13(9):817–824, 1983.

[215] John Tang. Understanding the telework experience of people with disabilities. Proc. ACM Hum.-Comput. Interact., 5(CSCW1), apr 2021.

[216] Yahya Tashtoush, Zeinab Odat, Izzat M Alsmadi, and Maryan Yatim. Impact of programming features on code readability. International Journal of Software Engineering and Its Applications, 2013.

[217] Bootstrap Core Team. Bootstrap, 2022.

[218] Josh Tenenberg, Wolff-Michael Roth, and David Socha. From I-Awareness to We-Awareness in CSCW. Computer Supported Cooperative Work (CSCW), 25(4):235–278, October 2016.

[219] Ted Tenny. Program readability: Procedures versus comments. IEEE Transactions on Software Engineering, 14(9):1271–1279, 1988.

[220] Jim Thatcher. Screen reader/2—programmed access to the gui. In International Conference on Computers for Handicapped Persons, pages 76–88. Springer, 1994.

[221] The Apache Software Foundation. Cordova. Apache, 2022.

[222] The GNOME Project. Orca Screen Reader. The GNOME Project, 2022.

[223] Anja Thieme, Cynthia L. Bennett, Cecily Morrison, Edward Cutrell, and Alex S. Taylor. "I can do everything but see!" – How People with Vision Impairments Negotiate their Abilities in Social Contexts. In Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI '18, pages 1–14, New York, New York, USA, 2018. ACM Press.

[224] Anja Thieme, Cecily Morrison, Nicolas Villar, Martin Grayson, and Siân Lindley. Enabling collaboration in learning computer programing inclusive of children with vision impairments. In Proceedings of the 2017 Conference on Designing Interactive Systems, pages 739–752, 2017.

[225] Christopher Toth and Tyler Spivey. Documentation nvda remote access, Sep 2018.

[226] Alexia Tsotsis. Meet phabricator, the witty code review tool built inside facebook, Aug 2011.

[227] Andries Van Dam. Post-wimp user interfaces. Communications of the ACM, 40(2):63–67, 1997.

[228] Janine van der Rijt, Piet Van den Bossche, Margje WJ van de Wiel, Sven De Maeyer, Wim H Gijselaers, and Mien SR Segers. Asking for help: A relational perspective on help seeking in the workplace. Vocations and learning, 6(2):259–279, 2013.

[229] Guido van Rossum, Nick Coghlan, and Barry Warsaw. Pep 8 – style guide for python code, Jul 2001.

[230] Bogdan Vasilescu, Andrea Capiluppi, and Alexander Serebrenik. Gender, representation and online participation: A quantitative study. Interacting with Computers, 26(5):488–511, 2014.

[231] Herman Wahidin, Jenny Waycott, and Steven Baker. The challenges in adopting assistive technologies in the workplace for people with visual impairments. In Proceedings of the 30th Australian Conference on Computer-Human Interaction, OzCHI '18, page 432–442, New York, NY, USA, 2018. Association for Computing Machinery.

[232] Annalu Waller, Vicki L. Hanson, and David Sloan. Including accessibility within and beyond undergraduate computing courses. In Proceedings of the 11th International ACM SIGACCESS Conference on Computers and Accessibility, Assets '09, page 155–162, New York, NY, USA, 2009. Association for Computing Machinery.

[233] April Yi Wang, Anant Mittal, Christopher Brooks, and Steve Oney. How data scientists use computational notebooks for real-time collaboration. Proceedings of the ACM on Human-Computer Interaction, 3(CSCW):1–30, 2019.

[234] Emily Q. Wang and Anne Marie Piper. Accessibility in action: Co-located collaboration among deaf and hearing professionals. Proc. ACM Hum.-Comput. Interact., 2(CSCW), November 2018.

[235] Xiaoran Wang, Lori Pollock, and K Vijay-Shanker. Automatic segmentation of method code into meaningful blocks to improve readability. In 2011 18th Working Conference on Reverse Engineering, pages 35–44. IEEE, 2011.

[236] Jeremy Warner and Philip J Guo. Codepilot: Scaffolding end-to-end collaborative software development for novice programmers. In Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, pages 1136–1141, Denver, CO, 2017. ACM.

[237] WebAccessibilizer. Bootstrap, 2022.

[238] Aaron Whyte, Bob Jervis, Dan Pupius, Erik Arvidsson, Fritz Schneider, and Robby Walker. Google javascript style guide.

[239] Michele A Williams, Caroline Galbraith, Shaun K Kane, and Amy Hurst. "just let the cane hit it"' how the blind and sighted see navigation differently. In Proceedings of the 16th international ACM SIGACCESS conference on Computers & accessibility, pages 217–224, 2014.

[240] Judith D Wilson, Nathan Hoskin, and John T Nosek. The benefits of collaboration for student programmers. ACM SIGCSE Bulletin, 25(1):160–164, 1993.

[241] Jacob O Wobbrock, Shaun K Kane, Krzysztof Z Gajos, Susumu Harada, and Jon Froehlich. Ability-based design: Concept, principles and examples. ACM Transactions on Accessible Computing (TACCESS), 3(3):1–27, 2011.

[242] Jacob O Wobbrock and Matthew Kay. Nonparametric statistics in human–computer interaction. In Modern Statistical Methods for HCI, pages 135–170. Springer, Berlin, Germany, 2016.

[243] Chien Wen Yuan, Benjamin V Hanrahan, Sooyeon Lee, Mary Beth Rosson, and John M Carroll. I didn't know that you knew i knew: Collaborative shopping practices between people with visual impairment and people with vision. Proceedings of the ACM on Human-Computer Interaction, 1(CSCW):1–18, 2017.

[244] Yuhang Zhao, Shaomei Wu, Lindsay Reynolds, and Shiri Azenkot. The effect of computer-generated descriptions on photo-sharing experiences of people with visual impairments. Proceedings of the ACM on Human-Computer Interaction, 1(CSCW):1–22, 2017.

[245] Yanyan Zhuang, Jennifer Baldwin, Laura Antunna, Yagiz Onat Yazir, Sudhakar Ganti, and Yvonne Coady. Tradeoffs in cross platform solutions for mobile assistive technology. In 2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM), pages 330–335. IEEE, 2013.

[246] Annuska Zolyomi, Anushree Shukla, and Jaime Snyder. Technology-mediated sight: A case study of early adopters of a low vision assistive technology. In Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility, ASSETS '17, page 220–229, New York, NY, USA, 2017. Association for Computing Machinery.

[247] Silvia Zuffi, Carla Brambilla, Giordano Beretta, and Paolo Scala. Human computer interaction: Legibility and contrast. In 14th International Conference on Image Analysis and Processing (ICIAP 2007), pages 241–246. IEEE, 2007.