

CodeTalk: Improving Programming Environment Accessibility for Visually Impaired Developers

Venkatesh Potluri
Microsoft Research India
Bangalore, India
t-vepot@microsoft.com

Priyan Vaithilingam
Microsoft Research India
Bangalore, India
t-prvai@microsoft.com

Suresh Iyengar
Microsoft Research India
Bangalore, India
supartha@microsoft.com

Y Vidhya
Vision Empower Trust
Bangalore, India
vidhya@visionempowertrust.in

Manohar Swaminathan
Microsoft Research India
Bangalore, India
swmanoh@microsoft.com

Gopal Srinivasa
Microsoft Research India
Bangalore, India
gopalsr@microsoft.com

ABSTRACT

In recent times, programming environments like Visual Studio are widely used to enhance programmer productivity. However, inadequate accessibility prevents Visually Impaired (VI) developers from taking full advantage of these environments. In this paper, we focus on the accessibility challenges faced by the VI developers in using Graphical User Interface (GUI) based programming environments. Based on a survey of VI developers and based on two of the authors' personal experiences, we categorize the accessibility difficulties into *Discoverability*, *Glanceability*, *Navigability*, and *Alertability*. We propose solutions to some of these challenges and implement these in CodeTalk, a plugin for Visual Studio. We show how CodeTalk improves developer experience and share promising early feedback from VI developers who used our plugin.

Author Keywords

Accessibility; Programming Environments; Visually Impaired; Audio Debugging

ACM Classification Keywords

H.5.2. Information interfaces and presentation: User Interfaces

INTRODUCTION

Software development is one of the fastest growing fields [10]. However, people with visual impairments are not very well represented in the field of computer science and software development: we are unaware of any formal study that confirms this.

However, we consider the surprise with which the fact of blind programmers is received (see for example the comments in [15]) as an empirical confirmation. The percentage of developers who have self-reported as being blind in the 2017 Stack Overflow survey is about 1% which is much more than the percentage of people with visual impairments in the general population [11]. We believe that the 1% reflects that blind developers are happy with the Stack Overflow question and answer website because it is accessible and consequently use it in higher numbers. According to the US National Bureau of Labor statistics [21] only about 2% of workers in the computing and mathematical professions have a disability compared to the percentage of people with disabilities in the general population of the US which is about 19% according to the US Census Bureau. There are several reasons for this under-representation, and in this paper, we address one of them, namely the poor accessibility of developer tools.

People with visual impairments, use Assistive Technology (AT) like screen readers, screen magnifiers, and braille displays to access computers. They have also been using the same to write computer programs. In recent times, GUI based Integrated Development Environments (IDEs) have become more widely used [11]. These modern IDEs have adopted many innovations to aid program comprehension and development by providing features like syntax highlighting, variable watch windows and ability to execute a code both forward and backward [13]. These feature-rich IDEs enable developers to be more productive and efficient. Though screen readers provide basic accessibility to IDEs¹, many features that make them so useful to sighted developers remain inaccessible to developers using screen readers.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

CHI 2018, April 21–26, 2018, Montreal, QC, Canada

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-5620-6/18/04...\$15.00

<https://doi.org/10.1145/3173574.3174192>

¹ From here on, we use IDEs interchangeably with GUI based IDEs

In this paper, we make the following contributions towards making programming environments more accessible to VI developers.

- We present a classification of accessibility issues in four headings: *discoverability*, *glanceability*, *navigability*, and *alertability* and provide illustrative examples of each kind. This classification was arrived at by combining the subjective experience of two of the authors with the results of a user survey on IDE accessibility.
- We propose solutions to address a subset of the identified issues and implement these solutions as CodeTalk, a plugin for Visual Studio. Unlike related work on accessibility of IDEs which address specific activities, we address accessibility issues across the entire spectrum of activities around software development from comprehending code, editing, debugging, and working with teams on large codebases.
- We present feedback that validates our approach by an exploratory user study with six VI developers using CodeTalk.

The paper is organized as follows: Section 2 describes the motivation for this work and summarizes related work. Section 3 presents a broad classification of accessibility issues in IDEs. In Section 4, we introduce our approaches to solve these and discuss details about CodeTalk, our Visual Studio plugin. In Section 5, we discuss an exploratory user study performed to get some initial user feedback on our approaches. Section 6 and Section 7 presents the key conclusions and highlights several directions for future research.

MOTIVATION AND RELATED WORK

The major motivating factors for this research are the personal experiences of two of the authors A and B. Author A is a novice programmer who primarily used a command line interface and a text editor to program. A's attempt to move to an IDE like Visual Studio was unsuccessful since the accessibility issues were found to be too daunting without continuous help from a sighted person. Author B has been programming using a screen reader and B's experiences with IDEs involved significant effort in tackling inaccessibility. The author was able to cope with just text-based tools for academic work and part-time projects. However, moving to a large organization as part of a product team required the use of an IDE used by the other team members to work efficiently. At this point, the author realized why sighted developers were able to work at a much faster pace. They were able to read code much faster than the screen reader user, quickly comprehend the structure of huge code bases, be informed about errors without explicit actions and move to any part of the code by pointing and clicking. Motivated by this experience, we looked at earlier research efforts that address accessibility issues in programming environments.

IDE accessibility for developers with visual impairments is still a new research area. There is very little exploration that has been done to improve the development and programming experience for VI developers. That said, there has been interest in both academia and industry to improve the accessibility of developer tools. IDEs like Eclipse [6], Apple's XCode [7] and Microsoft's Visual Studio [4] add accessibility support for screen reader users. However, this accessibility support is quite limited to having all buttons and UI elements spoken in some cases. Also, there have been several other attempts by researchers to improve the accessibility of developer tools. Emacspeak [14] is an early effort to improve developer tools accessibility. More recently, Baker et al., [2] address the difficulties faced by blind programmers while reading code. They describe StructJumper, an Eclipse plugin that displays an accessible tree-view of code structure with respect to the current line. This effort attempts to help VI developers get complete context with respect to a specific line of code. The plugin focuses on reading code effectively. Smith et al. [22] explain the problem of navigating hierarchical tree views in detail, and, propose requirements to make the tree views more usable. [22] complements our work on glanceability and navigability in CodeTalk. The key difference is that CodeTalk lays down a framework to address a broad spectrum of challenges faced by VI developers using IDEs while [22] does an in-depth investigation on nonvisual navigation of hierarchical data.

Sodbeans [19] and WAD [18] discuss approaches to use audio for debugging code. The Sodbeans plugin uses speech-based cues to enable VI developers to debug. WAD emphasizes on the developers' ability to comprehend the execution flow of the code.

Audio (both speech and non-speech) has been explored to enable VI developers to program. [16] explores the use of auditory cues (Spearcons) in reading source code. The researchers synthesized source code with different audio cues like speech, tones, and white noise, using NVDA's speech output and Audacity. They used combinations of these audio cues to represent the code file. The participants were asked to comprehend code using these audio files. This effort demonstrates that relying solely on screen-reading is not sufficient for VI developers to comprehend code.

[8] uses 3D printed models for VIPs to explore program output. Students wrote programs to generate tactile versions of the data to explore program output. Several efforts like [9] and [12] focus on teaching programming to blind students. As seen above, all the related research has focused on enabling VIPs to do specific tasks while programming. There is no work that addresses accessibility issues that arise across the complete program development cycle.

We do not address the larger challenge of building tools and languages that facilitate the learning of computer programming. However, we point to some interesting efforts in this direction: Quorum [20] started out as a

language that is easily accessible to screen readers but has since evolved to a much more general effort on evidence-based language design. The APL [17] is another effort to introduce programming to blind students. In this paper we focus on enhancing accessibility of IDEs to VIPs who have learnt the basics of programming and are currently users of IDEs.

To go beyond the specific experiences of the two authors mentioned and to understand the spectrum of accessibility issues that arise during the complete programming cycle, we conducted a user survey which we discuss in the next section.

Preliminary Survey

We conducted a preliminary survey with an objective to collect opinions from VI developers on IDE accessibility, with a specific focus on Microsoft Visual Studio. The survey was hosted online, and we made sure all parts of the survey were accessible to screen reader users. On completion of the survey, participants interested in giving more information could opt-in to participate in additional interviews by conveying their interest over email. Four out of the 20 participants of the survey participated in further detailed interviews. Details of the survey, including the questions, participants' demographic information and programming experience levels, etc., can be found in [1]. The learnings from the survey have been summarized in the next subsection.

Learnings from the Survey

The major observation we made when we collated the survey results and the interview responses, was that the accessibility issues were across the entire spectrum of software development. A sample of the responses to "list top 5 accessibility challenges" illustrates this very well:

- "watch windows are hard to use -specially the quick watch"
- "solution Explorer hangs on very large solutions when attempting to navigate within",
- "Sometimes controls don't have labels and report their class name"
- "access to breakpoint status while debugging"
- "There is no alternate way to get to things if you don't know one of the thousands of shortcut commands"
- "difficulty in moving from error screen to the editor where program is present (Control + tab) doesn't work"
- "Access to variable type and other info (usually accessed by hovering the mouse over the variable name)"

These responses were from VI developers with experience ranging from a year to more than 25 years. The issues people face range from simple ones like "Difficult to determine when code is folded up (hidden) and must be expanded" to that of an advanced user's "That comparison

tool is 100% inaccessible with screen readers so I have to configure my own code review tool in visual studio"

We then stepped back a bit to find if there is some structure to the numerous accessibility issues which will help us devise a solution process to handle them effectively. The result of this effort is the classification of accessibility challenges that we describe in the next section.

CLASSIFICATION OF ACCESSIBILITY CHALLENGES IN IDES

Based on the data from the accessibility survey, experience of the visually impaired authors, as well as related work on IDE accessibility, we classify accessibility challenges into four broad categories and give some example scenarios for each. We use examples from Visual Studio.

1. **Discoverability:** This is the ability with which a user can find features of the system to increase proficiency over time. Sighted users have many visual clues that indicate new features that could be useful for a given context, but VI developers need to depend on others to tell them about such features. Discoverability is an issue for sighted users as well but is exacerbated for VI developers. For instance, the author B was not aware of the variable watch window² and used console messages to find the variable values until pointed out by a sighted team member. The following are some examples of discoverability issues:
 - *Existing features:* Many features of the IDEs are overtly visible in the UI, but are hidden inside multiple levels of navigational hierarchy for screen reader users.
 - *New and modified features:* With every new version of the IDE, new features get added and existing features are modified. Many of these changes are visually represented, and there is no structured approach for VI developers to be informed of the same. This becomes more evident when IDEs do a complete UI over haul, changing the UI hierarchy and arrangement.
2. **Glanceability:** Visual Studio and most IDEs by definition, use the large real estate provided by high resolution monitors to present many aspects of the program development process in one screen. They depend on the ability of the developer to glance at various aspects of the development process at any given time. For sighted developers, glanceability is innate to the medium of information access, vision. The IDEs leverage the high bandwidth nature of visual input and provide features that enable sighted developers to make sense of information by quickly

² Watch window is used to evaluate variables and expressions during debugging.

glancing at the screen and the IDE's windows. Visual input, being a more active way of acquiring information gives an opportunity to unobtrusively provide information to the IDE's users without interrupting their current task. Unfortunately, these features are not available to the VI developers, and they often must consume information linearly. Following are some example situations:

- *Quick overview of the code structure:* Unlike sighted users, who can get the overview of the code structure by quickly scrolling up and down a page, the VI developer should go through the code line by line.
 - *Getting the context of the given line:* There are situations when the VI developer lands in an unknown line of code due to breakpoints or exceptions, or simply because the developer was distracted. On the other hand, based on the line number and the vertical slider bar's position, a sighted user has a notion of the size of the program file and the relative location of the cursor with respect to the beginning and the end of the file.
 - *Indentation level:* Indentation levels in whitespace dependent programming languages like Python, are easy to perceive for sighted users unlike VI developers, who must read the number of whitespaces for every line.
 - At any given point, sighted developers can look at multiple pieces of information (the console log window, stack traces, the actual code and a lot more information as per the developer's preference). VI developers using screen readers have to get this information by explicitly getting focus on to each window in sequence.
3. **Navigability:** An added advantage for sighted developers is the ability to quickly navigate through code using scroll, point and click. Screen reader users are limited to the search functionality and few other navigation features provided by the IDEs. This also extends to navigating between multiple panes within the IDE. Following are some example scenarios:
- *Skipping through large comments:* Sighted developers can skip through large code comments like documentation and licenses quickly as compared to screen reader users. It is difficult for VI developers to navigate to the end of these comments.
 - *Navigating through large blocks of code:* Sighted developers can scroll, point and click to navigate through blocks of code like if-else block, try-catch block. However, navigating through code within a block is not so intuitive and easy for VI developers using screen readers.
 - *Navigating across various windows:* Sighted developers can easily glance over multiple windows at the same time like the watch window,

call stack window, debug window, etc. On the other hand, VI developers must go through numerous keystrokes to access these windows.

```
static void Main(string[] args)
{
    var dictionary = new Dictionary<string, int>(5);
    dictionary.Add("cat", 1);
    dictionary.Add(mouse, 5)
}
```

Figure 1. Red squiggle shown for error in Visual Studio

4. **Alertability:** IDEs convey a significant amount of real time information through a completely visual interface [2]. Such information alerts the developer to issues that need immediate attention or actions that are in progress. The following examples enumerate few scenarios where VI developers do not get access to the real time information provided by the IDE:
- *Debugging Information:* Information related to debugging like values of variables and breakpoint information are not available to a VI developer unless explicit actions are performed.
 - *Error Information:* Syntax error information in IDEs is given by visual cues like red squiggle [Figure 1], which are not available to VI developers pro-actively.

These accessibility challenges result in a huge barrier for VI developers in exploiting the power of IDEs. Although one can bundle all these limitations and attribute them to the fact that sighted developers can either point-click or scroll-click while VI developers cannot, the above grouping helps us devise expedient alternates using a structured approach. We also note that these groupings helped organize our understanding of accessibility challenges and are not meant to be water tight compartments. In the next section we describe CodeTalk, our vehicle for addressing the above challenges.

CODETALK

CodeTalk is implemented as a Visual Studio plugin. CodeTalk works with Visual Studio versions 2015/2017 and supports C# and Python at the time of writing this paper. However, implementing support for newer languages is straightforward. We have chosen to implement CodeTalk as a Visual Studio plugin mainly due to the following reasons:

1. Visual Studio provides APIs that allows us to tap into all the IDE's features.
2. Visual Studio's increasing support for a variety of programming languages.
3. Free availability of Visual Studio community edition.
4. Visual Studio is the most popular IDE among developers [11].

In CodeTalk, we address the accessibility challenges categorized in the previous section by focusing on the root cause of the issues: Screen reader based access to

information is user dependent, unlike the use of a GUI by a sighted user. The user must actively seek out information from various components of the IDE. And since the information access with a screen reader is dependent on cursor focus, the user must explicitly set focus on the appropriate pane. In some situations, the VI developer might not be aware of the presence of a pane containing the information they are looking for. Our approach is to minimize the effort of the VI developer actively seeking information by proactive extraction and presentation of information or by introducing an audio channel distinct from the screen reader. CodeTalk extracts the information relevant to the context and makes it accessible to the developer with reduced effort. To this effect, we introduce new customizable keyboard shortcuts as shown in Table 1. We present below a few of the features of CodeTalk in detail.

Code Summary and Functions List

One of the first things a developer would want to do after opening a new code file is to glance at it and get information. Which file is this? What are the classes in this file? What are the functions in each class? VI developers get this information using standard navigation techniques (find function, read code one line at a time, etc.). In CodeTalk, we introduce a code summary feature. Using this, a developer gets an accessible tree view [Figure 2] containing the details about the namespaces, classes, and functions in the file. The developer³ using a screen reader can explore the tree view and get an overall understanding of the code structure. Additionally, they can also navigate to the desired code component by pressing the enter key. The code summary feature helps developers using screen readers get a “glance” of the different code constructs in the file.

We realized that one of the major constructs all developers frequently interact with are functions in a code file. To enable quick glanceability and navigability across functions in a file, we introduce a functions list view [Figure 3] that displays an accessible list view of all the functions in the current code file. Both the code summary and the functions list feature enable code glanceability and quick navigation of code.

Get Context of Current Line

Another important observation we made was that focus can move across lines or even code files while debugging or jumping into function definitions. In these scenarios, a VI developer might be interested to know the context with respect to the current line of code, at which the cursor is placed. Keeping this in mind, we introduce a feature that displays an accessible list view of context hierarchy

³ From here on, by mentioning developer we mean VI developer, unless explicitly stated otherwise.

containing the enclosing function, classes, and namespace that the current line of code belongs to.

Real-time Error Information

Most IDEs represent the syntax error in the code via syntax coloring. In Visual Studio, this is done via red squiggles [Figure 1]. We bring this visual information to VI developers via pro-active error tones informing the developer about syntax errors. The developer can then press a keyboard shortcut to get an accessible list of errors.

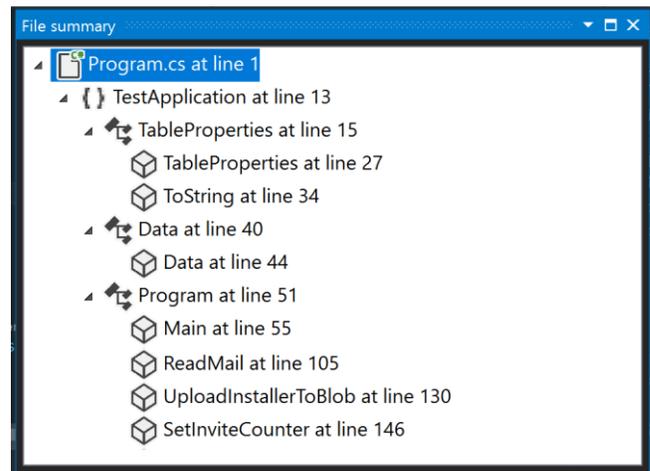


Figure 2. Code Summary containing tree view of code constructs

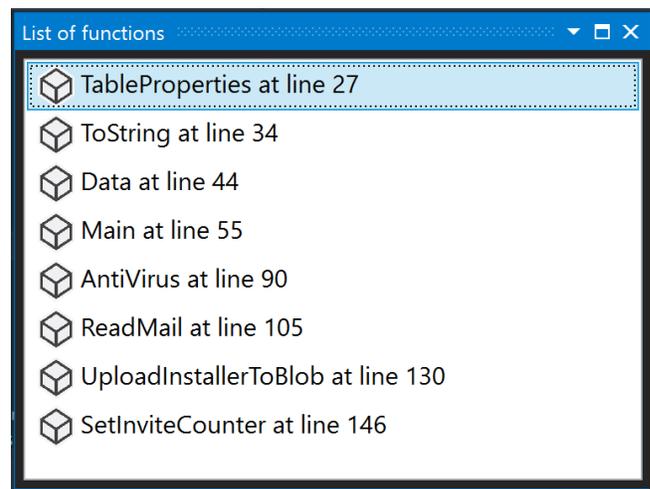


Figure 3. List of functions in the current code file.

Audio Debugging with TalkPoints

Debuggers are highly effective tools that assist developers to identify bugs in their code. However, using debugger tools is not a very accessible experience and VI developers prefer printing console messages for debugging instead of using a proper debugging tool [9]. Though “printf debugging” can get the job done for small projects, the process gets very cumbersome for larger projects. Also, the code gets cluttered with these debugging messages and needs to be cleaned up later. We propose a novel approach to debugging with audio using both speech and non-speech

cues. There have been tools like WAD [18] and Sodbeans [19] that explore audio for debugging source code. WAD, for instance, focuses on conveying the execution flow to the user. Though this is a very important piece of information, developers often need to know this piece of information with respect to very small parts of the code. Our approach to audio debugging is different as it (I) gives developers the option to choose between speech and non-speech based debugging and (II) gives developers information about specific variables or evaluates an expression in the execution context. (III) gives an option to break or continue execution after the audio cue. We have conceptualized and implemented 3 types of TalkPoints: *Speech Talkpoints*, *Tone Talkpoints* and *Expression Talkpoints*.

Feature	Keyboard Command
Code summary	Control + ~, Control + m
Functions list	Control + ~, Control + f
Get context	Control + ~, Control + g
Move to context	Control + ~, Control + j
Error information	Control + ~, Control + e
TalkPoints	Control + ~, Control + b

Table 1. CodeTalk keyboard shortcuts.

Steps to add a Talkpoint are as follows:

1. Invoke add TalkPoint dialog, from the desired cursor position by pressing a key combination. [Table 1]
2. Select the TalkPoint type.
3. Choose whether to pause or continue execution using the continue checkbox.
4. Activate the TalkPoint using the add button.

Speech TalkPoints

Speech TalkPoints are similar to adding trace statements. However, one small yet significant differentiating factor is that speech TalkPoints speak out the message set by the developer when they are hit without the developer having to explicitly switch focus and search in the trace window.

Tone TalkPoints

Our rationale behind proposing and implementing Tone TalkPoints was that developers often need to know only the execution path. For instance, the developer might want to know whether the execution entered an if, else or a catch block. The developer can select from the list of distinguishable tones while setting the TalkPoint. The tone is played when the TalkPoint is hit.

Expression TalkPoints

In many situations, developers are interested to know the value of a variable with respect to the execution context. With expression TalkPoints, we give developers the ability to have values of specific variables spoken to them when

these TalkPoints are hit. Assume the user wants to insert an expression in the following code:

```
int[] array = { 1, 2, 3, 4, 5, 6, 7, 8 };
int count = 0;
for (int i = 0; i < array.Length(); i++)
{
    count = count + array[i];
    //do something here.
}
```

Let us say the developer wants to track the value of the variable “count”. They can simply insert an Expression TalkPoint at line 5 as “*value of count is:*” + count. When the program is executed, the expression is run in the current breakpoint context, and the result is spoken to the developer. In the above case it will be: “value of count is 0”, “value of count is 1”, etc.

CodeTalk Design

CodeTalk’s design is both modular and extensible. Even though the current implementation is for Visual Studio IDE, CodeTalk can be easily implemented for other IDEs and even other languages. CodeTalk mainly consists of the following components.

- Keyboard manager
- Command objects
- Plugin outputs
- Language service and language specific implementations

Keyboard manager: This is responsible for capturing keyboard shortcuts, validating them and relaying it to the appropriate command objects

Command objects: These objects encapsulate the end to end functionality for a specific user command and send the output to the appropriate output block.

Plugin output: This module handles outputs from the command object. The output can be of various forms:

- *Dialogs:* Visual Studio dialog containing output entities in a list or tree view. For instance, function list command gives a dialog containing list view of all the functions.
- *Editor modifications:* Move the cursor to a specific line in the code file. For instance, move to context command moves the cursor to the beginning of the context block.
- *Audio:* Synthesize and send audio to the default system audio output using Microsoft Speech Synthesis APIs.

Language service and language specific implementations:

At the heart of CodeTalk design is the Language service block. This block defines a set of interfaces which are invoked by the command objects. The language-specific implementation modules implement these interfaces to add support to the corresponding language.

The C# implementation in CodeTalk leverages the Roslyn APIs [5] to implement the Language service interfaces. The implementation mainly consists of visitor patterns on the abstract syntax tree to accumulate the required entities and context.

For Python implementation, CodeTalk uses IronPython APIs [3]. For functionalities that require keyboard shortcuts, we chose key commands similar to those provided by Visual Studio. We also intend to make these keyboard shortcuts customizable.

Bootstrapping CodeTalk

One of our authors, B, has been implementing and using CodeTalk since its initial implementations. This exercise helped us evolve CodeTalk's feature set based on the author's needs. Also, the initial user survey was a reference to us to ensure that the features we implement would help a larger audience.

Author B was already familiar with using an IDE and was encouraged by the improvement in productivity due to CodeTalk right from the first set of features implemented: the functions list view and the code summary. The author used the plugin to implement subsequent features and could see immediate benefit while trying to make sense of code written by other members of the project.

The next set of features implemented were get context, move to context and error information. The get context and move to context features helped B quickly understand and navigate classes. Though a reasonably experienced programmer, B was relatively new to C# programming language. Prior to implementing the error list and real-time error information features, the author had to fix syntax issues only by building the project. This build and fix approach was a major productivity hiccup for B, as the project took at least 5 minutes to build and syntax errors were not available until the build completed. B observed a significant improvement in productivity due to the error list and real-time error information features as it didn't require explicitly building the project; Another major observation was that compiler error messages were easier to understand if attended to immediately as opposed to building after accumulating a few of them. Prior to implementing TalkPoints, the author B was very reluctant to use a debugger, often resorting to printf debugging. There were several occasions when B received code review comments asking for the removal of printf/log statements.

To verify if our approaches helped more developers, we performed an exploratory user study with 6 VI developers proficient with coding. We excluded novice programmers and those learning to program from this study since our current focus is not on discoverability, but to improve the productivity of already competent VI developers.

EXPLORATORY USER STUDY

We conducted an exploratory study with an objective of getting feedback from active Visual Studio programmers to

validate the direction we were taking and to get a preliminary idea of the utility of CodeTalk's features. As mentioned in the conclusion, a rigorous study is needed to identify the strengths and drawbacks of our approach. The study had four major components: Participant solicitation, user study without and with CodeTalk, and post user study online survey.

Participant solicitation

We circulated a short online survey to get basic information of interested participants. We wanted participants who code in C# or Python using Microsoft's Visual Studio 2015 and above.

We selected 6 participants who were reasonably experienced with writing code in C# and using Visual Studio. All the participants opted in to the study by sharing their email id and signed a consent form regarding our terms of study.

Setup for the Study

To observe developer's usage, we setup a remote Virtual Machine (VM) on Microsoft's Azure platform. The VM had the NVDA screen reader installed. For both the phases, we wanted developers to be in their most comfortable screen reader setup. To ensure this, we gave developers the credentials to connect to the VM a few hours in advance to the scheduled study time. Developers were also allowed to install any screen reader plugins and configure the screen reader to match their preferences. Most participants using NVDA preferred to connect using the NVDA Remote add-on. However, we requested participants to switch to Microsoft's remote desktop to perform tasks 3, 4, and 5 of phase 2 as the NVDA Remote add-on does not pass through system audio. However, switching to a remote desktop did not result in any change in screen reader behavior.

JAWS users, however could not use the remote VM as JAWS does not allow activations on Virtual Machines even with the remote desktop add-on. We allowed participants using JAWS to connect to a physical machine via JAWS Tandem or remote desktop.

Participants connected with us over a Skype audio call and shared their screen with us. This helped us observe user behavior. We recorded participant's microphone audio, our microphone audio and their screen's video for our observation and further analysis.

Phase 1: Performing programming tasks without CodeTalk

In this phase, participants were asked to perform 5 programming tasks using Visual Studio. We wanted to observe participants IDE usage without our enhancements. This phase also helped us better introduce our problem and plugin to the participant. Before performing the tasks, we asked participants about the general issues they faced as a VI developer when using IDEs.

The programming tasks we chose did not require developers to switch between multiple files. The participants performed the following tasks.

1. Give the hierarchical structure of a code file (namespace, classes, and methods) in a test project.
2. Go to a specified line in a code file using Visual Studio's go to line function and give us the context (enclosing method, class, and namespace information) with respect to the current line.
3. Open a code file and fix syntax errors in it.
4. Identify if running a project results in the execution flow going through a catch block. Participants were not allowed to modify the code. We allowed participants to modify the code if they were unable to perform the task without modifying the code.
5. Give the value of a variable in a loop in iteration "i" without modifying the code. The list in the loop was reading data from a file and participants were not allowed to look at the file. Participants were allowed to modify the code if they were unable to perform the task.

Phase 2: Performing programming tasks with CodeTalk Installed

On completion of phase 1, we introduced participants to CodeTalk, our accessibility plugin for Visual Studio. Participants were allowed to explore the plugin after our walk-through and we also made sure they had access to all the keyboard shortcuts in case they wanted to refer. Participants were given the same tasks as in the previous phase albeit with different code files. We did not make the use of CodeTalk mandatory for this phase. The participants could choose to use CodeTalk if they wanted to. We wanted to observe the developers' behavior given the tool. After the tasks, we asked the developers 3 questions.

1. How was your experience in doing the task in both Phase 1 and Phase 2?
2. Was there any more information you wish you had while doing this task?
3. How often do you encounter these tasks in your day to day programming?

After these questions, the participants were asked to give general feedback on the plugin and the user study. Towards the end of the call, we asked participants to fill a short online survey⁴.

Participant demographics

We had a total of 6 participants in the exploratory study. All participants have been coding for more than 1 year. Two of them have been programming for about 3-5 years, one for about 5-10 years and two for more than 10 years [Table 2]. All participants were male and completely blind. Five of the

⁴ We asked for their email ID in the survey for compensating them later and mentioned this in the survey.

participants reported they have been using a computer for more than 10 years. Participants were from the United States, United Kingdom, Spain, India and Romania. All participants were familiar with C#.

Observations from the User Studies

[Table 3] shows the average rating for our plugin's features. Participants were asked to rate the plugin's features on a scale of 10 (1 being not useful and 10 being very useful). CodeTalk's utility was rated on average 8.83 by the participants. We also describe our observations on participant's IDE usage while performing tasks in both phases.

Participant	Programming Experience
P1	1 – 2 years
P2	3 – 5 years
P3	3 – 5 years
P4	Above 10 years
P5	5 – 10 years
P6	Above 10 years

Table 2. Participant Demographics

Feature	Average Rating (on a scale of 10)
Navigability features (Code summary, Get context, Function list, etc.)	8.83
Real time error information (Pro-active error beeps and Error list)	8.33
Audio debugging (Tonal, Textual and Expression TalkPoints)	8.5

Table 3. Participant ratings of CodeTalk features.

Task 1: Reporting code summary

In phase 1, 3 out of the 6 participants navigated through code one line at a time to give us the summary. P3, P4 and P6 used one of the IDE's features to navigate through different class and functions of the file. P4 and P6 had developed their own navigation techniques using some of Visual Studio's features. P4 first navigated to the beginning of the namespace and then to the end. He followed a similar approach for all the blocks. However, this technique involved navigating through code one line at a time. P6, on the other hand, navigated by first folding the code and then navigating through the folded code. In phase 2 however, all participants preferred to use CodeTalk's code summary feature to report the summary. All participants except P2 mentioned that this is very useful to quickly understand large code files and code written by other developers. P2 however mentioned that they work on their own code most

of the time and so would not need to get the summary of code. However, they mentioned that this could come in handy in situations where they have to look at other people's code. P3, who used the IDE's feature to get the structure of code preferred to use CodeTalk as well. *"Using this code summary does not require me to move focus away from my IDE; I know that pressing enter or escape on the dialog box will get me back to the file I was working on."* was P3's feedback on completing task 1. P4 commented: *"Having a keyboard shortcut to get the tree structure, is very nice. It is just there. I do not have to use my methods anymore. This is better as it is right there and gives me just the summary."*

Task 2: Report context of a specific line

In this task, participants were asked to go to a line using Visual Studio's "go to" line feature. Participants were then asked to report the context (enclosing function, classes and namespaces) that the line belongs to. 3 out of the 5 participants preferred to navigate through the code one line at a time. The code had a nested class. 3 out of the 5 participants however, missed reporting this outer class as they had moved all the way to the top to report the namespace after finding one of the class's declaration statements. In phase 2 however, all participants chose to use CodeTalk's get context feature to complete the task. They mentioned that this feature would come in handy specially when they want to debug or when they are taken to a line of code by the IDE due to a breakpoint or exception.

Task 3: Fix syntax errors and build

In this task, developers were given code that had syntax errors. Participants had to fix the errors and then build the project. The initial action of all the participants excluding P2 was to try and read the code. Then, all participants except P2 built the project to check for syntax errors. P2 used other IDE features to fix the errors. In phase 2, all participants except P2 preferred to use CodeTalk's Error information features as it did not require building the project explicitly.

Task 4: Report whether the catch block is executed

In this task, developers were given a code file with a try and a catch block and were asked whether the catch block be executed if the code is run. The initial constraint for this task was that the participants could not modify code. The rationale behind putting this constraint is to see whether participants were familiar with breakpoints. 3 out of the 6 participants could not perform this task without modifying code; They mentioned that they did not find debuggers accessible and did not use breakpoints. They usually debugged by placing console statements (printf debugging). Participants could report the answer to us once we allowed them to modify code. In Phase 2, Participants were able to perform this task very easily and they chose to make use of CodeTalk's Tone TalkPoints to identify whether the catch block was executed. *"I like the idea of breakpoints not breaking, and simply continuing after playing the audio."*, exclaimed participant P2.

Task 5: Find value of a variable at runtime

In this task, developers were given code that iterates a list of numbers in a for loop and adds them to a variable sum. The numbers are populated in the list using a file. To perform this task, participants had two major constraints:

- Participants cannot modify code.
- Participants cannot read the file from which the values are loaded.

All participants except P1 could complete this task in both phases. In phase 1, 4 out of the 6 participants could not do it without modifying code. We then allowed participants to modify code. 3 out of these 4 participants could report the value by adding console statements (printf debugging) and 1 participant, P1, could not finish the task in both phases. In the second phase 4 of the 5 participants who finished the task used Tone TalkPoints whereas 1 participant, P5, used a combination of a Tone TalkPoint and Visual Studio's locals window to check for variable values.

Participants responded positively when they were asked whether they encountered these tasks as a part of their day to day programming. P2 however, did mention that they did not encounter task one (reporting the summary of the code) frequently as they mostly work on their own code, but also said *"this is definitely useful for situations where I have to look at other people's code"*. *"Yes, I find myself doing these things quite frequently, during my assignments"* was P3's feedback.

Participants were asked about their experience about the plugin and the user study in general. *"I never knew how much information I was not getting because I was using a screen reader. I had no clue sighted users had this much information available."* said P1. P1 also mentioned that they had difficulty in sorting through code in the post user study survey. *"I have difficulty to sort through code. Perhaps this is due to my vision impairment and not really an accessibility issue"* said P1. It was a surprising observation for us that these inaccessibilities were considered by VI developers as consequences of their impairment and not deficiencies in accessibility of the tool.

DISCUSSION

We believe that stepping back and looking at the nature of accessibility challenges in the use of IDEs has been very fruitful. The organization of these into four categories, discoverability, glanceability, navigability and alertability, has given us a structure to classify specific problems and to solve them using the accumulated tools built to solve earlier problems. In implementing CodeTalk we identified two key ideas to help address these problems: the first is to extract relevant information from the IDE that is spread around visually and present them directly in summary form to the VI developer. The second is to present additional information through a secondary audio channel distinct from the screen reader. A combination of these two ideas have been used to address a subset of the identified

challenges in the current version of CodeTalk. However, this systematic framework has opened numerous possibilities for future research that we outline below.

The notion of TalkPoints has tremendous promise, not just for VI developers, but even for sighted users. The introduction of the auxiliary audio channel has literally opened up additional bandwidth for the users. In particular, expression TalkPoint has the potential to monitor and announce subtle inter relations between functions and can be a powerful debugging tool.

Promising initial user feedback shows that our approach and CodeTalk have a positive impact on VI developers' productivity. It has also given us considerable feedback and additional insights which we intend to build on. However, we need to explore evaluation metrics for the effectiveness of these solutions and conduct more systematic user studies. How can we say CodeTalk has enhanced productivity? Do we measure the time taken to accomplish individual tasks with and without CodeTalk? Or since VI developers using IDEs depend extensively on keyboard shortcuts, should we measure this improvement by logging keystrokes? Do we just compare VI developers with and without CodeTalk or compare VI developers with sighted users since the ultimate goal of such accessibility work is to bridge the gap between the two? These are some of the many interesting questions that we have begun to grapple with in evaluating CodeTalk.

We also want focus on a broad class of issues that fall under discoverability. Currently, getting started with Visual Studio and similar IDEs requires significant hand-holding from sighted peers. The discoverability issues are a major reason for author A not switching to an IDE. Even experienced VI developers who have used an IDE for a long time are frequently surprised by new features they stumble upon accidentally. Given the complexity of modern IDEs, it is not practical to go through each one of the menu items or to exhaustively read the user manual to discover all the features.

In addition, this is rarely useful for a novice programmer and unproductive for experienced users. We need to devise new techniques that can gently induce the user to discover features when it is most useful. Such discoverability, even for sighted users, is still a challenge and it is a wide-open area of research.

Control of navigation granularity is a very widely used feature by screen reader users. Web navigation is generally through different HTML elements like headings, form controls, links etc. We would like to explore similar granular navigation techniques specific to code especially for easy navigation through classes, functions and inner code blocks.

Our choice of implementing CodeTalk as a plugin allows us to build these solutions in a manner that can easily be ported across IDEs. We have open sourced our implementation to facilitate further rapid development and

research⁵. Additionally, from user feedback, there is a need for CodeTalk to support more popular scripting languages like JavaScript.

CONCLUSION

We grouped the numerous accessibility challenges faced by VI developers in using GUI based programming environments into four categories, namely, discoverability, glanceability, navigability and alertability. We presented CodeTalk, a plugin for Visual Studio that enables VI developers to overcome some of these challenges. Participants in the exploratory user study have given very positive feedback on the utility and potential of CodeTalk to improve accessibility. We also presented several possible research directions that emerge from this work.

ACKNOWLEDGEMENTS

We would like to thank Indrani Medhi Thies, Saqib Shaikh and Sujeath Paredy for insightful discussions and suggestions.

REFERENCES

1. Y Vidhya. 2017. Preliminary Survey Responses. Supplementary material - SurveyResponses.pdf. (2017).
2. 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*. ACM, 3043–3052.
3. IronPython Community. 2017. IronPython. Retrieved September 19, 2017 from <http://ironpython.net/>.
4. Microsoft Corporation. 2017. Microsoft Visual Studio. Retrieved September 19, 2017 from <http://www.visualstudio.com>.
5. .Net Foundation. 2017a. Roslyn. Retrieved September 19, 2017 from <https://github.com/dotnet/roslyn>.
6. The Eclipse Foundation. 2017b. Eclipse. Retrieved September 19, 2017 <https://eclipse.org/>.
7. Apple INC. 2017. XCode. Retrieved September 19, 2017 from <https://developer.apple.com/xcode/>.
8. Shaun K Kane and Jeffrey P Bigham. 2014. Tracking@stemxcomet: teaching programming to blind students via 3D printing, crisis management, and twitter. In *Proceedings of the 45th ACM technical symposium on Computer science education*. ACM, 247–252.
9. Sean Mealin and Emerson Murphy-Hill. 2012. An exploratory study of blind software developers. In *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*. IEEE, 71–74.

⁵ <http://github.com/Microsoft/CodeTalk>

10. U.S. Bureau of Labor Statistics. 2016. Software Developers: Occupational Outlook Handbook:U.S. bureau of Labor Statistics. Retrieved September 19, 2017 from <https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm>.
11. Stack Overflow. 2017. Stack Overflow Developer Survey 2017. Retrieved September 19, 2017 from <https://insights.stackoverflow.com/survey/2017>.
12. Charles B Owen, Sarah Coburn, and J Castor. 2014. Teaching Modern Object-Oriented Programming to the Blind: An Instructor and Student Experience. In *ASEE Annual Conference*.
13. Guillaume Pothier, Éric Tanter, and José Piquer. 2007. Scalable omniscient debugging. *ACM SIGPLAN Notices* 42, 10 (2007), 535–552.
14. TV Raman. 1996. Emacspeak-A Speech Interface. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 66–71.
15. Reddit. 2017. Can I still be a Computer Scientist if I'm Blind?:cscareerquestions. Retrieved September 19, 2017 https://www.reddit.com/r/cscareerquestions/comments/3e844q/can_i_still_be_a_computer_scientist_if_im_blind/.
16. Dominic Roberts and Karlton Weaver. 2011. Audio Aids in Source Code. Retrieved September 19, 2017 from http://archive2.cra.org/Activities/craw_archive/dmp/awards/2011/Roberts/FinalPaper.pdf.
17. Jaime Sanchez and Fernando Aguayo. 2004. Listen what I do: blind learners programming through audio. *Memorias TISE* (2004), 120–124.
18. Andreas Stefik, Roger Alexander, Robert Patterson, and Jonathan Brown. 2007. WAD: A feasibility study using the wicked audio debugger. In *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*. IEEE, 69–80.
19. Andreas Stefik, Andrew Haywood, Shahzada Mansoor, Brock Dunda, and Daniel Garcia. 2009. Sodbeans. In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*. IEEE, 293–294.
20. Andreas M Stefik, Christopher Hundhausen, and Derrick Smith. 2011. 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*. ACM, 571–576.
21. U.S. Bureau of Labor Statistics. 2016. Economic News Release: Table 3. Employed persons by disability status, occupation, and sex, 2016 annual averages. Retrieved September 19, 2017 from <https://www.bls.gov/news.release/disabl.t03.htm>.
22. Ann C Smith, Justin S Cook, Joan M Francioni, Asif Hossain, Mohd Anwar, and M Fayezur Rahman. 2004. Nonvisual tool for navigating hierarchical structures. In *ACM SIGACCESS Accessibility and Computing*. ACM, 133–139.