**Product Design**

**Project Name:** Mind\_Reader (pending)

**Team Name:** Single Semester Snobs, We Don’t Byte, Five Guys, KJBK

(Sophia Drewfs, Mason Bone, Jake Grossman, Josiah Moses, Cal Wooten),

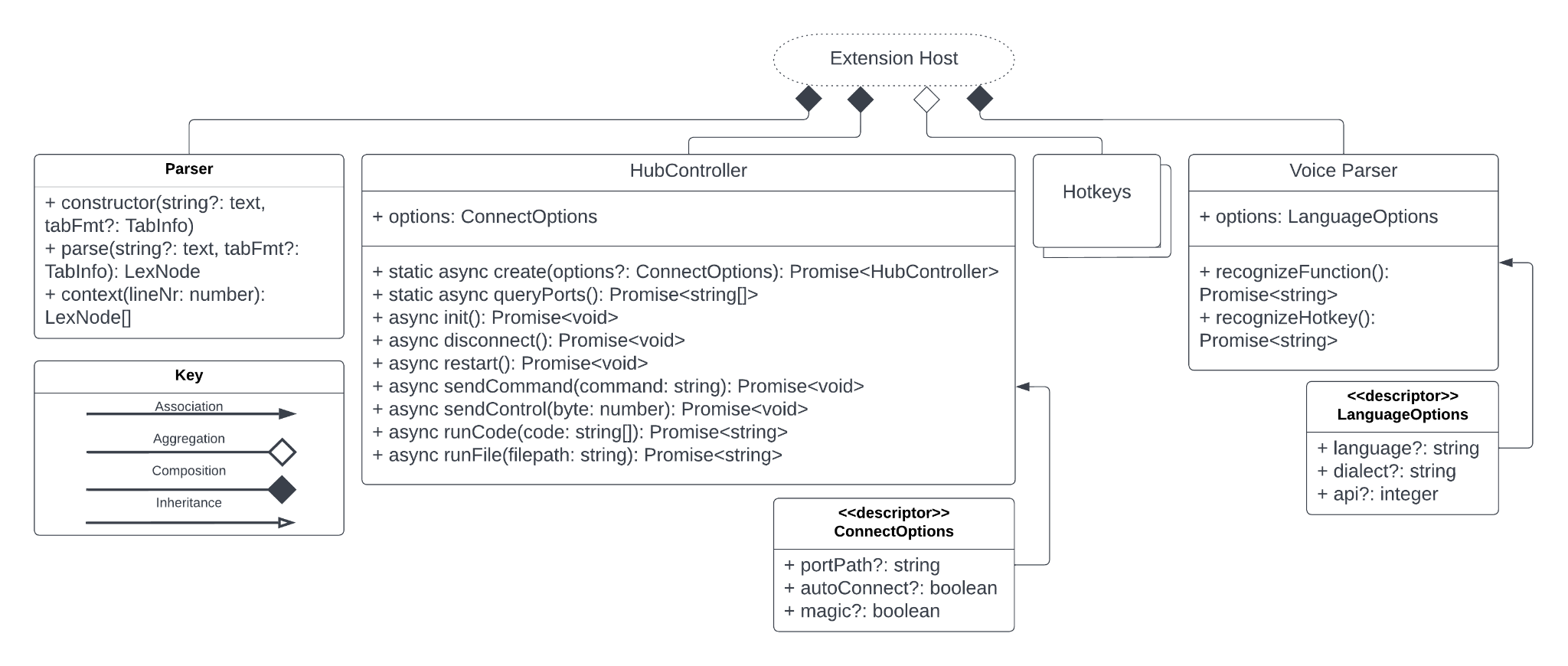
(John Breaux, Kendrick Johnson, Pedro Alvarez, Ryan Tolbert, Thomas Lane)

(Zachary Chenausky, Jigme Rinji Sherpa, Clay Lewis, Haris Javed, Saad Javed)

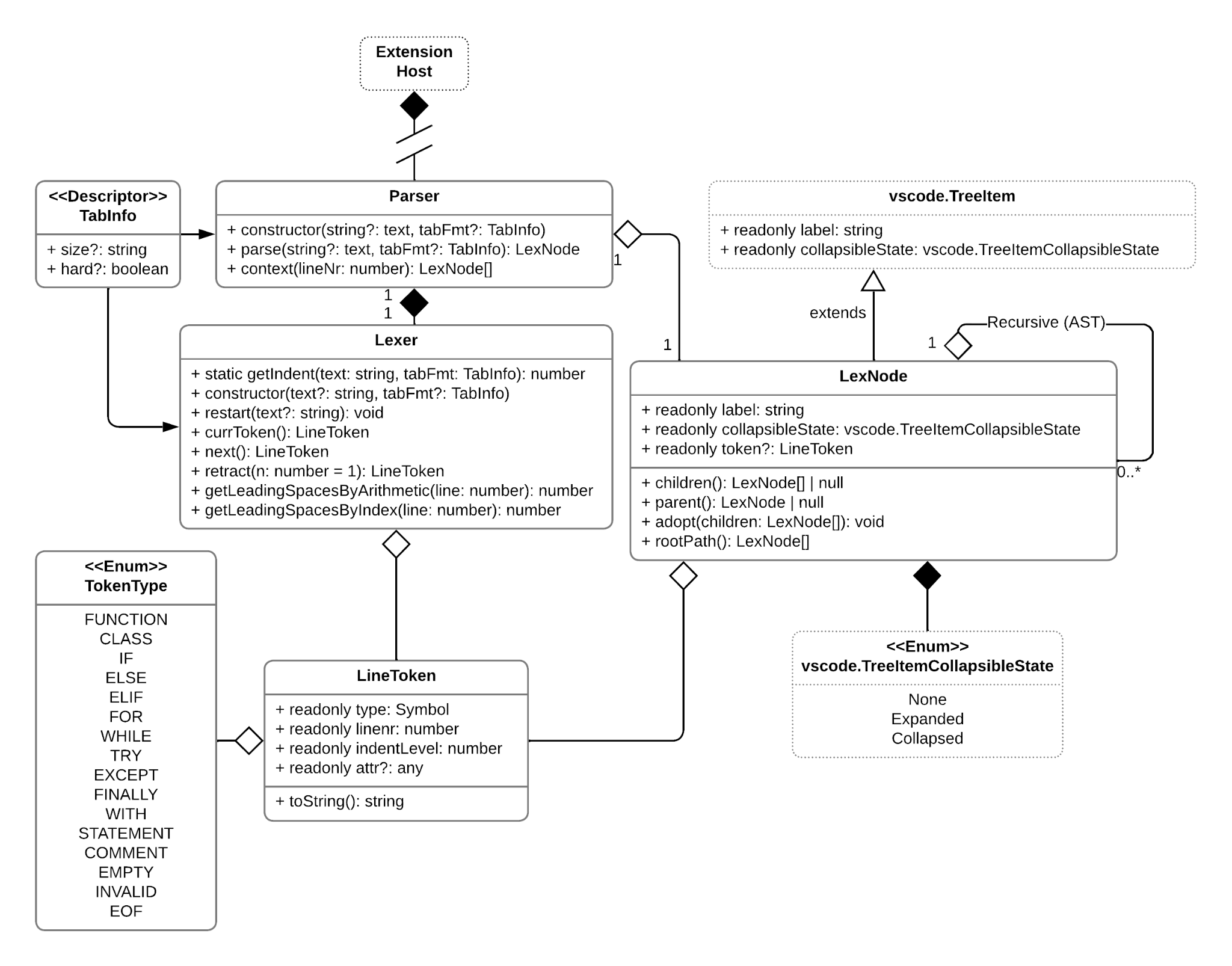
(Kyle Minter, Jacob Franks, Kennedy Boynton, Brett Berglund)

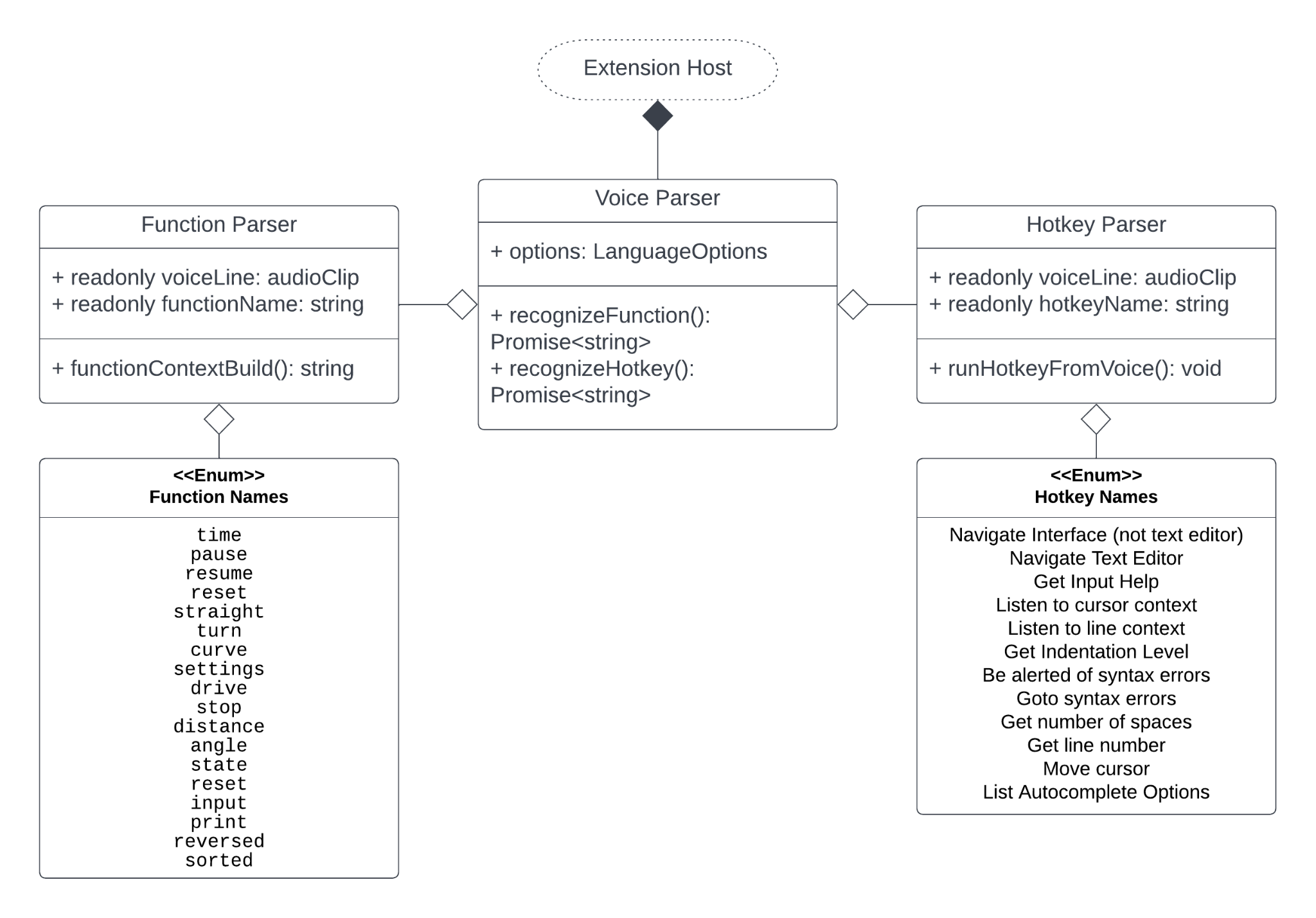
| Revision # | Revision Date | Summary of Changes | Author(s) |
| --- | --- | --- | --- |
| 0.1 | 10/12/2021 | Initial Design Doc | Mason Bone, Jake Grossman,  Sophia Drewfs, Cal Wooten,  Josiah Moses |
| 0.2 | 11/11/2021 | Updated Design of Diagrams | Mason Bone, Jake Grossman,  Sophia Drewfs, Cal Wooten,  Josiah Moses |
| 0.3 | 12/1/2021 | Final Update for Design Doc | Mason Bone, Jake Grossman,  Sophia Drewfs, Cal Wooten,  Josiah Moses |
| 0.4 | 3/5/2022 | Sprint 1 | John Breaux, Kendrick Johnson, Pedro Alvarez, Ryan Tolbert, Thomas Lane |
| 0.5 | 4/9/2022 | Sprint 2 | John Breaux, Kendrick Johnson, Pedro Alvarez, Ryan Tolbert, Thomas Lane |
| 0.6 | 5/7/2022 | Sprint 3:  Clarify wording of “the Hub” Update Information Architecture Diagram  Update Interface Wireframe(s)/Screenshot(s) | John Breaux, Kendrick Johnson, Pedro Alvarez, Ryan Tolbert, Thomas Lane |
| 0.7 | 10/8/2022 | Sprint 0  Update:  Information Architecture Diagram  Top Level Extension Diagram  Wireframe 3  Updates Made Section  Create:  List Autocomplete Options Sequence Diagram  Voice Hotkey Sequence Diagram  Speech to Function Sequence Diagram  Move Cursor Sequence Diagram  Voice Analysis System Class Diagram | Zachary Chenausky, Jigme Rinji Sherpa, Clay Lewis, Haris Javed, Saad Javed |
| 0.8 | 11/1/2022 | Continuing previous implementation of functional requirements that have not yet been completed. | Zachary Chenausky, Jigme Rinji Sherpa, Clay Lewis, Haris Javed, Saad Javed |
| 0.9 | 2/18/2023 | Added diagram explaining H13: indentation audio cue | Zachary Chenausky, Jigme Rinji Sherpa, Clay Lewis, Haris Javed, Saad Javed |
| 0.10 | 3/11/2023 | Continuing previous implementation of functional requirements as well as adding additional uses for the voice functions. | Zachary Chenausky, Jigme Rinji Sherpa, Clay Lewis, Haris Javed, Saad Javed |
| 0.11 | 4/8/2023 | Continuing previous implementation of functional requirements as well as adding additional uses for the voice functions. | Zachary Chenausky, Jigme Rinji Sherpa, Clay Lewis, Haris Javed, Saad Javed |
| 0.12 | 5/6/2023 | Only performing cleanup and bugfixing due to final presentation and deployment. | Zachary Chenausky, Jigme Rinji Sherpa, Clay Lewis, Haris Javed, Saad Javed |
| 0.13 | 10/3/2024 | Updated Information Architecture Diagram | Kyle Minter, Jacob Franks, Kennedy Boynton, Brett Berglund |
| 0.14 | 10/17/2024 | Updated Mind Reader Actions Menu Options Figure. | Kyle Minter, Jacob Franks, Kennedy Boyton, Brett Berglund |
| 0.15 | 10/30/2024 | Added Audio Flag System Diagram & Audio Flag Sequence Diagram to describe the design of the audio flag system | Kyle Minter, Jacob Franks, Kennedy Boyton, Brett Berglund |
| 0.16 | 10/31/2024 | Added Screenshot To Show Audio Selection When Adding Audio Flag to show how the UI prompt appears to the user | Kyle Minter, Jacob Franks, Kennedy Boyton, Brett Berglund |
| 0.17 | 12/04/2024 | Updated Audio Flag System Diagram, Audio Flag Selection Dropdown Screenshot, Mind Reader Actions Menu | Kyle Minter, Jacob Franks, Kennedy Boyton, Brett Berglund |
| .018 | 02/11/2025 | Updated Audio Flag System Diagram & Mind Reader Actions Menu.  Added Dropdown Menu for Searching for Audio Flags | Kyle Minter, Jacob Franks, Kennedy Boyton, Brett Berglund |
| .019 | 03/04/2025 | Updated Mind Reader Actions Menu & Dropdown for selecting note when place an audio flag | Kyle Minter, Jacob Franks, Kennedy Boyton, Brett Berglund |

# Class Diagrams

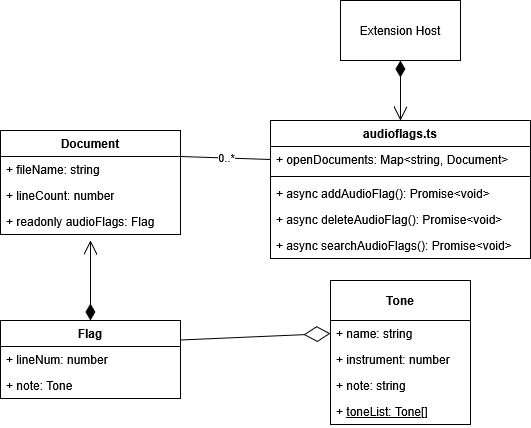


***Diagram 1:*** *Top Level Extension Diagram*

***Diagram 2:*** *Context Analysis System*



***Diagram 3:*** *Voice Analysis System*

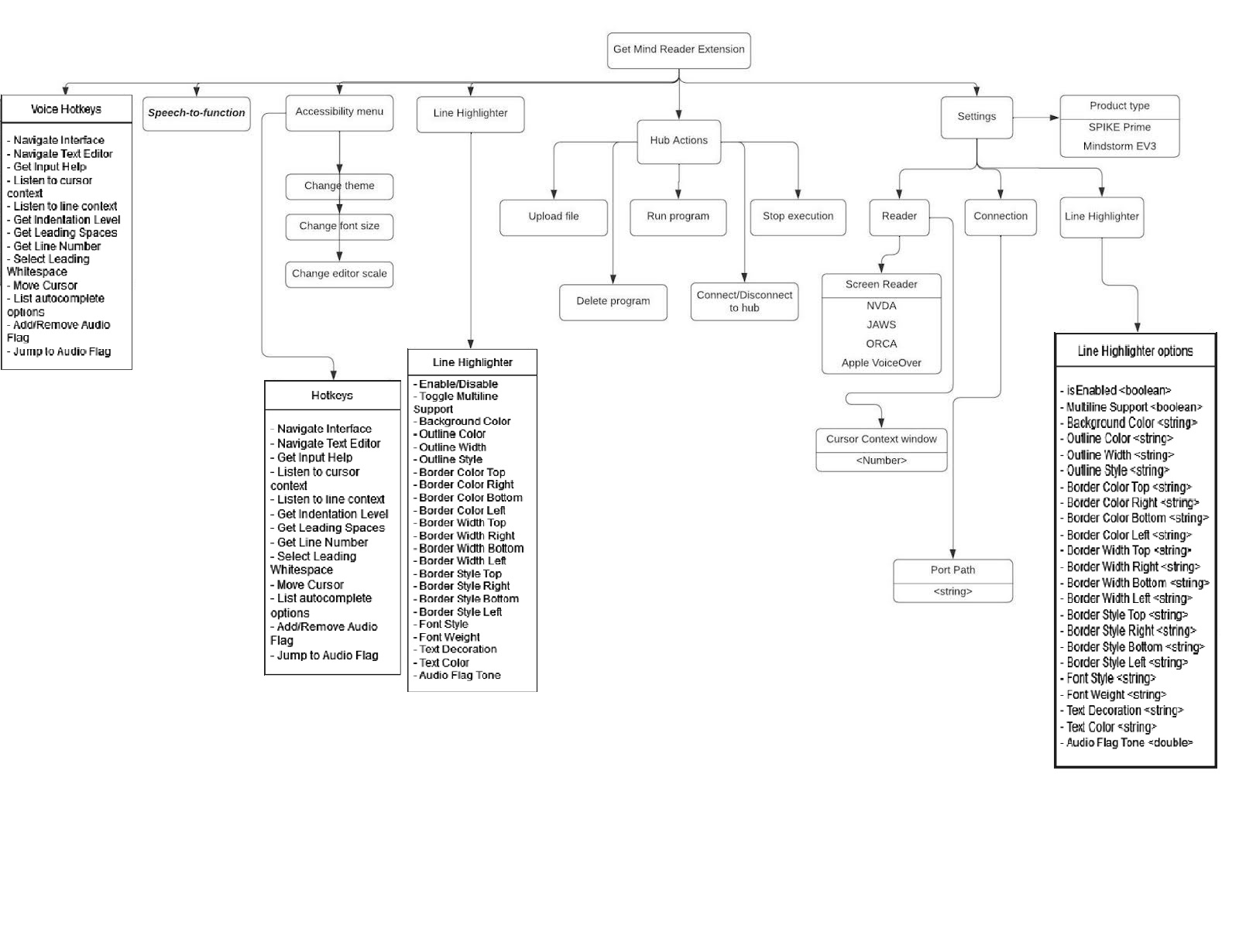


***Diagram 4:*** *Audio Flag System*

**ER Diagrams**

This extension does not utilize a database, therefore an ER Diagram has not been included.

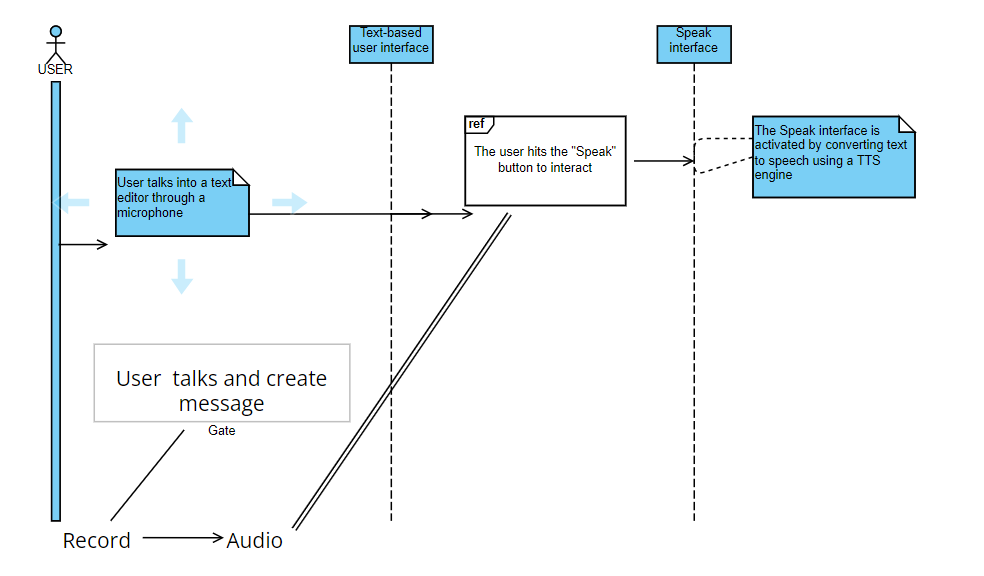
# Information Architecture Diagram



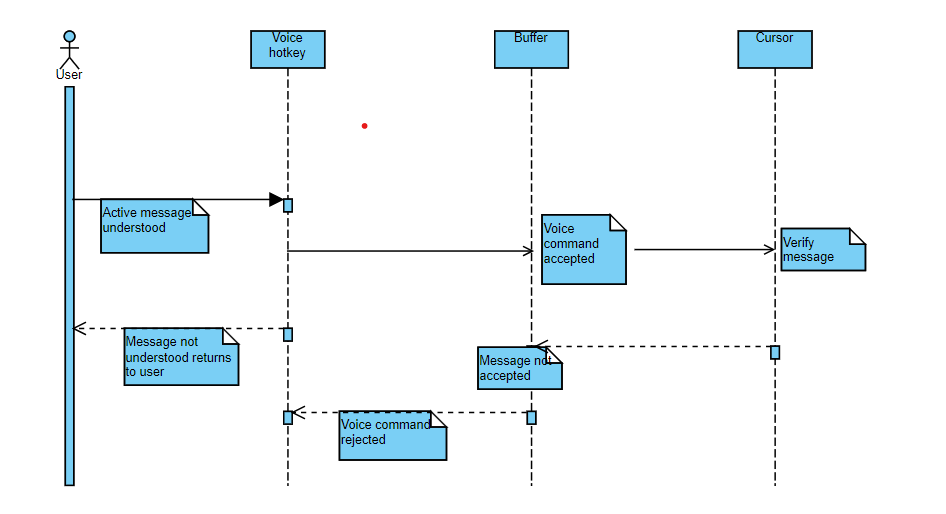
***Diagram 3:*** *Information Architecture*

# Sequence Diagrams:

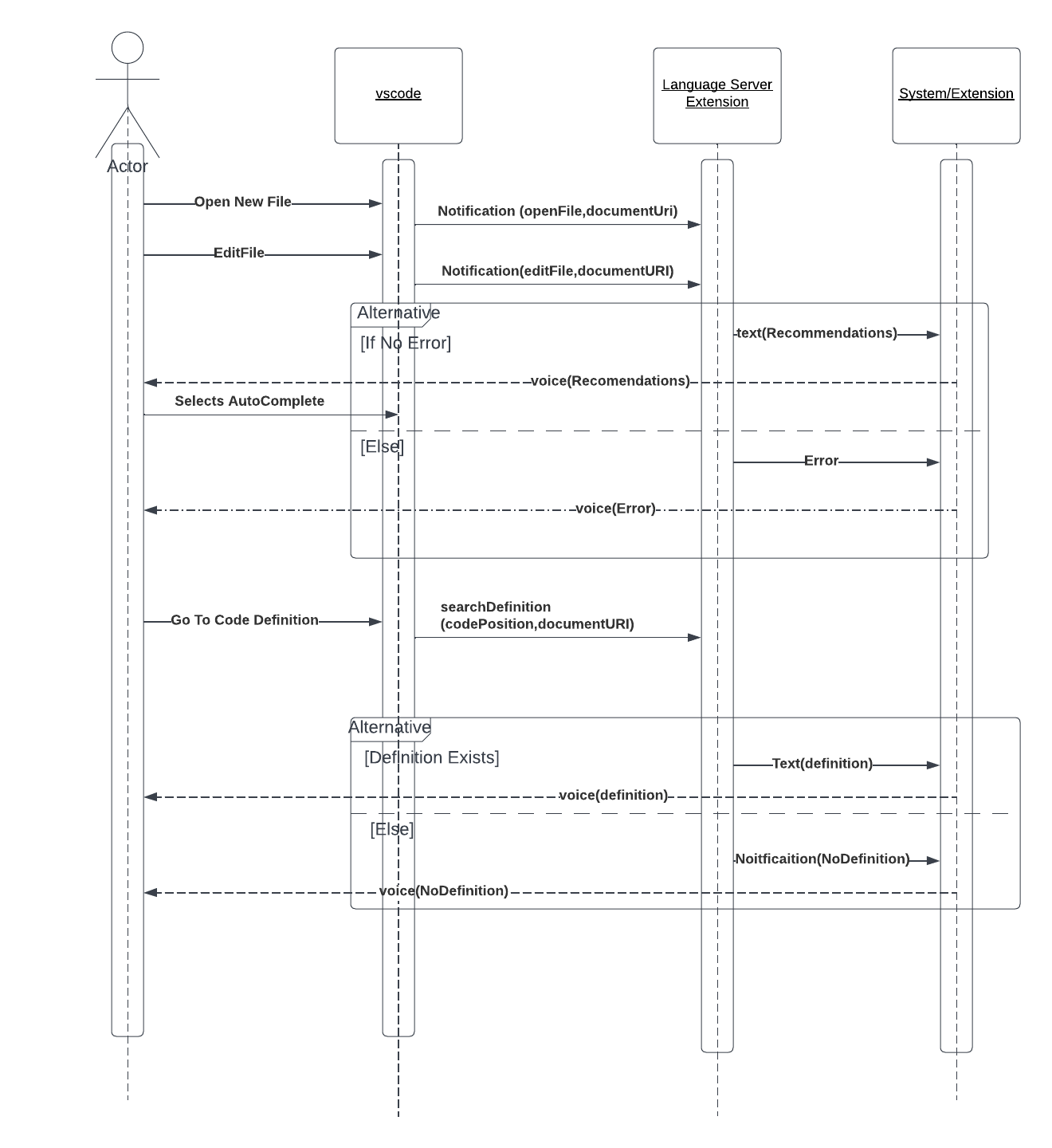
### Speech to function



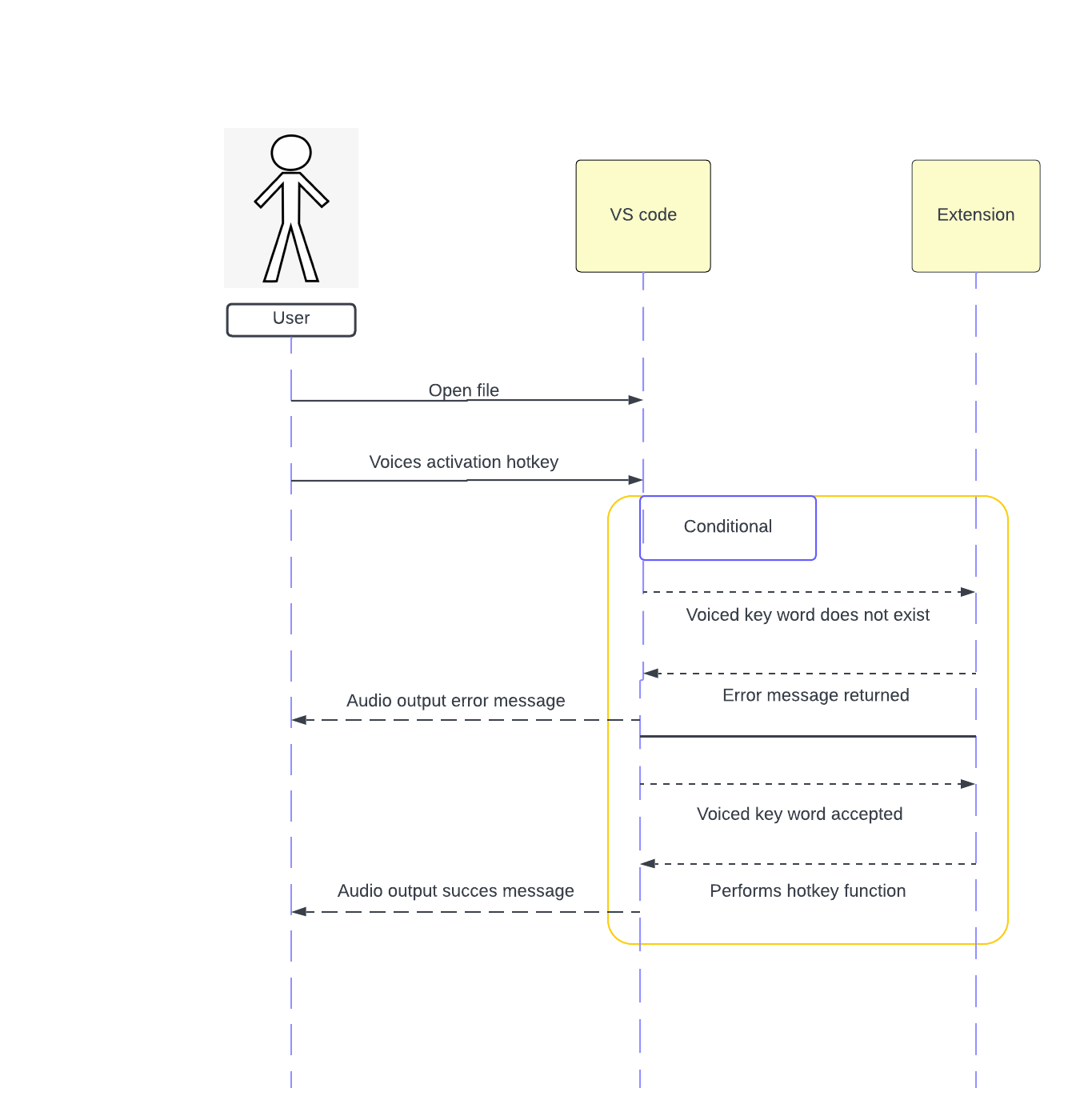
### Move Cursor Diagram

****

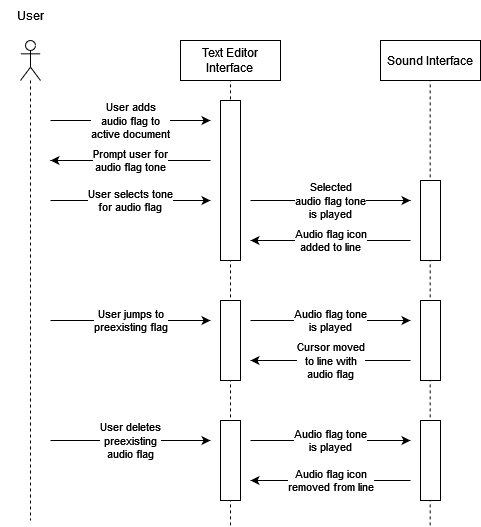
### Recommendation, Autocomplete



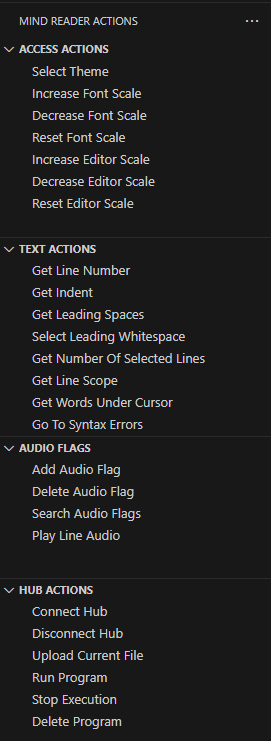
### Voice Hotkey

****

**Audio Flag System**



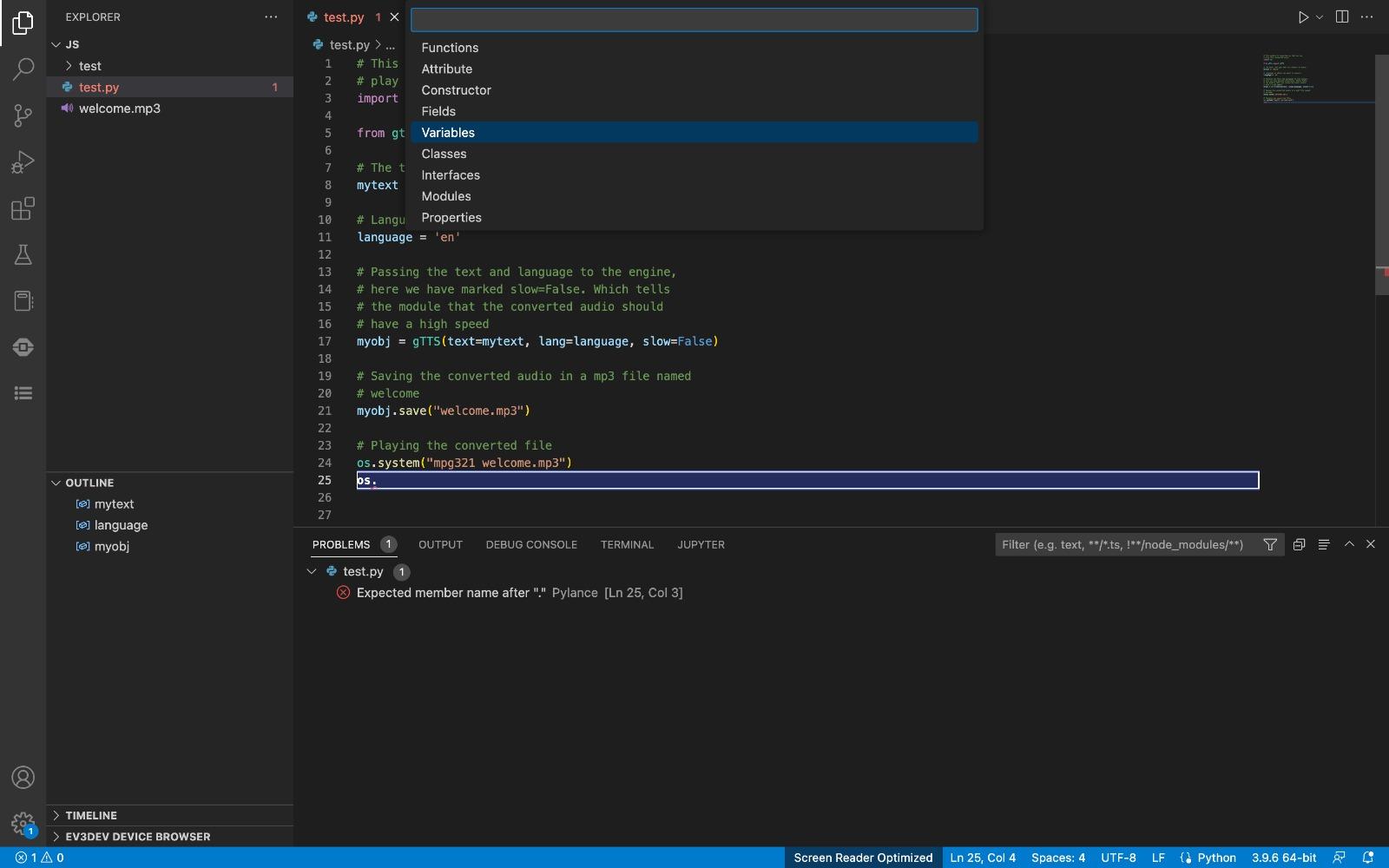
# User Interface Wireframe(s)/Screenshot(s)Visual Studio Code Settings (Representative Sample)



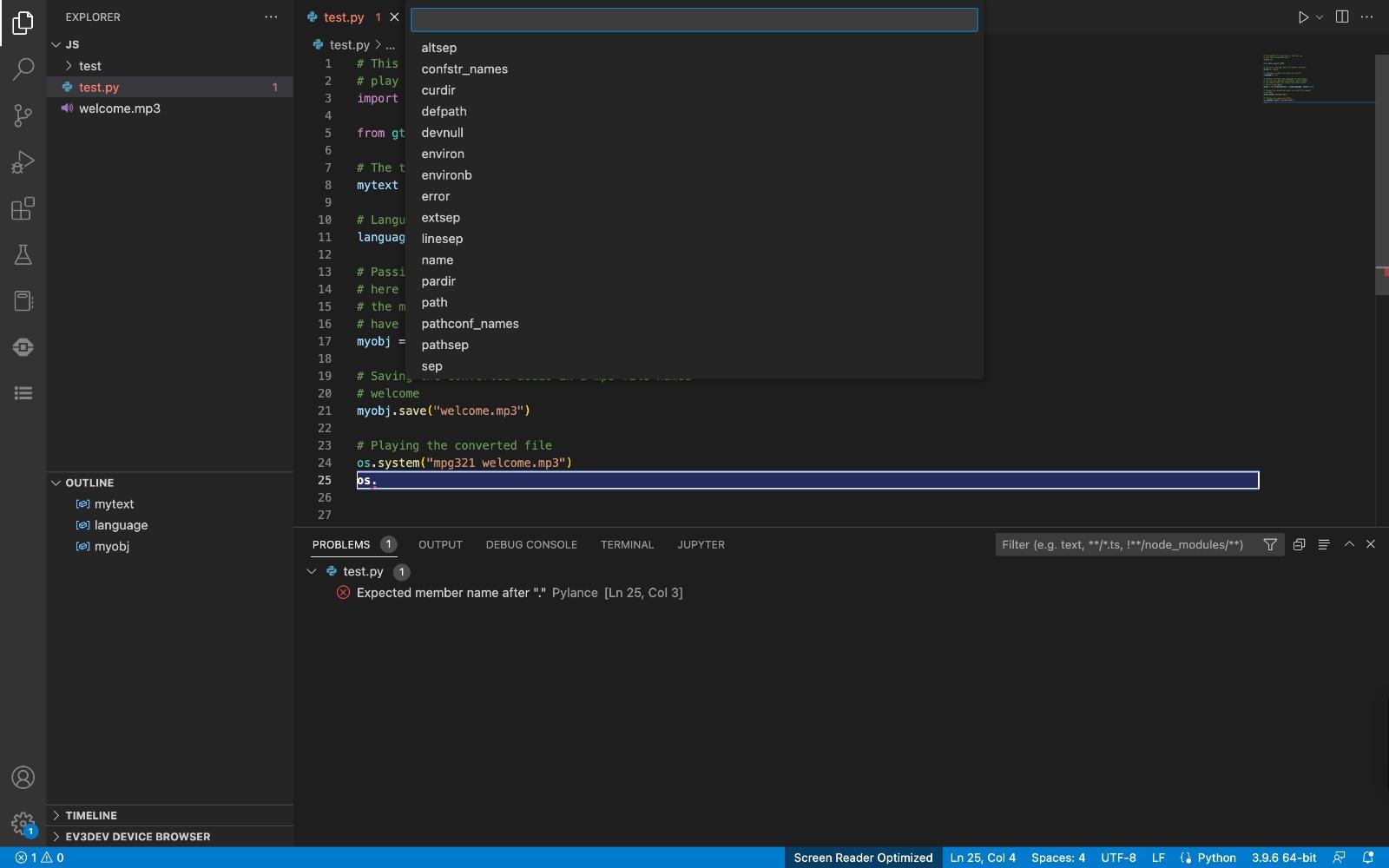
# Mind Reader Actions Menu Options

# 

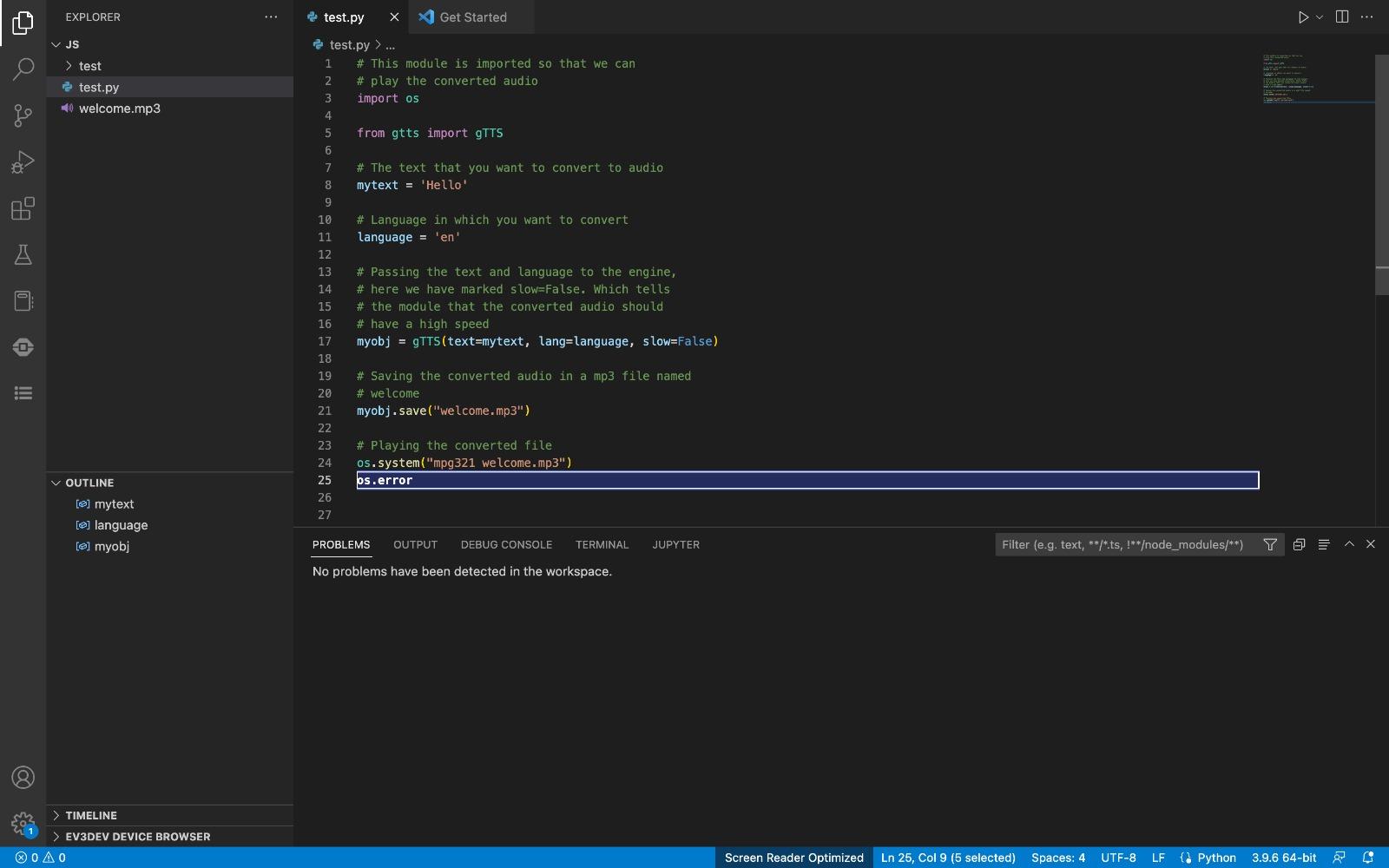
**Users select option for filter**

****

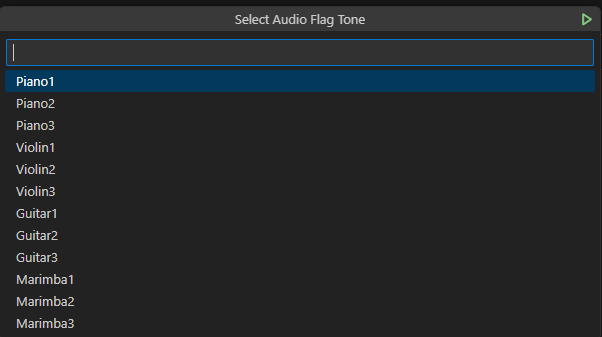
**Users select the type of suggestion**

****

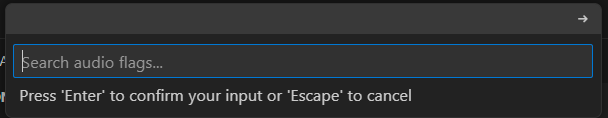
**Extension provides completion item based on their choice**

****

**The selected text gets completed on text editor**

****

**Dropdown for selecting note when placing an audio flag**

****

**Dropdown menu for searching for audio flags**

# Design Summary

To implement our application we will be creating a Visual Studio Code (VS Code) extension.

The *Information Architecture Diagram* provides an operational map to how this product acts and functions for users. It shows the hierarchy and organization of information in our extension.

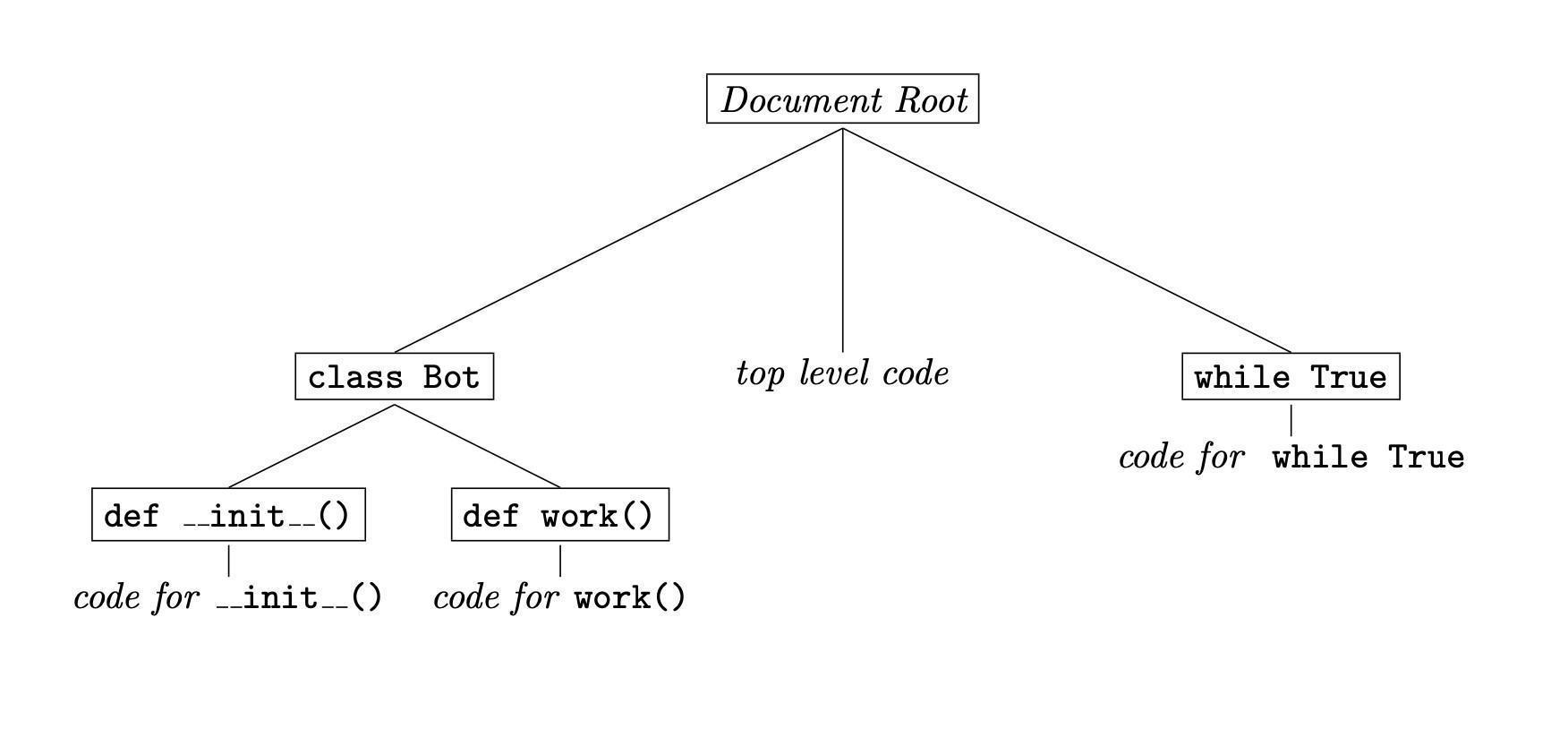
*Diagram 1* shows the top-level design of our extension. The “Extension Host” is the extension process that runs the extension and exposes the VS Code API. The extension consists of these elements: hotkeys for various actions within the editor and menus, a parser to provide context information about the user’s position in the code, such as the scope of the current line, and a hub controller to manage interaction with the LEGO SPIKE Prime Hub (“the Hub”).

Hotkeys:

Hotkeys comprise the majority of the functionality of our extension. They can inform the user of line indentation, navigate menus/editors, etc. The hotkeys will be implemented throughout the project and will help the user code.

Parser:

The **Parser** class constructs an abstract syntax tree of **LexNodes** that models the high-level structure of a Python source file. This AST is used to provide context for the user’s cursor position. The parser is the exposed portion of the context analysis component of the extension. A sample AST follows.

**class Bot(object):**

**def \_\_init\_\_(self, id):**

**this.name = id**

**def work(): print("Beep, Boop .-.")**

**b = Bot()**

**while True:**

**b.work()**

*(a). A short Python program for a robot. (b). The corresponding abstract syntax tree, with nodes enclosed in boxes.*

The **LexNode** class is a single node of the AST. It extends the **vscode**.**TreeItem** class to leverage its functionality, like accessibility information or invoking commands. Each **LexNode** contains a single **LineToken** object. In addition to the methods relating to parents and children that you expect for a tree item, it has a **rootPath()** method which provides a list of “this inside that” representing the path along the tree from the node to the root of the document.

The **Parser** uses the token stream generated by the **Lexer** to perform the syntax analysis while building the AST. The **Lexer** has an instance variable **tabFmt** that configures the style of tab to use while scanning. The **Lexer** has methods to restart scanning, advance and retract in the token stream, and calculate indentation for arbitrary text. It has instance variables for the current text, line position, and token from the stream. When text is passed to the **Lexer**, it is first processed by splitting it into an array of lines. The current line position then acts as a line pointer into the text being scanned.

The **TabInfo** class is simply a descriptor type containing two members:

* **size**: The width of a tab character, ***in bytes***
* **hard**: Whether to use literal tab characters or spaces

A **LineToken** represents a single line of a Python file. This is the finest level of precision required because the AST only considers the high-level constructs in Python, like **for**, **while**, or **try...catch**. A **LineToken** contains the following:

* A **TokenType**
* The indentation level for the line
* The line number
* An optional attribute (conditions, class names, function prototypes, etc.)

Each member of the **TokenType** enumeration represents a different construct (see *Diagram 2* for the full list). Most types are obvious in the way they map to Python constructs. **STATEMENT** represents a non-construct, non-whitespace, non-comment line; in other words, a regular line of code. This token represents the indentation, and its attribute holds the line of code. **COMMENT** represents a comment, and its attribute holds the comment. **EMPTY** indicates a line that is not significant to indentation or control flow (empty lines, whitespace lines) and its attribute is undefined. **EOF** represents the end of the token stream. The only token that will have a type of **EOF** is the special token constant **EOFTOKEN**. In the event that no token matches the current line, the tokenizer will return **INVALID**, which represents an invalid line. This invalid token was created to address the assumption that anything which isn’t a **STATEMENT** or a **COMMENT** is **EMPTY**, since those two symbols are now used in the context analysis system to allow reading the contents of a comment.

HubController:

The **HubController** class is responsible for communicating with and running Python code on the SPIKE Hub. It uses the raw mode of the MicroPython REPL to programmatically interact with the SPIKE Hub. All of its methods use **async**- and **await**-based promises to ensure that communications with the SPIKE Hub behave in a reliable, consistent order:

* **create()** is an asynchronous factory function that returns a new **HubController** instance. It takes an **options** argument of type **ControlOptions**, which configures how the **HubController** will interface with the SPIKE Hub.

**ControlOptions** has the following fields:

○ **portPath**: The port to use for the connection, or the fallback port if **magic** is true

○ **autoConnect**: Whether to automatically connect to the SPIKE Hub when the **HubController** is created.

○ **magic**: Attempt to automatically connect to the SPIKE Hub by probing serial connections and attempting to connect to valid ports. Implies **autoConnect**.

* **queryPorts()** probes serial connections for devices that broadcast their manufacturer as “LEGO System A/S” and returns a list of port paths for those devices.
* **init()** establishes and prepares a connection to a Hub. If a Hub is already connected, **init()** will disconnect from the SPIKE Hub before trying to establish the connection.
* **disconnect()** closes a connection to a Hub.
* **restart()** resets the SPIKE Hub to the initial state. This differs from **init()** in that it does not disconnect from the SPIKE Hub but instead reboots it while maintaining the serial connection.
* **sendCommand()** sends a single line of code to the SPIKE Hub.
* **sendControl()** sends a single byte control code to the SPIKE Hub. These correspond to the bytes sent when the Control key is pressed in conjunction with a letter.
* **runCode()** takes the lines of code to run as an array of strings and uses **sendCommand()** to send them to the SPIKE Hub to be run.
* **runFile()** takes the path of a file to send to the Hub, reads it into an array of strings, then uses **runCode()** to run the file.

**Design Rationale** HubController:

When designing the **HubController** class, there were issues with the initialization process. Because the SPIKE Hub took a non-negligible amount of time to boot up (~7 seconds), it was important to wait at least that long before starting to try interacting with the Hub, or the beginning of the data transmission would be dropped.

Using **setTimeout()** would not be sufficient in this case, since that would not synchronously wait before continuing execution, but instead would continue executing the code after the timeout, then calling the callback later in isolation. This causes messages to be received out of order by the Hub, at best, or crashing the Hub, at worst. Out-of-order data was a problem whenever interacting with the Hub, but was most severe at startup.

To work around this, all methods of the **HubController** that interact with the SPIKE Hub use **await**-based promises, which only resolve once any pending writes to the SPIKE Hub have finished. This is achieved by resolving the promise in the callback to the **SerialPort#drain()** function, which indicates that it is OK to write again. By using **await** when calling methods of the **HubController**, it can be ensured that the communication happens in a synchronous manner.

Creating the connection to the SPIKE Hub was initially a two-step process:

1. Create an instance of the HubController class: **let hub = new HubController()**
2. Initialize the SPIKE Hub with an asynchronous init function: **await hub.init()**

When implementing the **magic** option to automatically connect when creating the **HubController**, this posed a problem: TypeScript does not allow asynchronous constructors. This meant that it was not possible to use an ordinary constructor to both instantiate the HubController and connect to the SPIKE Hub.

This was solved with the use of an asynchronous factory function: the **HubController** constructor was made private, so the user cannot instantiate it themselves. Instead, the static **create()** method of the **HubController** class is used. **create()** is asynchronous and returns a Promise that is fulfilled with a new **HubController** instance once it is ready. This enabled automatic connection when first instantiating the **HubController**, without the need for a secondary function:

**let hub = await HubController.create()**