Analysis and Requirements

**User stories**

To guide the design of the tool, a set of user stories was developed to capture the needs and goals of potential users. These user stories provided valuable insights into the ways in which the tool could be used and the features that would be most important to its users. The user stories were used to inform the development of the tool's functionality and user interface, ensuring that the tool was tailored to the needs of its intended users. These user stories were then developed into specific scenarios that would serve as a useful testing framework further on in the development life cycle

**Requirements gathering: Asking computer science students to describe JSON files**

To develop the descriptions that would be produced by the tool, we recognized the importance of understanding the language used by the intended users of the tool. In order to gain insight into the terminology and language commonly used by programmers to describe JSON files, we conducted research with potential end users. We presented a short JSON file to a diverse group of 30 programmers and asked them to describe the file in their own words. This method provided us with valuable insights into the common words and phrases used by programmers to describe JSON documents. By analyzing the language used by programmers to describe JSON files, we were able to develop a formal mapping of the JSON grammar to natural language terms that would be most meaningful and relevant to the intended users of our tool. This research informed our design decisions regarding the levels of detail and customization options that we built into the tool, ensuring that the tool's descriptions would be both accurate and easily understood by its users.

**MoSCoW prioritisation**

To ensure that the tool we were designing would meet the most important needs of its intended users, we developed a MoSCoW feature prioritization for the JSONTalk tool. This process was guided by the user stories we had developed earlier in the design process, which provided us with valuable insights into the challenges faced by potential users when working with JSON files. The MoSCoW prioritization process helped us to identify and prioritize the most critical features and functionalities of the tool, based on the needs of its intended users. By categorizing features according to Must-Have, Should-Have, Could-Have, and Won't-Have priorities, we were able to focus our development efforts on the most important and high-priority features, ensuring that the tool would provide the greatest value to its users. This approach forced us to critically consider the priorities of the intended users of the JSONTalk tool and the challenges they faced when using existing tools, leading to a more user-centered and effective design.

Design section

**Choice to implement tool as command line interface**

We chose to implement the JSONTalk tool as a command-line tool for several reasons. First, command-line tools offer greater accessibility for users who require screen readers. This means that users can easily read the description with their own personal screen reader and keep their personal preferences. Additionally, command-line tools offer a more convenient solution for visually impaired programmers than graphical interfaces that rely on visual cues.

Moreover, command-line tools are generally faster and more efficient than graphical interfaces since they do not require the rendering or processing of graphical elements. This efficiency makes command-line tools scalable and adaptable to work with larger datasets or complex programming projects, providing a flexible solution that can meet various programming needs.

Furthermore, command-line tools can be easily integrated with other programming tools, improving collaboration among team members and streamlining programming tasks. This integration allows programmers to leverage other tools in their toolkit and work seamlessly between different workflows. Overall, implementing the JSONTalk tool as a command-line tool offers a highly accessible, efficient, flexible, and integrated solution for visually impaired programmers working with JSON data. However, it is important to note that even though command line interfaces are often deemed the most accessible for those with visual impairments, programmers still face a number of accessibility challenges when using the command line.

**Visitor design pattern**

We chose to use ANTLR4 for processing the input JSON files. The choice to use ANTLR4 with a visitor pattern for generating natural language descriptions of JSON files is a robust and flexible solution. The visitor pattern allows for a clear separation of concerns by decoupling the logic for traversing the parse tree from the actual nodes being traversed. ANTLR4, as a powerful parser generator, efficiently generates parse trees that can be easily navigated using the visitor pattern. This choice is particularly appropriate for complex JSON files, as ANTLR4 supports complex grammars and is capable of generating parsers for languages with features such as recursion, left recursion, and ambiguity. The visitor pattern also provides flexibility, allowing for easy modifications to the natural language descriptions, making it easier to accommodate changes in requirements or specifications.

In order to plan the syntax of the natural language description and determine the best approach for achieving it, we analyzed the abstract syntax trees generated by the ANTLR4 tool during the parsing of JSON files. We closely examined the relationships between the various elements and the order in which the nodes would be traversed in the parse tree. This approach helped us gain a deeper understanding of the underlying structure of the JSON files and how best to translate that structure into a clear and concise natural language description.

**Generating a formal description specification**

Prior to commencing coding, we formulated a formal mapping of the JSON grammar to natural language terminology. Considering the non-customizable nature of utilizing a screen reader with a structured text document, we made a design decision to allow the production of varying levels of descriptions with different levels of detail for our tool.

1. Top-level: This level provides a concise summary of the JSON file, offering users an overall view of the file. It serves as a quick reference point for users who want to understand the file's basic structure without delving into the specifics.
2. Structural: At this level, we describe the more complex elements of the JSON file, such as objects and arrays, and provide information on the type of elements contained within them. This level helps users understand the relationships and dependencies among different elements in the file.
3. Full: This level provides a detailed description of each element in the JSON file, including the key and value or just the value for anonymous elements contained within arrays. It is useful for users who need a comprehensive understanding of the file's content.

In addition to the three levels of description, we also added two more options that users could enable to customize the description further:

1. Depth option: This option allows users to specify an integer value that determines the number of levels of elements to describe. This option makes it possible to produce shorter, more understandable descriptions for larger files or files with a high level of nesting. It is an important design goal for our tool to provide users with a better overall view of large files.
2. Nesting option: If this option is enabled, each element's depth/nesting level is read out to the user, providing context about the information they are hearing. This option helps users understand the structure of the JSON file and the relationships among its elements more easily.

Implementation section

**Resources used**

To accelerate the project's development, we leveraged the benefits of various existing libraries. In particular, we utilized the following libraries:

* Antlr4: For the crucial initial processing of the JSON files. Specifically, for its lexer and parser components, which were instrumental in creating a parse tree from the input JSON files.
* FreeTTS: For the ReadAloud functionality of the tool, which enabled users to have the generated natural language descriptions read aloud.
* Picocli: For implementing the tool as a Command Line Interface (CLI), making it easier for users to access and operate the tool. By utilizing these libraries, we were able to expedite the development process, enhance the tool's functionality, and improve the overall user experience.

ANTLR4 played a critical role in implementing the JSONTalk tool by providing a powerful parser generator that allowed us to construct a parse tree from the input JSON file. Utilizing ANTLR4's grammar syntax, lexer, and parser, we were able to extract the syntactic structure of the input and produce an accurate representation of the data. The subsequent use of a Visitor implementation to traverse the parse tree and generate natural language descriptions of the input added significant value to the tool's functionality. ANTLR4's customizable features and reliable performance enabled us to parse the JSON input with precision and efficiency. The use of the Visitor design pattern in combination with ANTLR4 provides a flexible and extensible approach to processing complex data structures.

**JSON Parsing and Data Extraction Using Visitor Design Pattern and Inheritance**

The visitor design pattern was employed to traverse each node of the abstract syntax tree generated from parsing the input JSON files. By doing so, we were able to extract the necessary information for creating a file description from each node, which was then stored in various objects.

The jsonElement object was utilized to store information regarding primitive JSON key-value pairs, including both named and anonymous pairs. Such values were defined as being of type STRING, NUMBER, BOOLEAN, or NULL.

To store information about complex JSON key-value pairs, the jsonComplexElement class was created as an extension of the jsonElement class. The jsonComplexElement class was used to store values that were of type OBJECT and ARRAY.

Furthermore, the jsonObject class extended the jsonComplexElement class and provided additional functionality that catered specifically to JSON objects. Similarly, the jsonArray class was also created as an extension of the jsonComplexElement class and provided functionality tailored to JSON arrays.

In order to provide users with precise nesting and depth information for each JSON element, we implemented a system in which nodes were added to a list of their parent node's "children" while traversing the abstract syntax tree. This ensured that each parent node was always a jsonComplexElement.

By utilizing this approach and storing each element as a child of its respective parent, we were able to generate a comprehensive file description. Specifically, we iterated through each jsonComplexElement object that was created and listed the details of its children. This allowed us to accurately represent the hierarchical structure of the JSON file and provide the user with detailed information on the contents of each element.

**Command line interface**

As mentioned in the design section, we implemented the tool as a command line interface.