Informatics

Developing a Natural Language Interface for a UML Editor

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Simon Schwantler, BSc
Matrikelnummer 11703514

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof Dr. Dominik Bork
Mitwirkung: Dr. Stefan Klikovits

Wien, 29. Juni 2025

Simon Schwantler   Dominik Bork

TU WIEN Informatics

Developing a Natural Language Interface for a UML Editor

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Simon Schwantler, BSc
Registration Number 11703514

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof Dr. Dominik Bork
Assistance: Dr. Stefan Klikovits

Vienna, June 29, 2025

Simon Schwantler   Dominik Bork

Erklärung zur Verfassung der Arbeit

Simon Schwantler, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 29. Juni 2025

Simon Schwantler

Acknowledgements

Most importantly, I would like to thank my parents, Gabi and Klaus, who have supported me in countless ways. Their constant encouragement has been crucial in enabling me to complete my studies successfully.

I would also like to thank my supervisors, Dominik Bork and Stefan Klikovits, for their excellent guidance, valuable advice, and patience throughout the completion of this thesis.

Special thanks go to my friends and fellow students, David and Dominik, whose support made the journey through my studies truly memorable and enjoyable.

Finally, I want to thank Katharina. Without her, I would not have had the perseverance and motivation to reach this milestone.

Kurzfassung

Die weite Verbreitung der Unified Modeling Language (UML) in der Softwareentwicklung zeigt ihre Bedeutung für die Planung und Dokumentation von Softwareprojekten. Die Interaktion mit UML Editoren setzt jedoch meist die Verwendung von Maus und Tastatur voraus, was insbesondere für Menschen mit körperlichen Einschränkungen eine große Herausforderung darstellt. Diese Abhängigkeit schränkt die Zugänglichkeit zu derartigen Tools stark ein.

Natural Language Processing (NLP) bietet neue Möglichkeiten, Technologien barrierefreier und intuitiver nutzbar zu machen, indem es Computern erlaubt, menschliche Sprache zu verstehen und zu verarbeiten. Im Rahmen dieser Arbeit wird eine sprachbasierte Benutzungsschnittstelle (Natural Language Interface (NLI)) für UML-Editoren entwickelt, die moderne NLP-Modelle nutzt, um die Modellierung per Sprachbefehl zu ermöglichen.

Zunächst wird eine Taxonomie aller möglichen Operationen im Kontext von UML-Klassendiagrammen erstellt. Diese dient als Grundlage für die Definition konkreter Anfragen sowie für das Training von NL-Modellen, die in der Lage sind, Benutzerintentionen zu erkennen und relevante Informationen zu extrahieren, um entsprechende Aktionen im UML-Editor auszuführen.

Zur praktischen Umsetzung wird der quelloffene UML-Editor bigUML im Rahmen dieser Arbeit erweitert, sodass UML-Klassendiagramme ausschließlich über Sprachbefehle erstellt werden können. Dies zeigt, wie sich die Barrierefreiheit in der UML-Modellierung sowohl für Menschen mit körperlichen Einschränkungen, als auch für Nutzer, die eine effizientere Alternative zur herkömmlichen Bedienung suchen, verbessern lässt.

Eine empirische Evaluation des sprachgesteuerten Editors zeigt vielversprechende Ergebnisse hinsichtlich der Intentionserkennung, der Effektivität bei der Modellerstellung und der allgemeinen Nutzbarkeit.

Abstract

The widespread use of the Unified Modeling Language (UML) in software engineering shows its significance in planning and documenting software projects. However, the need for mouse and keyboard interaction when working with UML editors shows a big challenge, especially for users with physical disabilities. This reliance on traditional input devices limits accessibility.

Natural Language Processing (NLP) is at the forefront of making technology more accessible and intuitive, by allowing computers to understand and interpret human language. This thesis presents a Natural Language Interface (NLI) for UML editors that leverages state-of-the-art NLP technologies to enable interaction with UML models using voice commands.

First, this thesis will develop a taxonomy, defining all possible operations when working on a UML class diagram. This taxonomy is used to define concrete queries and train Natural Language (NL) models, capable of understanding the intent and extracting relevant information to perform operations in an UML editor.

To demonstrate the possibilities of such a NLI, the open-source UML modeling editor bigUML is extended to allow the creation of UML class diagrams using voice commands only. This highlights the improvements in accessibility for UML modeling, making it more efficient and user-friendly for a diverse audience, including those with physical disabilities and users seeking a more efficient alternative to traditional mouse and keyboard inputs.

An empirical evaluation of the NLI-enabled modeling editor is conducted, showing promising results in intent recognition, the effectiveness of model creation, and usability.

Contents

CHAPTER 1

Introduction

This chapter introduces the problem addressed in the thesis and explains its significance. It will establish specific research questions and outline the methodology that will be used. Finally, a brief overview of the remaining sections of the thesis will be provided.

1.1 Problem Statement and Motivation

The widespread use of the Unified Modeling Language (UML) in software engineering emphasizes its significance in planning and documenting software projects. However, the need for mouse and keyboard interaction with UML editors poses a big challenge, especially for users with physical disabilities. This reliance on traditional input devices limits accessibility. While recent years have seen notable efforts to improve accessibility in software and web development [NN11, W3C][1,2,3], and continuous evaluation shows improvements [Webb], the field of conceptual modeling lags behind [SMB23]. Accessibility in conceptual modeling has been largely neglected, resulting in the discipline being less inclusive and failing to accommodate the varied needs and abilities of all modelers. This oversight not only excludes many individuals with disabilities from participating in conceptual modeling but also overlooks the potential benefits of more accessible interfaces for all users [LBS+23a]. Enhancing accessibility in conceptual modeling can lead to more inclusive practices, allowing everyone, regardless of physical ability, to engage efficiently in modeling tasks, potentially offering improved interaction methods over traditional mouse and keyboard inputs.

Natural Language Processing (NLP) and intent recognition are at the forefront of making technology more accessible and intuitive, by enabling computers to understand and

[1] `https://www.apple.com/accessibility/`, last accessed 2025-06-14
[2] `https://blog.google/outreach-initiatives/accessibility/`, last accessed 2025-06-14
[3] `https://www.microsoft.com/en-us/accessibility/approach`, last accessed 2025-06-14

interpret human language [Hoy18]. These technologies face large challenges, including the complexity of human language. Accurately recognizing the intent of a user from their natural language inputs requires sophisticated algorithms that can handle ambiguity, slang, and dialectal differences [JM25]. Despite these challenges, the benefits of allowing users to interact with systems using natural language instead of fixed commands provide notable improvements [McT20]. It makes technology accessible to a bigger audience, including people without technical experience, and improves usability by enabling more and new interaction methods. This shift toward intuitive user interfaces can enhance the user experience, making it easier and more efficient for users to achieve their goals.

1.2 Aim of the Thesis and Expected Results

This thesis aims to develop a Natural Language Interface (NLI) for a UML editor. By leveraging NLP technologies and Python, the thesis seeks to enable speech commands to interact with UML models.

To achieve this goal, the thesis will extend bigUML by a natural language interface. bigUML [BIG, MB23] is an innovative open-source tool developed as an extension for Visual Studio Code (VS Code), utilizing the Graphical Language Server Protocol (GLSP) [Ecl, BLO23] to offer a dynamic and flexible environment for UML modeling. This integration with GLSP enables bigUML to provide a modeling experience that supports a wide range of UML diagrams. The tool's architecture is designed for extensibility, allowing future developers to increase and adapt its functionalities to meet their specific requirements. By leveraging the modular design of bigUML and its integration with GLSP, it becomes feasible to extend the tool with an NLI.

The extension includes the development of a middleware layer capable of translating standardized commands, generated from natural language inputs, into actions that bigUML's current implementation can recognize and execute without altering its core functionality. This approach ensures seamless integration, allowing users to interact with UML models using speech in a more intuitive and accessible manner. The research will focus on a mapping system that not only aims to understand the user's intent but also aligns with the existing implementation of bigUML, extending the tool's usability without compromising any existing features. The goal here is to increase accessibility for UML modeling, making it more efficient and user-friendly for a diverse audience, including those with physical disabilities and users seeking a more efficient alternative to traditional mouse and keyboard inputs. The main focus of this work is the UML class diagram.

1.2.1 Research Questions

According to the goals of this work, the following research questions arise:

(**RQ1**) : To what extent can a taxonomy capture all possible interactions with a UML model through NLP, allowing innovative user engagement?

- With the answer to this question, a taxonomy for the UML editor will be defined.

(**RQ2**) : In what ways can the BIGUML editor be extended to use Application Program Interface (API)-like interactions through standardized commands derived from natural language inputs, and what does this imply for the editor's extensibility?

- By answering this question, it is shown that the BIGUML editor provides sufficient extensibility for implementing innovative ways of interaction.

(**RQ3**) : What are the challenges in accurately interpreting natural language inputs for interacting with UML models, and how can these insights contribute to designing a more user-friendly interface?

- On the one hand, answering this question helps recognize general challenges in speech recognition, considering different speakers and dialects. On the other hand, it helps provide a way for intent recognition and entity extraction, allowing for a more intuitive interaction compared to predefined commands.

(**RQ4**) : To what extent does the integration of a natural language interface enhance the user experience in UML modeling, considering efficiency, inclusivity, and user satisfaction as metrics for optimization?

- Answering this question shows whether the solution provides sufficient results and indicates the completeness of the taxonomy.

1.3 Methodology

The methodological approach in this thesis follows design science research, as proposed by Hevner et al. [HMPR04], to create and evaluate innovative artifacts that aim to improve natural language interactions with UML models. This approach consists of four steps:

1. **Requirements Analysis**:
   In the first step, the requirements for the natural language interface must be gathered. This involves a detailed analysis of the existing BIGUML tool to identify the specific needs and constraints of integrating a NLI (**RQ2**). This phase will outline the functional and non-functional requirements, including accessibility features and compatibility with existing UML modeling functionalities.

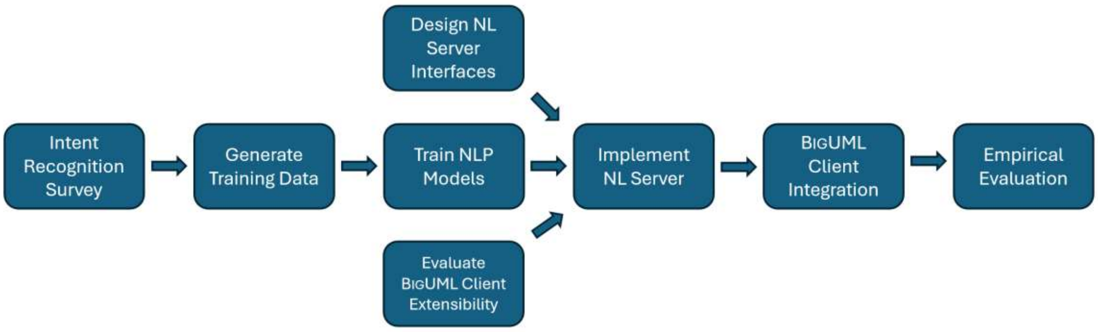


Figure 1.1: Research Outline

2. **Conceptualization**:
This stage focuses on designing the architecture of the NLI, including the development of a taxonomy that maps natural language commands to UML actions (**RQ1**). The conceptualization will also include the selection of technologies and tools for speech recognition, natural language processing, and the integration strategy with bigUML (**RQ2**). A user survey is conducted to identify possible inputs and ensure sufficient coverage of different wordings in user queries. A taxonomy can be derived from this data, describing all potential interactions with the editor. To ensure that all possible user interactions are covered, the survey will include closed questions concerning concrete modeling tasks (e.g., interaction with a specific object) as well as open questions on user requirements for interactions with a UML editor. This data will then be generalized and used to train the language model.

3. **Development**:
The development phase includes implementing the designed solution, beginning with the creation of a prototype (**RQ3**). This involves coding the speech-to-text module, the text-to-standardized-commands module, the standardized-commands-to-editor-commands module, and the communication interface with the bigUML editor. Each component will be developed iteratively, with ongoing testing for functionality and performance. A Bidirectional Encoder Representations from Transformers (BERT) Model [DCLT19] will be trained and used for the task of deriving standardized commands from natural language queries.

4. **Evaluation**:
Finally, the evaluation phase will assess the effectiveness and usability of the natural language interface through a combination of qualitative and quantitative methods (**RQ4**). Therefore, each module requires functional testing using excessive test data, spanning from pre-recorded voice to post-processing model editor commands. The evaluation will focus on the accuracy of intent recognition, user satisfaction, and any improvements in modeling efficiency compared to traditional interaction methods. The descriptive assessment will contain concrete modeling scenarios and test whether these cases can be covered utilizing the developed NLI. The evaluation

is planned to be conducted with users familiar with bigUML and evaluated by a questionnaire on how the new interaction methods change the user experience.

1.4 Structure

This thesis first discusses the background of model-driven engineering and general accessibility challenges in conceptual modeling. The following section discusses recent trends in natural language processing in software engineering. Chapter 3 provides an overview of current UML modeling editors and their accessibility features.

The next chapters provide details on the development process of the natural language interface for the bigUML editor. Chapter 4 discusses the requirements by laying out the main idea, explaining how a taxonomy for possible commands was gathered and used in designing a survey, conducted to gather possible commands formulated by users. Section 4.4 explains the process of deriving training data from the survey results. The development of a prototype is described in Section 4.5, highlighting the requirement for an advanced language processing model and evaluating speech-to-text frameworks. The technologies and frameworks used are outlined in Chapter 5, while Chapter 6 shows the application architecture design from an overall view, discussing the integration of the extension in the existing bigUML editor. It continues by highlighting each developed component in detail, discussing the development of the Natural Language Server and the integration process in bigUML. This work concludes with an evaluation section that compares the new interaction methods with traditional ones, highlighting the improvements and limitations.

CHAPTER 2

Background

This section provides an overview of the relevance of Model-Driven Engineering, highlighting its advantages in the software development process. Then, different model editors and their user interaction methods are discussed, pointing out their gaps for users with accessibility challenges. The section concludes by providing additional examples of natural language interfaces in software engineering, further highlighting recent trends in this field and their relevance in the whole software development life cycle.

2.1 Model-Driven Engineering in Software Development

Model-Driven Engineering (MDE) is an approach where models are treated as the primary artifacts in software development [Sch06, FR07]. Unlike traditional development, where code is written manually, MDE relies on structured representations of a system, which are used to generate executable code, documentation, and other artifacts automatically. This process allows developers to define abstract representations of their systems while ensuring consistency and automation throughout the development lifecycle. For instance, Zafar et al. [ZURKMN24] utilize MDE to generate the Graphical User Interface (GUI) code for mobile applications from UML diagrams. Another example is bigER [GB21], which provides a textual language to interact with entity-relationship models, as well as code generation for SQL tables.

MDE is particularly useful in domains requiring formalized modeling, such as software architecture, business process modeling, and UML-based development [MV06]. Through Model-to-Text (M2T) transformations, structured models can be converted into source code, while Model-to-Model (M2M) transformations enable different levels of abstraction in design.

One of the main advantages of MDE is its focus on automation [Sch06, FR07]. By defining a metamodel - a model that describes the structure and semantics of other models -

software developers can create tools that allow direct transformations of high-level designs into implementation-ready artifacts. However, despite its advantages, MDE also comes with challenges, particularly in terms of usability. Many modeling tools require extensive manual interaction, making them time-consuming and less intuitive for users unfamiliar with formal modeling practices.

In the context of this thesis, the primary focus is not on MDE itself but rather on how model editors can be improved by new interaction methods to support MDE in software development.

2.1.1 The Role of Model-Driven Engineering

The work presented in this thesis builds upon the principles of MDE but focuses specifically on user interaction within UML editors. The NLI for UML developed in this thesis introduces a new way of interacting with UML models, aiming to reduce manual effort and streamline the modeling process.

In traditional UML modeling, users must manually create and adjust diagram elements using mouse and keyboard-based interactions. While this approach works well for some user groups, others who prefer different input methods or are unable to use a mouse and keyboard properly are discouraged from engaging in MDE. The introduction of natural language commands addresses this issue by allowing users to perform actions such as:

- Creating classes and relationships using spoken or typed commands.
- Modifying attributes and methods without navigating through menus.
- Rearranging elements through verbal instructions.

By integrating NLP techniques into bigUML, this research demonstrates how voice and text-based interactions can improve usability in UML modeling environments.

2.2 Overview of existing Model Editors

Several tools have been developed to support UML modeling and model-driven development, each with its own strengths and limitations. This section introduces some of the most relevant tools, with a particular focus on their interaction paradigms and how they handle model manipulation.

2.2.1 bigUML: A Web-Based UML Model Editor

bigUML is an open-source UML modeling tool designed as an extension for VS Code [BIG, MB23]. It utilizes the Graphical Language Server Protocol (GLSP) to provide a dynamic and interactive modeling environment directly within VS Code. The modular architecture of BIGUML allows developers to build extensions that introduce new functionalities without modifying the core system.

One of the key advantages of BIGUML is its support for interactive modeling using a web-based approach. In contrast to traditional UML tools that operate as standalone desktop applications, BIGUML is integrated into modern development workflows, allowing users to work within their existing tools. This makes it particularly suitable for projects that require tight integration between modeling and code development.

However, BIGUML, like many UML modeling tools, relies heavily on traditional input methods such as mouse and keyboard interactions. While these input methods are sufficient for most users, they can become inefficient when working with complex UML diagrams, where frequent clicking and menu navigation are required. This limitation, as well as the need to improve accessibility, are primary motivations for introducing a natural language interface in this thesis.

2.2.2 Eclipse Modeling Framework: A Framework for Model-Driven Development

Eclipse Modeling Framework (EMF) is a widely used platform for model-driven software engineering [Weba]. Unlike BIGUML, which focuses on graphical UML modeling, EMF is a comprehensive framework that supports the definition, manipulation, and transformation of models.

EMF provides:

- A structured approach to defining metamodels.
- Code generation capabilities that allow models to be transformed into executable code.
- Interoperability with various modeling languages, including UML, XMI, and JSON-based representations.

EMF is particularly useful in environments where formal model transformations are required. It is widely adopted in enterprise software development, where models serve as the foundation for code generation and automated system integration. However, similar to BIGUML, user interaction in EMF is primarily based on manual editing, requiring a significant amount of user effort to create and modify models.

2.2.3 Comparison with other UML Tools

Beyond BIGUML and EMF, several other UML modeling tools exist, each with varying degrees of automation and user interaction paradigms. Some notable examples include:

- **PlantUML**[1]: A text-based UML editor where diagrams are generated from a

[1] https://plantuml.com/en/, last accessed 2025-05-26

structured textual description rather than direct graphical manipulation.

- **StarUML**[2]: A standalone UML modeling tool with an emphasis on extensibility through plugins.
- **Enterprise Architect**[3]: A commercial UML tool widely used in enterprise settings, offering a broad set of UML diagram types and integration with development environments.

While these tools provide different approaches to UML modeling, they rely on traditional interaction methods such as point-and-click interfaces or structured text-based inputs. The need for alternative interaction methods, such as voice commands and NLP, remains largely unexplored in mainstream modeling tools.

2.3 Accessibility Challenges in Conceptual Modeling

While accessibility is a well-established concern in web and software engineering, conceptual modeling tools often lack features that support individuals with disabilities. A systematic review of accessibility in conceptual modeling found that most existing tools focus on visual disabilities [PSB+02, BE98, KBC+04, BMS+14], while physical, auditory, speech, and cognitive impairments receive little to no attention [SMB23].

Improving accessibility in modeling can support more inclusive practices by enabling individuals of all physical abilities to participate more effectively. This may also introduce interaction methods that are more efficient than traditional mouse and keyboard inputs [LBS+23b].

The absence of disability-aware features in UML modeling tools creates barriers that prevent some users from fully engaging in conceptual modeling. These barriers are exceptionally high in interfaces that rely on mouse input, which lack integration of assistive technology and have limited keyboard navigation options. The review identified that most modern UML editors require fine cursor control and extensive use of graphical interfaces, making them difficult to use for individuals with mobility impairments [SMB23].

2.3.1 Interaction Challenges in UML Tools

Most UML modeling tools, including bigUML, EMF, and other web-based editors, rely heavily on traditional interaction methods, such as:

- Point-and-click interfaces that require precise cursor movement.
- Drag-and-drop operations for element placement.

[2] `https://staruml.io/`, last accessed 2025-05-26
[3] `https://www.sparxsystems.de/enterprise-architect/`, last accessed 2025-05-26

- Nested menus for accessing different modeling features.
- Shortcut-heavy workflows that are not optimized for users with disabilities.

These interaction patterns are challenging for users with physical disabilities who may struggle with precise mouse control. Similarly, individuals with visual impairments may find it difficult to interpret complex diagrams without adequate screen reader support or alternative text-based representations of diagrams [SMB23].

2.3.2 Current Accessibility Gaps in UML Modeling

Existing accessibility standards, such as the Web Content Accessibility Guidelines [W3C], have improved web usability for individuals with disabilities. However, similar efforts are largely absent in conceptual modeling. The review of current tools found that [SMB23]:

- No major UML modeling tool provides full keyboard navigation, limiting accessibility for users who cannot use a mouse.
- Screen reader support is inconsistent, making it difficult for visually impaired users to interpret models.
- Alternative interaction methods, such as speech input or tactile feedback, are almost nonexistent.

In contrast, other fields of software engineering have adopted accessibility-focused approaches by incorporating assistive technologies, automated accessibility evaluation tools, and alternative input mechanisms [SMB23]. Conceptual modeling, however, has lagged behind in these developments, highlighting the need for more inclusive interaction methods.

Given these challenges, a NLI for UML modeling presents a potential solution. This approach aligns with broader efforts to make digital tools more inclusive by integrating alternative interaction methods that accommodate a wider range of users. By incorporating accessibility-aware features into conceptual modeling tools, this research contributes to the broader goal of making software engineering tools more inclusive. Chapter 3 will discuss existing research on NLIs for UML and other related work, providing further context for the solution developed in this thesis.

2.4 Natural Language Processing in Software Engineering

NLP-powered tools enable users to interact with software development environments using text and voice commands, simplifying complex operations and reducing reliance on traditional input methods such as manual coding and graphical interfaces.

The use of voice and speech in programming environments has seen research interest since at least 2006 [DFN06]. These early approaches focused on precise instructions. Since then, various approaches have been developed, recently supported by powerful AI and LLM approaches [Now18], a trend that has very recently peaked in so-called *vibe coding*[4], whose goal is the exclusive use of LLMs and voice input to complete small-scale programming projects.

One of the most prominent developments in this area is GitHub Copilot Voice[5], a system that allows developers to write code, navigate files, and manage version control using spoken commands. Using NLP models, Copilot Voice translates natural language instructions into executable code, automating repetitive tasks and making software development more efficient and accessible. Although this tool focuses on software development rather than modeling, it further highlights the recent trend in software engineering to utilize voice interfaces, providing alternative means of interaction.

Another example of translating natural language to code is Cursor[6], a code editor with additional features designed to support development workflows through an NLI. Based on VS Code, Cursor enables developers to issue text commands such as requesting code refactorings, documentation, or multiple file changes, which the system then translates into actual code suggestions in any programming language. Rather than relying on autocomplete, Cursor integrates language models that understand the broader context of the codebase, which enables more complex operations.

[4] `https://arstechnica.com/ai/2025/03/is-vibe-coding-with-ai-gnarly-or-reckless-maybe-some-of-both/`, last accessed 2025-06-27

[5] `https://githubnext.com/projects/copilot-voice/`, last accessed 2025-05-26

[6] `https://www.cursor.so`, last accessed June 11, 2025

CHAPTER 3

State of the Art

Natural Language Processing has been increasingly applied in software engineering, particularly in automated UML modeling. The use of NL as input for automated derivation of UML diagrams has been studied before [MP12, MU06, DB09, DS11, ASBZ16, HH19], although these approaches are based on "classical" means of language processing (parse trees, word tagging, phrasal grammars). While many approaches focus on generating diagrams from textual requirements [FAA24, CTBV23, ABCS24], fewer have investigated interactive methods for manipulating UML models through natural language commands, for example by using chatbots [PGdL19, PGGdL19].

3.1 Related Work

This section reviews recent contributions in this field, highlighting their approaches and limitations.

3.1.1 Enhancing UML Class Diagrams with NLP

Utilizing ChatGPT, Rouabhia et al. [RH24] present an approach that automatically enriches UML class diagrams by extracting method details from text-based software specifications, provided as structured use case tables. This work demonstrates how large language models can assist in refining UML models based on structured text. However, it does not provide an interface for interactive modifications using natural language commands. Instead, the system processes static text inputs and updates existing UML diagrams. Since PlantUML[1] is used as an editor, no further refinement of the diagram is possible.

[1] `https://plantuml.com/en/`, last accessed 2025-05-26

3.1.2 ModelByVoice

Similarly to this work, ModelByVoice [LCA18] aims to increase accessibility for people with disabilities, especially blind people, by enabling users to create, edit, and navigate UML models using voice commands.

The system is built on three core components:

- **Voice Recognition Layer**: Captures spoken commands and converts them into structured text input.
- **Model Processing Layer**: Matches commands to predefined grammar rules that represent UML operations.
- **Speech Feedback Layer**: Provides audio responses to confirm actions, helping users understand the current model state.

Users can execute CRUD operations (Create, Read, Update, Delete), navigate through models and query elements. The tool works with a predefined grammar and does not focus on a specific diagram type. Therefore, a mapping from these predefined commands to the actual diagram elements must be provided. For example, the command "Create node" would translate to "Create Class" in the context of a UML class diagram and to "Create Action" in the context of a UML activity diagram.

A key aspect of this tool is its dependence on predefined grammar rules. Instead of Natural Language, the system requires users to follow a structured syntax when issuing commands:

```
create (diagram|node|link|compartment);
list diagram;
save diagram;
undo;
sleep;
help;
exit programm;

remove (navigation element|attribute|link);
create attribute;
update attribute;
update navigation element;
change navigation element;
detail navigation element;
say neighbours;
```

Listing 3.1: Grammar rules supported by ModelByVoice [LCA18]

Building on the ModelByVoice project, Carvalho et al. [dCA21] extend the concept of voice-based modeling by introducing a more adaptable interaction model that also supports non-vocal sounds and gestures alongside speech. Their work addresses usability limitations observed in the original ModelByVoice prototype by enabling users to customize their interaction with modeling editors based on their individual preferences and abilities. The proposed system supports a broader set of commands, offers both graph and table-based navigation, and allows for interaction through drawn gestures and user-defined sound signals.

3.1.3 Automated UML Class Diagram Generation from Textual Requirements Using NLP Techniques

A different approach by Meng et al. [MB24] focuses on creating UML class diagrams directly from textual requirements. Their methodology follows a rule-based extraction system, where predefined patterns and linguistic rules are applied to interpret domain-specific terms. This approach and its limitations are also discussed in the initial prototype implementation of this work, shown in Section 4.5. A notable feature of this implementation is that elements with the same name are automatically merged, which enhances the user experience and avoids having multiple elements with the same name. This shows a possible future improvement for the prototype developed in this work.

This method is limited to one-time generation, and once a diagram is created, users must manually adjust it through traditional interaction methods, as natural language-based modifications are not supported. As this work only supports classes and relationships, it lacks support for other class diagram notation elements shown in Section 4.2.

3.1.4 Towards Automatically Extracting UML Class Diagrams from Natural Language Specifications

Song et al. [YS22] introduce a modular pipeline for automatically extracting UML class diagrams from natural language software specifications. This pipeline allows for the systematic conversion of textual descriptions into structured UML models. However, similar to previous works, the lack of an interactive component means that users cannot refine diagrams dynamically using text or voice commands. The method also relies on predefined templates, limiting flexibility in modeling more complex software architectures.

3.1.5 UMLAI and Initial Steps Toward NLP-Driven UML Tools

UMLAI[2] is an open source project that allows users to input natural language descriptions, which are processed into UML diagrams using Google Gemini AI[3] and PlantUML[4]. Once a diagram is generated, further adjustments must be made either manually or by regenerating the entire model. As with other NLP-based UML tools, UMLAI operates as a one-time transformation tool rather than an interactive modeling assistant. While this does not allow users to model a diagram step-by-step, it demonstrates the capability to process more complex and concatenated commands, which is currently not supported

[2] `https://umlai.vercel.app/`, last accessed 2025-05-26
[3] `https://gemini.google.com/`, last accessed 2025-05-26
[4] `https://plantuml.com/en/`, last accessed 2025-05-26

in this work. Another advantage of this tool compared to other related work is that it is available online, promoting simple usage without the need to install any additional software.

3.1.6 Gaps and Opportunities

While recent research has advanced automated UML generation, several challenges remain unaddressed:

- **Lack of interactive modifications**
  Existing tools focus on diagram creation but do not support real-time changes using text or speech commands.
- **Limited accessibility features**
  Few implementations include voice support and rely on textual input.
- **Lack of integration with UML editors**
  Many approaches operate as standalone systems, requiring users to switch between environments rather than modifying models directly within existing UML tools.

This section shows the relevance of the prototype developed in this thesis, as related work addresses the general need for new interaction methods within UML editors, but has different gaps in their usability by either not integrating well in existing solutions or only supporting one-time generation of UML diagrams rather than real-time interaction methods.

The following chapters describe the process of developing the prototype presented in this work, from gathering requirements and training data and deciding on the technology stack to use, to implementation and integration details into the bigUML editor.

CHAPTER 4

Natural Language Server Requirements

This chapter describes the artifacts required to develop a NL server. The focus lies on the requirements for the language model, as well as on defining the technology and the interfaces used for developing the NL Server. These data provide the base for implementing a solution that can be integrated into existing modeling tools.

4.1 Main Idea

The first step is to set the focus on the supported UML diagram type, which is the UML class diagram. Supporting other diagram types follows a similar approach, but is not the aim of this work.

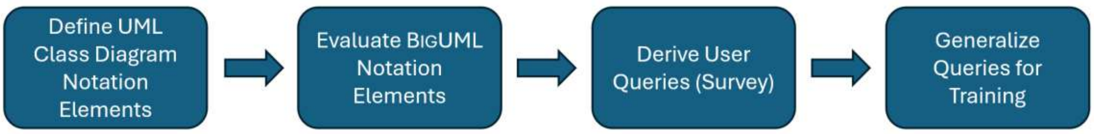


Figure 4.1: Process for Developing a Taxonomy

Gathering information on how users might interact with a modeling tool using voice control is a prerequisite for developing the NL Server. This data is required to derive a taxonomy covering all possible commands used to model a UML class diagram. Subsequently, different wordings for each taxonomy entry can be derived to gather a set of training data for the NL model. This data can then be used to train a natural language model.

In parallel, different speech-to-text frameworks are evaluated, and a first prototype is developed to understand the challenges in understanding and interpreting natural language.

4.2 Taxonomy

Two approaches are used to define a taxonomy. First, existing literature on the UML class diagram is reviewed. Seidl et al. [SSHK15] define a set of notation elements for a UML class diagram, shown in Table 4.1. This data serves as a starting point in gathering information on the possible actions performed on a UML class diagram.

Name	**Description**
Class	Description of the structure and behavior of a set of objects.
Abstract Class	Class that cannot be instantiated.
Association	Relationship between classes. Its navigability can be either undefined, in both directions or in one direction.
n-ary association	Relationship between n classes.
Association class	More detailed description of an association.
xor relationship	An object of a is either in relationship with an object of b or c, but not both.
Shared aggregation	Parts-whole relationship. One object is part of another object.
Strong aggregation, composition	Existence-dependent parts-whole relationship. One object is part of another object, but cannot exist alone.
Object	Instance of a class.
Link	Relationship between objects.

Table 4.1: Notation Elements - Class Diagram

In the next step, features of the bigUML tool are evaluated and compared to the notation elements defined in the literature. It can be observed that the bigUML tool provides sufficient support for almost all notation diagram elements defined in [SSHK15]. Not supported are the n-ary association, the XOR relationships, and the instantiation of classes as objects. Additionally, a graphical editor provides ways to interact with a model by selecting elements and changing their position in the diagram. These actions must also be considered when defining a set of intents for possible user interaction. While setting the focus on elements and changing their position is intuitively done using a mouse, using only voice commands is challenging. The following section provides an overview of all supported components. Each of the elements described can be added, updated, and deleted, as well as freely positioned in the diagram.

Containers

- **Primitive type**
  A primitive type has a name, visibility, and information about whether it is abstract. There are no predefined data types.
- **Data type**
  A data type has a name, visibility, and information about whether it is abstract.
- **Enumeration**
  An enumeration has a name, visibility, and information if it is abstract.
- **Class**
  A class element has a name, visibility, and information about whether it is abstract and active.
- **Interface**
  An interface has a name, visibility, and information if it is abstract.
- **Package**
  A package has a name, a URI, and a visibility.

Features

- **Enumeration literal**
  An enumeration can have multiple literals. Each literal has a name and visibility.
- **Operation**
  An operation can be added to classes, interfaces, data types, and primitive types. An operation has a name, visibility, concurrency, and whether it is abstract and static.
- **Property**
  A property can be added to classes, interfaces, data types, and primitive types. A property has a name, visibility, type, multiplicity, and if it is static, read-only, ordered, and unique.

Relations

- **Abstraction, Dependency**
  These relations connect two classes or interfaces and have a name and visibility.
- **Aggregation, Association, Composition**
  These relations connect two classes or interfaces and have a name and visibility. Additionally, name, visibility, and multiplicity can be defined on the source and target sides.

- **Generalization**
  A generalization connects two classes and has a property indicating whether it is substitutable.

- **Realization, Substitution, Usage**
  These relations connect two classes or interfaces and have a name and visibility.

- **Package Import, Package Merge** These relations connect two packages and have a property defining their visibility.

With the closed set of class diagram notation elements and their availability in the bigUML tool, concrete possible user queries can be derived for the implementation of the NL server. As these queries will be used to train and test the language model, it is important to ensure that this data does not include a bias.

To ensure a diverse set of commands, a survey is conducted among users of UML diagram editors to gather information on possible interaction patterns.

4.3 User Survey: Natural Language Commands for a UML Editor

A user survey was carried out as part of this thesis to learn more about the natural language instructions modelers might use when engaging with a voice-controlled UML class diagram editor. The main goal of the survey was to find various ways users articulate tasks typically performed in UML diagramming tools, such as creating and updating elements and their properties.

Participants were asked to provide complete sentences in natural language, as the goal was to simulate how users might naturally interact with such a tool using voice commands. Emphasis was placed on encouraging the use of synonyms and alternative phrasings, as users may describe the same action in various ways. For example, the terms "attribute", "field", and "property" can all refer to the same concept within a class. This is important information for later steps to cover a wide range of different wordings and avoid a bias in the capability of understanding queries.

The survey was structured around specific UML interactions, including creating classes, adding attributes and methods, modifying elements, and positioning diagram components. Additionally, the survey included an open-ended section asking respondents to recommend any other commands they believe are beneficial for using a UML editor effectively. This section will summarize the results, highlighting key patterns and trends in the natural language commands provided by the participants.

4.3.1 Survey Methodology

The survey was designed to capture a wide range of natural language commands related to interactions with UML class diagrams. It was shared with individuals familiar with

UML modeling to gather a variety of voice commands users might phrase to control a UML editor. The survey was structured into specific categories, each corresponding to a common action in UML class modeling, such as creating classes, adding attributes, and modifying methods.

Survey Design

The survey contained closed questions with one additional open question. Participants were asked to provide full, natural language sentences in response to the following interaction types:

- Create Class
- Add Attribute to Class
- Add Method to Class
- Rename Class
- Modify Class Attribute Name
- Modify Class Attribute Visibility
- Modify Class Method Name
- Modify Class Method Visibility
- Modify Class Method Parameters
- Add Relation
- Positioning

For each interaction type, respondents were encouraged to use varied phrasings and synonyms to capture the range of commands better. An open-ended question was included to allow participants to suggest any other commands they believed would be useful when interacting with a UML editor, capturing additional user needs that might not fit into predefined categories.

Participant Demographics

The survey was distributed online to people with previous experience in UML modeling. The survey was conducted between March and April 2024. Although specific demographic data was not collected, respondents were assumed to have a technical background, either from academic or industry settings. This approach ensured that the input provided reflected practical knowledge of UML and common interactions within modeling software. Information was collected from 11 individual participants, providing 242 user-defined natural language commands in total.

Data Collection and Processing

The responses were collected over several weeks, resulting in multiple entries for each interaction type. After collection, the data was reviewed to identify common patterns in phrasing and to extract information regarding how users express commands in natural language. Entries with incomplete or unclear responses were excluded from the analysis, ensuring that only meaningful data were used in the analysis.

4.3.2 Results Analysis

The survey responses provided details on how users might naturally phrase voice commands for various actions in a UML class diagram editor. A summary of the results is shown in Table 4.2. This section analyzes the responses for each interaction type, highlighting similarities and patterns, variations in phrasing, and notable trends across the different categories. The results offer a base for designing a voice-controlled interface that can accommodate a wide range of natural language queries.

```python
import nltk
import pandas as pd
import re
from collections import Counter
from nltk.corpus import stopwords
import matplotlib.pyplot as plt
nltk.download('stopwords')


def clean_tokenize(text):
    if text is None or not isinstance(text, str):
        return []

    text = text.lower()
    text = re.sub(r'[^\w\s]', '', text)
    words = text.split()
    stop_words = set(stopwords.words('english'))

    # also remove the word "class" since it is required for context, but not
    for the actual command
    stop_words.add("class")
    words = [word for word in words if word not in stop_words]
    return words


survey_data = pd.read_csv(file_path)

for column in survey_data.columns:
    survey_data[f"{column} Tokens"] = survey_data[column].apply(lambda x:
    clean_tokenize(x))

    all_intent_words = [word for tokens in survey_data[f"{column} Tokens"]
    for word in tokens]
    intent_word_count = Counter(all_intent_words)
```

Modeling Scenario	Responses	Sample Responses
Create a Class	26	– *Create a class called car.* – *Insert a class Vehicle into the current diagram.* – *I want to add a new Class to the diagram, where the new class' name is "Car".* – *Generate a class for "train".* – *Declare the class bike.* – *I would like to add a new class named/called/titled/designated 'Customer' to the diagram.*
Add an Attribute to a Class	24	– *Add an attribute called name of type string to the Employee class.* – *Add the string property name to the class Bike.* – *The new class (or "Car"-class) has the properties "color", "brand" and "type".* – *Attach the attribute motor to the class car.* – *I would like to introduce a new field called 'orderDate' with the type date in the 'Order' class.*
Add a Method to a Class	22	– *add a method named CalculateSalary with a return type of decimal to the Employee class* – *Create a new operation for the class Bike, which is named getBikeName. The operation expects two input properties which are both of type string.* – *add the functionality to train to move from one location to another, call this ability "go somewhere"* – *Define the method accelerate for the class car.*
Rename a Class	24	– *change the class name from Employee to Worker.* – *Set the name of the class Bike to Motorcycle.* – *I want to rename the already existing class "Car" to "Vehicle".* – *The class car should actually be called bike.* – *Please alter the name of the class 'Transaction' to 'FinancialTransaction'.*
Modify an Attribute Name of a Class	24	– *Rename the property "Color" of the class "Car" to "PrimaryColor".* – *Tyre would be a better name for the car class's wheel attribute* – *Rename the car's doors to door.* – *change the attribute name from name to employeeName in the Worker class.*
Modify an Attribute Visibility of a Class	24	– *make the employeeName attribute in the Worker class public.* – *Change/Update the visibility of the attribute name from the class Bike to public.* – *make the "rail with" visible to everyone* – *Make all attributes of the class car public.*
Modify a Method Name of a Class	23	– *rename the method CalculateSalary to ComputeSalary in the Worker class.* – *Edit the method name of the method getName inside the Bike class to change it to getSerialNr.* – *"go somewhere" should now be called "move tram"*
Modify a Method Visibility of a Class	24	– *set the ComputeSalary method in the Worker class to public.* – *I want to change the accessibility of the method "getPrize" of the class "Car" to "private".* – *Make the car class's start method private.* – *Adjust the visibility of the method 'getSupplierInfo' to public in the 'Supplier' class.*
Modify Method Parameters of a Class	26	– *In the method "DriveVihicle" change the parameter "speed" to "double" "kmh".* – *Add one int to the params of the car class's start method.* – *Update the name of the second parameter of the Bike class' method getWithTwoParams to secondParam.*
Add a Relation	24	– *create an association between the Worker and Manager classes.* – *create a generalization from the base class Person to the subclass Teacher.* – *Let the car inherit from vehicle.* – *Create a new relation between the classes car and driver with the following attributes: 1. The relation is called drives. 2. One driver drives one or more cars. 3. The relation type is dependency. A driver requires a car to drive.* – *Add a one to one association from car to person called drives.*
Modify the Positioning of a Class	31	– *The class "Engine" should be below the class "Car".* – *Put the car class to the top right of the diagram.* – *Move the class car 20% to the right.* – *Position the "Vehicle" class north of "Car" class.* – *position the Worker class below the Manager class*

Table 4.2: Responses to the user survey on the use of natural language commands for UML modeling scenarios.

```
32
33     intent_word_df = pd.DataFrame(intent_word_count.items(), columns=['Word',
        'Count']).sort_values(by='Count', ascending=False)
34
35     top_words = intent_word_df.head(10)
36
37     plt.figure(figsize=(10, 6))
38     plt.bar(top_words['Word'], top_words['Count'], color='skyblue')
39     plt.title(f"Top Words in {column} Commands")
40     plt.xlabel('Words')
41     plt.ylabel('Frequency')
42     plt.xticks(rotation=45)
43     plt.savefig(f".\\plots\\{column}.png")
```

Listing 4.1: Python script to plot most common used words

To determine the most frequently used words per command type, a simple Python script, as shown in 4.1, is used to analyze and visualize the most commonly used words. In lines 17 and 20, stopwords as well as the word "class" are defined to be ignored in the analysis, as these keywords will most likely be part of all queries without containing any valuable information. All words are collected into a list and counted in lines 30 and 31. The top ten most common words are selected in line 35 and visualized as bar charts in lines 37 to 43.

Create Class

Responses related to creating a class were diverse but generally followed a clear pattern. Most participants used verbs like "create" "add" or "new" to describe the action, followed by the name of the class. For example:

- "Create a class called car."
- "I want to add a new class called table to the diagram."

While the basic structure of the command is consistent, variations occurred in the choice of verbs, with some users preferring "insert" over "create" or "add". Additionally, some participants added more details, such as specifying the position of the new class in the diagram.

Add Attribute to Class

Participants provided commands that typically specified the class, the attribute name, and its type. The phrasing included either "add" or "insert", followed by the details of the attribute. Examples include:

- "Add an attribute called name of type string to the class Person."

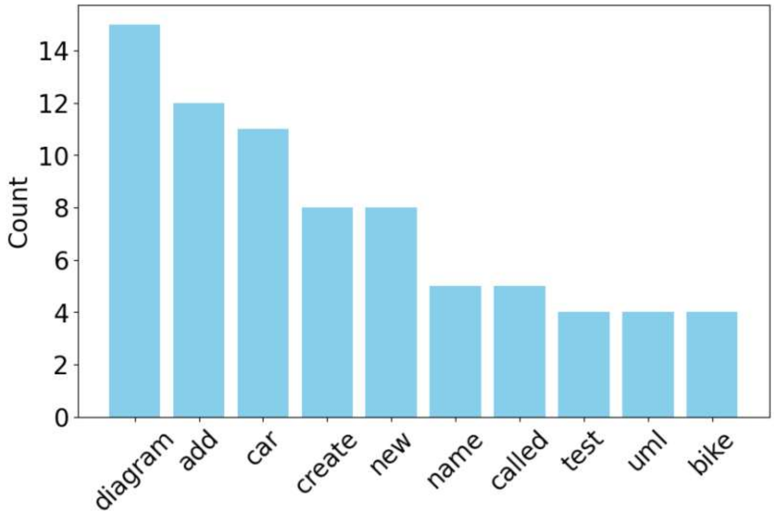


Figure 4.2: Most common words for intent "create class".

- "Please add an attribute named email of type string to the Customer class."

The inclusion of attribute types and class references was common, showing the need to provide focus on the element where the change should be applied. Some users also specified the visibility (e.g., public or private) for attributes, indicating a more advanced level of control over class properties.

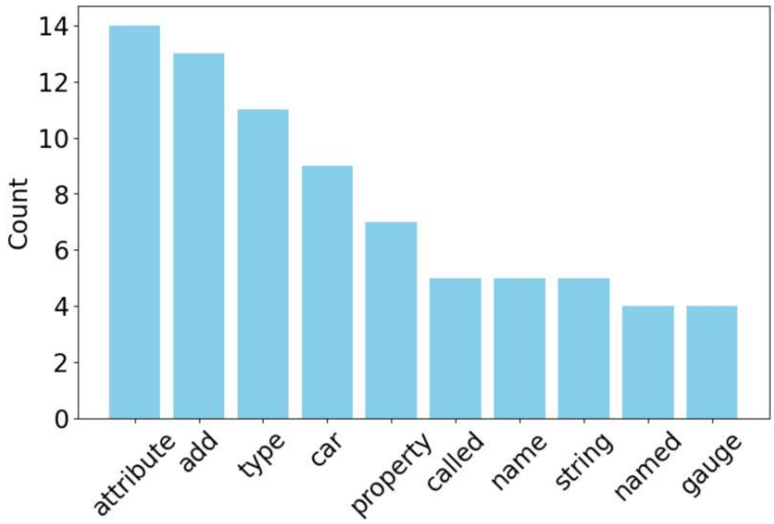


Figure 4.3: Most common words for intent "add attribute".

Add Method to Class

The responses for adding methods were structured similarly to the attribute commands, but included additional details, such as return types and parameters. A key takeaway here is that additional logic is required to determine the actual name of the method, as participants specified them in their expected form, concatenating words by using `snake-case` or `camelCase`. While this can be done by providing the query as text, it is not possible to do the same when providing the command using voice.

- "Add a public method named CalculateSalary with a return type double to the class Employee."
- "The new class has one function called startEngine."

It can again be observed that some users provide more details than others by specifying the visibility and return type of the added method.

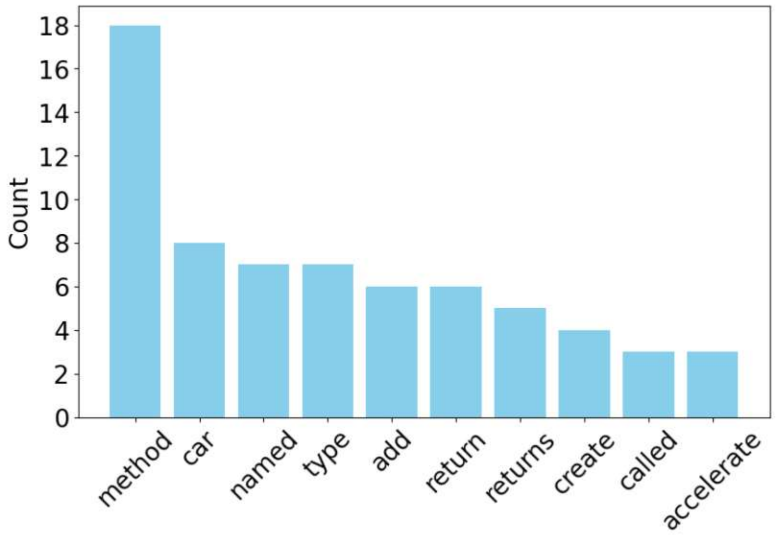


Figure 4.4: Most common words for intent "add method".

Rename Class

Respondents used straightforward commands to rename classes. Most answers followed a consistent format, indicating both the old and new class names. Common examples included:

- "Change the class name from Employee to Worker."
- "Set the name of the class Bike to Motorcycle."

These responses showed little variation, suggesting that renaming actions are typically expressed in a similar way between different modelers in natural language.

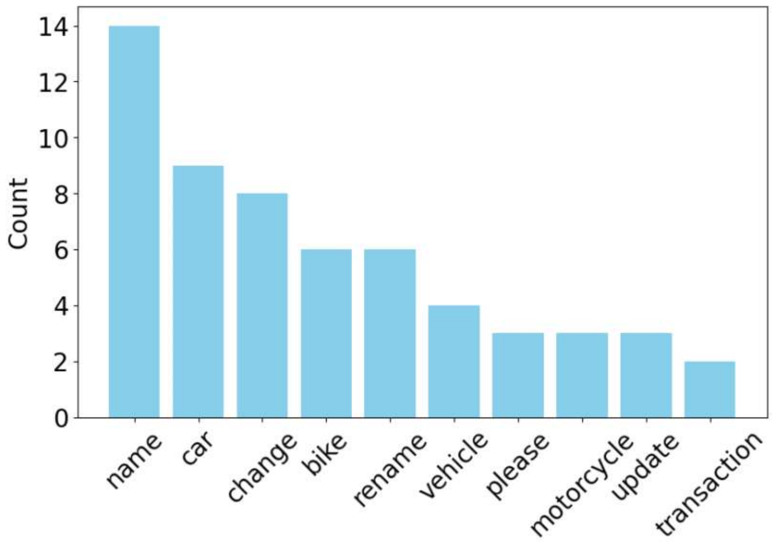


Figure 4.5: Most common words for intent "rename class".

Modify Class Attribute Name and Visibility

Commands in this category were more varied, particularly in terms of the language used to refer to visibility changes. Common commands included:

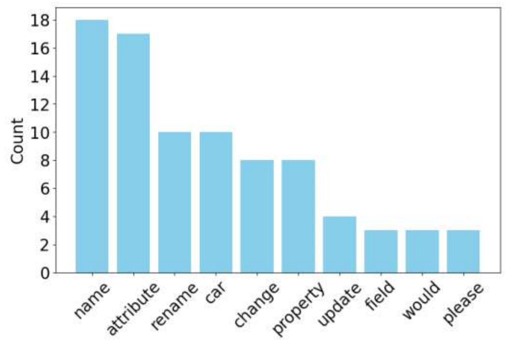
- "Change the attribute name from name to employeeName."
- "I want to rename an existing property of the class Car, where the old property name is type while the new property name is category."

While the wording for changing the visibility is similar to changing the name of attributes, the available entities are predefined, most commonly being "private", "protected", and "public".

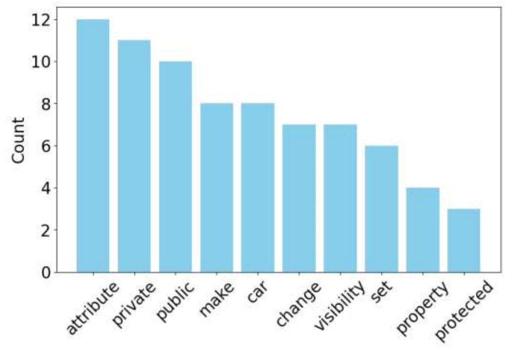
Modify Class Method Name, Visibility, and Parameters

Participants phrased commands for method modifications similar to attribute-related commands. Renaming methods were generally expressed similarly to renaming classes, while visibility changes included keywords like "public" or "private" as possible values. The name is again specified using `snake-case` or `camelCase`, which can only be addressed by written commands and not when using voice input.

- "Rename the method CalculateSalary to ComputeSalary."



(a) Most common words for intent "modify attribute name".



(b) Most common words in modify attribute visibility.

Figure 4.6: Most common words for intent "modify attributes".

- "Set the visibility of the method to private."

Parameter changes were often more detailed, with participants specifying both the parameter name and type:

- "Add a parameter of type decimal named amount to the method CalculateSalary."
- "Change the type of the first parameter of the method to integer."

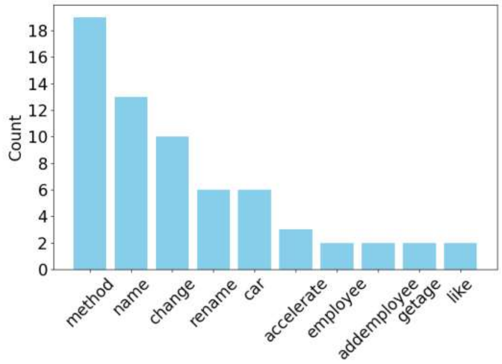
These responses again indicate different levels of detail in the user queries.

Add Relation

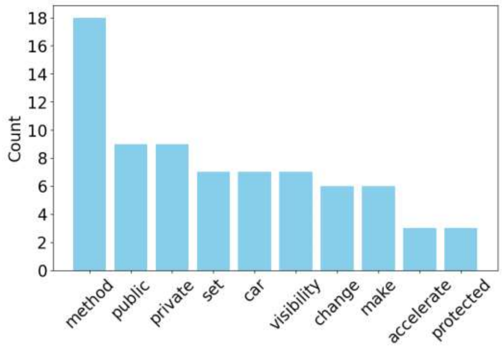
Commands for adding relations were highly variable, reflecting the complexity of UML relationships. With the number of different properties for relations, such as relation type, multiplicity, and names on both the source and the target of the relationship, this interaction is the most complex one. Participants described different types of relationships, such as generalizations, associations, and aggregations. Common examples included:

- "Create an association between the Worker and Manager classes."
- "Add a composition relationship from the 'ShoppingCart' class to the 'Item' class with the source multiplicity being 1 and the target multiplicity being 5."

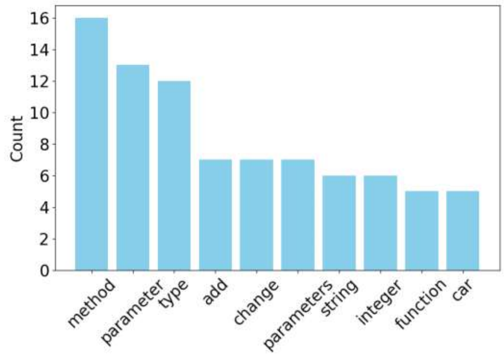
Users often included the type of relation, indicating that they expect the voice-controlled editor to handle complex interactions between classes. This category featured more diversity in phrasing, as participants described relationships with varying levels of detail and technical terminology.



(a) Most common words for intent "modify method name".



(b) Most common words for intent "modify method visibility".



(c) Most common words for intent "modify method parameters".

Figure 4.7: Most common words for intents "modify method".

Positioning

The positioning of elements within the UML diagram was another area where responses varied significantly, as participants either positioned elements relatively to other elements or globally within the diagram:

- "Position the Worker class below the Manager class."
- "Move the class Bike to the top right corner of the view."

Other Commands

The open-ended section yielded several interesting suggestions for additional commands. A few respondents suggested deletion or clearing functions, while others proposed commands for redo/undo previous commands:

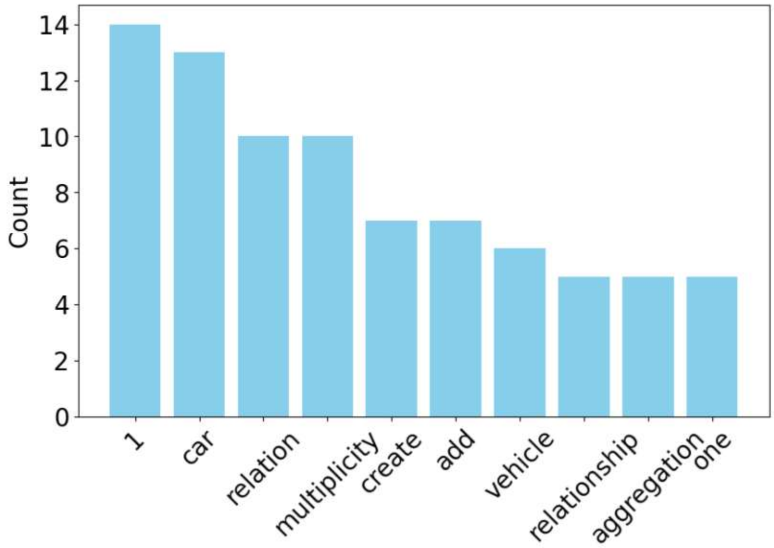


Figure 4.8: Most common words for intent "add relation".

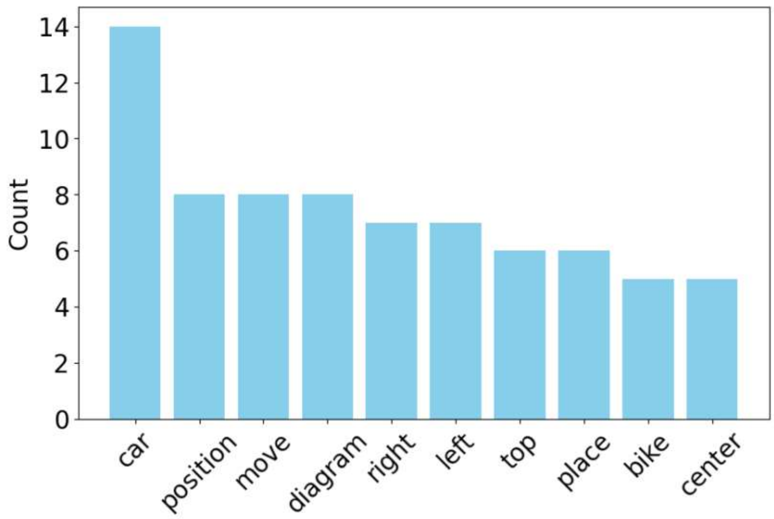


Figure 4.9: Most common words in positioning.

- "Remove the 'Client' class from the diagram."
- "Undo the last change."

These responses highlight potential gaps in the predefined categories and suggest areas where users expect additional functionality beyond basic diagram manipulation. The need to focus on specific elements to allow for shorter commands was also expressed.

Using traditional input methods like mouse and keyboard, setting the focus is intuitively provided, while it must be explicitly included in the set of voice commands.

4.3.3 Summary of Survey Results

The survey provided useful information on how users naturally express commands when interacting with a UML class diagram editor. While the diversity of responses demonstrated varying phrasings for common tasks, several key trends are shown that suggest practical approaches to designing a voice-controlled UML editor.

- **Consistency in Basic Commands**
  Across interaction types, such as creating classes, adding attributes, and renaming elements, participants showed a clear pattern in command structure. Most responses followed a "verb + object" format, such as "create class car" or "add an attribute called name". This consistency provides a solid foundation for implementing a basic command recognition system.

- **Use of Synonyms**
  As anticipated, participants used various synonyms and phrases to describe the same actions. For instance, "attribute" was interchangeably referred to as "field" or "property", and methods were occasionally referred to as "functions". Designing the system to handle these variations will be essential to ensure smooth user interaction.

- **Complexity in Modification and Relationships**
  Commands related to modifying attributes, methods, or creating relationships were often more detailed. Users included specific parameters and visibility settings, suggesting the need for the system to accommodate complex instructions with clarity and precision. Allowing the focus to be set on a specific element might also help reduce the complexity of queries.

- **Positioning and Layout Control**
  When it came to positioning elements within the diagram, participants used directional terms like "below", "next to", or specific coordinates. The expectation here was not only for basic placement but also for the ability to adjust layouts in relation to existing elements.

- **Focus as a Key Concept**
  One of the most important findings from the survey was the need to set a focus within the diagram. As one respondent noted, "being able to set a context for future commands (and being able to reset the context of course) would make using such a system a lot easier". This is particularly relevant in complex diagrams with nested classes, attributes, and methods, where long prompts could become cumbersome, both for users to phrase and for the system to parse and understand. Users expressed a desire for commands that allow them to focus on a specific element. Once the focus is set, subsequent commands could implicitly refer to

that element, simplifying interactions. For instance, instead of saying "rename the attribute weight in the class Car to mass", the user could issue two shorter commands: "focus on attribute weight" followed by "rename to mass". This method improves usability, particularly when handling more complex UML diagrams.

Since there is a limited set of notation elements for UML models in general - and in the context of this work for class diagrams - a set of possible operations can be derived for these elements by reviewing existing literature and evaluating existing editors like bigUML. The user survey contributed to finding concrete examples for all these operations, providing full coverage of all possible interactions. To answer (**RQ1**), it can be concluded that a taxonomy can be developed to define all possible interactions with a UML class diagram through NLP.

4.4 Derive training data

The previous section focused on gathering information both on notation elements and concrete queries users expect when working with a UML editor. In this section, the process of deriving training data for the NL model is described. This includes formulating example queries per intent as well as annotating these queries to allow further processing in the development process and training a language model.

This language model will serve two purposes:

- **Intent recognition**
  Each user query can be mapped to an intent defined in the taxonomy. This serves as the first step in extracting information from a user query. With the information gathered in the survey, intents might be grouped into the same intent, with the focused element providing the context. For example, there is no need to distinguish between renaming a class, a property, or a method. The intent is always the same, with the focused element providing the context.

- **Entity extraction**
  Entities within the query are information regarding element names, visibility, and types.

4.4.1 Approach

The survey defined groups of possible intents. With the additional request to provide focus, it is possible to group different actions into the same intent. For example, renaming a class and an attribute does not require an explicit distinction when parsing the user's intent if the focus is already set to the class or attribute. Therefore, the derived intents are as follows:

- **Create Container**
  Contains queries for creating primitive types, data types, enumerations, classes, interfaces, and packages.

- **Add Attribute**
  It contains queries for adding attributes and requires focus; the focus defines the element to which the attribute is added (class, interface, operation, etc.).

- **Add Method**
  Contains queries for adding methods; the focus must be set to a class or interface.

- **Change Name**
  Requires focus and changes the name of the selected element.

- **Change Visibility**
  Requires focus and changes the visibility of the selected element.

- **Change Datatype**
  Requires focus and changes the datatype of the selected element.

- **Add Relation**
  Contains queries for adding a relation. As these commands are the most complex ones given the number of different attributes, users must follow a stricter pattern when defining relations. The relation type must be specified alongside the source and target elements.

- **Delete**
  Deletes the focused element.

- **Focus**
  Sets the focus on an element.

- **Move**
  Moves the focused element in relation to another element within the diagram.

Setting the focus presents another challenge, as it is common for multiple elements to have the same name. Therefore, it is not sufficient to provide the name of the element to focus on, but also provide additional information such as type and container. There is also no formal constraint that element names must be unique within a diagram. For example, the query "focus on the method getName" might be ambiguous, as a method with this name might be present in multiple classes. A reworked query would be "focus on the method getName within the class person".

4.4.2 From Template Query to Training Data

This section describes how template queries are defined for each intent and how actual training data can be derived from these queries. On the example of the intent "Create Container", a template query might be:

`Create a` **`{container_type}`** `named` **`{container_name}`**`.`

The placeholders **{container_type}** and **{container_name}** can be replaced by actual examples, such as "class", "interface", or "datatype" for container type and nouns for the container name. The resulting queries might be:

`Create a` **`class`** `named` **`car`**`.`
`Create a` **`package`** `named` **`university`**`.`

This approach enables the generation of an arbitrary number of actual user queries without the need for manual phrasing. Providing multiple different intents - derived from queries formulated in the survey - allows for covering different wordings and supporting users when using different phrasing of queries for the same intent. See Listing 8.3 for a full list of the template queries and values for the placeholders.

In addition to the generated queries themselves, annotation information on the intent and entities must be stored. Table 4.3 shows a minimal example of generated training data from a single template query.

Query	Intent	Entities
Create a class named car	CreateContainer	container_type: class, container_name: car
Create a package named university	CreateContainer	container_type: package, container_name: university

Table 4.3: Generated training data set

This approach enables the generation of multiple thousands of queries per intent, along with annotated information on the intent and entities to expect. This data serves as the basis for training language models when implementing the NL server.

4.5 Prototype Development and Language Model Requirements

Before choosing a robust solution for intent recognition and entity extraction, a prototype was created in Python to understand the challenges of programmatically interpreting natural language. This prototype used a straightforward approach using the `spaCy`[1] library for tokenization and the `fuzzywuzzy`[2] library to match tokens with predefined keyword sets. The implementation details of the prototype are shown in Listing 8.1.

4.5.1 Prototype Design

The prototype is structured into three main components:

1. **Tokenization**
   Input queries were processed using Spacy's tokenization capabilities to extract individual words (tokens).

2. **Keyword Matching**
   Tokens were compared against predefined keyword sets, categorized as:

   - Actions (e.g., "create", "modify", "delete")
   - Objects (e.g., "class", "attribute", "method")
   - Directions (e.g., "above", "below", "left")
   - Identifiers (e.g., "name", "id", "label")

   Matching was performed using `fuzzywuzzy`, which calculated a similarity score for each token-keyword pair. Tokens exceeding a predefined similarity threshold were considered matches.

3. **Threshold Configuration**
   A threshold value was used to filter potential matches, ensuring a balance between false positives and missing relevant keywords.

4.5.2 Observation and limitations

Through testing, the prototype revealed several limitations:

- **Pattern Dependency**
  The approach performed well for queries that closely followed predefined patterns that used exact or near-exact keywords.

- **Scalability**
  Expanding the keyword set to accommodate diverse user expressions required a big manual effort and configuration. This was impractical for enabling true natural language understanding.

[1] `https://pypi.org/project/spacy/`, last accessed 2025-05-26
[2] `https://pypi.org/project/fuzzywuzzy/`, last accessed 2025-05-26

- **Varied User Expressions**
  The survey summarized in Section 4.3 showed that users often used varied and unpredictable wordings for similar intents. Given that the prototype only performed well when using a strict structure in the input queries, it could not generalize well, making it unsuitable for the goals of this thesis.

Extracting entities, such as names, proved particularly challenging. In the prototype, it was assumed that the name of an entity is always followed by its corresponding keyword.

For example, in the query "Create a class called car", the keywords "create" and "class" were correctly identified. However, the prototype incorrectly extracted "called" as the class name, as it was the token directly following "class".

Addressing this simple case required noteworthy effort and configuration, illustrating the complexity of accurately extracting entities in varied natural language inputs.

While the prototype offered a good learning experience, it shows the need for a more sophisticated approach to handle diverse and unstructured user inputs. This led to the decision to fine-tune a pre-trained BERT model, using its advanced language understanding capabilities for intent recognition and entity extraction.

This chapter describes how the requirements for developing the NL Server were obtained. First, a taxonomy for all supported operations for UML class diagrams is derived in general. Then the existing bigUML editor is reviewed with regard to whether and how these operations are available. The first user survey to gather example queries is evaluated, and general template queries are derived from it. Finally, a primitive first prototype is developed, highlighting the need for a more advanced NL model to support different wordings without overly restricting users.

CHAPTER 5

Technologies

This section provides an overview of the technologies and models used for developing the NLI, as well as developing an API to expose the features for integration in existing tools such as BIGUML.

5.1 Python

The natural language server is developed in Python, as it is considered the standard language for training and implementing machine learning tasks, which also includes the field of natural language processing. Python provides an extensive set of libraries, such as `TensorFlow` [ABC+16] and `Transformers` [WDS+20], that can be utilized for training a domain-specific model.

Additionally, the Python ecosystem extends beyond just training models. For example, Natural Language Toolkit `NLTK` [BKL09], and `spaCy` [HMVLB20] are two libraries providing support for tasks like tokenization and stemming, an important step for data preprocessing before training models.

The following sections provide an overview of the key libraries used in this work to build and train the language models, as well as to perform speech-to-text transformation. These libraries form the foundation of the thesis and handle various tasks, from tokenization to model training and deployment. While implementation details will be covered in Chapter 6, this section focuses on the general purpose and role of each library. Each of these libraries helped in developing and fine-tuning the BERT model. By using tools like `Transformers`, `Pandas`, and `Datasets`, it was possible to streamline the model training process, manage data efficiently, and focus on the quality of the training data. The combination of these libraries also allowed for a smooth integration between data preprocessing, model training, and evaluation.

5.1.1 Speech to Text Frameworks

As the user input will be provided by voice, speech-to-text processing is a key part of this work. To put as little load as possible on the client, this processing should also be done by the NL server. A key requirement is that the speech-to-text processing should be possible without the need to connect to an external service, using a model available offline within the NL Server.

Multiple engines and APIs provide speech-to-text functionality. The Python library `SpeechRecognition` [pyp] provides access to multiple of these services. To get an overview of the capabilities, each one is evaluated against a set of audio recordings, and the accuracy is compared. The APIs evaluated are:

- Google Speech to Text[1]
- Microsoft Azure Speech[2]
- IBM Speech to Text[3]
- Vosk API[4]
- Whisper API[5]

The selection of a suitable speech-to-text framework followed the requirement for offline processing using locally downloaded models. This criterion ruled out cloud-based services such as Google Speech Recognition, Microsoft Azure Speech, and IBM Speech to Text, all of which require authentication and account setup for access. The frameworks that met the offline requirement were Vosk API and Whisper API, both supporting locally hosted models for speech-to-text conversion.

A benchmark data set was used to evaluate these frameworks. The dataset was sourced from OpenSLR[6], containing a collection of pre-recorded audio files from various speakers, along with the corresponding text for evaluation.

A Python-based evaluation pipeline was developed for this purpose. Implementation details are shown in Listing 8.2. The pipeline processed each audio recording using the respective framework (Vosk or Whisper) to produce transcribed text, which was then compared against the expected ground truth. Text comparison was done using the `SequenceMatcher` class from the Python `difflib` package [dif], which computes a similarity ratio between two strings. Prior to comparison, a preprocessing step was

[1] `https://cloud.google.com/speech-to-text`, last accessed 2025-05-26
[2] `https://azure.microsoft.com/products/ai-services/ai-speech/`, last accessed 2025-05-26
[3] `https://www.ibm.com/products/speech-to-text`, last accessed 2025-05-26
[4] `https://alphacephei.com/vosk/`, last accessed 2025-05-26
[5] `https://github.com/openai/whisper`, last accessed 2025-05-26
[6] `https://www.openslr.org/12`, last accessed 2025-05-26

implemented to remove punctuation marks from the transcriptions, ensuring a focus on word-level accuracy rather than formatting discrepancies.

Each transcription was assigned a similarity score based on the computed ratios. The results of this evaluation indicated that while both APIs provide accurate speech-to-text functionality, the Whisper API slightly outperformed the Vosk API in terms of transcription accuracy across the test dataset. Consequently, Whisper was selected as the speech-to-text framework for this thesis due to its superior performance in the benchmark evaluation.

Expected	**Vosk Score**	**Whisper Score**
Stevie accustomed to move about disregarded had got up from the kitchen table carrying off his drawing to bed with him.	0.983	1.000
The prospect was as black as the window pane against which he was leaning his forehead.	0.983	0.983
On the dais under the golden dome the king cried out again racked by awful paroxysms.	0.982	0.988
The ladies in yellow dresses stand again in a ring about the emperor's daughter and are for the last time accosted by the singer with his lute.	0.982	0.982
He watched him gesticulating and murmuring in the kitchen.	0.982	0.860

Table 5.1: Example of Speech-to-Text Evaluation Scores using Vosk and Whisper. Text results for each model were omitted for better readability.

5.1.2 Bidirectional Encoder Representations from Transformers (BERT)

BERT is a language representation model introduced by Devlin et al. [DCLT19]. It demonstrates advancements in the field of NLP by enabling models to better understand the context of words in a sentence. Unlike earlier models that processed text in a unidirectional fashion (typically left-to-right or right-to-left), BERT uses a bidirectional approach. This means it considers the full context of a word by looking at the words both before and after it, resulting in a better understanding of language.

The architecture of BERT is based on the Transformer encoder, a neural network design introduced by Vaswani et al. [VSP+17]. The innovation of BERT lies in its pre-training methodology. It is trained on unlabeled text data using two unsupervised tasks: Masked

Language Model (MLM) and Next Sentence Prediction (NSP). In MLM, some tokens in the input are randomly masked, and the model learns to predict the original tokens based on surrounding words. In NSP, the model is trained to determine whether one sentence logically follows another, helping it learn relationships between sentences.

Once pre-trained, BERT can be fine-tuned with minimal adjustments for a wide range of downstream NLP tasks, such as sentiment analysis, named entity recognition, question answering, and text classification. This flexibility has contributed to its widespread adoption in both research and industry.

Popular implementations of BERT, such as those provided by Hugging Face's Transformers library [WDS+20], have made it easier to integrate it into applications.

In the context of this thesis, BERT is used for intent recognition and entity extraction from single-sentence inputs. Its bidirectional context modeling enables accurate identification of user intent by considering the full structure of a sentence. This makes BERT a powerful foundation for bridging natural language input to standardized UML commands. As a starting point, the pretrained model `bert-base-uncased`[7] is used in this work.

5.1.3 Pandas

`Pandas` [WM10] is one of the most popular Python libraries for data manipulation and analysis. It provides data structures like `DataFrames`, which allow for efficient data handling. In this thesis, `Pandas` was used to load the raw data to train and test the model. Sentences and their corresponding entity annotations were stored in a Comma-separated values (CSV) file with many thousands of entries, which was read and parsed to Python dictionaries using `Pandas` to create the input data for the BERT model.

5.1.4 Datasets Library

The `datasets` library [LVdMvP+21], developed by Hugging Face[8], is a lightweight and efficient tool for loading and managing datasets in machine learning workflows. In this thesis, the library was used to convert pre-processed text and labels into the format required for model training. Specifically, the `Dataset` class allowed for the structured representation of training and testing data, which was then fed into the model for training.

The key features of this library are:

- **Efficient Loading**
  It allows large datasets to be loaded into memory with minimal latency, making it suitable for real-time applications.
- **Dataset Manipulation**
  The library provides tools to manipulate datasets, such as tokenizing text and

[7] `https://huggingface.co/google-bert/bert-base-uncased`, last accessed 2025-05-28
[8] `https://huggingface.co/`, last accessed 2025-05-26

aligning labels to match the tokens, which was essential in pre-processing the data before training.

5.1.5 Transformers Library

The `Transformers` library [WDS+20], also developed by Hugging Face, is one of the most widely used Python libraries for NLP tasks. It provides pre-trained models and tools to easily implement state-of-the-art architectures, such as BERT and GPT. In this thesis, the `AutoTokenizer` and `AutoModelForTokenClassification` modules are used to handle tokenization and model loading for further training tasks.

- **AutoTokenizer**
  The tokenizer is responsible for converting input sentences into token IDs that the BERT model can process. In this thesis, the BERT tokenizer (specifically `bert-base-uncased`) was used to split sentences into subword tokens, which align well with the pre-training setup from BERT. This tokenizer also handles special tokens, such as `[CLS]` and `[SEP]` (beginning/end of a sentence), which are needed for BERT models to function correctly.
- **AutoModelForTokenClassification**
  This module from the `Transformers` library provides an easy way to load a BERT model pre-configured for token-level classification, such as entity extraction. It includes the necessary architecture for named entity recognition by assigning labels to each token in a sequence. In this thesis, the model was fine-tuned on labeled data to extract entities from the text.

The `Transformers` library makes it simple to download and fine-tune pre-trained models, reducing the complexity of training models from scratch.

5.1.6 Trainer and TrainingArguments

The `Trainer` and `TrainingArguments` classes, also from Hugging Face's `Transformers` library [WDS+20], are key components that simplify and automate the process of training models. These abstractions provide a high-level API that handles most of the repetitive tasks involved in fine-tuning pre-trained models, which allows to focus on model performance rather than the technical implementation of the training loop. In this thesis, these tools were used to fine-tune the BERT model for the intent recognition and entity extraction task.

Trainer

The `Trainer` class is used for managing the entire model training process. It handles the following tasks:

- **Data loading and batching**
  Automatically processing the training and evaluation datasets into batches to be fed into the model during each iteration.

- **Loss calculation**
  Computing the loss between model predictions and actual labels is important to monitor the model's accuracy.

- **Backpropagation**
  Automatically computing gradients and updating model parameters during training using optimization algorithms.

- **Evaluation**
  Running the evaluation dataset at regular intervals to monitor the model's performance during training.

The `Trainer` saves checkpoints and the final model, ensuring that the fine-tuned model is preserved. In this thesis, the `Trainer` was configured to train the BERT model using the pre-processed dataset. By specifying the model, training dataset, and evaluation dataset, the `Trainer` automatically handled the training of the language model. This simplified the implementation of the training and fine-tuning process.

TrainingArguments

The `TrainingArguments` class allows customization of the training process through various parameters. These arguments control key aspects of the model's training, evaluation, and saving behavior. In this thesis, several parameters were defined in `TrainingArguments` to tailor the training process:

- Output directory (`output_dir`)
  Specifies the directory where the trained model and tokenizer are saved after training. This ensures that the model is available for future use without retraining.

- Number of training epochs (`num_train_epochs`)
  Defines how many times the model will see the entire training dataset during training. For this thesis, three epochs were used, which is generally sufficient for fine-tuning pre-trained models like BERT.

- Training batch size (`per_device_train_batch_size`)
  Determines the number of examples processed by the model at once on each processor. A higher number improves training performance but requires more memory during training.

- Evaluation strategy (`evaluation_strategy`)
  Controls when the evaluation dataset is run during training. In this thesis, the

evaluation was run after each epoch to monitor model performance and prevent overfitting. This is important to ensure the model does not develop a bias and generalizes well to unseen data.

Another useful feature of the `TrainingArguments` is setting `push_to_hub`, which directly uploads the model to the Hugging Face Model Hub for version control.

This chapter discusses the various technologies and their frameworks used in this project. While the NLI Server was developed using Python, which provides good frameworks for speech-to-text processing and for training and testing NL models, extensions made in bigUML were developed using Java and JavaScript.

CHAPTER 6

Implementation

With the theoretical requirements defined as well as the technology to use in the NL Server, the next step is to design the applications' architecture as a whole. Special focus must be laid on integrating the NL Server in the bigUML tool. Therefore, clear interfaces must be defined for the interaction between the different components.

The implementation process itself is structured in three components. First, the NL Server is developed from scratch, using Python and exposing a REST API. Then, the bigUML Client gets extended by a new component allowing the user to input natural language queries by text and voice. As a final step, the bigUML Server is adapted to account for changed requirements in the commands sent by the Client.

The source code of NL Server[1] and the extended bigUML editor[2] is available on GitHub.

6.1 NL Server Architecture

Defining the overall application design serves as the first step in the development process. This includes defining interfaces that allow integration in the existing bigUML tool. The conceptual overall application design is shown in Figure 6.1. The sections highlighted in blue have been developed in this work. The main focus lies on the NL Server, responsible for parsing user queries to their respective intent and extracting the relevant entities contained. The NL Service exposes a REST API to allow a seamless integration in existing tools.

The bigUML client is responsible for user interaction and communication with the NL Server. Therefore, it is extended with a new component, allowing user input by written text as well as audio.

[1] `https://github.com/big-thesis/Schwantler.NLIforUML`, last accessed 2025-05-28
[2] `https://github.com/sschwantler/bigUML`, last accessed 2025-05-28

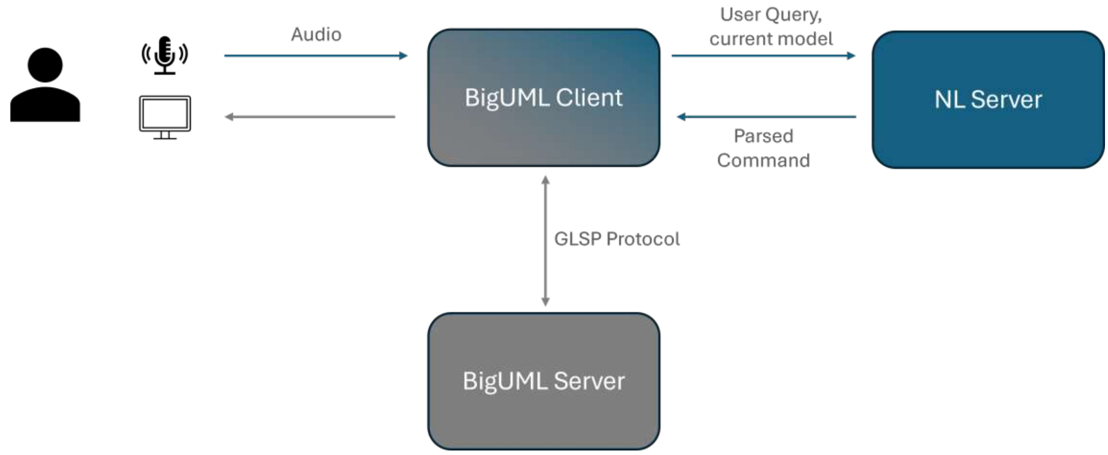


Figure 6.1: Overall Application Design

The bigUML Server should not require any changes from a conceptual point of view, as all required actions are already implemented and used when interacting with the bigUML client using the mouse and keyboard. However, as the next section will point out, small adaptations have to be made to allow a seamless integration of a natural language interface.

It must also be ensured that the new possibility of modeling with natural language commands does not interfere with any existing interaction methods. Users should always be able to decide whether to use classical interaction, keyboard shortcuts introduced by [SMB23], or the NLI.

To put as little load as possible on the bigUML client, the speech-to-text functionality will also be implemented at the NL Server. Therefore, the client is only responsible for capturing the audio, and further processing will be done on the server side. As the speech-to-text processing will be done by using a locally available model and no external services, integrating this model into the bigUML client would also contradict the design concept of keeping any responsibility apart from displaying the model out of the client. The addition of another dependency to the client would introduce a second point of failure to the process of modeling with NL. Keeping this functionality in the NL server helps to keep all NL functionality in one place. While the implementation of the NL Server is oriented towards the bigUML, this design allows the integration of the NL Server in other UML editors with little adaptations.

6.1.1 Interfaces

Since the NL Server and bigUML are developed using different technology stacks, a standardized interface must be defined to allow seamless integration. Also, users should not be required to install any additional tools when working with the natural language interface. The resulting two design considerations are:

- The NL Server functionality should be exposed by a REST API.
- The NL Server should be containerized using Docker[3].

This enables easy testing and deployment of the final application with minimal overhead.

6.2 NL Server Implementation

With the technical concept defined in Chapter 5 and the requirements from Chapter 4, the first step is to implement the NL Server.

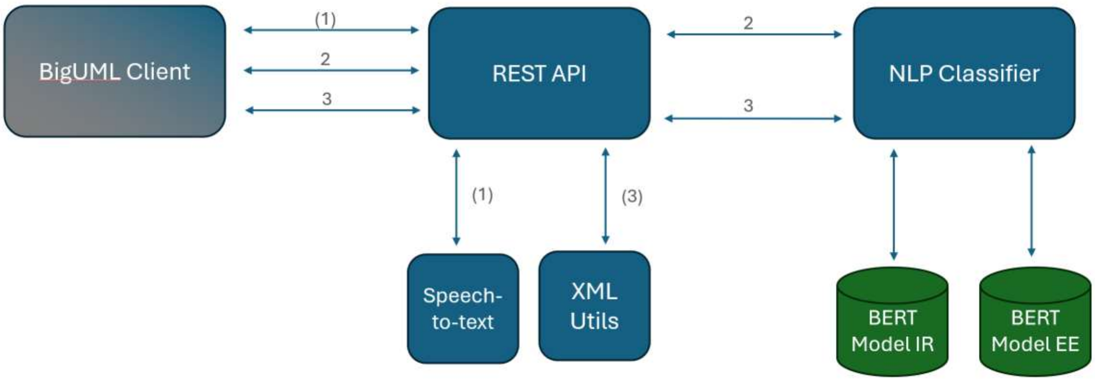


Figure 6.2: Application Design Natural Language Server

Before discussing the implementation of each component in detail, Figure 6.2 provides an overview of the final interaction pattern between the BIGUML client and the NL Server.

1. Perform speech-to-text processing, returning the user query as text.
2. Perform intent recognition.
3. Perform entity extraction. This might require the additional step of mapping the name of an entity to its ID in the UML model, since the name is not a unique identifier in UML class diagrams in general as well as in BIGUML.

The interface between client and server is provided as a REST API described in Section 6.2.6.

Multiple steps must be taken to implement the server. First, training data must be gathered, and the language models must be trained. Before making these models available, their performance and accuracy are evaluated using automated software tests. Once the testing results show sufficient performance, the language models are exposed through a standardized interface via a REST API.

[3] `https://www.docker.com/`, last accessed 2025-05-26

6.2.1 Training Data

In a first step, the functionality for generating training data is implemented. With such a service in place, thousands of queries can be generated in seconds without the need to annotate any data manually.

This generation of training data can be automated using a Python script. Listing 6.1 shows the code used for generating queries for the discussed intent "CreateContainer". Data generation for every other intent follows a similar pattern. Line 4 picks a random template query for creating a container (see templates defined in Listing 8.3). The next step is to select actual values for the placeholders in the query, which is done through lines 7 to 13. Note that the placeholder for container abstraction is only filled in every fifth generated query and left empty otherwise, with the intention of more accurately reflecting real-world examples. The generated data is finally stored in a CSV file, which can be utilized in the following training process. The data is formatted in a structured way, as shown in line 26. The final sentence is stored along with its intent and the relevant entities.

```
1  def generate_create_container_sentence():
2
3      # TEMPLATE QUERIES contain multiple queries per intent
4      template = random.choice(TEMPLATE_QUERIES[CREATE_CONTAINER_INTENT])
5
6      # CONTAINER_ABSTRACTIONS indicates if the container should be abstract
7      container_abstraction = random.choice(CONTAINER_ABSTRACTIONS) if decision
       (0.2) else ""
8
9      # CONTAINER_TYPES contain a list of available containers
10     container_type = random.choice(CONTAINER_TYPES)
11
12     # CONTAINER_NAMES contain a list of nouns
13     container_name = random.choice(CONTAINER_NAMES)
14
15     # Fill in entities into placeholders
16     sentence = template.format(container_abstraction=container_abstraction,
      container_type=container_type, container_name=container_name)
17
18     # Store keywords in dictionary
19     entities = {
20         "container_abstraction": container_abstraction,
21         "container_type": container_type,
22         "container_name": container_name,
23     }
24
25     # return generated sentence along with intent and keyword entities
26     return {"text": sentence, "intent": CREATE_CONTAINER_INTENT, "entities":
      entities}
```

Listing 6.1: Method for generating annotated training data for intent "CreateContainer"

Repeating this process for all defined intents generates two CSV files, one containing training data and one containing test data, which will both be utilized in the next step when training the BERT model. The complete script used for filling out these templates is attached in Listing 8.4.

6.2.2 Training the BERT Models

For the implementation of the NL server, two separate BERT models were trained: one for intent recognition and another for entity extraction. This choice was driven by the distinct nature of the two tasks. While they are related, they require different approaches and learning objectives.

Intent recognition involves identifying the user's high-level goal or action. It focuses on understanding the overall meaning of a query and classifying it into predefined categories. Entity extraction, on the other hand, deals with identifying specific details or elements within the query. This task requires fine-grained parsing and a precise understanding of the input.

Using separate models allowed each to specialize in its task. A single model handling both tasks might have struggled to balance the broader contextual understanding needed for intent recognition with the detailed focus required for entity extraction. Separate models ensured better performance by allowing each to optimize for its own specific requirements.

This approach also made the system more modular. Each model could be trained, updated, or replaced independently. This flexibility simplifies development and testing while reducing the risk of changes to one task impacting the other. This will also be backed by the evaluation, as the intent recognition model performed well from the beginning, while the more complex entity extraction required some fine-tuning and improvement of the training data. As these models were separated, the time-consuming task of training had to be done only for one model.

While the approach for training both models shows similarities, there are noteworthy distinctions. These differences are discussed in the next two sections.

Intent Recognition

The intent recognition model was designed to classify the desired action from natural language queries. The detailed Python code used is shown in Listing 8.5. The goal was to map each query to one of several predefined intent categories. The training process began by preparing a dataset of queries annotated with their respective intents. This dataset was loaded in CSV format, containing separate files for training and testing data. A tokenizer based on the pretrained `bert-base-uncased` model was used to process the input text. The tokenizer transformed each query into a sequence of token IDs, ensuring compatibility with the BERT architecture. Input sequences were padded and truncated to a maximum length of 128 tokens for consistency.

A configuration for the model was created, specifying the number of intents and associating them with unique labels. If a pretrained model checkpoint was available locally or from the Hugging Face Hub, it was loaded and validated against the current label set to ensure consistency. If no such checkpoint was available, the training started from the base BERT model.

The fine-tuning was performed using the Hugging Face Trainer API, which simplified the setup of the training process. Key parameters included batch sizes of 64 for both training and evaluation, and training over three epochs. An epoch represents one complete pass through the entire training dataset. During each epoch, the model processes every data sample in batches, updating its weights iteratively to reduce the loss. Evaluating the model on the validation dataset at the end of each epoch helps track its performance and prevents overfitting. The batch size was chosen as a trade-off between training speed and available memory. The training itself was performed using Google Colab and an A100 GPU[4].

The optimal parameter set, including learning rate, batch size, and number of epochs, was determined through experimentation and monitoring of validation performance. In this case, three epochs were chosen to balance sufficient training time with generalization capability, minimizing the risk of overfitting.

Once training was complete, the model and tokenizer were saved locally for deployment. If required, the model could also be pushed to the Hugging Face Hub for versioning and sharing. This approach provided a robust and efficient solution for classifying intents, ensuring that the system could reliably interpret user goals.

Entity Extraction

Compared to training the model for intent recognition, training for entity extraction requires additional steps in preparing the training data. The detailed Python code used is shown in Listing 8.6. Each query cannot be evaluated as a whole but has to be checked word by word. While certain keywords have a high value, others can be ignored. This can best be described by the following example:

```
Create a new abstract interface named skateboard
```

contains seven words, but only three contain relevant information concerning entity extractions, while others can basically be ignored. By using this example, the next subsections show how the query is processed.

Tokenization

Tokenization plays an important role in preparing user queries for training a BERT language model. The objective of this pre-processing step is to transform raw textual data into a format that can be understood by the model while preserving the structure of entities.

[4] `https://cloud.google.com/compute/docs/gpus`, last accessed 2025-05-28

The training data consists of textual queries along with associated intent labels and entity annotations. Each query represents a command or statement, and the entities within the query are labeled with specific categories.

Continuing with the example from above, training data is provided in the structure shown in Listing 6.2.

```
Text: Create a new abstract interface named skateboard
Entities: {container_abstraction: abstract, container_type:
    interface, container_name: skateboard}
```

Listing 6.2: Example Training data

BERT uses `WordPiece` tokenization [hugc], a subword-based approach that allows the model to handle out-of-vocabulary words effectively. The tokenizer splits words into smaller units, ensuring that even unseen words can be represented using known subword components. If the tokenizer does not recognize a word like "Skateboard" as a whole word, it may be split up as follows:

$$"skateboard" \longrightarrow "skate", "\#\#board"$$

While "skate" is treated as an independent token, "board" is prefixed with "##", indicating it is a continuation of a previous token. This approach ensures that common subwords are reused, enhancing efficiency and handling rare words effectively.

Note that by default, the tokenizer adds special tokens to the beginning and end of each input. These tokens are `[CLS]` and `[SEP]` (beginning/end of a sentence). Since the training data is limited to one sentence, these special tokens are not considered in the training or when using the model.

Once tokenization is complete, the next step is to align entity labels with the corresponding tokens. Since entity spans may be split into multiple tokens, a Begin-Inside-Outside (BIO) tagging scheme is used:

1. B-Entity (Beginning of an entity)
2. I-Entity (Inside an entity)
3. O (Outside of any entity)

The first token of an entity is labeled as "B-entity", and any subsequent token is annotated with "I-entity". Tokens with no information are labeled "O". Table 6.1 shows the tagged result of the running example.

Since machine learning models do not process text directly, each label is mapped to a numerical ID. This mapping must also be provided when using the model in a later stage, to ensure recognised tokens are mapped correctly back to entities.

Tokenized Text	Label
Create	O
a	O
new	O
abstract	B-container_abstraction
interface	B-container_type
named	O
Skate	B-container_name
##board	I-container_name

Table 6.1: Example of BIO Tagging for Tokenized Text.

After tokenization and alignment, the training data consists of a list of tokenized words and a list of corresponding labels mapped to IDs.

The actual training of the model follows a similar approach to that described for intent recognition.

Training Artifacts

After training, several files are generated to store the trained model and its associated components. These files are necessary for loading the model later for inference. Below is a description of each file:

- **config.json**
  This file contains the model's configuration, including the number of labels, model architecture, and other hyperparameters. It ensures that the model can be loaded with the correct settings when used for inference.

- **model.safetensors**
  Stores the trained model weights in a safe and efficient format. Safetensors format is preferred over binary files as it prevents memory corruption issues and loads faster [hugb].

- **special_tokens_map.json**
  Defines any special tokens used by the tokenizer. This file was not changed during training and is the same as for the base model.

- **tokenizer_config.json**
  This file stores the tokenizer's configuration details, such as vocabulary size, truncation settings, and special token handling. It is the same as the base model and is not changed during training.

- **vocab.txt**
  This is a vocabulary file listing all WordPiece tokens used by the tokenizer. Each line corresponds to a token and its index in the vocabulary.

These files are essential for loading and using the trained model for entity extraction and intent recognition, discussed in the next section.

6.2.3 Usage of the BERT models

Once the models for intent recognition and entity extraction have been trained and testing shows sufficient performance, they can be used for inference on new input text. This section describes the inference process and provides a step-by-step explanation of how predictions are made.

Before making predictions, the trained models and their respective tokenizers must be loaded.

The models are loaded from their respective directories using Hugging Face's AutoClassifiers [huga]. The corresponding tokenizers are also loaded from the trained models, ensuring that the same tokenization rules are applied as during training.

This functionality is bundled in a single class, allowing easy usage in other components, as shown in Section 6.2.4.

Performing Intent Recognition

Intent recognition aims to classify an input sentence into one of the predefined intent categories.

- **Tokenization**
  Tokenization is performed in the same way as used to train the model, splitting each word into a token and optionally further into subwords.
- **Model Inference**
  The tokenized input is passed to the intent recognition model. The model outputs raw scores for each intent class ID. These IDs are mapped to the actual intents using the ID-to-label information stored in the model configuration.
- **Softmax Activation**
  The logits are transformed into probabilities using the softmax function. The intent with the highest probability is selected.

Performing Entity Extraction

Entity extraction identifies specific parts of the sentence that correspond to predefined entity types. The steps are similar to intent recognition, with an additional step of reconstructing split-up tokens:

- **Tokenization**
  Tokenization is performed in the same way as used to train the model, splitting each word into a token and optionally further into subwords.

- **Model Inference**
  The tokenized input is fed into the entity extraction model, which assigns a label ID to each token. These IDs are mapped to the actual entity labels using the ID-to-label information stored in the model configuration.

- **Label Reconstruction**
  The previous step provides a list of predictions for each token. Listing 6.3 shows how these predictions are applied to the tokens.

```
1 tokens = ['Create', 'a', 'new', 'abstract', 'interface', 'named', 'skate
    ', '##board']
2 predicted_labels = ['O', 'O', 'O', 'B-container_abstraction', 'B-
    container_type', 'O', 'B-container_name', 'I-container_name']
3
4 prediction = {}
5 for token, label in zip(tokens, predicted_labels):
6     if token.startswith("##") and current_label is not None:
7         prediction[current_label] += token.replace("##", "")
8     elif label.startswith("B-") and not token.startswith("##"):
9         current_label = label[2:]
10        prediction[current_label] = token
11    elif label.startswith("I-") and current_label is not None:
12        if token.startswith("##"):
13            token = token.replace("##", "")
14        else:
15            token = f" {token}"
16        current_label = label[2:]
17        prediction[current_label] += token
18    else:
19        # O-Label without ##
20        current_label = None
21        pass
```

Listing 6.3: Reconstruction of labels from prediction

A notable part here is that errors in the prediction can be corrected, as seen on line 6. For example, a subword (starting with ##) should always be considered if the previous token is relevant. For the running example, this means that even if "##board" is not recognized and falsely labeled "O", it would still be considered. The final prediction result is shown in Listing 6.4.

```
    {
      "container_abstraction": "abstract",
      "container_type": "interface",
      "container_name": "skateboard"
    }

```

Listing 6.4: Example entity extraction output

This approach enables accurate intent classification and entity extraction, making the model applicable in real-world scenarios.

6.2.4 NlpClassifier - Unified Interface for Model Interaction

The `NlpClassifier` class (see Listing 8.7) is a centralized interface that bundles the functionality of the intent recognition and entity extraction models. Its purpose is to provide a standardized and simplified way to interact with the models, ensuring that future developments, such as API integrations, can be implemented seamlessly. This also helps in keeping the system extensible in the future, as the actual language model can potentially be replaced by any other model as long as it complies with the interface.

This class allows for the initialization and use of the trained models with minimal overhead, as shown in Listing 6.5.

```
classifier = NlpClassifier(
    ir_model_path="", # path to intent recognition model
    ee_model_path="", # path to entity extraction model
)

intent = classifier.predict_intent(user_query)
entities = classifier.predict_entities(user_query)
```

Listing 6.5: Usage of `NlpClassifier`

The class is designed to handle both intent recognition and entity extraction within a single structure. This eliminates the need to manage each model separately, reducing code complexity and enhancing maintainability.

By providing clear and consistent methods, the class ensures that all model interactions follow the same protocol. This standardization simplifies future integration with external systems.

One of the key advantages of the `NlpClassifier` is its modular design, which allows for easy replacement or upgrading of models without impacting the surrounding codebase. As long as new models implement the existing method interfaces, they can be integrated with minimal changes, allowing the system to use potentially improved models in the future.

The current implementation of the interface methods follows the steps described in Section 6.2.3.

6.2.5 Utility Methods

In addition to the core functionalities for intent recognition and entity extraction, the system includes key utility methods that support tasks such as speech-to-text conversion and mapping UML entity names to their corresponding IDs and metadata.

Speech-to-Text Conversion

As discussed in Section 5.1.1, Whisper is used for speech-to-text processing. The implementation is lightweight, as the model can be loaded and utilized with just a few lines of code. Despite its simplicity, this component is critical for converting spoken queries into text that can be processed by the NLP models.

Since the speech-to-text process relies entirely on Whisper's capabilities, it does not require additional custom preprocessing beyond providing the input in the expected binary format.

XML Parsing for UML Model Mapping

Since the user queries will specify elements by their name, but processing them will require their unique identifier, functionality for mapping names to IDs is implemented. This is achieved using `etree` from the lxml library[5], an Extensible Markup Language (XML) parsing tool that allows querying and data extraction.

It's important to note that the XML parsing component is heavily tailored to support the bigUML editor. This customization is necessary because there is no standard in how UML models are represented in such editors and therefore use unique XML schemas and data structures.

The bigUML editor uses two files containing an XML representation of the UML model. One `.uml`-file containing unique IDs for UML elements (e.g., classes, attributes, methods) and one `.unotation`-file which holds additional metadata such as position, size, and graphical details of UML elements.

The parsing workflow begins by searching for an element using its name. This returns the corresponding unique identifier for the UML element. Once the ID is obtained, metadata such as the element's size and position is obtained. This data is then available in downstream processes for diagram manipulation.

6.2.6 REST API

A REST API is exposed using `FastAPI`[6] to provide a standardized interface for interacting with the intent recognition and entity extraction models. This API allows external applications (like bigUML) to send requests and receive predictions in a standardized way, serving as a key component in the architecture of the application. The design ensures modularity, scalability, and future-proofing.

The API is structured with a clear separation between:

- **Business Logic** where requests are processed using voice-to-text and the `NlpClassifier` described in Section 6.2.4.

[5] `https://lxml.de/`, last accessed 2025-05-26
[6] `https://fastapi.tiangolo.com/`, last accessed 2025-05-26

- **OpenAPI specification**[7] ensuring consistency in request/response formats. OpenAPI provides a language-independent specification that describes the structure, endpoints, and data models of the API. This ensures that the API can be easily understood, documented, and integrated with applications developed in various programming languages. It also comes with a simple User Interface (UI) out of the box (see Figure 6.3).
- **Data Models** can be generated from the OpenAPI specification, which reduces manual coding and potential errors.

6.2.7 Testing the BERT Models and the NL Server as a whole

After training the intent recognition and entity extraction models, it is important to evaluate their performance to identify potential issues and help further fine-tune them. The testing process combines automated testing with randomly generated data and manual validation to ensure robust model performance. This section outlines the testing approach using Python's testing framework `PyTest`[8], the evaluation methodology, and the role of manual testing in addressing potential biases.

Automated Testing using PyTest

Automated tests allow for the fast evaluation of model performance across thousands of queries with minimal manual effort. The script used for testing is shown in Listing 8.8. The tests focus on providing details on any issues and allow performance tracking regarding accuracy and time consumed. The testing process uses automatically generated test data created using the approach described in Section 6.2.1. This ensures that the models are exposed to a wide range of input variations, covering different intents and entity structures. Using the same approach to generate training and test data holds the risk of not recognizing a bias in the generation of the queries. This shows the importance of additional manual testing with actual users in a later stage.

During testing, the intent recognition and entity extraction models process each query from the test dataset. The predictions are then compared with the expected outputs. For intent recognition, the predicted intent is checked for an exact match with the expected intent. For entity extraction, the comparison involves verifying that all extracted entities correspond correctly to their expected labels and values.

The testing framework checks the correctness of the predictions and tracks the performance metrics for each intent. After each test run, the results are aggregated to calculate accuracy scores as percentages per intent, providing an overview of model performance across different intent categories. This automated approach allows the evaluation of thousands of test cases, reducing the time and effort required for model validation.

[7] `https://www.openapis.org/`, last accessed 2025-05-26
[8] `https://docs.pytest.org/`, last accessed 2025-05-26

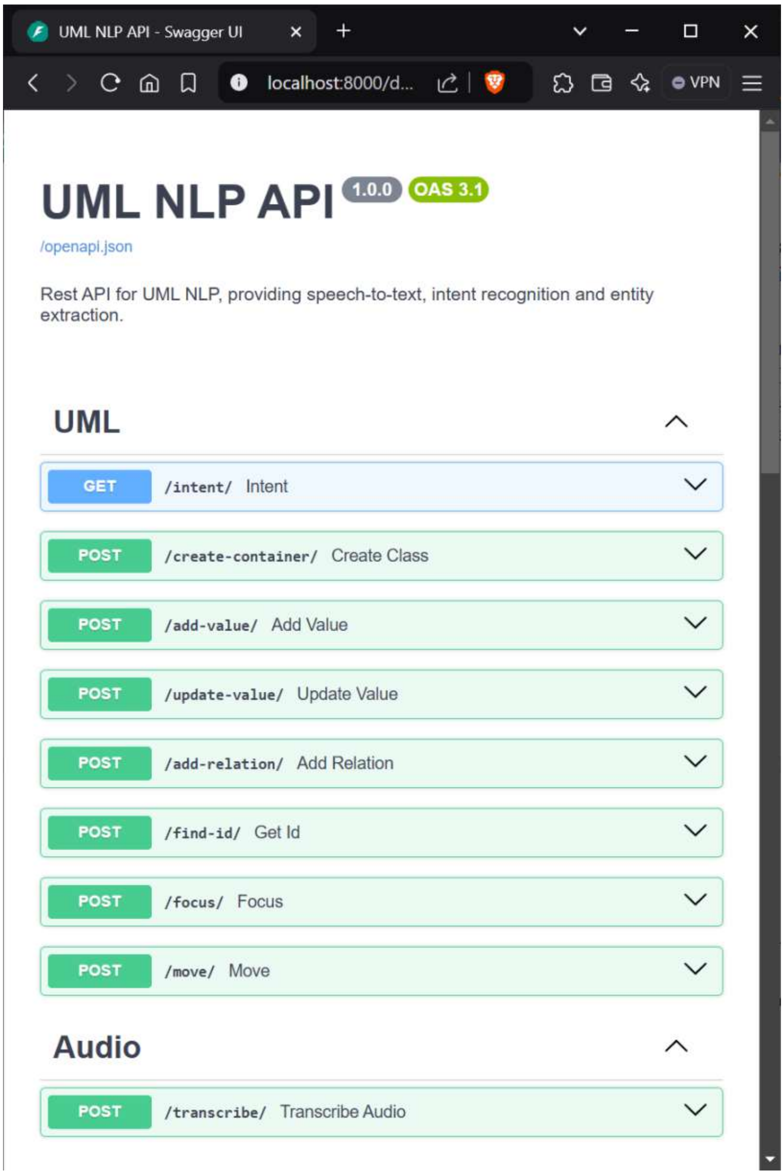


Figure 6.3: NL Server UI (Swagger)

When the models produce incorrect predictions, these errors are captured in log files. Each log entry includes the original query, the expected results, the actual predictions made by the models, and performance metrics such as the time taken to generate each prediction.

An example log message is shown in Listing 6.6. Lines 1 to 7 show the log message for an error. It can be observed that the intent was predicted correctly (lines 4 and 5), while the entities show a difference for the extracted method name (lines 6 and 7). In contrast, lines 9 to 15 show the result of a successfully tested query, where the expected and actually extracted values match. One test run contains multiple such debug and error logs for each intent. Line 17 displays a summary of a test run, where the percentage of correct predictions is summarized by intent.

```
1  2025-02-01 10:22:48,769 ERROR
2      Time(IR): 0.0261s | Time(EE): 0.0650s
3      ’Develop a private function disconnect with return type short’
4      Expected: AddMethod
5      Actual: AddMethod
6      Expected: {’datatype’: ’short’, ’attribute_method_name’: ’disconnect’, ’
       visibility’: ’private’}
7      Actual:   {’datatype’: ’short’, ’attribute_method_name’: ’disconnect with
       ’, ’visibility’: ’private’}
8
9  [...]
10
11 2025-02-01 10:52:05,573 DEBUG
12   Time(IR): 0.0541s| Time(EE): 0.0820s
13   ’Create a data type called Pen’
14   Expected: CreateContainer
15         Actual: CreateContainer
16   Expected: {’container_type’: ’data type’, ’container_name’: ’Pen’}
17         Actual: {’container_type’: ’data type’, ’container_name’: ’pen’}
18
19 [’CreateContainer: 100.0%’, ’AddAttribute: 100.0%’, ’AddMethod: 97.0%’, ’
       ChangeName: 99.8%’, ’ChangeVisibility: 100.0%’, ’ChangeDatatype: 100.0%’,
        ’Delete: 100.0%’, ’AddRelation: 97.8%’, ’Focus: 100.0%’, ’Move: 99.0%’,
       ’Undo: 100.0%’]
```

Listing 6.6: Example shortened log output of a test run

This structured logging enables easy error analysis. By reviewing the logs, patterns can be identified in incorrect predictions, such as recurring issues with specific intents or consistent misclassifications of certain entities. For example, if the model frequently mislabels a particular entity type, this could indicate the need to adjust the training data or fine-tune the model parameters.

In addition, tracking any errors allows specific test cases to be easily replicated. The ability to reproduce errors also helps to track the effectiveness of model updates, ensuring that improvements are consistently reflected in the test results.

Although automated testing provides good coverage and efficiency, manual testing is an essential component of the evaluation process. Automated tests based on generated data can introduce biases that may not reflect real-world usage. These biases might result from patterns in the data generation process that the model unintentionally learns, leading to artificially high performance on test cases that are similar to the training data. Manual testing helps to reduce these risks by exposing the models to unpredictable real-world input variations.

End-to-End Test using PyTest

The general performance of the BERT models can be evaluated on its own, as shown in the previous section. Since some operations require an existing model, and therefore its representation, additional end-to-end tests are performed. In this setup, the `NlpClassifier` is not called directly, but all calls are made against the actual REST API, to simulate the integration in bigUML. The full script for this test is shown in Listing 8.9.

These tests are also implemented in Python and describe an example use case, as shown in Listing 6.7. The provided data includes the endpoint to call, the user query, and, optionally, the files representing the current model state. In addition to evaluating the performance of intent recognition and entity extraction, the utility functions, such as parsing names to IDs in the model's representations, were also assessed.

```
("/create-container", "Create an abstract class named Person")
("/add-value", "Add the private attribute name of type string", "1.uml", "1.unotation")
("/add-value", "Add the private attribute size of type string", "2.uml", "2.unotation")
("/focus", "Select the attribute size", "3.uml", "3.unotation")
("/update-value", "Change the name to weight")
("/update-value", "Change the visibility to protected")
("/create-container", "Create a class named Student")
("/focus", "Select class Student", "4.uml", "4.unotation")
("/move", "Place it right of class Person", "5.uml", "5.unotation")
("/add-value", "Add the private attribute student id of type string", "6.uml", "6.unotation")
("/add-value", "Create the public method get student id with return type string", "7.uml", "7.unotation")
("/add-relation", "Create a generalisation from Student to Person", "8.uml", "8.unotation")
```

Listing 6.7: Use-Case Description for end-to-end test

Listing 6.8 shows the result of the test. It can be observed that the NL Server was able to process all queries and extract the relevant information from the provided model files. As this test goes a step further and works with actual model representations, the results include metadata information for further processing. For example, lines 16 to 18 demonstrate how an element is selected. The provided UML diagram is searched for an attribute with the name "size", and once it is found, all its metadata, such as ID, position, and size, are returned.

```
1 Create an abstract class named Person
2 {"intent":"CreateContainer"}
3 {"element_name":"person","element_type":"class","is_abstract":true}
4
5
6 Add the private attribute name of type string
7 {"intent":"AddAttribute"}
8 {"element_name":"name","value_datatype":"_PqgqkDPeEfCMOsDMQjZukA","value_visibility":"private"}
```

```
Add the private attribute size of type string
{"intent":"AddAttribute"}
{"element_name":"size","value_datatype":"_PqgqkDPeEfCMOsDMQjZukA","value_visibility":"private"}

Select the attribute size
{"intent":"Focus"}
{"element_id":"_lbzMoDPeEfCMOsDMQjZukA","element_name":"size","x_coord":17.0,"y_coord":20.0,"height":45.0,"width":60.0}

Change the name to weight
{"intent":"ChangeName"}
{"new_value":"weight"}

Change the visibility to protected
{"intent":"ChangeVisibility"}
{"new_value":"protected"}

Create a class named Student
{"intent":"CreateContainer"}
{"element_name":"student","element_type":"class","is_abstract":false}

Select class Student
{"intent":"Focus"}
{"element_id":"_92h2IDPeEfCMOsDMQjZukA","element_name":"student","x_coord":218.0,"y_coord":109.0,"height":45.0,"width":107.10677337646484}

Place it right of class Person
{"intent":"Move"}
{"x_coord":618.0,"y_coord":108.0}

Add the private attribute student id of type string
{"intent":"AddAttribute"}
{"element_name":"student id","value_datatype":"_PqgqkDPeEfCMOsDMQjZukA","value_visibility":"private"}

Create the public method get student id with return type string
{"intent":"AddAttribute"}
{"element_name":"get student id","value_datatype":"_PqgqkDPeEfCMOsDMQjZukA","value_visibility":"public"}

Create a generalisation from Student to Person
```

```
{"intent":"AddRelation"}
{"class_from_id":"_92h2IDPeEfCMOsDMQjZukA","class_to_id":"
    _as5Z8DPeEfCMOsDMQjZukA","relation_type":"Generalisation","relation_name
   ":""}
```

Listing 6.8: End-to-End Test Result

This test can be seen as an intermediate step to semi-automatically test the integration into bigUML, without any actual dependencies on the client. With this approach, it was possible to conclude that the implementation on the NL Server was sufficient to handle an example use case.

6.3 bigUML Client

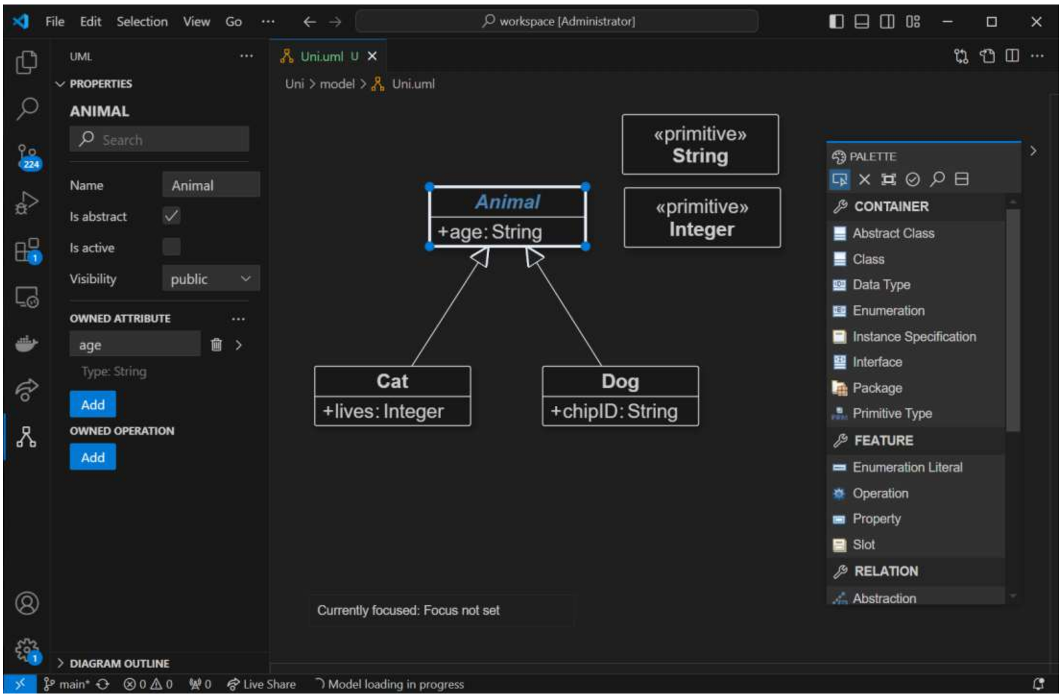
While the NL Server was developed from scratch, bigUML is an existing UML editor running in VS Code [BIG]. It enables users to create various UML diagrams through traditional interaction methods such as using a mouse and keyboard. The aim of this integration is to improve bigUML by adding NL capabilities, allowing users to create and modify UML diagrams through voice input and text-based commands powered by the NL Server.

6.3.1 Extending the User Interface

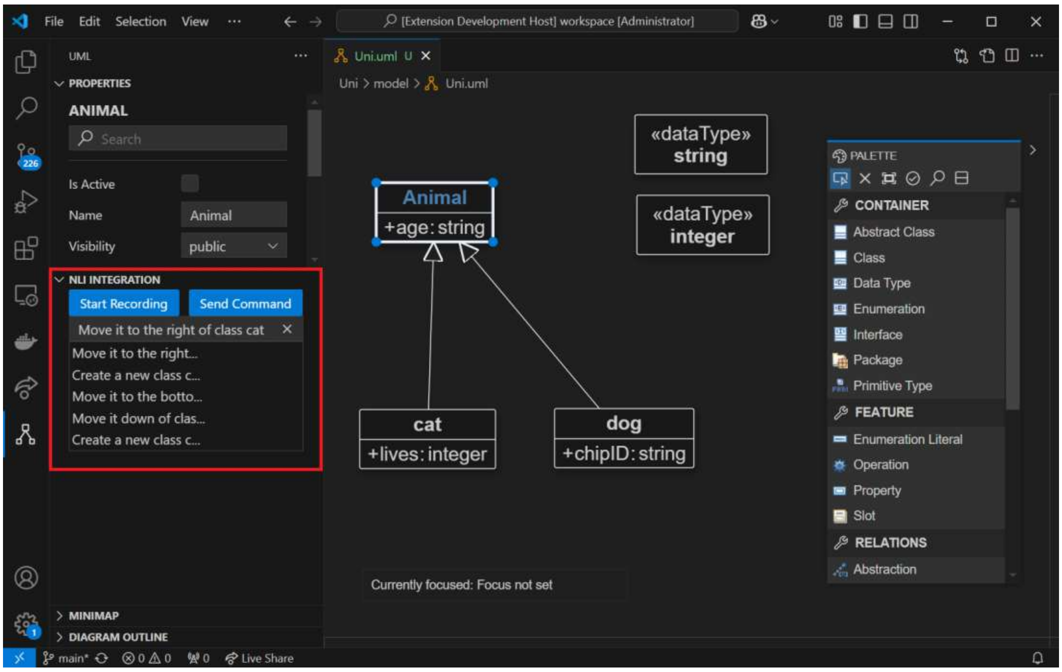
Before implementing any business logic or communication with the NL Server, the first step involved adding a simple, intuitive UI. The UI changes are implemented as a new component, following the existing modular architecture of bigUML to maintain consistency with the overall application structure. Figure 6.4 shows a comparison between the original UI and the extended UI after integrating the NL Server functionality.

The new UI component consists of four main elements, designed to enable seamless interaction with the NL Server while keeping the interface clean and minimalistic:

- **A text field** which allows users to enter commands manually using the keyboard. This is useful for situations where voice input may not be practical or when users prefer typed instructions.
- **Send Command Button** to execute the command written in the text field.
- **Start Recording Button** which activates the speech-to-text functionality, enabling users to issue commands via voice. This triggers a five-second audio recording, which is automatically processed. The transcribed text is also displayed in the text field.
- **Query history** displays each processed query and allows the user to select recently executed queries. This is especially useful for commands that are used multiple times in the process of modeling, such as setting the focus on a specific element.



(a) BIGUML before integration.



(b) BIGUML after integration.

Figure 6.4: BIGUML UI Changes

While the changes to the UI are kept to a minimum on purpose, the implementation details, including how these UI elements communicate with the NL Server and handle audio processing, are described in the following section.

6.3.2 Integration Implementation

The integration of the NL Server into bigUML required modifications to two existing packages. The `uml-components` package is extended by a new component (referred to as webview), responsible for handling user input, communicating with the NL Server, and updating the UML model based on recognized intents and extracted entities. Changes in the `uml-vscode-intergration` package (referred to as the provider) are responsible for accessing features like the microphone for voice recordings.

Communication Flow

When a user interacts with the editor, it follows a sequence of steps, as shown in Figure 6.2. The steps to record, process, and transcribe the audio are optional, as the query can also be provided as text directly.

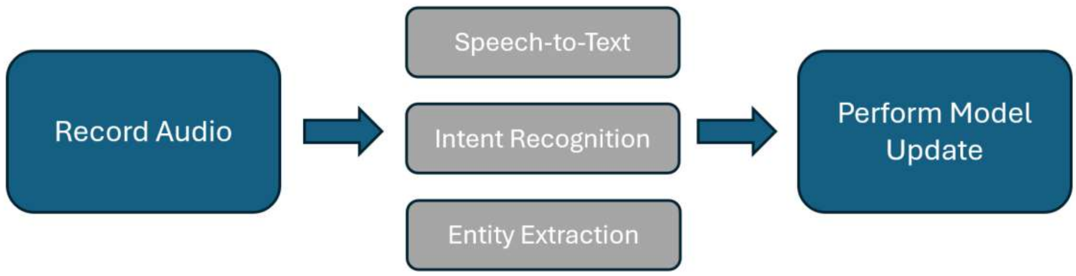


Figure 6.5: bigUML Client Flow

Once the audio recording is triggered, an action message is sent from the webview to the provider for the actual recording. Listing 6.9 shows the action method to trigger the recording by creating a new notification of the kind "startRecording".

```
protected async onRecordActionMessageStart(): Promise<void> {
    this.sendNotification({ kind: 'startRecording' });
}
```

Listing 6.9: JavaScript Listing - Start Recording

The request is processed by the provider, as shown in Listing 6.10. The audio is recorded using the sound package `sox`[9]. As shown in line 4, a five-second audio snippet is recorded and stored locally. Lines 7 to 10 show how this recording is parsed and sent out as an action message, for the webview to catch and further process.

[9] `https://sourceforge.net/projects/sox/`, last accessed 2025-05-26

```
private startRecording(): void {
    // ...
    this.recordingProcess = exec(
        `sox -d -b 16 -e signed -c 1 -r 16k "${outputPath}" trim 0 5`,
        (error, stdout, stderr) => {
            // ...
            const fileBuffer = fs.readFileSync(outputPath);
            const uint8Array = new Uint8Array(fileBuffer);
            const action = AudioRecordingCompleteAction.create(outputPath,
    uint8Array);
            this.webviewViewConnection.send(action);
            // ...
        }
    );
    vscode.window.showInformationMessage('Recording started...');
}
```

Listing 6.10: JavaScript Listing - Record Audio using Sox

The webview then forwards the recording to the NL Server for transcription, as shown in Listing 6.11. The request is built in line 6, converting the raw audio data to a file which is accepted by the API. Lines 16 to 18 show the processing of the response, where the transcription is extracted and the text field in the UI is populated.

```
1 protected async transcribeAudio(): Promise<void> {
2     // ...
3     formData.append("file", new File([this.audioBlob], "recording.wav", {
      type: "audio/wav" }));
4     const response = await fetch(NLI_SERVER_URL + '/transcribe/', {
5         headers: {
6             'accept': 'application/json',
7         },
8         method: 'POST',
9         body: formData,
10    });
11
12    if (!response.ok) {
13        throw new Error(`HTTP error! status: ${response.status}`);
14    }
15
16    const data = await response.json();
17    console.log(`Transcription: ${data.transcription}`);
18    this.inputText = data.transcription;
19 }
```

Listing 6.11: JavaScript Listing - Transcribe Audio

Once transcribed, Listing 6.12 shows the further processing of the query for intent recognition. The NL Server is called to predict the intent in line 2, and the method to handle the intended action is invoked in line 12.

```
1 protected async onStartIntent(): Promise<void> {
2     const response = await fetch(NLI_SERVER_URL + `/intent/?user_query=${this
      .inputText}`, {
```

```
        headers: {
            accept: 'application/json'
        }
    })
    if (!response.ok) {
        console.error(response.text)
    }
    const json = await response.json();

    await this.handleIntent(json.intent);
}
```

Listing 6.12: JavaScript Listing - Intent Recognition

Depending on the intent, a specific action has to be taken, and the corresponding endpoint for entity extraction is called. Listing 6.13 shows the case for creating a new container. First, the root element of the current diagram needs to be selected. This is done in line 1. Then, in lines 4 to 13, the NL Server is called to extract the entities from the user query. Finally, the desired action is triggered in lines 17 to 33, where the event to create a new element is dispatched. This event contains the type of container to create, along with its position and properties.

```
1 protected async createContainer() {
2     const root_json = await this.findIdByName("root", "root");
3
4     const response = await fetch(NLI_SERVER_URL + `/create-container/?
    user_query=${this.inputText}`, {
5         headers: {
6             accept: 'application/json'
7         },
8         method: "POST"
9     });
10    if (!response.ok) {
11        console.error(response.text);
12    }
13    const json = await response.json();
14
15    const containerType = umlTypesMap.get(json.container_type) ?? `
    CLASS__Class`;
16
17    this.dispatchEvent(
18        new CustomEvent('dispatch-action', {
19            detail: CreateNodeOperation.create(containerType,
20            {
21                containerId: root_json.id,
22                location: {
23                    x: 0,
24                    y: 0
25                },
26                args: {
27                    name: json.container_name,
28                    is_abstract: json.is_abstract
29                }
```

```
            })
        })
    );
}
```

Listing 6.13: JavaScript Listing - Execute Action

Finally, Listing 6.14 shows how an action message is triggered to update the model. This ensures that the latest changes are correctly displayed in the UI.

```
this.sendNotification({ kind: 'requestModelResources' });
```

Listing 6.14: JavaScript Listing - Update Model

To answer (**RQ2**), it can be concluded that the architecture of the existing bigUML editor is developed in a way that allows easy extensibility by adding new components. In this work, the UI of the editor was extended with a new component, providing a way to record audio queries and directly provide queries as text. Since the processing of queries is done within the NL Server, the bigUML editor only needs to forward these commands via the REST interface provided and apply model updates according to the result. It was possible to reuse this functionality from the existing implementation for classical user interaction using mouse and keyboard with minor changes, showing good extensibility within the existing tool.

6.4 bigUML Server

The bigUML Server, which handles model updates and keeps track of the state, required only minor changes to fully support the new NLI. In its current implementation, creating new elements is a two-step process, where the elements are first created with placeholder values using the mouse and are updated afterwards by providing names as text input. Since a NL query already contains all the required information, the functionality to create new elements is extended by allowing optional parameters specifying properties. Listing 6.15 shows the newly added arguments `name` and `is_abstract` for the example of creating a new class (lines 9 and 10). The same has been implemented for all other container types, as well as named relations, as shown in line 20.

```
1  CreateNodeOperation.create("CLASS__Class",
2      {
3          containerId: 1234,
4          location: {
5              x: 0,
6              y: 0
7          },
8          args: {
9              name: "Skateboard",
10             is_abstract: false
11         }
12     })
13
```

```
CreateEdgeOperation.create(
            {
                elementTypeId: "CLASS__association",
                sourceElementId: 1234,
                targetElementId: 5678,
                args: {
                    name: "drives"
                }
            })
```

Listing 6.15: Added Parameters for element creation

This chapter provided an overview of the development process for the NLI for bigUML. It described in detail how the training data was generated, how the BERT models were trained, tested, and exposed via a REST API, and how the NLI was integrated in bigUML.

CHAPTER 7

Evaluation

The goal of this chapter is to evaluate the effectiveness and usability of the integrated NLI within bigUML. To achieve this, a series of experiments are conducted. First, it is shown that all supported elements of bigUML can be created using voice commands. Then, the usability is shown by attempting to recreate a UML class diagram from the book "UML @ Classroom" [SSHK15], as shown in Figure 7.7. This approach enables a direct comparison between traditional interaction methods, which use a mouse and keyboard, and the newly introduced voice control system. Finally, a survey is conducted to gather feedback on the new NLI introduced in this work. Participants are asked to create the UML class diagram shown in Figure 7.8 from scratch using only voice commands.

The evaluation focuses on assessing both the functional capabilities and the practical limitations of the voice-controlled system in real-world scenarios.

7.1 Supported Interactions using the NLI

Before creating complete class diagram examples, each interaction is shown in an isolated example in bigUML.

7.1.1 Create container types

All supported container types can be created using voice commands. This forms the basis for further interaction, as the creation of containers is a basic operation within the process.

```
Create an abstract class named person.
Define a data type called list.
Create a new enumeration named grade.
Add an abstract interface named template to the diagram.
Make a package with the title software.
```

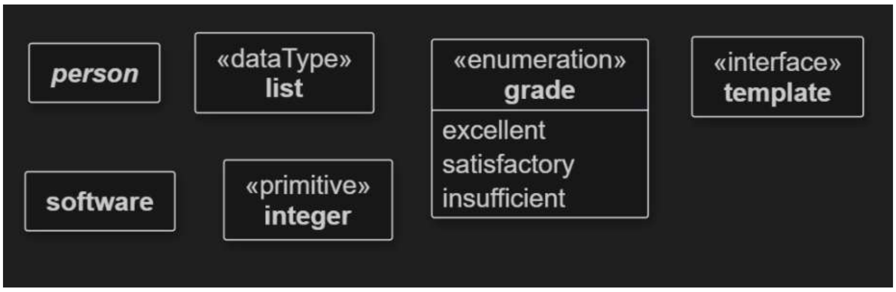


Figure 7.1: Container Types

```
Insert a primitive type integer into the current diagram.
```

Listing 7.1: Commands for Figure 7.1

After the container has been created, adding attributes to classes is supported. Examples of attributes are shown in the following example. Note that each data type used must be explicitly created.

Figure 7.2: Simple Class with Properties

```
Create a new class called lecture.
Add the data type float to the diagram.
Create a new data type called string.
Move it to the right of class lecture.
Focus on data type float.
Place it at the bottom of datatype float.
Set the focus on class lecture.
Add a new private attribute called name of type string.
Set the focus on class lecture.
Create the private variable hours with datatype float.
```

Listing 7.2: Commands for Figure 7.2

The next step is to extend this example by two public operations. Note that operations do not provide return values. This limitation is inherent to the editor itself and is not introduced by the NLI.

```
Set the focus on class lecture.
Add the public function get ects.
Focus on class lecture.
```

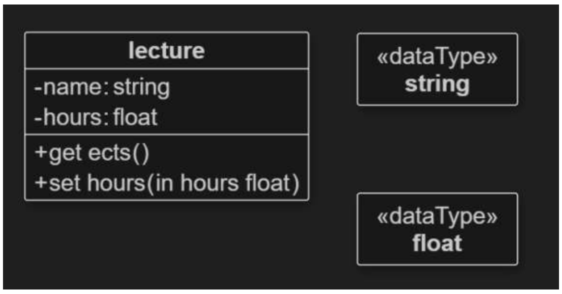


Figure 7.3: Simple Class with Operations

```
Create the public method called set hours.
Add the parameter hours of type float.
```

Listing 7.3: Commands for Figure 7.3

7.1.2 Creating Relations

BIGUML supports multiple relation types, as shown in Section 4.2. Continuing the example from above, all these relation types are supported by the NLI. Figure 7.4 shows an association relation.

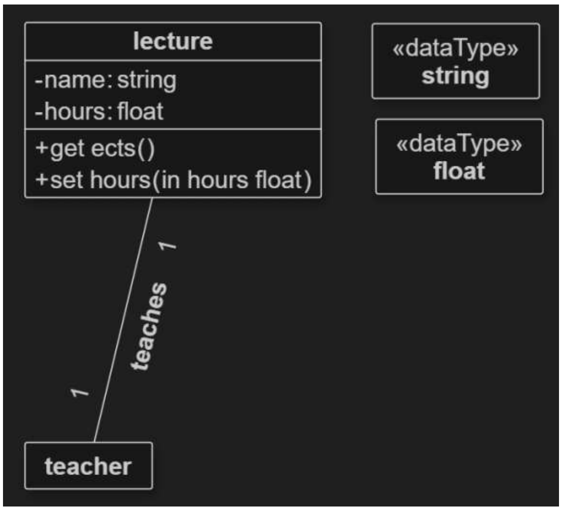


Figure 7.4: Association

```
Create a new association called teaches from teacher to lecture.
```

Listing 7.4: Commands for Figure 7.4

Relations containing multiplicities show the first limitation of the voice control system, as this is not supported in the current version of the NLI. Additionally, some relation types, such as generalizations, do not support setting a name and therefore cannot be selected.

Relations can be created between different classes, as well as for the same class, representing a self-association, see Figure 7.5. Again, multiplicities have to be set manually.

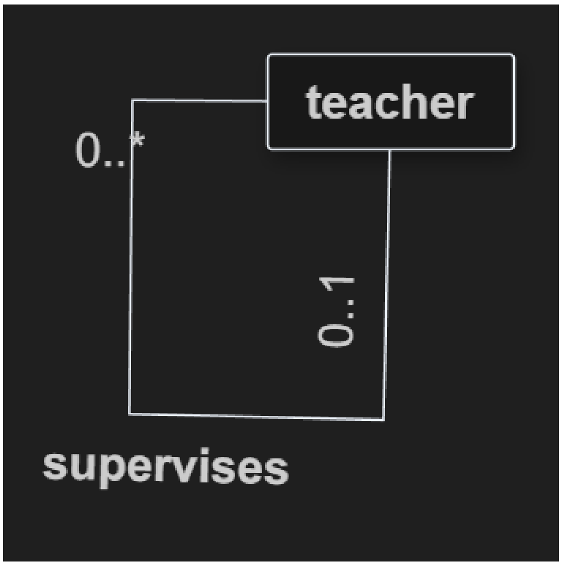


Figure 7.5: Self Association

```
Create an association called supervises from teacher to teacher.
```

Listing 7.5: Commands for Figure 7.5

All other relations supported by BIGUML can also be added using voice control. Figure 7.6 shows all an example of all supported relation types.

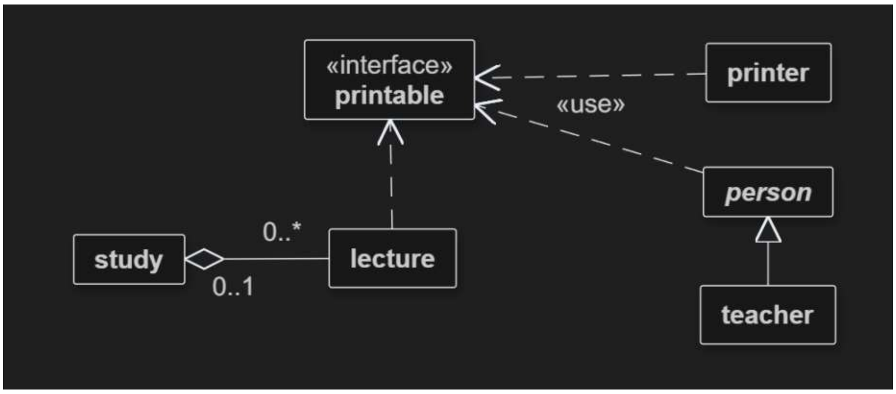


Figure 7.6: Other Relation Types

```
Add a usage between printer and printable.
Create a dependency from lecture to printable.
Between person and printable, create a new dependency.
```

```
Connect lecture and study with a strong aggregation.
Add a generalization from teacher to person.
```

Listing 7.6: Commands for Figure 7.6

In the process of creating elements, it may be necessary to make adjustments. The NLI supports renaming and deleting elements. The focus can be set on named elements, and the position can be adjusted, relative to other elements in the diagram. The utility method "Undo" is also recognized, rolling back the last operation performed. This is particularly useful when voice commands are not picked up correctly.

7.2 Creating a Real-World Example

The previous sections described all supported operations in theory. In this section, a real-world example is selected and modeled using only voice or text input.

7.2.1 University Information System

This example is described in [SSHK15], capturing a University Information System. Figure 7.7 shows the final result using the commands listed in Listing 7.7. Note that the final result resembles the example, with the only difference being the setting of multiplicities, as this is currently not supported in the prototype.

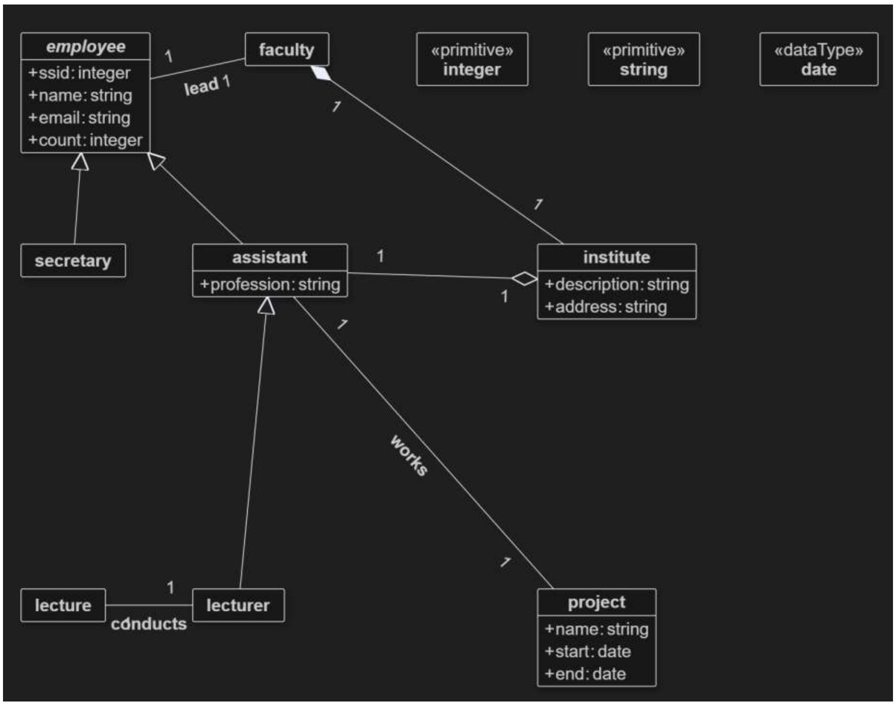


Figure 7.7: Example: University Information System [SSHK15]

```
Create a new abstract class called Employee.
Create a new class called Faculty.
Move it to the right of class Employee.
Create a new primitive datatype integer.
Move it to the right of class Faculty.
Create a new primitive datatype string.
Move it south of datatype integer.
Create a new data type date.
Move it south of datatype string.
Focus on class Employee.
Create a new public property called ssid of type integer.
Create a new public property name of type string.
Create a new public property email of type string.
Create a new public property called count of type integer.
Create a class called Secretary.
Move it to the bottom of Employee.
Create a new class called Assistant.
Move it to the right of class Secretary.
Add the public attribute profession of type string.
Create a new class called Institute.
Move it to the right of class Assistant.
Add the public attribute description of type string.
Add the public attribute address of type string.
Add a new class called Lecturer.
Move it to the bottom of Assistant.
Add the new class lecture to the diagram.
Move it to the bottom of Secretary.
Create a new class called Project.
Move it to the bottom of class Institute.
Add the public attribute called name of type string.
Add the public attribute called start of type date.
Add the pubic attribute called end of type date.
Add a generalization from secretary to employee.
Add a generalization from assistant to employee.
Add an association from employee to faculty called lead.
Add a composition from institute to faculty.
Add an aggregation from assistant to institute.
Add a generalization from lecturer to assistant.
Add an association from lecturer to lecture called conducts.
Add an association from assistant to project called works on.
```

Listing 7.7: Commands for University Information System (see Figure 7.7)

7.3 User Survey

With the final implementation completed, an evaluation was performed using a survey similar to the one described in Section 4.3. Users were tasked to create the class diagram shown in Figure 7.8 using only voice input.

```
Create a new data type called string.
Create a new data type called integer.
Move it to the right of data type string.
```

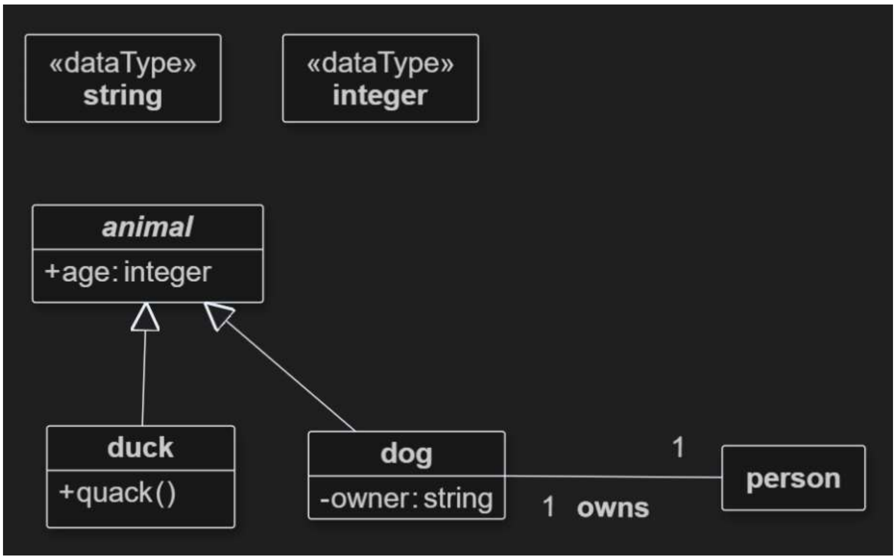


Figure 7.8: Survey Task - Animals

```
Create a new abstract class called animal.
Move it to the bottom of data type string.
Add a public property called age of type integer.
Create a new class called Duck.
Move it to the bottom of class animal.
Add a public method called quack.
Create a new class called dog.
Move it to the right of class duck.
Add a private property called owner of type string.
Add a new class person.
Move it to the right of class dog.
Create a generalization from duck to animal.
Create a generalization from dog to animal.
Create an association called Owns from person to dog.
```

Listing 7.8: Commands necessary for creating the Diagram in Figure 7.8

A detailed setup guide was provided[1], and participants were asked general questions about their experience in conceptual modeling and their usage pattern of voice assistant tools in their daily lives.

For the task itself, a detailed description of the class diagram was provided along with example commands to provide an introduction to the new tool.

To conduct the evaluation, users were asked to upload a screenshot of their final result, as well as the logging information of their modeling session. These logs contain the transcription history of the used commands, as well as the audio recordings. This data

[1] `https://github.com/sschwantler/bigUML/blob/main/NLI_README.md`, last accessed 2025-05-28

provides useful information for further improvements, highlighting commands that are working well and commands where users had problems.

The survey concluded with a feedback section, asking participants to rate their general experience and to provide written feedback on any potential problems or feature gaps they might have encountered. Finally, users were asked to complete a System Usability Scale (SUS) [Bro96], allowing standardized evaluation and calculation of a usability score. The SUS is a widely adopted and standardized questionnaire for evaluating the perceived usability of a system. It contains ten questions, which users rate on a five-point scale, ranging from strongly disagree to strongly agree. Statements alternate between positive and negative phrasing to minimize response bias and encourage thoughtful responses.

The evaluation results are presented in the following section.

Since the pipeline from recording the users' voice input to executing and displaying the intended action in the UI includes multiple steps, there are multiple points of failure in the system (see Figure 6.2). Each step requires the previous step to be successful, and issues introduced in any step cannot be corrected in later stages.

The evaluation of the user survey in the next section focuses on the performance of the language models in intent recognition and entity extraction. Incomplete user queries or wrong results of voice-to-text transcription prevent the language model from making correct predictions.

7.3.1 User Survey Evaluation

The evaluation is structured into four parts, each addressing a different aspect of the survey and user interaction.

The first part focuses on general participant information, particularly their prior experience with conceptual modeling and UML diagrams. In addition, details about their level of experience with voice assistants are requested, including the frequency of use in daily life, the types of assistants they use, and the specific tools or platforms they rely on.

The second part evaluates the task execution itself - whether participants were able to complete the modeling task using voice commands alone, how they rated the overall experience, and what insights can be drawn from their written feedback.

The third part analyzes the results uploaded by participants. This includes their final UML diagrams, which are submitted as screenshots, and the corresponding interaction logs. The logs contain all transcriptions as well as the predicted intents and entities extracted by the language model. These results are aggregated to identify recurring patterns, highlighting which voice commands were reliably interpreted and which ones led to errors or confusion.

Finally, the SUS [Bro96] is applied to derive a usability score for the system, providing a quantitative measure of the perceived ease of use.

Participants' Experience with Conceptual Modeling and Voice Assistance

The survey was distributed within the same user group as the previous survey described in Section 4.3. With 23 participants, the participation rate is significantly higher compared to the 11 participants in the previous survey. The results show that most participants had experience with both conceptual modeling and UML modeling. For conceptual modeling, the majority selected "Good experience" (10 responses), followed by "Some experience" (8 responses). Three participants reported having "Little experience", and two stated having "no experience". No one indicated having "Expert-level experience".

The results for experience in UML modeling follow a similar pattern, with "Good experience" being the most common choice (13 participants), followed by "Some experience" (7 participants). "No experience" was selected twice, while "Little experience" was chosen once. Again, "Expert-level experience" was never selected.

This distribution suggests that the sample group consisted of users with a moderate to strong background in modeling, making them well-suited to evaluate the usability of the voice-based modeling system.

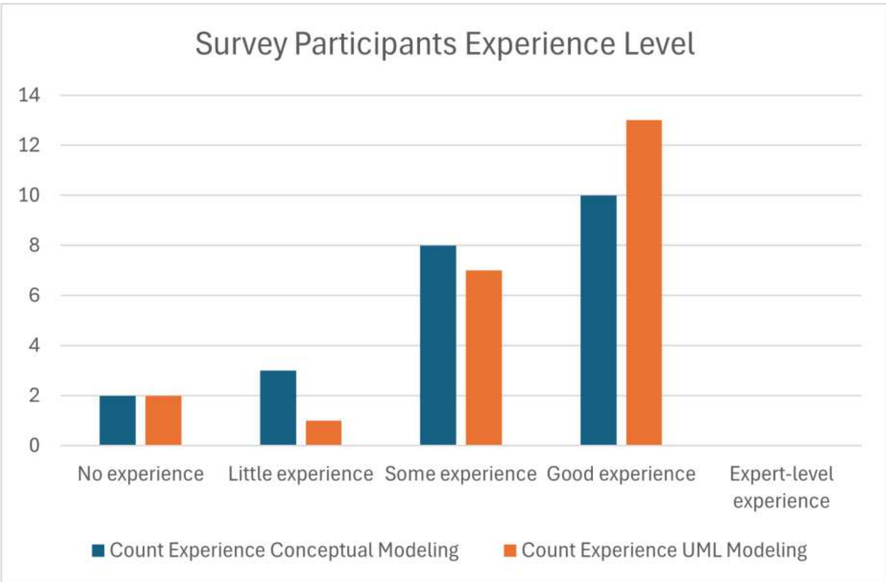


Figure 7.9: Survey Evaluation - User Experience

To assess participants' familiarity with voice interaction, the survey included a question on how often they use voice assistants. Most participants had at least limited experience. Nine reported having tried them once or a few times, five stated they use them several times per week, and four indicated usage several times per month. Two participants reported using them daily, while another two used them only a few times per year. One participant stated they had never used voice assistants.

Participants were also asked which systems they had used. Google Assistant and Alexa were selected most frequently (11 responses each), followed by Siri and voice translation tools (9 responses each). One respondent selected 'Other', and one person indicated that they were not using any assistants at all. The answers show that the group had used a variety of different systems, and most of the participants reported some prior interaction with voice-based tools.

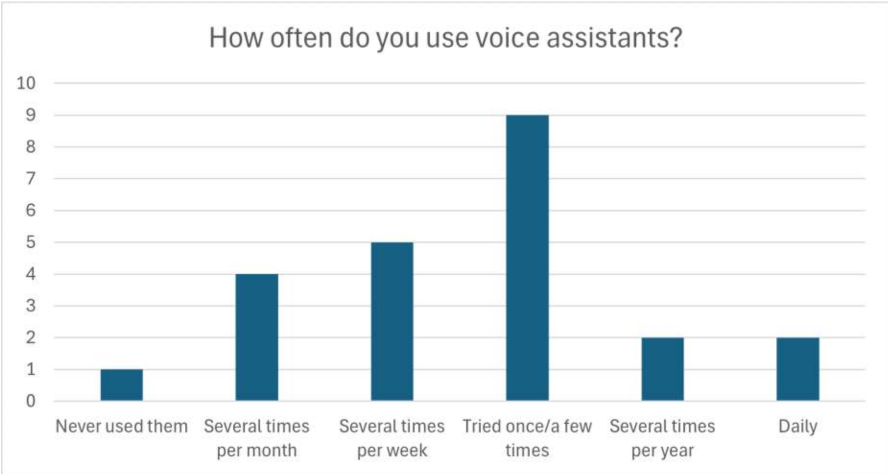


Figure 7.10: Survey Evaluation - Voice Assistant Usage

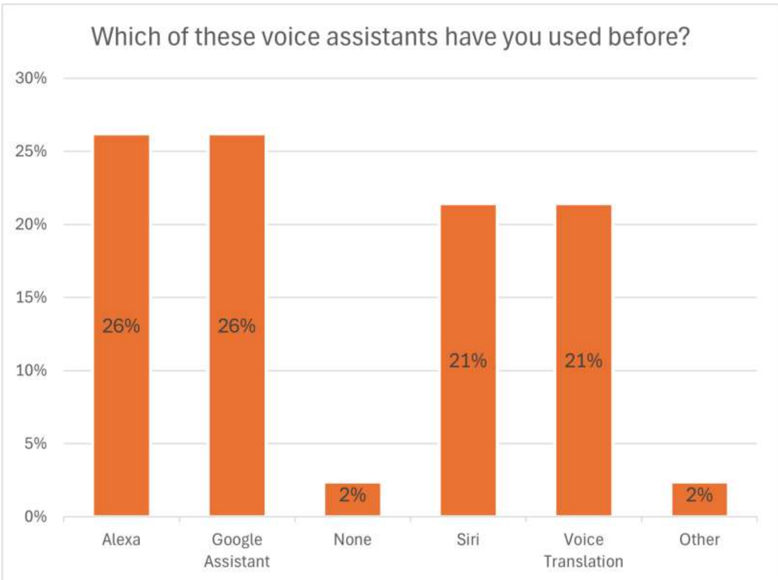


Figure 7.11: Survey Evaluation - Types of Voice Assistance

User Experience

When asked to rate the usability of the speech-based interface, participants gave a mixed response. Ten chose "Neutral", five rated it as "Good", and another five selected "Poor". The lowest possible rating, "Very Poor", was selected twice, and the highest rating, "Very Good", was selected once, suggesting that the system performed reasonably but did not fully meet expectations across the board.

Regarding task completion, thirteen participants stated they were able to finish the modeling task using voice or text input without switching to a mouse or keyboard. Ten were not able to complete the task this way. These numbers show that most users managed to work with the system successfully, but a noticeable portion still ran into issues that required traditional interaction methods.

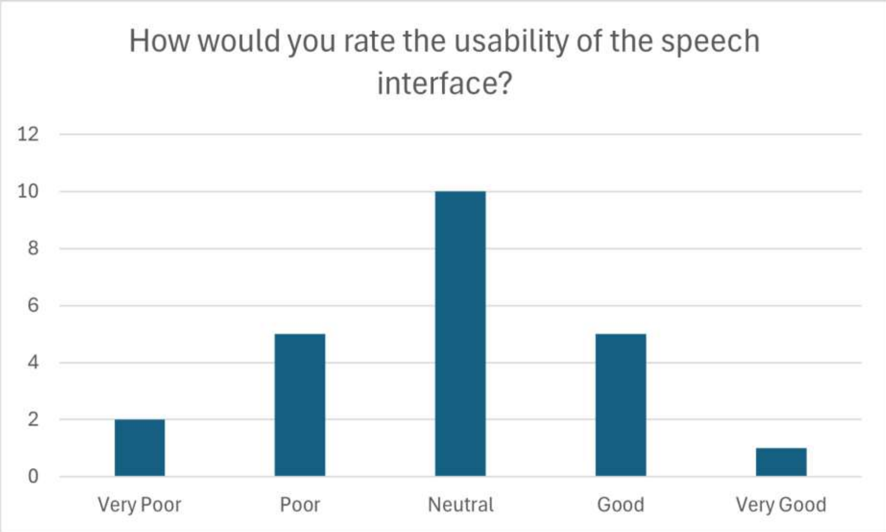


Figure 7.12: Survey Evaluation - Usability

System Usability Scale

To evaluate the usability of the voice-controlled modeling system, the SUS was used. This questionnaire includes ten items, alternating between positive and negative statements, such as "I thought the system was easy to use" and "I found the system unnecessarily complex". Participants rate each statement on a five-point scale. The final score ranges from 0 to 100, with higher values reflecting better usability.

The average SUS score across all participants was 66.85, with a median of 70.00. While many participants rated the system positively, two reported notably low scores (30.00 and 37.50), which lowered the overall average. These results suggest that while the system worked well for most users, a few encountered difficulties that affected their assessment.

To examine whether prior modeling experience influenced usability ratings, the average of each participant's experience in conceptual modeling and UML modeling was compared

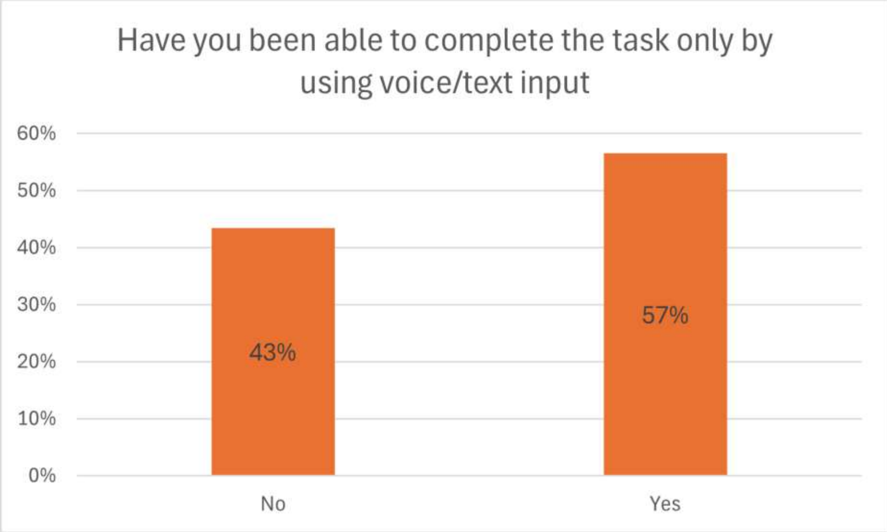


Figure 7.13: Survey Evaluation - Task Completion

to their SUS score. The corresponding chart (Figure 7.14) shows a loose upward trend: participants with more modeling experience often rated the system more favorably. However, the distribution also includes two outliers: One participant with no experience who gave a high score, and another with moderate experience who rated the system poorly. Although the trend points in a consistent direction, the relationship is not strictly linear. Due to the small sample size, these results should be interpreted with caution.

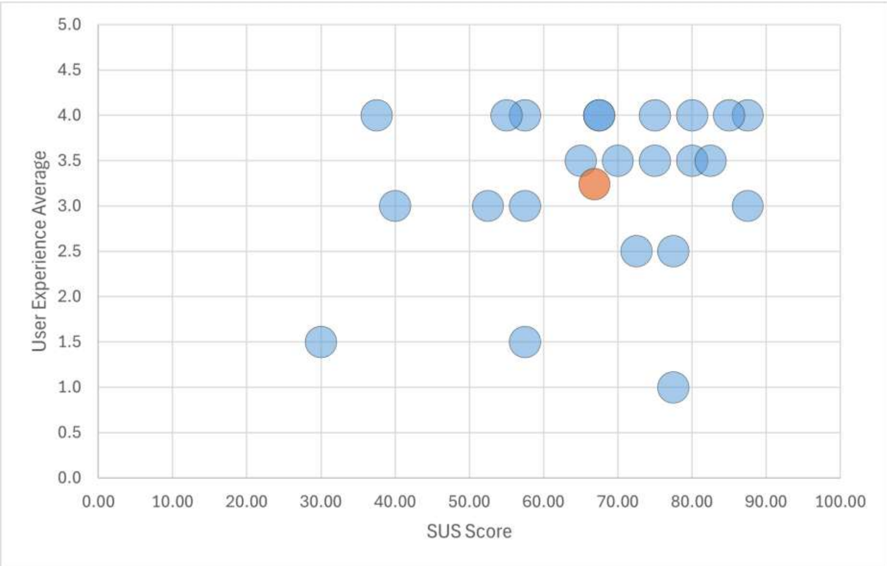


Figure 7.14: Survey Evaluation - Correlation between SUS Score and User Experience

Question	++	+	/	–	––
I think that I would like to use the system frequently	0	6	4	10	3
I found the system unnecessarily complex	0	1	3	7	12
I thought the system was easy to use	6	9	5	2	1
I think that I would need the support of a technical person to be able to use this system	0	3	2	8	10
I found the various functions in this system were well integrated	2	16	4	1	0
I thought there was too much inconsistency in this system	8	4	4	7	0
I would imagine that most people would learn to use this system very quickly	6	14	1	1	1
I found the system very cumbersome to use	1	4	5	3	10
I felt very confident using the system	1	8	6	6	2
I needed to learn a lot of things before I could get going with this system	0	3	1	14	5

Table 7.1: NLI Prototype – System Usability Scale

7.3.2 Discussion

This section provides an overview of the survey, summarizing participants' written feedback and interpretations of the log data.

Interpretation of Log Data

Since every participant provided logging data of their modeling session along with all captured voice recordings, these can be analyzed for patterns in NL understanding performance. The logs are structured with timestamps - useful for associating log entries with the corresponding voice recordings - and contain the transcribed query text as interpreted by the speech recognition system.

Each query was processed by the NL server to extract two core elements: the intended modeling action (intent) and the relevant information (entities). Then, the accuracy of these results was manually evaluated based on what the user actually tried to express in their command. Each query was then marked as correct or incorrect, allowing quantitative evaluation of the model's performance between users and between different command types.

For every entry, the predicted intent and the extracted entities were compared to what the user was actually trying to express. In the case of incorrect entity extraction, the specific type of error was not further categorized. It is not relevant whether the error was caused by a wrong entity key or by an incorrect or missing value. Regardless of the cause, any mismatch that would lead to incorrect behavior in the modeling environment was marked as incorrect. This binary assessment reflects whether a query would be understood correctly by the system in practice. Table 7.3 shows different errors. The first one is missing an entity containing the direction to move, the second one is extracting the wrong entities (relation type and name are wrong), and the third one does not understand the attributes' names correctly.

It is important to note that the performance of the voice-to-text transcription was not evaluated as part of this analysis, as also stated in Section 7.3.3. The assessment focused solely on whether the interpreted intent and entities would lead to the correct modeling operation, regardless of the quality or correctness of the transcription itself. If the possibly wrong transcription still resulted in a correct prediction by the NL server, the query was marked as correct. For example, the transcribed query "rename new value to H" may not reflect what the user originally intended to say, but since the predicted intent was "ChangeName" and the extracted entity was "H", this query was considered correct for both intent and entity extraction.

Intent	**Count of Intent**	**Avg. Intent Correct**	**Avg. Entity Correct**
AddAttribute	99	78.8%	65.7%
AddMethod	72	55.6%	69.4%
AddRelation	124	95.2%	72.6%
ChangeDatatype	9	22.2%	22.2%
ChangeName	53	100.0%	88.7%
ChangeVisibility	3	100.0%	100.0%
CreateContainer	231	92.2%	85.3%
Delete	53	94.4%	62.3%
Focus	118	85.6%	76.3%
Move	160	100.0%	45.6%
Undo	10	90.0%	90.0%
Total	**932**	**88.7%**	**70.7%**

Table 7.2: Evaluation Modeling Use Case Log Data

Across all participants, intent recognition achieved an average accuracy of 88.73%, while entity extraction was slightly lower, with an average accuracy of 70.71%. Several participants achieved perfect scores for intent detection, but entity extraction showed more variation, as shown in Figure 7.15. For example, a user achieved 96% accuracy for intent, but only 57% for entity extraction. This pattern appeared in multiple cases, suggesting that identifying what the user wants to do is generally more reliable than

User	Transcription	Interpretation of Intent	Comment
A	Select class stack	Select class duck	Accent
B	Move it to the left of the glass anymore	Move it to the left of class animal	Accent
C	Add a public method for WAC 2	Add a public method called quack to duck	Accent, Quiet and bad Microphone
D	Rename it to any matter	Rename it to animal	Accent, Reverberation and Echo Effects
E	Place it to the left of class torque	Place it to the left of class dog	Accent

Table 7.3: Examples of Wrong Transcriptions

correctly capturing all details of their command. This highlights that entity extraction is more complex and prone to error compared to intent recognition.

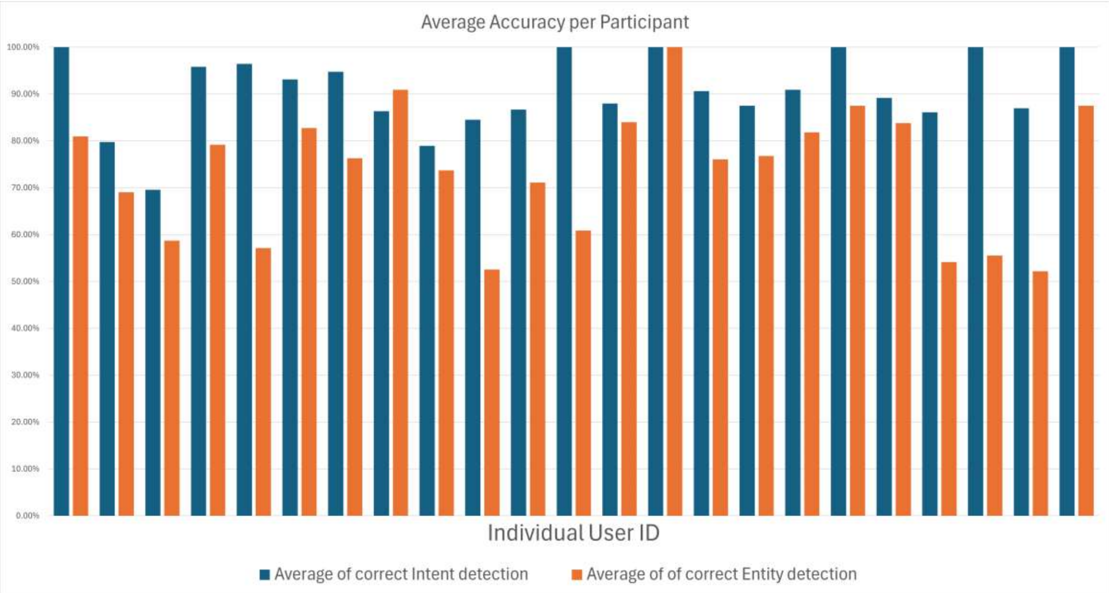


Figure 7.15: Survey Evaluation - Logs Accuracy per User

A breakdown by intent (Table 7.2) highlights differences between command types. Intents like `ChangeName` and `Move` performed well, with intent accuracy reaching 100%. `Delete`, `CreateContainer` and `AddRelation` also reached an intent accuracy over 90%. In contrast, `ChangeDatatype` showed much lower performance, with only 22.22% accuracy for intent and 22.22% for entity recognition. `AddMethod` also proved challenging, with intent accuracy at 55.56% and entity accuracy at 69.44%. In total, the system processed 932 queries. Commands that occurred more frequently, such as `CreateContainer` (231 times) or

`AddRelation` (124 times), generally performed more consistently, though performance still varied depending on phrasing and recognition quality.

Note that some intents have high accuracy but were used very rarely by participants, as shown in Table 7.2. Therefore, these results are not used in any interpretation.

Accents and Dialects

Accents and dialects had a strong influence on voice transcription. Table 7.3 shows example results of wrong transcriptions with their interpretation of what was intended, along with an evaluation comment when listening to the audio recording. Note that every example is from a different participant. Recurring issues were poor pronunciation and poor microphone quality. Moreover, the transcription model[2] did not put any emphasis on the participants being nonnative English speakers, which might explain why these errors were observed across different users.

Interpretation of Written Feedback

Participants provided detailed written feedback that helped identify recurring issues and areas for improvement. One of the most frequently mentioned problems involved moving elements within the diagram. Users found the current method unintuitive, as it only allows positioning elements relative to others. An example of user interaction is shown in Listings 7.9 and 7.10. This often led to frustration, especially when users attempted commands like "move below" without success. Even when phrased correctly, results were sometimes unpredictable. Several participants suggested allowing more precise control, such as pixel-based positioning or supporting commands like "switch positions".

```
Move the data type integer to the right
Move it right off datatype string
Move it left
Select data type integer
Move it right
Select data type integer integer
Move it right
Move it right of datatype string
```

Listing 7.9: Survey logs 1 - Position element

```
Move person to the center of the diagram
Move class person to the right
Move class person to the right of animal
Move class person under the class animal
Move class person under the class lock
```

Listing 7.10: Survey logs 2 - Position element

Another widely reported limitation was the audio recording workflow. Having to start each recording manually and being limited to five seconds per command affected the flow

[2] https://huggingface.co/openai/whisper-base.en, last accessed 2025-05-26

of interaction. Multiple users ran into problems with longer commands being cut off, forcing them to repeat inputs or switch to text input. A preferred alternative would be manual control over starting and stopping the recording or automatic detection of when a user finishes speaking. Some also suggested supporting multiple commands in a single recording, separated by keywords like `STOP`.

The speech recognition quality was another area of concern. Words like "animal", "dog", "duck", or "age" were often misheard, causing incorrect or unexpected system responses (see Listing 7.11). In some cases, users attempted to work around this by spelling out words, which only worked if they were aware that this was supported. Clearer documentation could help here. Participants also experienced issues with commands being ignored due to small variations in phrasing. For instance, some commands only worked when including optional words such as "a" ("create a class" versus "create class") or when spoken in a specific structure. Multiple users noted that the system required exact command phrasing and recommended that it become more flexible or context-aware in future versions.

```
Create an abstract class called Anywhere
Rename it to any male
Rename it to any man
Rename it to anywhere
Rename it to any metal
Rename it to any matter
Rename is to any mile
Rename it to any male
...
Rename it to animal
```

Listing 7.11: Survey logs 3 - Incorrect transcription example of the word "Animal"

Setup difficulties also affected the experience. Some participants experienced difficulties when installing and configuring `Sox`, which is required for voice recording, while others encountered Java-related errors. In several cases, the system failed to record audio entirely or returned backend errors. Although many resolved these issues eventually, they reported spending considerable time debugging. Users suggested bundling required services or providing clearer guidance during setup to reduce entry barriers.

The lack of feedback and confirmation during use was also brought up. Some users wanted to see the recognized text before it was processed into a command so that they could correct mistakes. Others recommended a short delay between recognition and execution or the ability to cancel and retry commands if needed.

Finally, many users wanted more examples and command references directly within the interface. While a `README` with a detailed explanation existed, several mentioned they did not realize it contained the information they needed or found it inconvenient to switch between the editor and external documentation. Participants suggested embedding command examples, syntax hints, or a quick-reference guide directly into the tool.

On the positive side, many participants saw strong potential in the concept. Some noted that the text input worked surprisingly well and that the system was able to handle fairly complex commands. One participant mentioned they were impressed by how well the system processed longer, detailed instructions, even though it sometimes took multiple attempts. Others highlighted that the tool could be especially useful in accessibility contexts, and that combining speech control with intelligent assistance (such as follow-up prompts or auto-completion) could make the interface more intuitive. Several users described the idea as interesting and promising, especially once the handling of phrasing and recognition improves with time.

To answer (**RQ3**), all stages in the processing of NL inputs - from recording and transcribing to extracting the intent and entities - must be considered. As the survey evaluation in Section 7.3 shows, the accuracy differs between intents, as well as between different users. Reasons are the quality of the audio recording and the use of accents and dialects, which are more present for some speakers than for others.

Another challenge for users was the fixed audio recording time, as more complex queries require a longer timeframe to formulate correctly. These insights show that a more user-friendly interface can be designed by providing a more intuitive and natural way for audio recording, as well as improving the speech-to-text framework to account for the challenges observed by the user group. In addition, the training data for the language models can be improved with the actual user queries from the survey, providing better results for a more diverse set of wordings within queries.

7.3.3 Limitations

While the developed NLI shows promising results, there are certain limitations of the current prototype and its evaluation.

User Selection Bias

The user survey was conducted through personal invitations of participants, which means that the selection is not completely representative of the general population of modeling tool users. Additionally, the user group was inhomogeneous, comprising primarily participants from academic backgrounds with some additions from our industry partner. This could lead to biased feedback that does not accurately represent real-world experiences.

Exclusion of Accessibility Considerations

Furthermore, the survey did not include individuals with accessibility issues. Even though the tool is in the early development stage, the insights of users who are reliant on speech input on the one hand, but also experienced in the use of voice interfaces on the other, would be a significant improvement.

Uncontrolled Evaluation Variables

Lastly, evaluation did not account for variables such as microphone quality or noisy environments. These uncontrolled factors can significantly impact the quality of the recordings and, therefore, the user experience and the reliability of the feedback.

Evaluation Realism

The experiment was conducted using a small UML model with only a few classes. This does limit the generalizability of the findings to realistic scenarios, as the survey did not cover all UML modeling elements. Thus, the evaluation should be considered in this simplified setting, and it remains uncertain how well the NLI would handle larger, real-world models with more complex demands for good navigation and element placement.

To summarize, this chapter provides an overview of the supported operations, including examples in bigUML and the required commands for their creation. The evaluation of the user survey showed the strengths and weaknesses of the developed prototype with a small real-world example. The user survey feedback was evaluated in a quantitative way by providing insights on the user responses and by calculating a SUS score, and qualitatively by reviewing the user logs manually and checking for patterns or other issues which might not be explicitly mentioned in the written feedback.

CHAPTER 8

Conclusion

In this chapter, the thesis will be concluded by summarizing the key findings. Finally, future work will be discussed.

8.1 Conclusions and Findings

The main goal of this thesis was to develop a prototype that extends an existing UML editor with a Natural Language Interface. A key requirement for this solution was to be modular and not interfere with any existing functionality, allowing users to both use the existing interface, as well as the newly added NL features. Each component of the developed prototype is summarized in the next paragraphs. A requirement for the NLI was to allow interacting with the editor with minimal to no use of traditional input methods like the mouse and keyboard.

The NL Server was designed to be a stand-alone component, exposing a REST API for easy integration in existing editors. During the development phase, the first step was to gather different user queries by conducting a survey. The survey results were then used to train two language models: One for intent recognition and one for entity extraction. Both models are based on a pre-trained BERT model. The NL Server included a testing framework to validate the performance of the trained models against actual user queries.

The existing UML editor bigUML was then extended to interact with the NL Server via the developed REST API. A new UI component was added to the tool, which allowed users to input their queries by either recording their voice or by providing the query as text. The input was processed entirely in the NL Server, keeping the required changes within the editor as little as possible. Once the user query was interpreted and the results were sent back to the editor, the corresponding actions were executed immediately.

The prototype was then evaluated by a user survey. Users were tasked with modeling a predefined UML class diagram using only the new interface. This survey highlighted the strengths and weaknesses of the prototype and showed possibilities for future improvements, which will be discussed in the next section.

More than half the participants of the survey described in Section 7.3 succeeded in modeling the UML class diagram using the NLI, and many of the remaining participants nearly completed the task, creating large parts of the model with the NLI. It can be concluded that the tool has the potential to serve as an effective alternative input source to the mouse and keyboard of modeling editors. Despite that, it should be considered that there are still over 40% of users who could not completely solve the task.

Given the early development stage of the bigUML NLI, it was expected that the usability would point out some limitations. However, the evaluation of the SUS (see Section 7.3.1) shows that users found the system easy to use, the functionalities well integrated, and that it would be easy to learn for most people.

Across the 932 user-defined NL queries, the approach was able to correctly identify 88.7% of the modeling intents and 70.7% of the entities. To answer (**RQ4**), it is therefore concluded that the NL server is capable of handling speech input correctly for many user queries in the evaluation case. It can also be observed that the accuracy changes between participants.

8.2 Future Work

Each component of the system, from speech recognition and interpretation to integration in bigUML, offers opportunities for further development. The current voice-to-text component relies on the Whisper model, which generally performs well. However, its behavior has not been closely reviewed within the specific context of UML modeling. Since speech recognition errors were one of the most common causes of incorrect commands, it would be useful to test whether domain-specific tuning, vocabulary adjustments, or alternative models can reduce recognition mistakes, especially for technical terms and similar-sounding words.

The recording workflow was another frequent point of criticism. Currently, users are limited to recordings of exactly five seconds in length. The survey highlighted that many participants found this too restrictive, particularly when trying to give more detailed or carefully phrased commands. A more flexible solution would allow users to start and stop recordings manually. Alternatively, the system could stop recording automatically when a brief pause in speech is detected. Some users also suggested speaking multiple commands in one go, separated by a keyword like stop. These approaches would make longer interactions feel smoother and not require starting every recording manually.

The language models for intent recognition and entity extraction can also be improved. While initial training data came from a smaller set of example queries, the second survey evaluating the prototype produced a more diverse and extensive dataset. This new data could help train better models, potentially improving both robustness and generalization. Some command types already reach high accuracy, but especially those involving less common actions or more complex phrasing remain error-prone. Improving the models may also allow the system to better handle variations in how users phrase commands. Since the data from the evaluation survey contains the raw audio recordings as well as the transcribed result, this data can further be used to improve the voice-to-text processing.

Several users pointed out that the focus mechanism in the prototype is inconvenient. Currently, some commands require an element to be focused, and each interaction with an element sets the focus to this particular element. This mechanism can be replaced by a more intuitive solution. Ideally, the system should infer what the user intends to focus on based on the spoken input. Implementing this would require coordination between intent recognition, context tracking, and the graphical editor. Automating focus would reduce interruptions and make interaction more natural.

Feedback also highlighted limitations in the editor integration itself. Some commands were not supported because the bigUML client does not expose the required functionality. Extending the editor's command set would enable more interactions and reduce the number of failed operations. In parallel, the UI could be made more tolerant by accepting partial or ambiguous input and asking clarifying questions rather than rejecting commands that do not meet strict formatting.

In the long run, the current intent-based design could evolve toward a dialogue-driven system. Instead of relying on short, isolated commands, users could issue more complex instructions and receive interactive feedback. For example, the system might respond with suggestions or ask for clarification if a query is incomplete. This would move the interaction closer to a real conversation, allowing a smoother and more flexible modeling experience. Participants already mentioned this as a desirable direction, especially when reflecting on situations where multiple corrections were necessary to achieve the intended outcome.

Overview of Generative AI Tools Used

Writefull[1] and Grammarly Pro[2] where used for spell checking and grammar fixes. Style suggestions from Grammarly Pro were used to improve the wording of sentences.

For the thesis, ChatGPT Plus[3] was used to get a rough outline on how to structure chapters. During development, it was used to get an overview of new technologies to use.

[1] https://www.writefull.com/, last accessed 2025-06-28
[2] https://app.grammarly.com/, last accessed 2025-06-28
[3] https://chatgpt.com/, last accessed 2025-06-28

List of Figures

List of Tables

Acronyms

AI Artificial Intelligence. 12

API Application Program Interface. xiv, 3, 37–39, 41, 45, 47, 50, 55–57, 60, 65, 68, 89, 135, 136

BERT Bidirectional Encoder Representations from Transformers. 4, 36, 37, 39–42, 49–51, 57, 60, 68, 89

BIO Begin-Inside-Outside. 51, 52, 97

CSV Comma-separated values. 40, 48, 49

EMF Eclipse Modeling Framework. 9, 10

GLSP Graphical Language Server Protocol. 2, 8

GPT Generative Pre-trained Transformer. 41

GUI Graphical User Interface. 7

JSON JavaScript Object Notation. 9

LLM Large Language Model. 12

M2M Model-to-Model. 7

M2T Model-to-Text. 7

MDE Model-Driven Engineering. 7, 8

MLM Masked Language Model. 39, 40

NL Natural Language. ix, xi, 5, 13, 14, 17, 20, 32, 34, 36, 38, 43, 45–47, 49, 57, 58, 60, 62, 64–67, 81, 82, 86, 89, 90, 95, 109

NLI Natural Language Interface. ix, xi, 2–4, 8, 11, 12, 37, 43, 46, 67–71, 73, 81, 86, 87, 89, 90, 97

NLP Natural Language Processing. ix, xi, 1–3, 8, 10–13, 15, 32, 39, 41, 56

NSP Next Sentence Prediction. 40

REST Representational State Transfer. xiv, 45, 47, 56, 60, 67, 68, 89, 135, 136

SQL Structured Query Language. 7

SUS System Usability Scale. 76, 79–81, 87, 90, 95, 97

UI User Interface. 57, 58, 62–65, 67, 76, 89, 91, 95

UML Unified Modeling Language. ix, xi, 1–5, 7–11, 13–18, 20–23, 28, 29, 31, 32, 36, 40, 46, 47, 55, 56, 60, 62, 64, 69, 76, 77, 79, 87, 89, 90, 97

URI Uniform Resource Identifier. 19

VS Code Visual Studio Code. 2, 8, 12, 62

XMI XML Metadata Interchange. 9

XML Extensible Markup Language. 56

Bibliography

[ABC+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 265–283, USA, 2016. USENIX Association.

[ABCS24] Pasquale Ardimento, Mario Luca Bernardi, Marta Cimitile, and Michele Scalera. A rag-based feedback tool to augment uml class diagram learning. In *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, pages 26–30, 2024.

[ASBZ16] Chetan Arora, Mehrdad Sabetzadeh, Lionel Briand, and Frank Zimmer. Extracting domain models from natural-language requirements: approach and industrial evaluation. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pages 250–260, 2016.

[BE98] Paul Blenkhorn and D Gareth Evans. Using speech and touch to enable blind people to access schematic diagrams. *Journal of Network and Computer Applications*, 21(1):17–29, 1998.

[BIG] BIGModelingTools. bigUML Modeling Tool. `https://marketplace.visualstudio.com/items?itemName=BIGModelingTools.umldiagram`. Accessed: 02.02.2024.

[BKL09] Steven Bird, Ewan Klein, and Edward Loper. *Natural Language Processing with Python*. O'Reilly Media, Inc., 01 2009.

[BLO23] Dominik Bork, Philip Langer, and Tobias Ortmayr. A vision for flexibile glsp-based web modeling tools, 2023.

[BMS+14] Suzanne P Balik, Sean P Mealin, Matthias F Stallmann, Robert D Rodman, Michelle L Glatz, and Veronica J Sigler. Including blind people in computing through access to graphs. In *Proceedings of the 16th international ACM SIGACCESS conference on Computers & accessibility*, pages 91–98, 2014.

[Bro96] John Brooke. SUS-A quick and dirty usability scale. *Usability evaluation in industry*, 189(194), 1996.

[CTBV23] Javier Cámara, Javier Troya, Lola Burgueño, and Antonio Vallecillo. On the assessment of generative ai in modeling tasks: an experience report with chatgpt and uml. *Software and Systems Modeling*, 22(3):781–793, 2023.

[DB09] Deva Kumar Deeptimahanti and Muhammad Ali Babar. An automated tool for generating uml models from natural language requirements. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 680–682. IEEE, 2009.

[dCA21] João Fonseca de Carvalho and Vasco Amaral. Towards a modelling workbench with flexible interaction models for model editors operating through voice and gestures. In *IEEE 45th Annual Computers, Software, and Applications Conference, COMPSAC 2021, Madrid, Spain, July 12-16, 2021*, pages 1026–1031. IEEE, 2021.

[DCLT19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[DFN06] Alain Désilets, David C Fox, and Stuart Norton. Voicecode: An innovative speech interface for programming-by-voice. In *CHI'06 extended abstracts on Human factors in computing systems*, pages 239–242, 2006.

[dif] Difflib — docs.python.org. `https://docs.python.org/3/library/difflib.html`. [Accessed 05-17-2025].

[DS11] Deva Kumar Deeptimahanti and Ratna Sanyal. Semi-automatic generation of uml models from natural language requirements. In *Proceedings of the 4th India Software Engineering Conference*, pages 165–174, 2011.

[Ecl] EclipseSource. Glsp. `https://www.eclipse.org/glsp/`. Accessed: 02.02.2024.

[FAA24] Alessio Ferrari, Sallam Abualhaijal, and Chetan Arora. Model generation with llms: From requirements to uml sequence diagrams. In *2024 IEEE 32nd International Requirements Engineering Conference Workshops (REW)*, pages 291–300. IEEE, 2024.

[FR07] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *Future of Software Engineering (FOSE '07)*, pages 37–54, 2007.

[GB21] Philipp-Lorenz Glaser and Dominik Bork. The biger tool - hybrid textual and graphical modeling of entity relationships in vs code. In *2021 IEEE 25th International Enterprise Distributed Object Computing Workshop (EDOCW)*, pages 337–340, 2021.

[HH19] Zahra Abdulkarim Hamza and Mustafa Hammad. Generating uml use case models from software requirements using natural language processing. In *2019 8th International Conference on Modeling Simulation and Applied Optimization (ICMSAO)*, pages 1–6. IEEE, 2019.

[HMPR04] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, 2004.

[HMVLB20] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. spaCy: Industrial-strength Natural Language Processing in Python. 2020.

[Hoy18] Matthew Hoy. Alexa, siri, cortana, and more: An introduction to voice assistants. *Medical Reference Services Quarterly*, 37:81–88, 01 2018.

[huga] Auto Classes — huggingface.co. `https://huggingface.co/docs/transformers/model_doc/auto`. [Accessed 29-01-2025].

[hugb] Safetensors — huggingface.co. `https://huggingface.co/docs/safetensors/index`. [Accessed 29-01-2025].

[hugc] WordPiece tokenization - Hugging Face NLP Course — huggingface.co. `https://huggingface.co/learn/nlp-course/chapter6/6`. [Accessed 29-01-2025].

[JM25] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*. 3rd edition, 2025. Online manuscript released January 12, 2025.

[KBC+04] Alasdair King, Paul Blenkhorn, David Crombie, Sijo Dijkstra, Gareth Evans, and John Wood. Presenting uml software engineering diagrams to blind people. In *Computers Helping People with Special Needs: 9th International Conference, ICCHP 2004, Paris, France, July 7-9, 2004. Proceedings 9*, pages 522–529. Springer, 2004.

[LBS+23a] Roman Lukyanenko, Dominik Bork, Veda Storey, Jeffrey Parsons, and Oscar Pastor. Inclusive conceptual modeling: Diversity, equity, involvement, and belonging in conceptual modeling. 11 2023.

[LBS+23b] Roman Lukyanenko, Dominik Bork, Veda C. Storey, Jeffrey Parsons, and Oscar Pastor. Inclusive conceptual modeling: Diversity, equity, involvement, and belonging in conceptual modeling (short paper). In Claudenir M. Fonseca, José Borbinha, Giancarlo Guizzardi, David Aveiro, Sotirios Liaskos, C. Maria Keet, Estefanía Serral, Fernanda Baião, João Araújo, Tiago Prince Sales, Miguel Mira da Silva, Sergio de Cesare, H. Sofia Pinto, Ladjel Bellatreche, and Simon Hacks, editors, *Companion Proceedings of the 42nd International Conference on Conceptual Modeling: ER Forum, 7th SCME, Project Exhibitions, Posters and Demos, and Doctoral Consortium co-located with ER 2023, Lisbon, Portugal, November 06-09, 2023*, volume 3618 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2023.

[LCA18] João Lopes, João Cambeiro, and Vasco Amaral. Modelbyvoice-towards a general purpose model editor for blind people. In *MoDELS (Workshops)*, pages 762–769, 2018.

[LVdMvP+21] Quentin Lhoest, Albert Villanova del Moral, Patrick von Platen, Thomas Wolf, Mario Šaško, Yacine Jernite, Abhishek Thakur, Lewis Tunstall, Suraj Patil, Mariama Drame, Julien Chaumond, Julien Plu, Joe Davison, Simon Brandeis, Victor Sanh, Teven Le Scao, Kevin Canwen Xu, Nicolas Patry, Steven Liu, Angelina McMillan-Major, Philipp Schmid, Sylvain Gugger, Nathan Raw, Sylvain Lesage, Anton Lozhkov, Matthew Carrigan, Théo Matussière, Leandro von Werra, Lysandre Debut, Stas Bekman, and Clément Delangue. Datasets: A Community Library for Natural Language Processing. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 175–184. Association for Computational Linguistics, November 2021.

[MB23] Haydar Metin and Dominik Bork. Introducing biguml: A flexible open-source glsp-based web modeling tool for uml. In *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, page 40–44. IEEE Press, 2023.

[MB24] Yang Meng and Ainita Ban. Automated uml class diagram generation from textual requirements using nlp techniques. *JOIV : International Journal on Informatics Visualization*, 8:1905, 11 2024.

[McT20] Michael McTear. *Conversational AI*. Synthesis Lectures on Human Language Technologies. Springer International Publishing, Cham, Switzerland, October 2020.

[MP12] Priyanka More and Rashmi Phalnikar. Generating uml diagrams from natural language specifications. *International Journal of Applied Information Systems*, 1(8):19–23, 2012.

[MU06] GS Anandha Mala and GV Uma. Automatic construction of object oriented design models [uml diagrams] from natural language requirements specification. In *PRICAI 2006: Trends in Artificial Intelligence: 9th Pacific Rim International Conference on Artificial Intelligence Guilin, China, August 7-11, 2006 Proceedings 9*, pages 1155–1159. Springer, 2006.

[MV06] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005).

[NN11] Julius Nganji and Shawulu Nggada. Disability-aware software engineering for improved system accessibility and usability. *International Journal of Software Engineering and Its Applications*, 5:47–62, 07 2011.

[Now18] Anna Nowogrodzki. Speaking in code: how to program by voice. *Nature*, 559(7712):141–143, 2018.

[PGdL19] Sara Pérez-Soler, Esther Guerra, and Juan de Lara. Flexible modelling using conversational agents. In Loli Burgueño, Alexander Pretschner, Sebastian Voss, Michel Chaudron, Jörg Kienzle, Markus Völter, Sébastien Gérard, Mansooreh Zahedi, Erwan Bousse, Arend Rensink, Fiona Polack, Gregor Engels, and Gerti Kappel, editors, *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019*, pages 478–482. IEEE, 2019.

[PGGdL19] Sara Pérez-Soler, Mario González-Jiménez, Esther Guerra, and Juan de Lara. Towards conversational syntax for domain-specific languages using chatbots. *J. Object Technol.*, 18(2):5:1–21, 2019.

[PSB+02] Helen Petrie, Christoph Schlieder, Paul Blenkhorn, Gareth Evans, Alasdair King, Anne-Marie O'Neill, George T Ioannidis, Blaithin Gallagher, David Crombie, Rolf Mager, et al. Tedub: A system for presenting and exploring technical drawings for blind people. In *Computers Helping People with Special Needs: 8th International Conference, ICCHP 2002 Linz, Austria, July 15–20, 2002 Proceedings 8*, pages 537–539. Springer, 2002.

[pyp] SpeechRecognition — pypi.org. `https://pypi.org/project/SpeechRecognition/`. [Accessed 04-12-2024].

[RH24] Djaber Rouabhia and Ismail Hadjadj. Enhancing class diagram dynamics: A natural language approach with chatgpt, 2024.

[Sch06] D.C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006.

[SMB23] Aylin Sarioğlu, Haydar Metin, and Dominik Bork. How inclusive is conceptual modeling? a systematic review of literature and tools for disability-aware conceptual modeling. In *Conceptual Modeling: 42nd International Conference, ER 2023, Lisbon, Portugal, November 6–9, 2023, Proceedings*, page 65–83, Berlin, Heidelberg, 2023. Springer-Verlag.

[SSHK15] M. Seidl, M. Scholz, C. Huemer, and G. Kappel. *UML @ Classroom: An Introduction to Object-Oriented Modeling*. Undergraduate Topics in Computer Science. Springer International Publishing, 2015.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

[W3C] W3C. Web content accessibility guidelines. https://www.w3.org/TR/WCAG21/. Accessed: 02.02.2024.

[WDS+20] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-art natural language processing. In Qun Liu and David Schlangen, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[Weba] EclipseWeb Web. Eclipse EMF — projects.eclipse.org. `https://projects.eclipse.org/projects/modeling.emf.emf`. [Accessed 02-03-2025].

[Webb] WebAIM. The webaim million. https://webaim.org/projects/million/2024. Accessed: 14.06.2025.

[WM10] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010.

[YS22] Song Yang and Houari Sahraoui. Towards automatically extracting uml class diagrams from natural language specifications. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages* *and Systems: Companion Proceedings*, MODELS '22, page 396–403, New York, NY, USA, 2022. Association for Computing Machinery.

[ZURKMN24] Haroon Zafar, Saif Ur Rehman Khan, Atif Mashkoor, and Habib Un Nisa. MOBICAT: a model-driven engineering approach for automatic GUI code generation for android applications. *Front. Comput. Sci.*, 6, May 2024.

Appendix

The whole project is available at GitHub:

- NL Server: `https://github.com/big-thesis/Schwantler.NLIforUML`, last accessed 2025-05-28.
- Extended BIGUML editor: `https://github.com/sschwantler/bigUML`, last accessed 2025-05-28.

Text Processing Prototype

```python
import spacy
from fuzzywuzzy import fuzz
from spacy.cli.download import download

# Load English tokenizer, tagger, parser and NER
model_name = "en_core_web_sm"

action_keywords = ["create", "modify", "delete", "change", "add"]
object_keywords = ["class", "attribute", "method", "property", "association"]
direction_keywords = ["above", "below", "left", "right"]
identifier_keywords = ["name", "id", "label"]

matching_threshold = 59


def find_best_match(word, keyword_list, index):
    matches = [(keyword, fuzz.ratio(word, keyword)) for keyword in
    keyword_list]
    best_match = max(matches, key=lambda x: x[1])

    if best_match[1] >= matching_threshold:
        return (best_match[0], index)
    else:
        return None


def process_text_string_to_command(txt: str):
    if model_name not in spacy.util.get_installed_models():
```

```
        print(f"Downloading spaCy model: {model_name}")
        download(model_name)

    nlp = spacy.load(model_name)

    doc = nlp(txt)

    tokens = [token.text.lower() for token in doc]

    action_matches = []
    object_matches = []
    direction_matches = []
    identifier_matches = []
    identifier_value = []
    attribute_name = []

    for i, token in enumerate(tokens):
        action_matches.append(find_best_match(token, action_keywords, i))
        object_matches.append(find_best_match(token, object_keywords, i))
        direction_matches.append(find_best_match(token, direction_keywords, i
))
        identifier_matches.append(find_best_match(token, identifier_keywords,
 i))

    action_matches = [e for e in action_matches if e is not None]
    object_matches = [e for e in object_matches if e is not None]
    direction_matches = [e for e in direction_matches if e is not None]
    identifier_matches = [e for e in identifier_matches if e is not None]

    if len(identifier_matches) > 0:
        for match in identifier_matches:
            index = match[1] + 1
            identifier_value.append((match[0], tokens[index]))

    if len(object_matches) > 0:
        for match in object_matches:
            index = match[1] + 1
            attribute_name.append((match[0], tokens[index]))

    print(txt)
    print("Action Keywords:", action_matches)
    print("Object Keywords:", object_matches)
    print("Direction Keywords:", direction_matches)
    print("Identifier Keywords:", identifier_matches)
    print("Identifier Value:", identifier_value)
    print("Attribute Name:", attribute_name)
    print("\n")
```

Listing 8.1: Text Processing Prototype

Speech to Text Evaluation Script

```
import csv
```

```python
import glob
import io
import logging
import json
import re

import torch
import vosk
import whisper
import numpy as np
import soundfile as sf
import speech_recognition as sr

from difflib import SequenceMatcher
from time import sleep
from pathlib import Path

STORE_OUTPUT = False
RECORDER_ID = "11703514"
AUDIO_PATHS = [
    {
        "input": Path("..."),  # path to recording
        "expected": Path("..."),  # path to expected transcriptions
    }
]

logger = logging.getLogger()
logging.basicConfig(level=logging.DEBUG, format="%(message)s")

r = sr.Recognizer()


def remove_punctuation(text):
    pattern = r"[^\w\s]"  # This pattern matches anything that is not a word character or whitespace
    cleaned_text = re.sub(pattern, "", text)
    return re.sub(" +", " ", cleaned_text).strip()


def read_expected_audio_csv(
    path: Path, delimiter: str, skip_first_row: bool = True
) -> dict:
    ret_val = {}
    with open(path, newline="\n") as csvfile:
        reader = csv.reader(
            csvfile,
            delimiter=delimiter,
            quotechar='"',
        )

        if skip_first_row:
            next(reader)

```

```
        for row in reader:
            if len(row) != 2:
                raise Exception(
                    "CSV must contain 2 entries per row, one with file id and one with expected value"
                )
            ret_val[row[0]] = {}
            ret_val[row[0]]["expected"] = row[1]

    return ret_val


def to_text_vosk(audio_data: bytes):
    # Load Vosk vosk_model (downloaded from https://alphacephei.com/vosk/models)
    model_path = Path("..\\offline_models\\vosk_model")
    if model_path.exists():
        model = vosk.Model(str(model_path))
    else:
        model = vosk.Model(model_name="vosk-model-en-us-0.22-lgraph")

    # Initialize Vosk recognizer
    rec = vosk.KaldiRecognizer(model, 16000)

    # Process the entire audio data
    rec.AcceptWaveform(audio_data)

    # Finalize and print the transcript
    result_raw = rec.FinalResult()

    result = json.loads(result_raw)

    return remove_punctuation(result["text"])


def to_text_whisper(audio_data: np.ndarray | Path):
    model = whisper.load_model("base")
    result = model.transcribe(audio_data, fp16=torch.cuda.is_available(), language="en")
    return remove_punctuation(result["text"])


if __name__ == "__main__":
    for dict_audio_paths in AUDIO_PATHS:
        expected_path = None

        result = read_expected_audio_csv(
            path=dict_audio_paths["expected"], delimiter=",", skip_first_row=True
        )

        all_in_files = glob.glob(f"{dict_audio_paths['input']}\\*.wav")
        for file in all_in_files:
```

```
            file_path = Path(file)
            audio = sr.AudioFile(str(file_path))
            with audio as source:
                audio = r.record(source)

            key = file_path.stem.rsplit("_", 1)
            wav_bytes = audio.get_wav_data(convert_rate=16000)
            wav_stream = io.BytesIO(wav_bytes)
            audio_array, sampling_rate = sf.read(wav_stream)
            audio_array = audio_array.astype(np.float32)
            result[key[0]][key[1]] = {}
            result[key[0]][key[1]]["vosk"] = to_text_vosk(
                audio.get_raw_data(convert_rate=16000, convert_width=2)
            )
            result[key[0]][key[1]]["whisper"] = to_text_whisper(audio_array)
            result[key[0]][key[1]]["vosk_result"] = SequenceMatcher(
                None,
                result[key[0]]["expected"].lower(),
                result[key[0]][key[1]]["vosk"].lower(),
            ).ratio()
            result[key[0]][key[1]]["whisper_result"] = SequenceMatcher(
                None,
                result[key[0]]["expected"].lower(),
                result[key[0]][key[1]]["whisper"].lower(),
            ).ratio()
            logger.info(f"ID:            {file_path.stem}")
            logger.info(f"Expected:      {result[key[0]]['expected']}")
            logger.info(f"Vosk:          {result[key[0]][key[1]]['vosk']}")
            logger.info(f"Whisper:       {result[key[0]][key[1]]['whisper']}")
            logger.info(f"Vosk Score:    {result[key[0]][key[1]]['vosk_result']}")
            logger.info(f"Whisper Score: {result[key[0]][key[1]]['whisper_result']}")
            sleep(1)
```

Listing 8.2: Speech to Text Evaluation Script

Template Queries for Training Data

```
CREATE_CONTAINER_INTENT = "CreateContainer"
ADD_ATTRIBUTE_INTENT = "AddAttribute"
ADD_METHOD_INTENT = "AddMethod"
CHANGE_NAME_INTENT = "ChangeName"
CHANGE_VISIBILITY_INTENT = "ChangeVisibility"
CHANGE_DATATYPE_INTENT = "ChangeDatatype"
ADD_RELATION_INTENT = "AddRelation"
DELETE_INTENT = "Delete"
FOCUS_INTENT = "Focus"
MOVE_INTENT = "Move"
UNDO_INTENT = "Undo"

TEMPLATE_QUERIES = {
```

```
    CREATE_CONTAINER_INTENT: [
        "Create a {container_abstraction} {element_type} named {element_name}",
        "Define a {container_abstraction} {element_type} called {element_name}",
        "Create a new {container_abstraction} {element_type} named {element_name}",
        "Add a {container_abstraction} {element_type} named {element_name} to the diagram",
        "Make a {container_abstraction} {element_type} with the title {element_name}",
        "Define a {container_abstraction} {element_type} and name it {element_name}",
        "Create a {container_abstraction} {element_type} called {element_name}",
        "Add a new {container_abstraction} {element_type} named {element_name}",
        "Make a {container_abstraction} {element_type} and name it {element_name}",
        "Insert a {container_abstraction} {element_type} {element_name} into the current diagram",
        "I want to add a new {container_abstraction} {element_type} named {element_name} to the diagram",
        "Declare a {container_abstraction} {element_type} {element_name}",
        "I would like to add a new {container_abstraction} {element_type} named {element_name}",
        "Please insert a {container_abstraction} {element_type} called {element_name} into the diagram",
        "Could you create a {container_abstraction} {element_type} titled {element_name}?",
        "Let's add a new {container_abstraction} {element_type} named {element_name} to our diagram",
        "Please create a {container_abstraction} {element_type} and name it {element_name}",
    ],
    ADD_ATTRIBUTE_INTENT: [
        "Add a {visibility} {element_type} named {element_name} {datatype}",
        "Introduce a {visibility} {element_type} {element_name} {datatype}",
        "Define a {visibility} {element_type} {element_name} {datatype}",
        "Add {element_name} as a {visibility} {element_type} {datatype}",
        "Include {element_name} as a {visibility} {element_type} {datatype}",
        "Create the {visibility} {element_type} {element_name} {datatype}",
        "Add the {visibility} {element_type} {element_name} {datatype}",
        "Please add an {visibility} {element_type} named {element_name} {datatype}",
        "I would like to introduce a new {visibility} {element_type} called {element_name} {datatype}",
        "Can you insert a {visibility} {element_type} named {element_name} {datatype}?",
        "Include a {visibility} {element_type} {element_name} {datatype}",
        "I need you to add a {visibility} {element_type} named {element_name} {datatype}",
        "Please introduce a {visibility} {element_type} called {element_name}
```

```
 {datatype}",
        "Add the {visibility} {element_type} {element_name} {datatype}",
        "Create the {visibility} {element_type} {element_name} {datatype}",
    ],
    ADD_METHOD_INTENT: [
        "Define a {visibility} {element_type} {element_name}",
        "Add {element_name} as a {visibility} {element_type}",
        "Include the {visibility} {element_type} {element_name}",
        "Implement a {visibility} {element_type} named {element_name}",
        "Create a new {visibility} {element_type} called {element_name}",
        "Introduce a {visibility} {element_type} {element_name}",
        "Insert a {visibility} {element_type} named {element_name}",
        "Develop a {visibility} {element_type} {element_name}",
        "Construct a {visibility} {element_type} {element_name}",
        "Establish a {visibility} {element_type} {element_name}",
        "Generate a {visibility} {element_type} {element_name}",
        "Add a {visibility} {element_type} named {element_name}",
        "Please add a {visibility} {element_type} named {element_name}",
        "I would like to include a {visibility} {element_type} called {
element_name}",
        "Can you insert a {visibility} {element_type} titled {element_name}?"
,
        "Lets introduce a {visibility} {element_type} named {element_name}",
    ],
    CHANGE_NAME_INTENT: [
        "Update the name to {new_value}",
        "Rename to {new_value}",
        "Change the name to {new_value}",
        "Replace the name with {new_value}",
        "Please rename to {new_value}",
        "I would like to change the name to {new_value}",
        "Can you update the name to {new_value}?",
        "I need it renamed to {new_value}",
        "Edit the name to {new_value}",
        "Rename to {new_value}",
        "Please change the name to {new_value}",
        "I would like to rename it to {new_value}",
        "Can you update the name to {new_value}",
        "Let's change the name to {new_value}",
    ],
    CHANGE_VISIBILITY_INTENT: [
        "Change the visibility to {new_value}",
        "Revise the visibility to {new_value}",
        "Set the visibility to {new_value}",
        "Please change the visibility to {new_value}",
        "I would like to set the visibility to {new_value}",
        "Can you set the visibility {new_value}?",
    ],
    CHANGE_DATATYPE_INTENT: [
        "Alter the datatype to {new_value}",
        "Modify the datatype to {new_value}",
        "Change the type to {new_value}",
    ],
```

```
    DELETE_INTENT: [
        "Remove {element_type} {element_name}",
        "Delete {element_type} {element_name}",
        "Erase {element_type} {element_name}",
    ],
    ADD_RELATION_INTENT: [
        "Add a {relation_type} called {element_name} between {class_name_from} and {class_name_to}",
        "Between classes {class_name_from} and {class_name_to}, create a {relation_type} with the name {element_name}",
        "Establish a {relation_type} relationship with the name {element_name} from {class_name_from} to {class_name_to}",
        "Create a {relation_type} called {element_name} linking {class_name_from} with {class_name_to}",
        "Form a {relation_type} named {element_name} between {class_name_from} and {class_name_to}",
        "Connect {class_name_from} and {class_name_to} with a {relation_type} called {element_name}",
        "Set up a {relation_type} with the name {element_name} from {class_name_from} to {class_name_to}",
        "Implement a {relation_type} between {class_name_from} and {class_name_to} named {element_name}",
        "Define a {relation_type} relationship called {element_name} linking {class_name_from} to {class_name_to}",
        "Construct a {relation_type} with the name {element_name} that connects {class_name_from} to {class_name_to}",
    ],
    FOCUS_INTENT: [
        "Focus on {element_type} {element_name}",
        "Select the {element_type} {element_name}",
        "I want to set the focus to {element_type} {element_name}",
        "Please select {element_type} {element_name}",
        "Can you focus on {element_type} {element_name}?",
        "Switch focus to {element_type} {element_name}",
    ],
    MOVE_INTENT: [
        "Move it to the {direction} of the {element_type} {element_name}",
        "Place it {direction} of the {element_type} {element_name}",
        "Position it {direction} to {element_type} {element_name}",
    ],
    UNDO_INTENT: [
        "undo",
        "undo last operation",
        "undo the last operation",
        "rollback last operation",
        "rollback the last operation",
    ],
}

CONTAINER_ABSTRACTIONS = ["abstract"]
CONTAINER_TYPES = [
    "class",
    "interface",
```

```
    "data type",
    "enumeration",
    "instance",
    "package",
    "primitive type",
]

ATTRIBUTE_TYPES = ["attribute", "field", "property", "parameter"]
METHOD_TYPES = ["function", "method", "operation"]  # have the same meaning
CONTAINER_NAMES = [
    "Vehicle",
    "Employee",
    "Order",
    "Customer",
    "Mountain",
    "Smartphone",
    "Espresso",
    "Rainforest",
    "Novel",
    "Solar",
    "Symphony",
    "Volcano",
    "Airplane",
    "Satellite",
    "Icecream",
    "Skateboard",
    "Coral",
    "Vaccine",
    "Robot",
    "Bamboo",
    "Algorithm",
    "Tsunami",
    "Sculpture",
    "Windmill",
    "Aurora",
    "Telescope",
    "Origami",
    "Drone",
    "Cactus",
    "Guitar",
    "Printer",
    "River",
    "Bridge",
    "Book",
    "Chair",
    "Laptop",
    "Jacket",
    "Wallet",
    "Bicycle",
    "Clock",
    "Pen",
    "Glass",
    "Phone",
```

```
    "Spoon",
    "Table",
    "Bottle",
    "Shirt",
    "Shoe",
    "Hat",
    "Sock",
    "Pillow",
    "Lamp",
    "Fork",
    "Key",
]
VISIBILITIES = ["public", "private", "protected"]
ATTRIBUTE_NAMES = [
    "id",
    "name",
    "email",
    "address",
    "Size",
    "Age",
    "Rating",
    "Color",
    "Quantity",
    "Price",
    "Weight",
    "Height",
    "Length",
    "Width",
    "Depth",
    "Material",
    "Condition",
    "Texture",
    "Taste",
    "Smell",
    "Sound",
    "Brightness",
    "Hardness",
    "Temperature",
    "Speed",
    "Capacity",
    "Durability",
    "Flexibility",
    "Efficiency",
    "Accuracy",
    "Complexity",
    "Rarity",
    "Popularity",
    "Reliability",
    "Brand",
    "Origin",
    "Warranty",
    "Power",
    "Voltage",
```

```
    "Energy",
    "Frequency",
    "Bandwidth",
    "Resolution",
    "Contrast",
    "Saturation",
    "Latency",
    "Throughput",
    "Coverage",
    "Range",
    "Connectivity",
    "Compatibility",
    "Security",
    "Portability",
    "Functionality",
]
ACTION_NAMES = [
    "calculate total",
    "update profile",
    "save changes",
    "delete account",
    "fetch data",
    "validate user",
    "refresh page",
    "render view",
    "get details",
    "set preferences",
    "clear cache",
    "encrypt data",
    "decrypt data",
    "submit form",
    "load resources",
    "initiate connection",
    "disconnect",
    "archive records",
    "generate report",
    "log error",
    "send notification",
    "parse input",
    "format output",
    "schedule task",
    "cancel operation",
]
RELATION_TYPES = [
    "abstraction",
    "aggregation",
    "strong aggregation",
    "association",
    "composition",
    "dependency",
    "generalisation",
    "usage",
]
```

```
DATATYPES = [
    "Integer",
    "double",
    "float",
    "boolean",
    "Character",
    "byte",
    "short",
    "long",
    "Date",
    "String",
    "List",
    "Map",
    "Set",
]
DIRECTIONS = ["north", "west", "east", "south", "top", "left", "bottom", "
    right"]
```

Listing 8.3: Template Queries used for Training Data

Script for Generating Training & Testing Queries

```
import csv
import random
from pathlib import Path

import pandas as pd

from src.nlp.variables import (
    ACTION_NAMES,
    ADD_ATTRIBUTE_INTENT,
    ADD_METHOD_INTENT,
    ADD_RELATION_INTENT,
    ATTRIBUTE_NAMES,
    ATTRIBUTE_TYPES,
    CHANGE_DATATYPE_INTENT,
    CHANGE_NAME_INTENT,
    CHANGE_VISIBILITY_INTENT,
    CONTAINER_ABSTRACTIONS,
    CONTAINER_NAMES,
    CONTAINER_TYPES,
    CREATE_CONTAINER_INTENT,
    DATATYPES,
    DELETE_INTENT,
    DIRECTIONS,
    FOCUS_INTENT,
    METHOD_TYPES,
    MOVE_INTENT,
    RELATION_TYPES,
    TEMPLATE_QUERIES,
    UNDO_INTENT,
    VISIBILITIES,
)
```

```python
from src.utils.utils import decision, remove_punctuation

NUMBER_QUERIES_PER_INTENT = 5000  # Adjust as needed
INTENTS = TEMPLATE_QUERIES.keys()
TRAINING_FILE_PATH = "..\\..\\training_data\\generated_queries\\nlp_training_data.csv"
TEST_FILE_PATH = "..\\..\\training_data\\generated_queries\\nlp_test_data.csv"


def generate_create_container_sentence():
    template = random.choice(TEMPLATE_QUERIES[CREATE_CONTAINER_INTENT])
    container_abstraction = (
        random.choice(CONTAINER_ABSTRACTIONS) if decision(0.2) else ""
    )
    element_type = random.choice(CONTAINER_TYPES)
    element_name = random.choice(CONTAINER_NAMES)

    sentence = template.format(
        container_abstraction=container_abstraction,
        element_type=element_type,
        element_name=element_name,
    )

    entities = {
        "container_abstraction": container_abstraction,
        "element_type": element_type,
        "element_name": element_name,
    }

    return {"text": sentence, "intent": CREATE_CONTAINER_INTENT, "entities": entities}


def generate_add_attribute_sentence():
    template = random.choice(TEMPLATE_QUERIES[ADD_ATTRIBUTE_INTENT])

    datatype = random.choice(DATATYPES)
    element_name = random.choice(ATTRIBUTE_NAMES)
    visibility = random.choice(VISIBILITIES) if decision(0.5) else ""
    element_type = random.choice(ATTRIBUTE_TYPES)

    datatype_prefix = random.choice(
        ["of type ", "with datatype ", " with the type ", " with the datatype "]
    )

    datatype_str = datatype_prefix + datatype if decision(0.7) else ""

    sentence = template.format(
        datatype=datatype_str,
        element_name=element_name,
        visibility=visibility,
```

```
        element_type=element_type,
    )

    entities = {
        "datatype": datatype if datatype_str != "" else "",
        "element_type": element_type,
        "element_name": element_name,
        "visibility": visibility,
    }

    return {
        "text": sentence,
        "intent": ADD_ATTRIBUTE_INTENT,
        "entities": entities,
    }


def generate_add_method_sentence():
    template = random.choice(TEMPLATE_QUERIES[ADD_METHOD_INTENT])

    element_name = random.choice(ACTION_NAMES)
    visibility = random.choice(VISIBILITIES) if decision(0.5) else ""
    element_type = random.choice(METHOD_TYPES)

    sentence = template.format(
        element_name=element_name,
        visibility=visibility,
        element_type=element_type,
    )

    entities = {
        "element_name": element_name,
        "visibility": visibility,
        "element_type": element_type,
    }

    return {
        "text": sentence,
        "intent": ADD_METHOD_INTENT,
        "entities": entities,
    }


def generate_update_value_sentence(intent, target):
    template = random.choice(TEMPLATE_QUERIES[intent])

    new_value = random.choice(target)

    sentence = template.format(new_value=new_value)

    entities = {"new_value": new_value}

    return {"text": sentence, "intent": intent, "entities": entities}
```

```python


def generate_add_relation_sentence():
    template = random.choice(TEMPLATE_QUERIES[ADD_RELATION_INTENT])

    class_name_from = random.choice(CONTAINER_NAMES)
    class_name_to = random.choice(CONTAINER_NAMES)
    relation_type = random.choice(RELATION_TYPES)
    element_name = random.choice(
        ATTRIBUTE_NAMES
    )  # todo check if attribute names represent actual values

    sentence = template.format(
        class_name_from=class_name_from,
        class_name_to=class_name_to,
        relation_type=relation_type,
        element_name=element_name,
    )

    entities = {
        "class_name_from": class_name_from,
        "class_name_to": class_name_to,
        "relation_type": relation_type,
        "element_name": element_name,
    }

    return {"text": sentence, "intent": ADD_RELATION_INTENT, "entities": entities}


def generate_focus_or_delete_sentence(intent, element_types, target):
    template = random.choice(TEMPLATE_QUERIES[intent])

    element_name = random.choice(target)
    element_type = random.choice(element_types) if decision(0.5) else ""

    sentence = template.format(
        element_type=element_type,
        element_name=element_name,
    )

    entities = {
        "element_type": element_type,
        "element_name": element_name,
    }

    return {"text": sentence, "intent": intent, "entities": entities}


def generate_move_sentence():
    template = random.choice(TEMPLATE_QUERIES[MOVE_INTENT])

    direction = random.choice(DIRECTIONS)
```

```
    element_type = "class"
    element_name = random.choice(CONTAINER_NAMES)

    sentence = template.format(
        direction=direction, element_type=element_type, element_name=
element_name
    )

    entities = {
        "direction": direction,
        "element_type": element_type,
        "element_name": element_name,
    }

    return {"text": sentence, "intent": MOVE_INTENT, "entities": entities}


def generate_undo_sentence():
    sentence = random.choice(TEMPLATE_QUERIES[UNDO_INTENT])

    return {"text": sentence, "intent": UNDO_INTENT, "entities": {}}


def generate_sentence(intent) -> list:
    retVal = []
    if intent == CREATE_CONTAINER_INTENT:
        retVal.append(generate_create_container_sentence())
    elif intent == ADD_ATTRIBUTE_INTENT:
        retVal.append(generate_add_attribute_sentence())
    elif intent == ADD_METHOD_INTENT:
        retVal.append(generate_add_method_sentence())
    elif intent == CHANGE_NAME_INTENT:
        retVal.append(
            generate_update_value_sentence(CHANGE_NAME_INTENT,
CONTAINER_NAMES)
        )
        retVal.append(generate_update_value_sentence(CHANGE_NAME_INTENT,
ACTION_NAMES))
        retVal.append(
            generate_update_value_sentence(CHANGE_NAME_INTENT,
ATTRIBUTE_NAMES)
        )
    elif intent == CHANGE_VISIBILITY_INTENT:
        retVal.append(
            generate_update_value_sentence(CHANGE_VISIBILITY_INTENT,
VISIBILITIES)
        )
    elif intent == CHANGE_DATATYPE_INTENT:
        retVal.append(generate_update_value_sentence(CHANGE_DATATYPE_INTENT,
DATATYPES))
    elif intent == DELETE_INTENT:
        retVal.append(
            generate_focus_or_delete_sentence(
```

```
                DELETE_INTENT, CONTAINER_TYPES, CONTAINER_NAMES
            )
        )
        retVal.append(
            generate_focus_or_delete_sentence(
                DELETE_INTENT, ATTRIBUTE_TYPES, ATTRIBUTE_NAMES
            )
        )
        retVal.append(
            generate_focus_or_delete_sentence(DELETE_INTENT, METHOD_TYPES, ACTION_NAMES)
        )
    elif intent == ADD_RELATION_INTENT:
        retVal.append(generate_add_relation_sentence())
    elif intent == FOCUS_INTENT:
        retVal.append(
            generate_focus_or_delete_sentence(
                FOCUS_INTENT, CONTAINER_TYPES, CONTAINER_NAMES
            )
        )
        retVal.append(
            generate_focus_or_delete_sentence(
                FOCUS_INTENT, ATTRIBUTE_TYPES, ATTRIBUTE_NAMES
            )
        )
        retVal.append(
            generate_focus_or_delete_sentence(FOCUS_INTENT, METHOD_TYPES, ACTION_NAMES)
        )
    elif intent == MOVE_INTENT:
        retVal.append(generate_move_sentence())
    elif intent == UNDO_INTENT:
        retVal.append(generate_undo_sentence())
    else:
        raise NotImplementedError()
    for s in retVal:
        s["text"] = remove_punctuation(s["text"])

    return retVal


if __name__ == "__main__":
    training_data = []
    test_data = []

    for _ in range(NUMBER_QUERIES_PER_INTENT):
        for curr_intent in INTENTS:
            training_data.extend(generate_sentence(curr_intent))
            test_data.extend(generate_sentence(curr_intent))

    df_training = pd.DataFrame(training_data)
    df_test = pd.DataFrame(test_data)

```

```
    Path(TRAINING_FILE_PATH).parent.mkdir(parents=True, exist_ok=True)

    df_training.to_csv(
        TRAINING_FILE_PATH,
        index=False,
        quoting=csv.QUOTE_NONNUMERIC,
    )
    df_test.to_csv(
        TEST_FILE_PATH,
        index=False,
        quoting=csv.QUOTE_NONNUMERIC,
    )

    print(f"Generated data at: \n\t {TRAINING_FILE_PATH} \n\t {TEST_FILE_PATH
    }")
```

Listing 8.4: Script for Generating Training & Testing Queries

Training Script for Intent Recognition

```
from pathlib import Path

from datasets import load_dataset
from transformers import (
    AutoConfig,
    AutoModelForSequenceClassification,
    AutoTokenizer,
    Trainer,
    TrainingArguments,
)

from src.nlp.variables import TEMPLATE_QUERIES

MODEL_ID = "uml_ir"
LOCAL_MODEL_PATH = "" # path to model
TRAINING_DATA_PATH = "" # path to training data
TESTING_DATA_PATH = "" # path to test data
PUSH_MODEL_TO_HUB = False
TRAIN_EXISTING_FROM_HUB = False


HUGGINGFACE_USERNAME = "" # hugging face username to directly push results


def transform_labels(example):
    example["labels"] = label2id[example["intent"]]
    return example


def tokenize_function(examples):
    return tokenizer(
        examples["text"], padding="max_length", truncation=True, max_length
    =128
```

```
    )


if __name__ == "__main__":

    dataset = load_dataset(
        "csv",
        data_files={
            "train": TRAINING_DATA_PATH,
            "test": TESTING_DATA_PATH,
        },
    )

    label_names = sorted(TEMPLATE_QUERIES.keys())
    num_labels = len(label_names)

    model_url = "bert-base-uncased"
    id2label = {str(i): label for i, label in enumerate(label_names)}
    label2id = {label: i for i, label in enumerate(label_names)}

    config = AutoConfig.from_pretrained(
        model_url, num_labels=num_labels, id2label=id2label, label2id=label2id
    )

    dataset = dataset.map(transform_labels)
    tokenizer = AutoTokenizer.from_pretrained(model_url)
    model = AutoModelForSequenceClassification.from_pretrained(model_url, config=config)

    tokenized_datasets = dataset.map(tokenize_function, batched=True)

    training_args = TrainingArguments(
        output_dir=LOCAL_MODEL_PATH,
        eval_strategy="epoch",
        learning_rate=2e-5,
        per_device_train_batch_size=64,
        num_train_epochs=3,
        weight_decay=0.01,
        push_to_hub=PUSH_MODEL_TO_HUB,
        report_to="none",
        fp16=True,
        save_strategy="epoch",
        save_total_limit=2,
    )

    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=tokenized_datasets["train"],
        eval_dataset=tokenized_datasets["test"],
        tokenizer=tokenizer,
    )
```

```
    trainer.train()

    model.save_pretrained(LOCAL_MODEL_PATH)
    tokenizer.save_pretrained(LOCAL_MODEL_PATH)

    if PUSH_MODEL_TO_HUB:
        trainer.push_to_hub()
```

Listing 8.5: Training Script for Intent Recognition

Training Script for Entity Extraction

```
import pandas as pd

from ast import literal_eval
from datasets import Dataset
from transformers import (
    AutoConfig,
    AutoModelForTokenClassification,
    AutoTokenizer,
    Trainer,
    TrainingArguments,
)


MODEL_URL = "bert-base-uncased"
MODEL_ID = "uml_ee"
LOCAL_MODEL_PATH = "" # path to model
TRAINING_DATA_PATH = "" # path to training data
TESTING_DATA_PATH = "" # path to test data
HUGGINGFACE_USERNAME = "" # hugging face username to directly push results
PUSH_MODEL_TO_HUB = False
TRAIN_EXISTING_FROM_HUB = False


def align_tokens_and_entities(text, entity_annotations, tokenizer, labels):
    tokens = tokenizer.tokenize(text)
    token_labels = ["O"] * len(tokens)

    for entity, annotation in entity_annotations.items():
        if not annotation:
            continue

        start_char = text.find(annotation)

        if start_char != -1:
            entity_tokens = tokenizer.tokenize(annotation)
            start_token = None

            for i in range(len(tokens)):
                if tokens[i : i + len(entity_tokens)] == entity_tokens:
                    start_token = i
```

```
                    break

            if start_token is not None:
                end_token = start_token + len(entity_tokens)
                token_labels[start_token] = f"B-{entity}"

                for i in range(start_token + 1, end_token):
                    token_labels[i] = f"I-{entity}"

    for e in set(token_labels):
        if e not in labels:
            labels.append(e)
    return tokens, token_labels


def preprocess_data(path, tokenizer):
    # Load the dataset
    df = pd.read_csv(path)
    df["entities"] = df["entities"].apply(literal_eval)

    tokenized_texts = []
    tokenized_labels = []
    labels_list = []

    for _, row in df.iterrows():
        text = row["text"]
        entities = row["entities"]
        tokens, labels = align_tokens_and_entities(
            text, entities, tokenizer, labels_list
        )

        tokenized_texts.append(tokens)
        tokenized_labels.append(labels)

    return tokenized_texts, tokenized_labels, labels_list


def encode_and_align_labels(examples, tokenizer, label2id):
    tokenized_inputs = tokenizer(
        examples["texts"],
        truncation=True,
        padding="max_length",
        is_split_into_words=True,
    )
    labels = []

    for i, label_seq in enumerate(examples["labels"]):
        label_ids = [label2id[lab] for lab in label_seq]
        input_len = len(tokenized_inputs["input_ids"][i])
        label_ids += [label2id["O"]] * (input_len - len(label_ids))
        labels.append(label_ids)

    tokenized_inputs["labels"] = labels
```

```python
    return tokenized_inputs


if __name__ == "__main__":

    tokenizer = AutoTokenizer.from_pretrained(MODEL_URL)

    tokenized_texts_train, tokenized_labels_train, labels_train =
    preprocess_data(
        TRAINING_DATA_PATH, tokenizer
    )
    tokenized_texts_test, tokenized_labels_test, labels_test =
    preprocess_data(
        TESTING_DATA_PATH, tokenizer
    )

    label_names = set(labels_train + labels_test)
    num_labels = len(label_names)

    train_data = {"texts": tokenized_texts_train, "labels":
    tokenized_labels_train}
    train_df = pd.DataFrame(train_data)
    train_dataset = Dataset.from_pandas(train_df)

    test_data = {"texts": tokenized_texts_test, "labels":
    tokenized_labels_test}
    test_df = pd.DataFrame(test_data)
    test_dataset = Dataset.from_pandas(test_df)

    id2label = {str(i): label for i, label in enumerate(label_names)}
    label2id = {label: i for i, label in enumerate(label_names)}

    config = AutoConfig.from_pretrained(
        MODEL_URL, num_labels=num_labels, id2label=id2label, label2id=
    label2id
    )

    train_dataset = train_dataset.map(
        lambda examples: encode_and_align_labels(examples, tokenizer,
    label2id),
        batched=True,
        remove_columns=["texts", "labels"],
    )
    test_dataset = test_dataset.map(
        lambda examples: encode_and_align_labels(examples, tokenizer,
    label2id),
        batched=True,
        remove_columns=["texts", "labels"],
    )

    model = AutoModelForTokenClassification.from_pretrained(MODEL_URL, config
    =config)

```

```
    training_args = TrainingArguments(
        output_dir=LOCAL_MODEL_PATH,
        num_train_epochs=3,
        per_device_train_batch_size=64,
        learning_rate=3e-5,
        warmup_steps=500,
        eval_strategy="epoch",
        logging_dir="./logs",
        push_to_hub=False,
        report_to="none",
        save_strategy="epoch",
        fp16=True,
        logging_steps=100,
        save_total_limit=2,
    )

    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=train_dataset,
        eval_dataset=test_dataset,
        tokenizer=tokenizer,
    )

    trainer.train()

    model.save_pretrained(LOCAL_MODEL_PATH)
    tokenizer.save_pretrained(LOCAL_MODEL_PATH)

    if PUSH_MODEL_TO_HUB:
        trainer.push_to_hub(repo_name=MODEL_ID)
```

Listing 8.6: Training Script for Entity Extraction

NlpClassifier

```
import logging
from pathlib import Path

import torch
from transformers import (
    AutoModelForSequenceClassification,
    AutoModelForTokenClassification,
    AutoTokenizer,
)

logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)


class NlpClassifier:
    ir_model = None
    ir_tokenizer = None
```

```
    ir_config = None

    ee_model = None
    ee_tokenizer = None
    ee_config = None

    def __init__(self, ir_model_path=None, ee_model_path=None):
        if ir_model_path and Path(ir_model_path).is_dir():
            self.ir_model = AutoModelForSequenceClassification.from_pretrained(
                ir_model_path
            )
            self.ir_tokenizer = AutoTokenizer.from_pretrained(ir_model_path)
            self.ir_config = self.ir_model.config
        else:
            logger.warning("Intent recognition model not initialized: incorrect path")

        if ee_model_path and Path(ee_model_path).is_dir():
            self.ee_model = AutoModelForTokenClassification.from_pretrained(
                ee_model_path
            )
            self.ee_tokenizer = AutoTokenizer.from_pretrained(ee_model_path)
            self.ee_config = self.ee_model.config
        else:
            logger.warning("Entity extraction model not initialized: incorrect path")

    def predict_intent(self, text):
        if self.ir_model is None:
            logger.error("Intent recognition model not initialized")
            return

        inputs = self.ir_tokenizer(
            text, padding=True, truncation=True, max_length=128, return_tensors="pt"
        )

        with torch.no_grad():
            outputs = self.ir_model(**inputs)

        probs = torch.nn.functional.softmax(outputs.logits, dim=-1)

        predicted_label_idx = torch.argmax(probs, dim=1).item()

        predicted_label = list(self.ir_config.label2id.keys())[
            list(self.ir_config.label2id.values()).index(predicted_label_idx)
        ]

        logger.debug(f"Predicted intent: {predicted_label}")

        return predicted_label

```

```python
    def predict_entities(self, text):
        if self.ee_model is None:
            logger.error("Entity extraction model not initialized")
            return

        inputs = self.ee_tokenizer(
            text,
            return_tensors="pt",
            truncation=True,
            padding="max_length",
            max_length=128,
        )

        with torch.no_grad():
            outputs = self.ee_model(**inputs.to(self.ee_model.device))
        predictions = outputs.logits.argmax(dim=-1).squeeze().tolist()

        tokens = self.ee_tokenizer.convert_ids_to_tokens(
            inputs["input_ids"].squeeze().tolist(), skip_special_tokens=True
        )

        predicted_labels = [self.ee_config.id2label[id] for id in predictions
    ]

        logger.debug(tokens)
        logger.debug(predicted_labels)

        current_label = None
        ret_val = {}
        for token, label in zip(tokens, predicted_labels):
            if token.startswith("##") and current_label is not None:
                ret_val[current_label] += token.replace("##", "")
            elif label.startswith("B-") and not token.startswith("##"):
                current_label = label[2:]
                ret_val[current_label] = token
            elif label.startswith("I-") and current_label is not None:
                if token.startswith("##"):
                    token = token.replace("##", "")
                else:
                    token = f" {token}"

                if current_label == label[2:]:
                    current_label = label[2:]
                    ret_val[current_label] += token
                else:
                    # I-label does not belong to prev B-label -> ignore it
                    pass
            else:
                # O-Label without ##
                current_label = None
                pass

        logger.debug(f"Predicted entities: {ret_val}")
```

```
        return ret_val
```

Listing 8.7: NlpClassifier

PyTest Script for Testing Intent Recognition and Entity Extraction

```
import logging.handlers
from datetime import datetime

import pytest

from src.nlp.nlp_classifier import NlpClassifier
from tests.test_utils.test_utils import compare_dicts_case_insensitive,
    load_test_data

formatter = logging.Formatter("%(asctime)s %(levelname)-8s \n\t%(message)s")
logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)
fhi = logging.FileHandler(
    f"logs\\test_nlp_{datetime.now().strftime('%Y_%m_%d_%H_%M_%S')}.log"
)
fhi.setFormatter(formatter)
logger.addHandler(fhi)

TEST_DATA_PATH = ""  # Path to Test Data
test_data = load_test_data(TEST_DATA_PATH)

result_counter = {}

classifier = NlpClassifier(
    ir_model_path="",  # Path to Intent Recognition Model
    ee_model_path="",  # Path to Entity Extraction Model
)


@pytest.fixture(scope="session", autouse=True)
def run_around_tests():
    yield
    logger.info(
        [
            f"{intent}: {res['correct'] / res['count'] * 100}%"
            for intent, res in result_counter.items()
        ]
    )


@pytest.mark.parametrize("sentence,expected", test_data)
def test_nlp_models(sentence, expected):
    if expected["intent"] not in result_counter:
        result_counter[expected["intent"]] = {"correct": 0, "count": 0}

```

```python
    start_ir = datetime.now()
    intent = classifier.predict_intent(sentence)
    end_ir = datetime.now()

    start_ee = datetime.now()
    entities = classifier.predict_entities(sentence)
    end_ee = datetime.now()

    is_correct = (
        True
        if intent == expected["intent"]
        and compare_dicts_case_insensitive(entities, expected["entities"])
        else False
    )

    result_counter[expected["intent"]]["correct"] += 1 if is_correct else 0
    result_counter[expected["intent"]]["count"] += 1

    log_string = (
        f"Time(IR): {format((end_ir - start_ir).microseconds / 1000000, '.4f')}s| "
        f"Time(EE): {format((end_ee - start_ee).microseconds / 1000000, '.4f')}s"
        f"\n\t'{sentence}'"
        f"\n\tExpected: {expected['intent']} Actual: {intent}"
        f"\n\tExpected: {expected['entities']} Actual: {entities}"
    )

    logger.debug(log_string) if is_correct else logger.error(log_string)

    assert (
        is_correct
    ), f"\nExpected intent: {expected['intent']} \nActual Intent: {intent} \nExpected entities: {expected['entities']} \nActual Entities: {entities}"
```

Listing 8.8: PyTest Script for Testing Intent Recognition and Entity Extraction

PyTest Script for End-to-End Test via the REST API

```python
import json
import os

from fastapi.testclient import TestClient

from src.api.main import app
from tests.test_utils.test_utils import load_xml_files_to_dict

# Set environment variables for model paths in the test environment
os.environ["LOCAL_MODEL_PATH_IR"] = ""  # Path to Intent Recognition Model
os.environ["LOCAL_MODEL_PATH_EE"] = ""  # Path to Entity Extraction Model

client = TestClient(app)

```

```python

def test_use_case_1():
    expected_model_state_dict = load_xml_files_to_dict("test_data/use_cases/use_case_1")

    user_story = [
        ("/create-container", "Create an abstract class named Person"),
        ("/add-value", "Add the private attribute name of type string", "1.uml", "1.unotation"),
        ("/add-value", "Add the private attribute size of type string", "2.uml", "2.unotation"),
        ("/focus", "Select the attribute size", "3.uml", "3.unotation"),
        ("/update-value", "Change the name to weight"),
        ("/update-value", "Change the visibility to protected"),
        ("/create-container", "Create a class named Student"),
        ("/focus", "Select class Student", "4.uml", "4.unotation"),
        ("/move", "Place it right of class Person", "5.uml", "5.unotation"),
        ("/add-value", "Add the private attribute student id of type string", "6.uml", "6.unotation"),
        ("/add-value", "Create the public method get student id with return type string", "7.uml", "7.unotation"),
        ("/add-relation", "Create a generalisation from Student to Person", "8.uml", "8.unotation"),
    ]

    for tup in user_story:
        data = {"user_query": tup[1]}
        body = {}
        if len(tup) > 2:
            if tup[2] in expected_model_state_dict:
                body["uml_model"] = expected_model_state_dict[tup[2]]
            if tup[3] in expected_model_state_dict:
                body["unotation_model"] = expected_model_state_dict[tup[3]]

        intent = client.get("/intent", params=data)
        print(tup[1])
        print(intent.text)

        if tup[0] is not None:
            response = client.post(tup[0], params=data, json=body)
            assert response.status_code == 200, f"Status Code is {response.status_code}"
            print(response.text)

            if tup[0] == "/focus":
                payload = json.loads(response.text)
                assert (
                    "element_id" in payload
                ), f"element_id not found in payload:  {response.text}"
        print("\n")
```

Listing 8.9: PyTest Script for End-to-End Test via the REST API