Accessibility of Command Line Interfaces

Harini Sampath
Google Inc.
Seattle
harinis@google.com

Alice Merrick
Google Inc.
Seattle
amerrick@google.com

Andrew Macvean
Google Inc.
Seattle
amacvean@google.com

ABSTRACT

Command-line interfaces (CLIs) remain a popular tool among developers and system administrators. Since CLIs are text-based interfaces, they are sometimes considered accessible alternatives to predominantly visual developer tools like IDEs. However, there is no systematic evaluation of the accessibility of CLIs in the literature. In this paper, we describe two studies with 12 developers on their experience of using CLIs with screen readers. Our findings show that CLIs have their own set of accessibility issues - the most important being CLIs are unstructured text interfaces. Based on our results, we provide a set of recommendations for improving the accessibility of command-line interfaces.

CCS CONCEPTS

• **Human-centered computing → Command line interfaces**; **Empirical studies in accessibility**.

KEYWORDS

accessibility, command-line interfaces, screen reader users, developers with visual impairments

ACM Reference Format:
Harini Sampath, Alice Merrick, and Andrew Macvean. 2021. Accessibility of Command Line Interfaces. In *CHI Conference on Human Factors in Computing Systems (CHI '21), May 8–13, 2021, Yokohama, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3411764.3445544

1 INTRODUCTION

Though command-line interfaces (CLIs) are less popular end-user interfaces than GUIs, they remain an important tool for developers, and system administrators [4, 20]. For these technical users, CLIs enable more efficient interactions than GUIs and allow them to create custom workflows through scripting [20]. Many popular developer tools (e.g., git, docker) and almost all major cloud providers offer command-line interfaces (e.g., GCP. AWS, Azure, Digital Ocean) for their technical users.

Text is the primary mode of interaction with CLIs. Users input commands, which are lines of text that adhere to a specific syntax and receive a response from the command, which is also in the form of lines of text. Interaction with a CLI is mostly through the keyboard.

Both these qualities - a) being a text-based interface and b) being an interface that can be accessed entirely through a keyboard - make CLIs an attractive option in terms of accessibility for developers with visual impairments (DWVI). In literature around the accessibility of developer tools and personal accounts of programmers with visual impairments, CLIs are often considered accessible alternatives to other visual developer tools (e.g., IDEs) [7, 22, 25].

Perhaps due to this assumption that CLIs are inherently more accessible, there has been no systematic evaluation of their accessibility to date. While being a text- and keyboard-based interface contributes to improved accessibility compared to graphical user interfaces, this does not necessarily translate into a completely accessible user experience. For instance, being text-based and keyboard-operable would meet only a small subset of the WCAG [5] criteria.

In the research described in this paper, our goal is to understand if command-line interfaces are accessible to developers with visual impairments (DWVI) who use screen readers. We describe two studies with DWVI who use CLIs and describe their experience of using CLIs with screen readers. We conclude by articulating a set of recommendations for building accessible CLIs.

2 RELATED RESEARCH

2.1 Accessibility of Programming

In a recent survey, around 1.5 % [23] of software developers self-report as having some form of visual impairment. However, many modern programming tools are visual (e.g., IDEs, code review tools and source code repositories). While programming is not inherently a visual activity, it has been argued that the advent of visual tools has made it inaccessible to individuals with visual impairments [16, 28].

Existing research into making programming accessible has investigated multiple facets of the problem like - a) understanding the needs of developers with visual impairments [1, 19] , b) building accessible developer tools [3, 25, 31], c) making programming education accessible [10, 17], d) making visual artifacts associated with programming (e.g., UML diagrams) accessible [9, 15].

In this paper, we review two particular areas of literature most relevant to our research on CLI accessibility - i) understanding the needs of developers with visual impairments and ii) building accessible developer tools. When applicable, we also highlight the assumption that CLIs are accessible alternatives to visual programming tools.

2.1.1 Understanding needs of developers with visual impairments. Research into understanding the needs of DWVI has sought to identify the issues developers face and the strategies and workarounds they use while programming [1, 19].

In interviews with eight software developers, Mealin et al. [19] found that DWVI use a variety of assistive technologies in their work, rely heavily on API documentation to get an overview of the code structure, and feel they excel at tasks that do not require vision like algorithm design. They also note the lack of development tool usage by DWVI - most of them used text editors instead of IDEs, *printf* debugging instead of debuggers, used temporary text buffers for editing code outside the IDE, and preferred languages like Python primarily due to the availability of the interpreter. Albusays et al. [1] conducted a survey of 69 blind developers and a follow-up study with more in-depth observations [2]. The most frequent issues found were around IDE accessibility, debugging, and code navigation. Common workarounds included *printf* debugging and usage of text editors instead of complex IDEs.

It is interesting to note that many of the workarounds used, such as text editors, *printf* debugging, and using a Python interpreter, all require developers to interact with a command-line interface.

2.1.2 Building accessible developer tools. Writing, comprehending, and debugging code are canonical developer tasks [29]. There have been multiple efforts in literature to make these experiences accessible to DWVI by building accessible developer tools.

Making code structure accessible: Many cues about the structure of source code like indentation and nesting hierarchies are encoded visually, making it difficult for DWVI to comprehend and navigate code with screen readers efficiently. SructJumper [3], is an Eclipse plugin that creates a tree-based representation based on the hierarchical structure of source code. Developers can use this tree view to comprehend the code's high-level structure and switch to a text editor to get the details. Schanzer et al. [27] extend this approach of making the code structure accessible for multiple programming languages with a language-independent, cloud-based toolkit. The toolkit generates a block editor and audio descriptions for the code structure. Codetalk [25], is a Visual Studio plugin that aims to make multiple facets of IDE and thus software development accessible like code comprehension, debugging, and working in a team. In their formative studies, the authors found that DWVI use CLIs and a text editor as an alternative to using IDEs.

Multimodal cues for encoding structure : A related approach to improving source code comprehension is to encode signals about code structure in the auditory and tactile modalities. Tactile Code Skimmer (TCS) [8] uses haptic cues to convey the *shape of code* with the intent of reducing the *hearing load* on developers who already use screen readers. Stefik et al. [32] evaluated auditory representation of a program using 'lexical scoping cues' that convey the nesting structure of code. They also introduce the concept of 'artifact encoding' to measure the auditory comprehensibility of auditory program representations. Hutchinson et al. [14] investigated the effectiveness of speech, non-speech, and spearcons (rapid speech-based auditory icons) to convey code structure. They found non-speech sounds to be useful in identifying code constructs. Ludi et al. [18] investigated the role of auditory cues in improving visual programming languages like Scratch. They found that spearcons resulted in the fastest time-to-completion, speech resulted in the most accurate understanding of the code.

Accessible Debugging : Sodbeans [31] is a plugin for NetBeans IDE that uses auditory cues to make debugging accessible. In motivating the work, the authors state command-line based tools are more usable for DWVI than visual programming tools. WAD [30] is an auditory debugger for Visual Studio IDE. It uses auditory cues to improve comprehension during dynamic behaviors in debugging, like tracking program flow and change in variable values.

Research into improving accessibility of developer tools has been concentrated on writing, comprehending, and debugging code. However, in the real world, developers use multiple tools outside their IDEs like documentation, command-line interfaces, source code repositories, code review tools, etc. There is a lack of research into understanding and improving the accessibility of these tools. In this paper, we concentrate on one of these tools - command-line interfaces.

2.2 CLI Accessibility

As stated in the previous section, in the literature surrounding the accessibility of developer tools, CLIs are often considered a more accessible alternative to graphical interfaces. Many workarounds that blind developers use, like working with text editors, using printf debugging, or using python interpreters for coding, often imply the use of command-line interfaces [2, 19, 25]. There is also the sentiment that CLIs are inherently more accessible than visual programming tools [28, 31].

To reiterate the possible reasons behind this assumption, CLIs do not suffer from two of the most common accessibility issues found in graphical user interfaces: a) they are mostly keyboard accessible and b) there is no need for providing alternative labels for non-text content. However, this does not necessarily mean that the experience of using a CLI with a screen reader is efficient.

To date, there has been no systematic evaluation of the accessibility of CLIs. The most relevant prior work is accessibility research on text-based video games [13]. In this work, the author describes a case study based on their experience, making a text-based interactive game accessible. They argue that their findings could translate to command-line interfaces due to the similar pattern of text-based interaction in both CLIs and text-based games.

Our primary intent in this research was to address this gap in the literature around the accessibility of command-line interfaces. In particular, our research goals were to

(1) Understand the experience of using CLIs with screen readers
(2) Understand the accessibility issues developers with visual impairments face when using CLIs
(3) Articulate a set of guidelines to improve CLI accessibility

To this end, we report findings from two studies conducted with developers with visual impairments who use screen readers.

3 METHODOLOGY

To investigate our research questions, we conducted two studies. The first study was a set of semi-structured interviews with six developers who use CLIs and screen readers to understand their perception on CLI accessibility. In the second study, six developers with visual impairments used a CLI for a period of two weeks and reflected on the accessibility issues they encountered in an interview session.

As part of Study 2, participants recorded some task data (e.g. task time, success metrics) as well. These are reported in Section 4 . Analysis of qualitative data from Study 1 and Study 2 revealed significant overlap in themes. These are reported together in Section 5.

3.1 Study 1 : User Interviews with DWVI

3.1.1 Participants. In this formative study, we interviewed six software developers. The inclusion criteria were that participants self-identified as - a) software developers, b) using screen readers, and c) using command-line interfaces as part of their development workflow. Two of the participants reported using a braille display. Five participants had some form of visual impairment. One participant did not have a visual impairment, but extensively used screen readers as part of their work. The participant details and their assistive technology usage are available in Table. 1. Informed consent was obtained from all participants. Each participant received $175 for their time.

3.1.2 Procedure. The interviews were semi-structured. We asked participants about - a) their workspace and assistive technology usage, b) code development practices, c) the role CLIs played in their developmental workflow, d) the issues they faced with using CLIs, and e) what would be their ideal experience of using CLIs with screen readers.

Four interviews happened remotely over Google Meet. Two participants preferred to be interviewed over the phone. Each interview session lasted approximately 90 minutes. All the interview sessions were recorded and transcribed.

3.2 Study 2 : Usability evaluation of the CLI

We conducted a second evaluative study to further refine and confirm the findings from Study 1. In this study, developers used a CLI for two weeks and reported on their experiences.

3.2.1 Participants. Six developers took part in this study. The inclusion criteria were identical to the first study - software developers who used CLIs and screen readers as part of their development workflow. All participants in the study reported having some form of visual impairment. All participants self-identified as legally blind. One participant had some light perception. The participant details are available in Table. 2. Each participant received $450 for their time.

3.2.2 Procedure. In the second study, six developers remotely completed four tasks with the *gcloud* CLI [24] of Google Cloud Platform - a) Install and setup the CLI , b) create a virtual machine in the cloud, c) recognize text from a given image using a public machine learning service and d) deploy a docker container to the cloud infrastructure. All participants chose to use JAWS screen reader on a Windows machine to complete their tasks.

The second study consisted of three parts: a) an onboarding interview explaining the tasks, a brief interview of their code development workflows, and provision of credentials and access to a cloud platform to complete the tasks, b) the participants spent two weeks working on the tasks and recorded their experiences, and c) a final interview reflecting on their experiences of working with CLIs and their accessibility. All communications with the participants used plain text email. Participant P2.1 dropped out of the study after the onboarding session, so their entries are not reported here.

3.2.3 Task Entries. Participants were provided the following structure for recording their entries - a) Give a brief overview of the steps you used to perform this task, b) Did you complete this task?, c) How long did you spend on this task?, d) What accessibility or usability issues did you encounter on this task?, and e) What resources or documentation did you use for this task?

They also recorded their perceived workload in completing each task using a modified NASA TLX scale. The NASA Task Load Index [11] measures subjective mental workload on five 7-point scales with 21 gradations. For this study, we used three NASA TLX scales: performance (success), effort, and frustration. The scales used were 7-point Likert scales, with one being lowest and seven highest. Participants reported their TLX metrics after the completion of each task.

We did not include the *physical demand (e.g., physical activity like pulling, pushing)* scale from the NASA TLX as this construct was not very relevant to our tasks. We also did not include the *temporal demand* scale. Instead, we explained the four tasks to the participants during the onboarding process and asked them to estimate how long they anticipated each task would take them to complete. We asked participants to keep track of the time they started and stopped working on each task. After completing the tasks, participants recorded the actual time the task took them. They reflected on the difference between the estimated and reflected time during the off-boarding interview.

3.3 Analysis

The transcripts from the interviews in Study 1 and the on-boarding and off-boarding interviews in Study 2 were all coded using a grounded theory approach [6]. The codes were maintained in nViVo. We found common themes emerging from both the studies. These are reported together in Section.5. Any theme that emerged from just one of the studies is noted explicitly.

4 RESULTS - TASK METRICS

4.1 Study 2 task metrics

Participants' response to the NASA TLX scale is shown in Table.3, Table.4, and Table.5. Figure.1 shows the estimated and actual time taken for each participant for each of the tasks.

Participant P2.1 dropped out of the study after the onboarding study, so their entries are not reported here. P2.3 and P2.5. had the CLI already installed in the development environment they were working on, even though they did not have prior experience with the CLI. Hence, we do not report the TLX metrics for these participants for Task 1.

Task 1 was the installation task. The installer for the CLI evaluated in the study was GUI-based. The *actual time* users reported for this task reflected their environment's network latency rather than actual time on task. Also, since the installer itself was GUI-based, it did not add any additional insight into CLI accessibility, so we chose not to report Task 1 time metrics in Figure. 1

The actual time to complete Task 2 ($Mean(SD) : 129(111)\ minutes$) was significantly higher ($t(4) = 2.34, p = 0.039$) than the anticipated time ($Mean(SD) : 21(13.4)\ minutes$). The time differences between anticipated and actual task times trended towards significance for Task 3 ($Anticipated : 32(26.1)\ minutes, Actual :$

Table 1: Participant Details - Study 1

Participant	Assistive tech used for development work	Platform or OS
P1.1	JAWS, NVDA, braille display	Windows
P1.2	JAWS, NVDA, VoiceOver, braille display	Windows OS on Mac
P1.3	JAWS, NVDA, VoiceOver	Windows, Mac
P1.4	JAWS, NVDA	Windows
P1.5	JAWS, NVDA	Windows
P1.6	ChromeVox, ORCA, VoiceOver	Linux, Mac

Table 2: Participant Details - Study 2

Participant	Assistive tech used for development work	OS	Terminal
P2.1	JAWS	Windows	desktop-based
P2.2	JAWS, NVDA, Narrator, braille display, braille notebook	Windows	desktop-based
P2.3	JAWS, NVDA, braille display, braille embosser	Windows	web-based
P2.4	JAWS, NVDA, ChromeVox, braille display, braille notebook	Windows	desktop-based
P2.5	JAWS, Window-Eyes	Windows	web-based
P2.6	JAWS, NVDA, braille display	Windows	desktop-based

Table 3: Perceived Success. A higher score indicates higher perceived success.

	P2.1	P2.2	P2.3	P2.4	P2.5	P2.6
Task 1	-	7	-	4	-	7
Task 2	-	6	7	6	2	6
Task 3	-	7	7	4	1	6
Task 4	-	5	6	3	1	4

Table 4: Perceived Effort. A higher score indicates higher perceived effort.

	P2.1	P2.2	P2.3	P2.4	P2.5	P2.6
Task 1	-	5	-	7	-	3
Task 2	-	6	1	7	6	6
Task 3	-	2	1	6	5	7
Task 4	-	5	3	5	7	7

$107(113.9)\ minutes, t(4) = -1.86, p = 0.06$) and were statistically significant for Task 4 ($Anticipated : 33(24.9)\ minutes, Actual : 132(98.6)\ minutes, t(4) = -2.68, p = 0.02$).

4.1.1 Discussion. While the task success rates in Study 2 were high as shown in Table. 3, the effort (shown in Table. 4) and frustration (shown in Table. 5) to achieve the success was also relatively high. This is also reflected in the difference in estimated and actual time in Fig. 1. The relatively high success rate concurs with the perception that CLIs are accessible. However, the high effort and high frustration show that the user experience of using a CLI with a screen reader is sub par. The analysis of open-ended interview responses from Study 1 and Study 2 discussed in Section shed light into the accessibility issues participants encountered.

Table 5: Perceived Frustration. A higher score indicates higher perceived frustration.

	P2.1	P2.2	P2.3	P2.4	P2.5	P2.6
Task 1	-	2	-	7	-	1
Task 2	-	2	3	5	6	5
Task 3	-	2	1	7	5	5
Task 4	-	3	7	6	7	6

5 RESULTS - THEMES FROM STUDY 1 AND STUDY 2

5.1 CLI Workflows

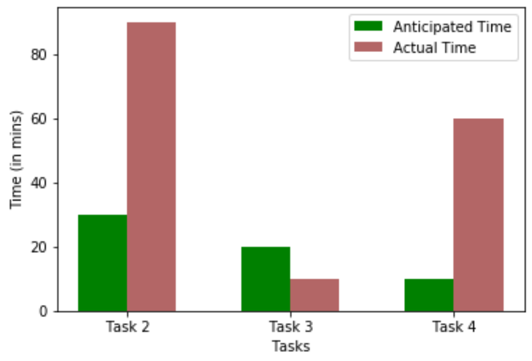
We asked participants about the type of tasks for which they used CLIs. Participants mentioned using CLIs to compile and run their code and scripts, source code management (e.g., working with git), and connecting to remote machines (e.g., ssh-ing).

P1.3 : *command-line is essentially a lot of script running, running the code, I don't write code using the command-line, I never do that, almost never do that. Something like git commits, or git pushes, or set up, running codes these kinds of development tasks is what, yeah.*

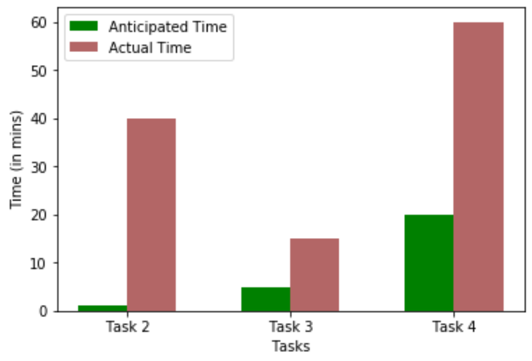
P1.4 : *File management, connection to remote resources, occasionally downloading the files with wget and curl etc, logging into remote systems, programming, short calculations by simply instantiating Python.*

5.2 Unstructured Text

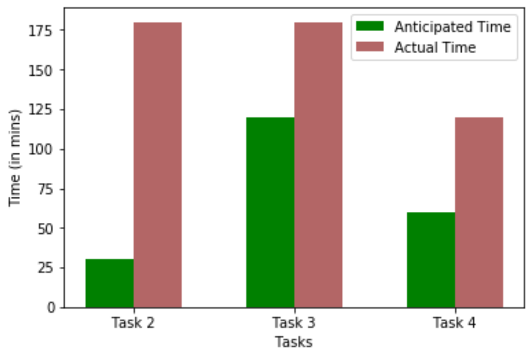
While CLIs are text-based interfaces, much of the text representation in CLIs is unstructured. This means there is no underlying structure for screen readers to take advantage of. For instance, in web pages, html tags like h1 and h2 help screen readers infer the structure and hierarchy of the pages and use hotkeys to navigate



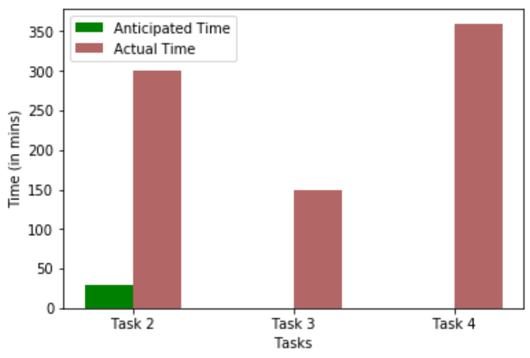
(a) Participant P2.2



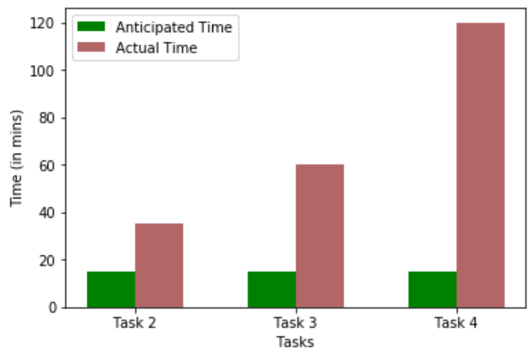
(b) Participant P2.3



(c) Participant P2.4



(d) Participant P2.5



(e) Participant P2.6

Figure 1: Anticipated time on task vs Self-reported time on task

Figure 1: Anticipated time on task vs Self-reported time on task

quickly to the next heading or another semantic tag (e.g., header, nav).

The lack of underlying structure in the CLI results in users having to parse content linearly, which is extremely inefficient. Users mentioned three common areas where such unstructured text becomes problematic: man pages, tables, and long outputs from CLI commands.

5.2.1 Navigation in terminal. Participants in both Study 1 and Study 2 consistently noted the difficulty in navigating the terminal with a screen reader. P1.1 and P1.2 noted in particular that simple tasks like scrolling could be painful. P2.2 explained that it is almost like trying to simulate a mouse with a keyboard and often does not work reliably.

P1.1 : *Scrolling in terminal is generally a pain*

P1.2 : *Sometimes the screen reader can't, it's almost as if you can't scroll past a certain boundary. So you have this buffer and you're not able you can't scroll up past a certain point, or you can't scroll down past a certain point.*

P2.2 : *Because sometimes the limitation of a JAWS cursor, and I think that's the biggest limitation is like, it's using the mouse simulator so sometimes it gets stuck at one point on the screen and then you cannot navigate very well. It kind of sees what it's seeing and then it can only go so far, it doesn't move very well. It's almost like, what we call a mouse cursor getting locked, almost. So, I couldn't navigate everything like I can on the web page.*

Given that even simple navigation tasks like scrolling are difficult in a terminal with screen readers, one of the common accessibility issues mentioned was the difficulty in consuming unstructured text that often appears as output of commands.

5.2.2 Man Pages. Man (short for manual) pages are a form of in-terminal documentation for CLI. They have the advantage of being available from within the terminal without having to context switch. They are usually invoked by `man [command]` or `[command name] --help`.

Participants in Study 1 called out man pages as a canonical example of unstructured text in a CLI. Fig.2 shows the man page for the *ls* command that lists the contents of a directory. While the content in Fig.2 appears formatted, it is simply not possible to navigate this content like a web page equivalent. For instance, users can not jump to headings or skip sections to reach the location they want. They would have to painstakingly read this content linearly multiple times to get the information they are seeking.

Workarounds : The workarounds participants use for man pages were diverse depending on their background. P1.1 mentioned having to write their own script to render man pages in a browser while P1.3 mentioned they never use man pages and rely only on web-based documentation.

P1.1 : *I wrote a script to render man pages in browser*

P1.3 : *I just fall back to the web if I need to read documentation, I do a web search and go through it that way because man pages are very unstructured in terms of navigating with a screen reader, so it gets very cumbersome to navigate man pages.*

This theme emerged primarily from Study 1, since the CLI used in Study 2 offered html version of all reference documentation. So users did not have to rely on just man pages.

5.2.3 *Tables.*

This theme emerged in both Study 1 and Study 2.

Tables are a common form of unstructured text in CLIs. Many CLI commands present their output in a tabular format. An example is shown in Fig. 3 . While this appears structured with a title, row, and columns, this is just visual formatting, and there is no inherent structure here that is available to screen readers. Users can not easily tab through to a desired row and column or have a row heading announced. They would have to parse linearly and rely on their memory to contextualize the current element at their cursor location.

Most participants' common workaround was to try to memorize the column names and hypothesize which column an element belongs to as they parse the table linearly.

P1.1: *Usually, I just try to keep track of it all in my head because I haven't been able to figure out a good way to navigate tables. So I'd just have to memorize the column names and then manually navigate between each piece of information and try to remember what column it's associated with.*

P1.5: *A good example is the top command where it has the six or seven columns going across the screen to show the PID and the memory percentage and the CPU percentage and all that, that, those are a little bit challenging to read because, since a screen reader won't be able to interpret that as a table, you have to memorize which numbers go with which columns and which order they will appear in. So, anything that's tabular like that is hard*

Some participants mentioned that this process becomes easier as they repeatedly use a command, but this could be an arduous task if the number of columns is large or the first time they encounter the table.

P1.4 on relying on their past experience to read the content in Fig. 3 : *you always know the process line that comes first, you know the process then comes next, you know the amount of CPU usage of the next column, memory is the next one*

5.2.4 *Ideal Experience.*

When asked what the ideal experience would look like, participants wished that the CLIs would be flexible enough to convert tables into a format more suitable for consumption via screen readers. Popular choices included the ability to transform to flattened tables for easier parsing or being able to export these tables as html or comma-separated files so that they could read these in a browser or spreadsheet software.

P1.4 : *I've done stuff like that before where I've written a full script, to just take the output of something that was tabular in nature and literally make it into a table by wrapping it in the table and some <td>s and <tr>s and calling it a day.*

P2.5 : *I love flattened tables. I absolutely do because thatś just the way that JAWS, interacts in a non-tabular form most of the time. If that exported as a CSV, would pop straight up into [spreadsheet software] and be on my way.*

5.2.5 *Long output text.*

This theme also emerged in both Study 1 and Study 2.

Many CLI commands provide long output texts as the response. Due to the difficulties navigating in the terminal, users reported copying the terminal contents to a notepad to review the content. This workaround makes it easier to perform tasks like searching for a keyword.

P1.5: *I will have it display all of the text, so even if it's four or five screens ... I will select the entire scrollback buffer and copy it and paste it in Notepad, and then I'll review it in Notepad - it's easier to do that in a Notepad editor window. Or if I want to copy and paste text, that's much easier to do from the editor.*

P1.4 : *Let's say I'm doing a Port Scan of a network and I have really long output, then what I would do is I would output it to a file*

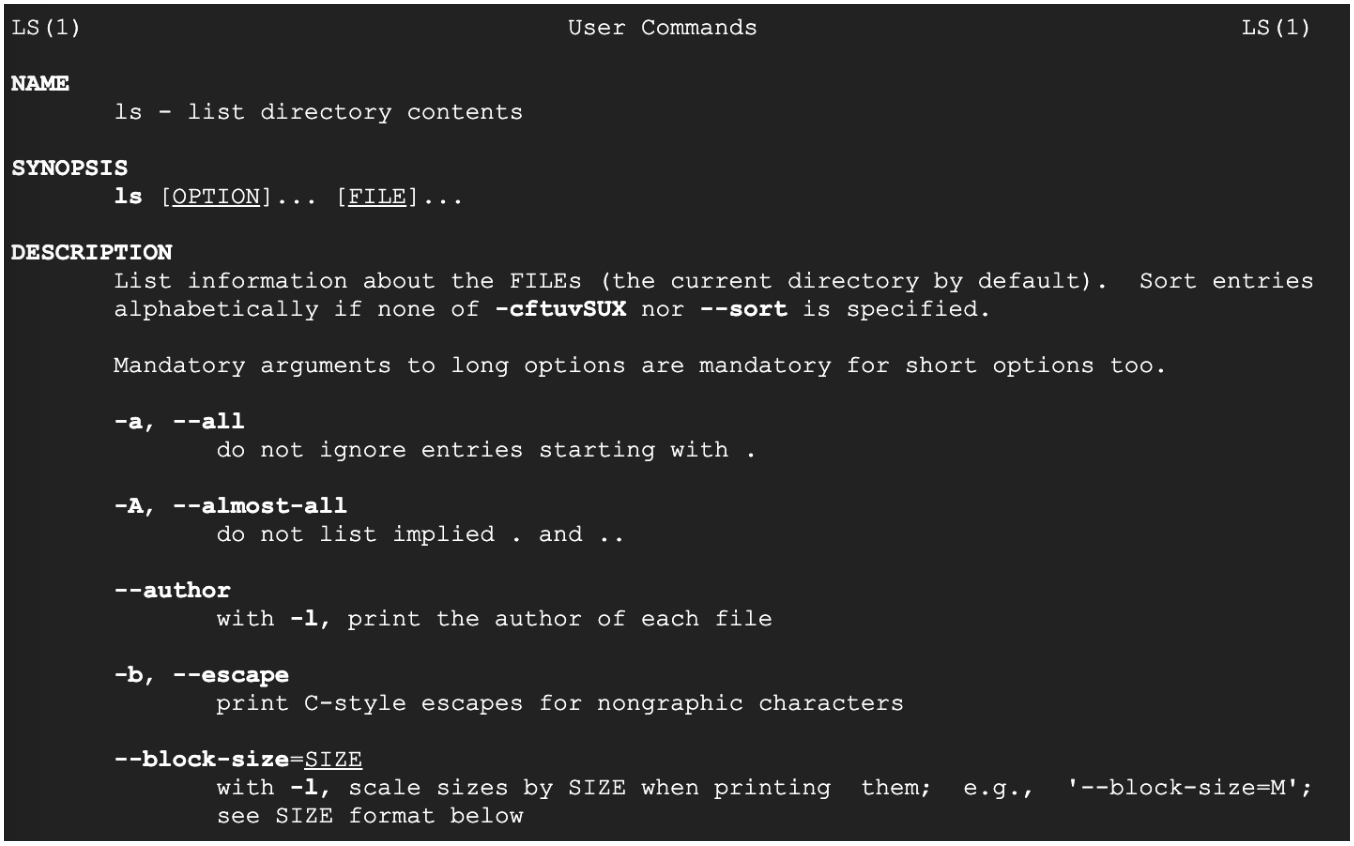
Multiple utilities would allow users to search for a piece of text in a long output from within a CLI (e.g. grep). For instance, in the output snippet illustrated in Fig. 4, the field "description" contains the most useful information. Using a utility like grep (e.g., `[command] | grep 'description'`) would make it easy to parse just that line of information.

However, in our study, participants did not report using these utilities. This was because they did not have a mental model of the output they are looking at until they review the content at least once.

Unlike APIs, most CLI commands do not document their output contract. So this means a user has no idea what to expect out of a command until they actually execute it. They do not have a schema in their mind ahead of time for the output they are parsing, which makes it very difficult to comprehend the output and, in particular, search for a specific value.

For instance, PC described his experience of obtaining the output in Fig.4 like this :

P2.3 : *But the results were pretty scrambled, I mean, they, I couldn't make much sense out of it other than the fact that they were, something to do with a graph, Y-axis with an X-axis, coordinates. They were like, Y, left brace, right brace, 500, then XX, then like, it was weird, I thought it was a bit weird.*



```
LS(1)                         User Commands                         LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List information about the FILEs (the current directory by default).  Sort entries
       alphabetically if none of -cftuvSUX nor --sort is specified.

       Mandatory arguments to long options are mandatory for short options too.

       -a, --all
              do not ignore entries starting with .

       -A, --almost-all
              do not list implied . and ..

       --author
              with -l, print the author of each file

       -b, --escape
              print C-style escapes for nongraphic characters

       --block-size=SIZE
              with -l, scale sizes by SIZE when printing  them;  e.g.,  '--block-size=M';
              see SIZE format below
```

Figure 2: A typical man/help page.

5.3 Status and Progress Indication

This theme also emerged in both Study 1 and Study 2.

5.3.1 Lack of Status and Progress Indication. Linux has the convention where a silent return without any errors is considered success [26]. This convention translates to many commands, not providing explicit progress and status indication. Unsurprisingly, participants found this paradigm difficult with screen readers. For instance, since the screen readers did not vocalize any status, P2.4 resorted to observing any brightness changes on their screen.

P2.4: *Yeah, it might have been something on the screen, but the screen reader didn't say anything. Because I can tell there's much that when I was doing there, the screen would get a little bit darker. That would tell me something is on there, but I don't know, I can't see enough, so I don't know what. But then when I moved my focus away from it the screen would brighten up normal, so there's probably something happening, but I can't tell what's going on.*

P1.1 observed this lack of status indication as a general issue when working with CLIs.

P1.1 : *sometimes it'll not report status information, and so I've had to guess as to whether it was, in the middle of compiling something then sometimes it would hang. And I wouldn't tell if it had really froze or whether it was still doing something.*

P2.6 stated that they ended up creating multiple resources because there was no status message.

P2.6: *No it did not tell me. It did not, it didn't, it wouldn't, it didn't tell me and so I thought maybe I hadn't created it. So I created a second, not realising.*

5.3.2 Lack of screen reader friendly progress indication. Sometimes CLI commands do provide progress indication. In these cases, they try to simulate GUI-like progress indication mechanisms like progress bars (Fig.5) and spinners (Fig.6). However, screen readers do not recognize them as progress indicators and vocalize the underlying Unicode characters.

P2.5 described their experience with the spinner like progress indicator in Fig. 6 as

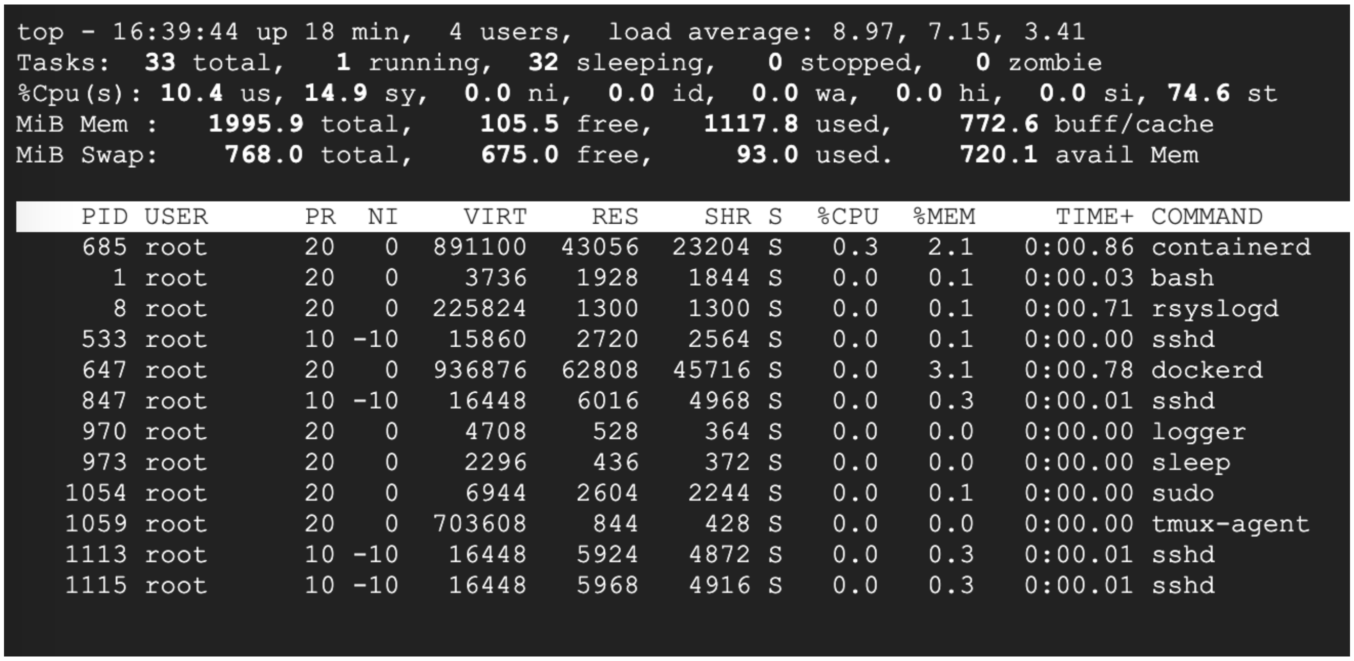
P2.5 : *Little dots submit, little blah, blah, blah, blah, completed with status failure.*

5.4 ASCII Art

This theme also emerged from both Study 1 and Study 2.

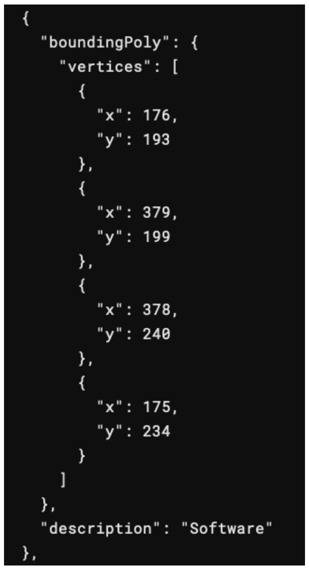
Sometimes CLIs use ASCII characters to decorate tables and other output entities to make them visually appealing. However, our study participants found this experience very distracting and adding to the already difficult task of reading a table. For instance, in reference to the table in Fig.7 P2.5 said :

P2.5: *So it's in a table, all the decorators are distracting. I think the main content here isn't too bad; it's just the way that it's formatted.*



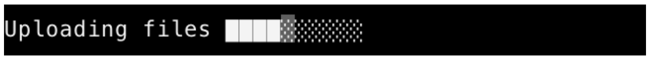
```
top - 16:39:44 up 18 min,  4 users,  load average: 8.97, 7.15, 3.41
Tasks:  33 total,   1 running,  32 sleeping,   0 stopped,   0 zombie
%Cpu(s): 10.4 us, 14.9 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si, 74.6 st
MiB Mem :   1995.9 total,    105.5 free,   1117.8 used,    772.6 buff/cache
MiB Swap:    768.0 total,    675.0 free,     93.0 used.    720.1 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
    685 root      20   0  891100  43056  23204 S   0.3   2.1   0:00.86 containerd
      1 root      20   0    3736   1928   1844 S   0.0   0.1   0:00.03 bash
      8 root      20   0  225824   1300   1300 S   0.0   0.1   0:00.71 rsyslogd
    533 root      10 -10   15860   2720   2564 S   0.0   0.1   0:00.00 sshd
    647 root      20   0  936876  62808  45716 S   0.0   3.1   0:00.78 dockerd
    847 root      10 -10   16448   6016   4968 S   0.0   0.3   0:00.01 sshd
    970 root      20   0    4708    528    364 S   0.0   0.0   0:00.00 logger
    973 root      20   0    2296    436    372 S   0.0   0.0   0:00.00 sleep
   1054 root      20   0    6944   2604   2244 S   0.0   0.1   0:00.00 sudo
   1059 root      20   0  703608    844    428 S   0.0   0.0   0:00.00 tmux-agent
   1113 root      10 -10   16448   5924   4872 S   0.0   0.3   0:00.01 sshd
   1115 root      10 -10   16448   5968   4916 S   0.0   0.3   0:00.01 sshd
```

Figure 3: An example of tables in the CLI. Output of *top* command.



```
{
  "boundingPoly": {
    "vertices": [
      {
        "x": 176,
        "y": 193
      },
      {
        "x": 379,
        "y": 199
      },
      {
        "x": 378,
        "y": 240
      },
      {
        "x": 175,
        "y": 234
      }
    ]
  },
  "description": "Software"
},
```

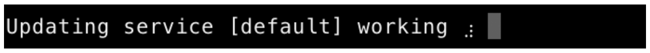
Figure 4: Example of a Long output



```
Uploading files ████▓░░░░░
```

Figure 5: Example CLI Progress Bar

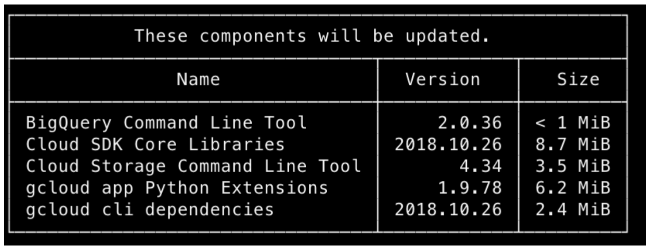
P1.4 observed that ASCII arts in general, were annoying. P1.4: *ASCII Art is beyond annoying, and it is very common, so [CLI] had this thing for a little while where they would have, during an error, it*



```
Updating service [default] working ⠏ █
```

Figure 6: Example CLI Spinner

would have this error, and it would help with like ASCII Art and so all you're doing as a screen reader user is $ %̂ % $, you know what I mean, it's total nonsense.



```
┌──────────────────────────────────────────────────────────────────┐
│                These components will be updated.                 │
├──────────────────────────────────────┬────────────────┬──────────┤
│                 Name                 │    Version     │   Size   │
├──────────────────────────────────────┼────────────────┼──────────┤
│ BigQuery Command Line Tool           │         2.0.36 │  < 1 MiB │
│ Cloud SDK Core Libraries             │     2018.10.26 │  8.7 MiB │
│ Cloud Storage Command Line Tool      │           4.34 │  3.5 MiB │
│ gcloud app Python Extensions         │         1.9.78 │  6.2 MiB │
│ gcloud cli dependencies              │     2018.10.26 │  2.4 MiB │
└──────────────────────────────────────┴────────────────┴──────────┘
```

Figure 7: A CLI table with ASCII decorators

5.5 Quality of Error Messages

This was a theme that emerged only from Study 2. Participants in Study 1 did not explicitly call out error messages as an issue.

The role of error messages in API developer experience is well documented in literature [12, 21]. Participants in our study also encountered issues with clarity and usefulness of error messages. The issues ranged from error messages that were difficult for screen readers to verbalize to error messages not being actionable.

For instance, in Task 2, P2.2 experienced a regular expression as part of the error message and described the experience as *So, yeah,* *it was just saying like, A-Z 1-0, you know, like it was saying, you have to have in between these characters and symbols and letters. It would give a range. So, it wasn't still clear.*

Participants heavily relied on cues from error messages to troubleshoot, and the lack of clear, actionable messages cost them a lot of effort. P2.5 mentioned that experiencing an unclear error message resulted in them spending six hours debugging on a task they expected to take 30 minutes.

Researcher : *On your log entry for the final task, on a scale of one to seven, how physically demanding was the task, and you said eight. What made that task so physically demanding?*

Participant 2.5 : *I put eight just because I was going back through the original instructions and said oh, this was supposed to take a half hour and it took me six hours . . . it was very clear what was supposed to happen except it was very unclear what, how to get around the errors. So pretty straightforward, but it took a long time, so that's why.*

6 DISCUSSION AND RECOMMENDATIONS

Even though CLIs are text-based and keyboard-operable interfaces, the experiences of the developers we spoke to show that they do not always provide a positive user experience. For instance, while the tasks in Study 2 were achievable (Table. 3), they took longer time to complete than participants expected (Fig. 1) and involved high amounts of effort (Table. 4) and frustration (Table. 5).

The barriers participants encountered were that CLIs are unstructured text-based interfaces, making them inefficient for use with screen readers. Comprehending such unstructured text from a CLI command often happens outside the CLI, in text files or html documents where navigation is much easier. Status and progress indication may not always be available in CLIs and may not be screen reader friendly when available. Error messages also may not be screen reader friendly or actionable.

Users persevere with slow line-by-line reading, relying on their memory and experience and by being generally resilient to issues they encounter. But all this results in a far from the optimal user experience. P1.3 described encountering such accessibility roadblocks as P1.3 : *I'm very, very resilient to accessibility bugs. I'm like, bring it on, I'm going to navigate this, but again, sometimes even my patience is put to the test.*

It would be a better user experience if CLIs could enable some of the common workarounds that users already use off-the-shelf, see a table as a csv file, and export a long output as an HTML file with appropriately leveled headings. To this end, we propose a set of recommendations for building accessible CLIs.

Recommendation 1 : Ensure that a HTML version of all documentation is available.

Description : The output of `--help` or `man` is long unstructured text and screen reader users do not use them for documentation. They rely on the HTML / online version of this documentation. Assume that if the reference documentation resides only within the terminal, it is not available to screen reader users.

Recommendation 2: Provide a way to translate long outputs into another accessible format.

Description: As described earlier, it is difficult to scroll in the terminal with a screen reader. Users will have a hard time scrolling across the entire output and reading/copying all the text available to them. Being able to export this into a more accessible format like text or html would be a better user experience. This would allow them to search or navigate more quickly to the content they need.

Recommendation 3 : Document the output structure for each command.

Description: To effectively parse and comprehend the output from a command, users need to know what the command's output looks like ahead of running the command.

Recommendation 4: Provide a way to translate tables in CLI output into another accessible format

Description: Even though a table in CLI output appears structured, it is just visually formatted. It is still unstructured content for a screen reader user. They would have to linearly parse all the elements to understand the structure and then parse and navigate this structure again to understand the content. It is cognitively demanding, and inefficient. Providing the ability to translate tables to other formats like html or csv would make parsing this efficient.

Recommendation 5: Ensure that all commands provide status and progress indication.

Description: All commands need to provide some form of status indication and progress indication when appropriate. When a command does not provide any status or progress indication, screen reader users receive no feedback on the system status.

Recommendation 6: Ensure that all status and progress indicators used are screen reader friendly.

Description: Screen readers do not read ASCII-art or non-ASCII characters as intended for sighted developers. This means that if any of these are used to represent progress, a screen reader might not render this content correctly.

Recommendation 7 : Ensure that error messages are understandable when read aloud.

Description: Error (and output) messages from CLIs could contain a lot of jargon (e.g. regular expressions, domain-specific acronyms, URLs) that might be difficult for a screen reader to verbalize. This might make it challenging to comprehend the error message.

7 CONCLUSION

This paper investigated the accessibility of command-line interfaces through two studies with developers with visual impairments. We found that despite being text-based and keyboard-operable, CLIs suffer from their own set of accessibility issues: unstructured text, lack of status and progress indication, lack of screen-reader friendly progress indicators and inaccessible error messages. Based on these findings, we provided a set of recommendations for improving the accessibility of CLIs.

Study 2 was based on users using a particular CLI. Nevertheless, the overlap in themes between Study 1 and Study 2 reassures that our findings and recommendations can generalize to other CLIs. However, we also found themes that emerged only in Study 2. It is possible that a study with another CLI choice would have resulted in new findings. Hence, our immediate future goal is to replicate some of the findings with other CLIs.

8 ACKNOWLEDGMENTS

We thank Kevin Storer, Sarah D'Angelo and Youyang Hou for their valuable feedback on this paper's early drafts. We also thank the Google Cloud CLI tools team for their support of this work.

REFERENCES

[1] Khaled Albusays and Stephanie Ludi. 2016. Eliciting programming challenges faced by developers with visual impairments: exploratory study. In *Proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering*. 82–85.

[2] Khaled Albusays, Stephanie Ludi, and Matt Huenerfauth. 2017. 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*. 91–100.

[3] Catherine M Baker, Lauren R Milne, and Richard E Ladner. 2015. Structjumper: A tool to help blind programmers navigate and understand the structure of code. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. 3043–3052.

[4] Rob Barrett, Eser Kandogan, Paul P Maglio, Eben M Haber, Leila A Takayama, and Madhu Prabaker. 2004. Field studies of computer system administrators: analysis of system management tools and practices. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*. 388–395.

[5] Ben Caldwell, Michael Cooper, Loretta Guarino Reid, Gregg Vanderheiden, Wendy Chisholm, John Slatin, and Jason White. 2008. Web content accessibility guidelines (WCAG) 2.0. *WWW Consortium (W3C)* (2008).

[6] Kathy Charmaz. 2014. *Constructing grounded theory.* Sage.

[7] Parham Doustdar. 2016. The Tools of a Blind Programmer. https://www.parhamdoustdar.com/2016/04/03/tools-of-blind-programmer/, Last accessed Sep 2020.

[8] Olutayo Falase, Alexa F Siu, and Sean Follmer. 2019. Tactile Code Skimmer: A Tool to Help Blind Programmers Feel the Structure of Code. In *The 21st International ACM SIGACCESS Conference on Computers and Accessibility*. 536–538.

[9] Filipe Del Nero Grillo and Renata Pontin de Mattos Fortes. 2014. Tests with blind programmers using awmo: An accessible web modeling tool. In *International Conference on Universal Access in Human-Computer Interaction*. Springer, 104–113.

[10] Alex Hadwen-Bennett, Sue Sentance, and Cecily Morrison. 2018. Making Programming Accessible to Learners with Visual Impairments: A Literature Review. *International Journal of Computer Science Education in Schools* 2, 2 (2018), 3–13.

[11] Sandra G Hart. 2006. NASA-task load index (NASA-TLX); 20 years later. In *Proceedings of the human factors and ergonomics society annual meeting*, Vol. 50. Sage publications Sage CA: Los Angeles, CA, 904–908.

[12] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R Klemmer. 2010. What would other programmers do: suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1019–1028.

[13] Michael James Heron. 2015. A case study into the accessibility of text-parser based interaction. In *Proceedings of the 7th ACM SIGCHI symposium on engineering interactive computing systems*. 74–83.

[14] Joe Hutchinson and Oussama Metatla. 2018. 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*. 1–6.

[15] Alasdair King, Paul Blenkhorn, David Crombie, Sijo Dijkstra, Gareth Evans, and John Wood. 2004. Presenting UML software engineering diagrams to blind people. In *International Conference on Computers for Handicapped Persons*. Springer, 522–529.

[16] Mario Konecki, Alen Lovrenčić, and Robert Kudelić. 2011. Making programming accessible to the blinds. In *2011 Proceedings of the 34th International Convention MIPRO*. IEEE, 820–824.

[17] Richard E Ladner and Andreas Stefik. 2017. AccessCSforall: making computer science accessible to K-12 students in the United States. *ACM SIGACCESS Accessibility and Computing* 118 (2017), 3–8.

[18] Stephanie Ludi, Jamie Simpson, and Wil Merchant. 2016. 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*. 279–280.

[19] Sean Mealin and Emerson Murphy-Hill. 2012. An exploratory study of blind software developers. In *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 71–74.

[20] Sandra R Murillo and J Alfredo Sánchez. 2014. Empowering interfaces for system administrators: Keeping the command line in mind when designing GUIs. In *Proceedings of the XV International Conference on Human Computer Interaction*. 1–4.

[21] Brad A Myers and Jeffrey Stylos. 2016. Improving API usability. *Commun. ACM* 59, 6 (2016), 62–69.

[22] Tuukka Ojala. 2017. Software development 450 words per minute. https://www.vincit.fi/en/software-development-450-words-per-minute/, Last accessed Sep 2020.

[23] Stack Overflow. 2019. Stack Overflow Developer Survey. https://insights.stackoverflow.com/survey/2019, Last accessed Sep 2020.

[24] Google Cloud Platform. [n.d.]. gcloud command-line tool overview. https://cloud.google.com/sdk/gcloud, Last accessed Dec 2020.

[25] Venkatesh Potluri, Priyan Vaithilingam, Suresh Iyengar, Y Vidya, Manohar Swaminathan, and Gopal Srinivasa. 2018. Codetalk: Improving programming environment accessibility for visually impaired developers. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–11.

[26] The Linux Documentation Project. 2020. Exit Codes With Special Meanings. https://tldp.org/LDP/abs/html/exitcodes.html, Last accessed Sep 2020.

[27] Emmanuel Schanzer, Sina Bahram, and Shriram Krishnamurthi. 2019. Accessible AST-based programming for visually-impaired programmers. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 773–779.

[28] Robert M Siegfried. 2006. Visual programming and the blind: the challenge and the opportunity. *ACM SIGCSE Bulletin* 38, 1 (2006), 275–278.

[29] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. 2010. An examination of software engineering work practices. In *CASCON First Decade High Impact Papers*. 174–188.

[30] Andreas Stefik, Roger Alexander, Robert Patterson, and Jonathan Brown. 2007. WAD: A feasibility study using the wicked audio debugger. In *15th IEEE International Conference on Program Comprehension (ICPC'07)*. IEEE, 69–80.

[31] Andreas Stefik, Andrew Haywood, Shahzada Mansoor, Brock Dunda, and Daniel Garcia. 2009. Sodbeans. In *2009 IEEE 17th International Conference on Program Comprehension*. IEEE, 293–294.

[32] Andreas Stefik, Christopher Hundhausen, and Robert Patterson. 2011. An empirical investigation into the design of auditory cues to enhance computer program comprehension. *International Journal of Human-Computer Studies* 69, 12 (2011), 820–838.