



Universiteit
Leiden
The Netherlands

Computer Science & Economics

A conversational agent for fixing omissions in requirements specification models

Max W. Vogt
s2098660

Supervisors:

First supervisor: Dr. G.J. Ramackers

Second supervisor: Prof.dr.ir. J.M.W. Visser

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)

www.liacs.leidenuniv.nl

06/08/2022

Abstract

This thesis defines and implements a conversational environment for the user to address issues regarding incompleteness and ambiguity in natural language, for an automatic model-generating environment. The goal of this research is to address issues in requirements specifications. To achieve this a conversational agent (chatbot) will be offered, which can make modifications to a specification model. The user will be engaged in a dialogue with the chatbot, the goal of these dialogues is to fix issues such as ambiguity and incompleteness. The chatbot will realize the modifications and change the model in the database, visualizing the results afterward. Furthermore, the chatbot can fill the gaps in the knowledge of the user, concerning the modeling language (UML). One can ask the chatbot for information about a specific model or general definitions within UML and its components. The chatbot will also proactively reach out to the user with questions regarding incompleteness in the (by the user) provided natural language source. The goal of this approach is that completeness for the model is achieved and that a user (who is not a modeling expert) can change the UML model in a user-friendly way. Bridging the gap between the domain expert and the specification model.

The system proposed in this thesis is an addition to the ngUML project of Leiden University. Where the focus is on eliciting requirements specifications and transforming them into specification models and source code. Natural language is processed via Natural Language Processing (NLP) models to generate the UML models and the source code. As a part of this thesis, the proposed design and functionality have been developed into a working prototype. For the implementation within ngUML, a rule-based approach has been chosen. With the communication happening via a Websocket. The structure of this system can be expanded or reused for future development or usage with other modeling languages.

Acknowledgements

This thesis project has been executed as part of the Bachelor of Computer Science and Economics at LIACS (Leiden Institute of Advanced Computer Science), the Computer Science faculty of Leiden University. Many thanks to dr. G.J. Ramackers and prof.dr.ir. J.M.W. Visser, for supervising and supporting this project. And to all the ngUML team members who supported the development of this system, with special thanks to Pepijn Griffioen and Willem-Pieter van Vlokhoven for the intense guidance and help.

Contents

1	Introduction	1
1.1	Context	1
1.2	Problem statement	1
2	Background & related work	2
2.1	UML	2
2.1.1	Class diagram	2
2.1.2	Activity diagram	3
2.2	ngUML project	5
2.3	NLP	8
2.4	Chatbots	8
2.5	Chatbot classifications	11
3	Logical system design	15
3.1	System requirements	15
3.2	Architecture overview	15
3.3	Front-end	19
3.4	Back-end	20
3.5	Rule engine	21
4	Technical system design	25
4.1	System technical components	25
4.1.1	Django Framework	25
4.1.2	Websocket communication	25
4.1.3	RegEx library	26
4.1.4	NLTK library	26
4.2	Database storage & interaction	27
4.3	Reusability	29
5	System validation	30
5.1	Worked example	30
5.2	Prototype testing & experiments	33
6	Conclusion	35
6.1	Conclusion	35
6.2	Constraints	36
6.3	Future work	36
	References	40
A	System expansion manual	41
B	Test cases & instruction	44
C	Questionnaire	55

1 Introduction

1.1 Context

In today's digital society software is playing an ever-growing role. One might say that the current society cannot function without it. For example businesses, organizations and governments depend on software systems. The software that is developed is becoming more complex and the systems are larger than ever. But the way we develop software (Software Engineering) has not changed a lot in the last decades. Software Engineering is still done (mostly) by coding, using programming languages such as Python, C++, PHP, etc. Although these languages have changed and are more mature than they were decades ago, the principle of how they work and how the user should program is still the same.

One of the issues with programming is that the programmer or coder needs to have extensive knowledge before he or she can program any complex program or algorithm. Therefore only a small portion of the world has knowledge of coding or the ability to create programs or models. A trend of the last couple of years has been low-code platforms and applications. Allowing users to create programs with less programming knowledge or without any programming knowledge at all.

Low code offers a shortcut for creating programs and applications. But that is not all the material that is used for Software Engineering. Another big part is capturing requirements effectively, typically resulting in a specification model. One of the most commonly used model languages is UML. A graphic way to show for example classes, attributes, and relationships between them. To make these models in an easy way without the business specialist user needing any programming knowledge the Prose to Prototype / ngUML project was founded at LIACS (Leiden Institute of Advanced Computer Science), a faculty of Leiden University. This project is further explained in a later section of this thesis.

1.2 Problem statement

With the current software of the ngUML project, a business specialist can generate a UML model by providing plain text or spoken text to the web app. The problem with this is that spoken and written language has a lot of incompleteness and ambiguity within them. This is something that cannot be included in a computer program if we want to generate a UML model. A developer can already fix these issues manually. But the developer will have to go into the back-end/ developer environment to make changes. So the user (business specialist or developer) will need programming knowledge to do this, therefore the goal of the project has not been reached this way. To fix these issues the project needs a conversational component to talk to the business specialist about the text that is provided. Therefore this thesis proposes a chatbot for the ngUML project to give the business specialist a conversational environment to fix issues surrounding the UML model. This can be about incompleteness, ambiguity, or setting variables. The system will also play a role in the identification of potential issues in the provided text of the business specialist. At the same time the system will offer a environment to quickly change the model to the modeler/developer user.

2 Background & related work

2.1 UML

Unified Modeling Language (UML) was created by the Object Management Group (OMG) and is a language to visually display models. UML diagrams are used for visually displaying systems based on objects and the relationships between them. With UML there can be (for example) a representation of how a user interacts with a system. UML defines a big number of different kinds of diagrams. For this thesis, the focus will be on 2 types of UML diagrams: class diagrams and activity diagrams.

2.1.1 Class diagram

A class diagram visualizes different classes, their attributes, and their operations (or methods) and shows the relationships between them [1]. The main components of a class diagram are:

- Classes: the square objects in the model.
- Attributes: part of a class, attributes are shown in the first partition of the class.
- Operations/Methods: part of a class, operations are shown in the second partition of the class.
- Relationships: the connections (lines) between the different classes. The different kinds of relationships are shown in Figure 1:

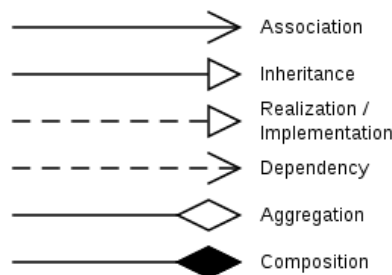


Figure 1: Relationships in UML class diagram [2]

Figure 2 shows an example of a class diagram in UML. In this model, a system for an ATM is shown. It shows all the classes and the relationships between them. The numbers and stars next to the relationships represent the cardinality of the relationship. A star represents For instance the model shows that a Customer can only have 1 Bank, but a Bank can have multiple customers (looking at the "has" relationship). Furthermore, we see associations and inheritance type relationships. The relationships that are represented with "just" a line, are bidirectional associations. The classes consist of three parts. The first part is the top, representing the name. The second (middle) one represents the attributes of the class. The third part shows the operations for the class.

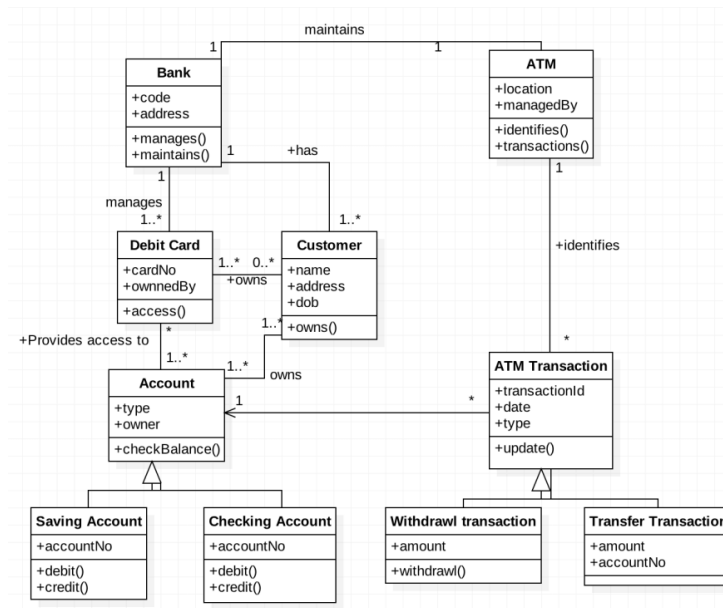


Figure 2: Simple example of a class diagram in UML [3]

2.1.2 Activity diagram

UML activity models are designed to model computational and organizational processes, such as workflows [4]. The diagram should represent the flow of a certain process or activity. The diagram represents the metadata that is located in the activity model. Figure 3 shows the most basic components of a UML activity diagram. And figure 4 shows a simple example of a UML activity diagram. In this example, the verification process of the documents of a student is described.















Symbol	Name	Use
	Start/ Initial Node	Used to represent the starting point or the initial state of an activity
	Activity / Action State	Used to represent the activities of the process
	Action	Used to represent the executable sub-areas of an activity
	Control Flow / Edge	Used to represent the flow of control from one action to the other
	Object Flow / Control Edge	Used to represent the path of objects moving through the activity
	Activity Final Node	Used to mark the end of all control flows within the activity
	Flow Final Node	Used to mark the end of a single control flow
	Decision Node	Used to represent a conditional branch point with one input and multiple outputs
	Merge Node	Used to represent the merging of flows. It has several inputs, but one output.
	Fork	Used to represent a flow that may branch into two or more parallel flows
	Merge	Used to represent two inputs that merge into one output
	Signal Sending	Used to represent the action of sending a signal to an accepting activity
	Signal Receipt	Used to represent that the signal is received
	Note/ Comment	Used to add relevant comments to elements

Figure 3: Basic symbols used for a UML activity diagram [5]

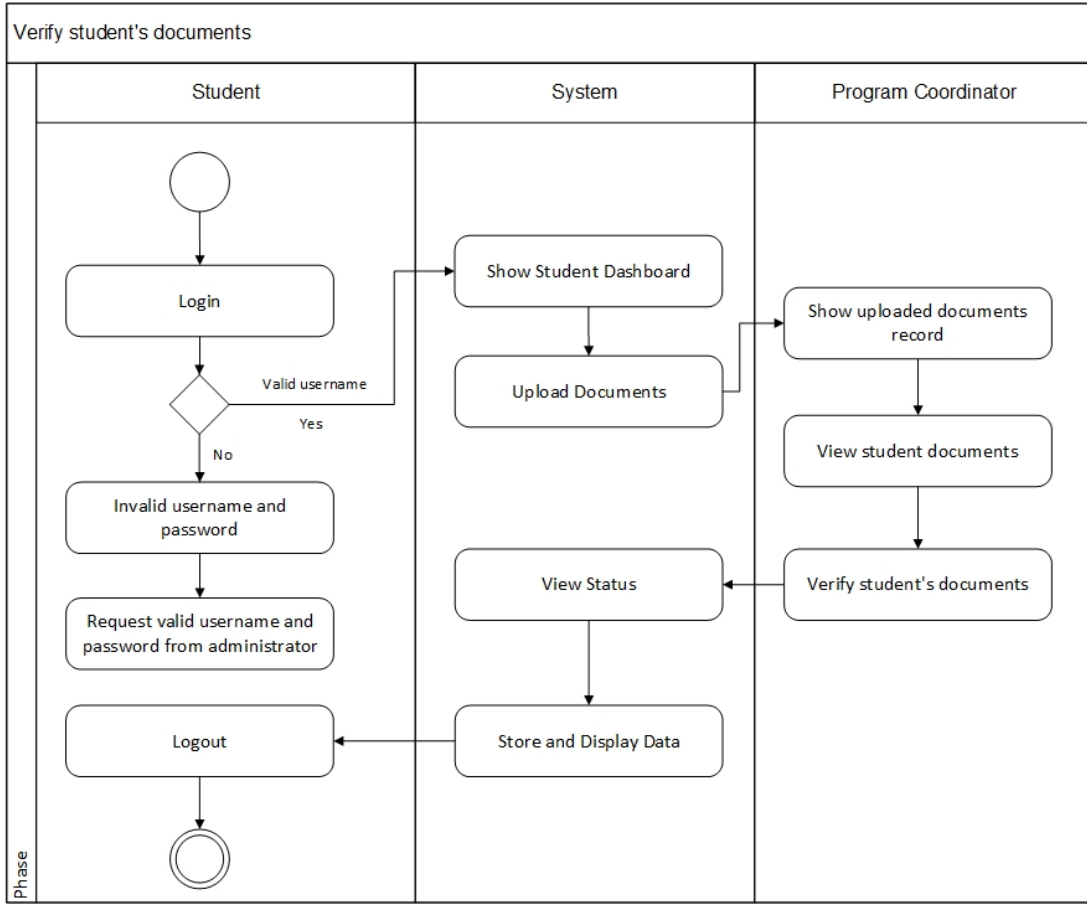


Figure 4: Simple example of an UML activity model

2.2 ngUML project

This research and thesis are additions to the **ngUML (Next Generation UML) project** of the LIACS (Leiden Institute of Advanced Computer Science) faculty of Leiden University. In this article, [6] more information about the project and relating to this system can be found. The current ngUML tool is a web-based application, mainly programmed in Typescript and Python (Django). In this web application, a business specialist can provide a requirements text by either typing it in or by uploading a file. In this text, all the requirements for the model should be stated. Then the text is analyzed and processed, as stated in [6]: "we first apply NLP to a large text. Our finding is that text in natural language always contains ambiguities, noise, and omissions that make it impossible for any NLP approach to produce a correct and complete model automatically. Therefore, we present the resulting model to the developer for feedback and refinement. Our approach includes showing the model in a UML editor and utilizing a (simple) executable prototype of the resulting system. We believe that we can achieve a high level of automation in this way, and offer mechanisms for rapid feedback and refinement." The generated source code is not in scope for this thesis. The ngUML project has different kinds of users: business specialists and developers. Business specialists are the users that provide the text or speech from which the model will be generated.

The developers are the users that can manually change the requirements or modify the model (using the diagram editor in figure 6). These are people with programming or UML knowledge. Below are screenshots of the ngUML tool. Figure 5 shows the environment where the business specialist can provide the requirements text, Figure 6 shows the environment where the developer can modify the model.

The screenshot displays the 'ngUML Requirements preprocessing' web application. The top navigation bar includes links for 'Home', 'Manage requirements', and 'Runnable prototype'. A left sidebar shows the current step as 'Review extraction Step 3'. The main content area is divided into three sections: 'Project name' with the value 'Testproject', 'Project description' with the text 'Project for testing the chatbot', and 'Write requirements'. The 'Write requirements' section contains a text area with the following text: 'An order is placed by a specific customer. A customer has a first name, last name, address, and birth date. The order consists of multiple line items. Each order has an order number, an entry date, a delivery status, and a description. A line item specifies a particular product, and defines the quantity that is ordered. A product is characterized by a name, a description, a product number, a price, a location. Restricted products and flammable products are types of products. Each order is shipped by a delivery company. The delivery company has a name and an address.' To the right of the text area is an 'Upload requirements' section with a file upload box and instructions: 'Max file size is 500kb. Only .txt or .wav files are supported.' and 'Drag and drop your file here or click to upload'. An 'OR' button is located between the text area and the upload section.

Figure 5: A screenshot of the class generator of the ngUML tool

To create this model out of the requirements text. The tool first identifies the metadata from the provided text. The NLP approach to identifying the metadata from the requirements text is based on pre-trained unsupervised language models. After the metadata is collected, the UML model is built. This model is then shown in a visual environment in the diagram editor. Here the developer can make changes to the model. By clicking on a class a menu with different options appears. A developer can for instance change the datatype of an attribute here.

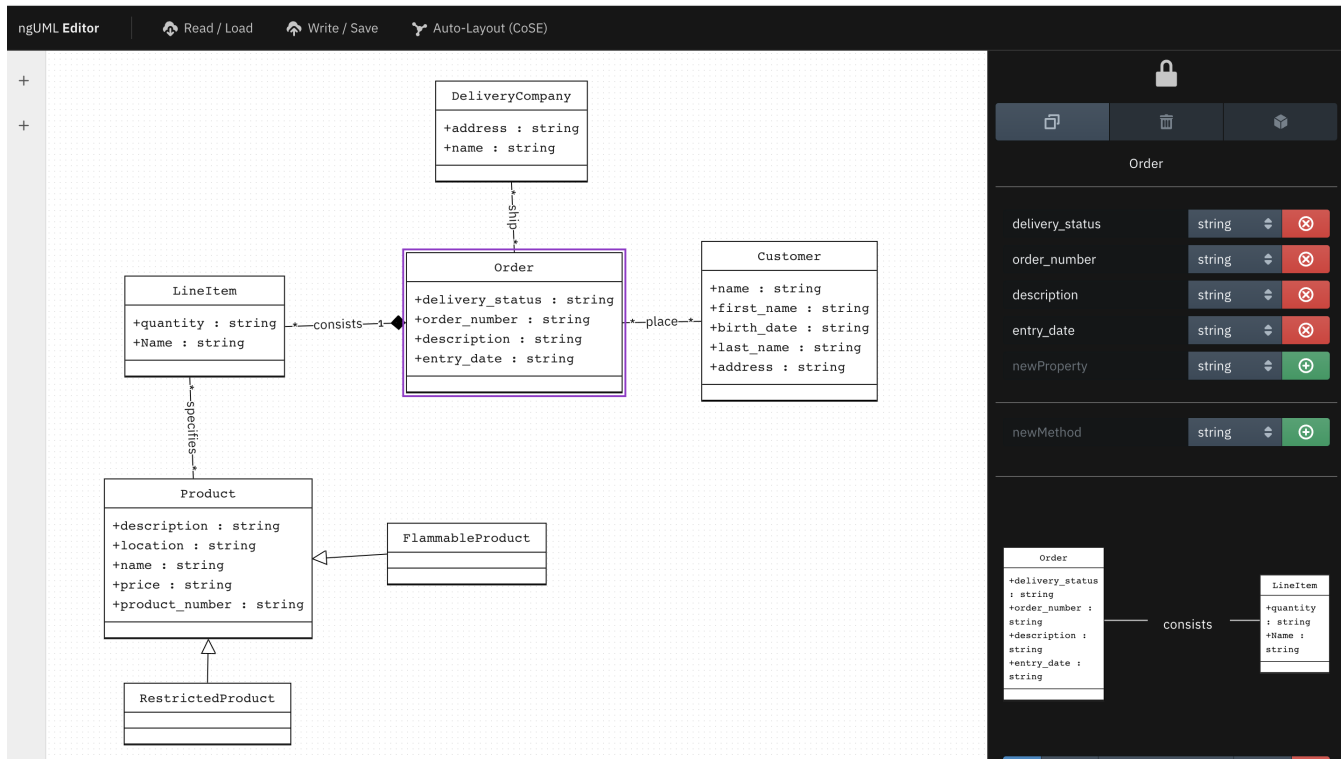


Figure 6: A screenshot of the diagram editor of the ngUML tool

Below there is an overview of all the components of the project (Figure 7). The chatbot will fulfill the role of the conversational component (part of the interactive components).

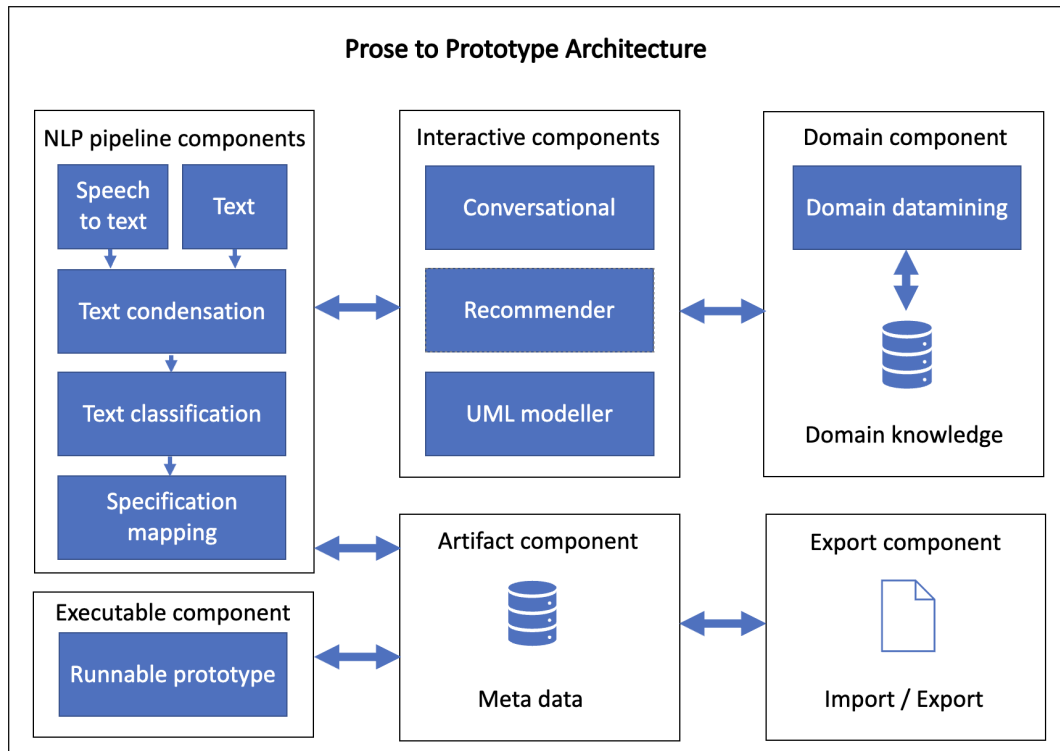


Figure 7: A schematic overview of the ngUML project and all its components [6]

2.3 NLP

Natural Language Processing is a field within artificial intelligence and linguistics. Khurana et al [7] describe NLP as: "Natural Language Processing (NLP) is a tract of Artificial Intelligence and Linguistics, devoted to making computers understand the statements or words written in human languages." Traditional rule-based NLP has 2 blocking issues: natural language has a very big and unstructured nature and the approach with predefined rules is not competent to handle human errors in the natural language [8]. To tackle these issues, NLP based on machine learning algorithms gained popularity. These techniques use algorithms based on probabilities. These systems gave good results because they learn from the data. They do not require a big set of predefined rules and they respond better to unfamiliar and unexpected input. NLP is used in this project to transform the supplied text into a UML model. In the Rule Engine section, the use of NLP for the chatbot is discussed.

2.4 Chatbots

Definition

Lokman and Aamedeen describe chatbots as: "Chatbot (abbreviated Chatting Robot) is a computer

system that allows human to interact with computer using Natural Human Language" [9]. Other terms used for chatbots are dialogue systems, conversational agents, chatterbot, or bot (although bot can also be used to describe other kinds of systems). For this thesis, the term chatbot or bot will be used to represent the options named above.

History

In 1950 Alan Turing wrote his "Computing Machinery and Intelligence". In this piece he discussed the question: "Can Machines think?" [10]. Turing focused on testing whether a computer can communicate in a way that is indistinguishable from the way a human communicates [11]. To test this he came up with the "imitation game" also known as the "Turing test".

This imitation game is played with two people (person A and B) and one interrogator. The interrogator stays in a separate room from the two people. The interrogator can ask person A and B questions and receive answers from them (via a teletypewriter) [12]. The interrogator's goal is to tell if person A and/ or B is a human or a computer program. Adamopoulou and Moussiades [13] argue that the question Turing discussed was the idea that started the development of chatbots. And that the computer programs that could participate in the imitation game can be called a chatbot.

One of the most famous and first chatbots is ELIZA, created in 1966 by Joseph Weizenbaum [14]. Which tried to imitate a psychotherapist, by replying to the human user in the interrogative form [13]. Since then the amount of chatbots and their performance has increased significantly, by using new machine learning technologies such as deep learning and neural networks [14]. The development of the internet, the access to large amounts of data, and more powerful computers led to a lot of improvements for chatbots and an increase in the number of chatbots. Below (Figure 8) is an overview of how many documents were created per year that included keywords like "chatbot", in the Scopus database [15].

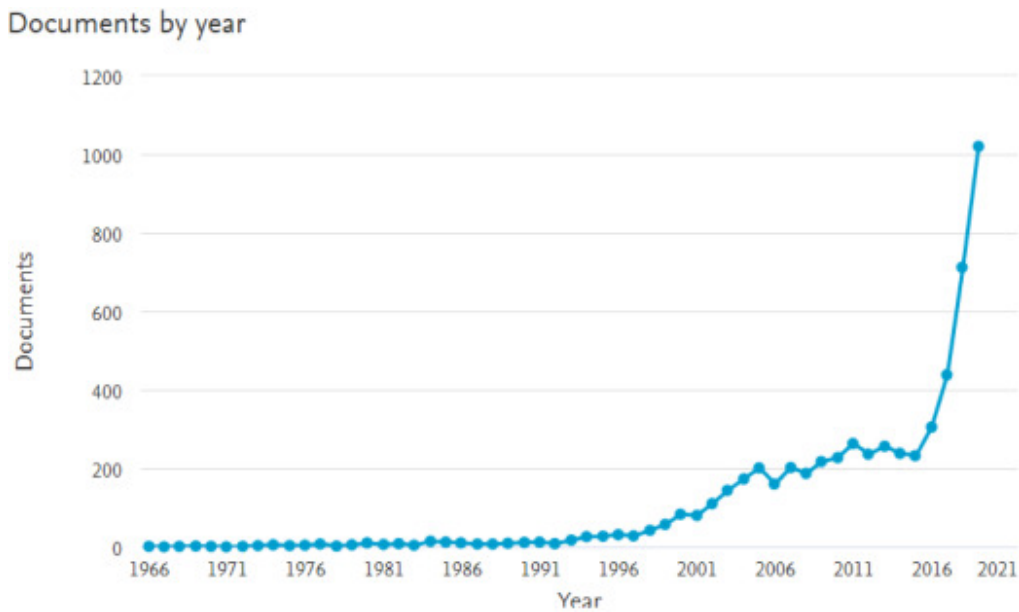


Figure 8: Search Results in Scopus (Scopus preview—Scopus—Welcome to Scopus, 2020), from 1966 to 2019 for the keywords “chatbot” or “conversation agent” or “conversational interface”. [13]

Interaction with chatbots

HCI stands for human-computer interaction and is a field of study about how human users interact with computers and software. How well human behavior is mimicked is vital for how a human user experiences the interaction with a chatbot. Chatbots open a new field of possibilities but also present new challenges for HCI. Because users experience the interaction with chatbots differently and compared to the interaction with other types of software, according to Brandtzaeg and Følstad [16]. Just adding human aspects is not a guarantee for success. As Ciechanowski et al. [17] found when they experimented with 2 different chatbots. One only replied to the human user with text messages (Figure 9), and the other had an animated human avatar that spoke to the human user (Figure 10). The results showed that the test subjects experienced the contact with the first chatbot as more pleasant. Because the second chatbot felt more inhumane and weird. Because of these results, the decision was made to let the chatbot for this thesis look more like the one in Figure 9 than the one in Figure 10. For more information about the design, see section 3.3.



Figure 9: Screenshot of the TEXT chatbot that participants interacted with [17]

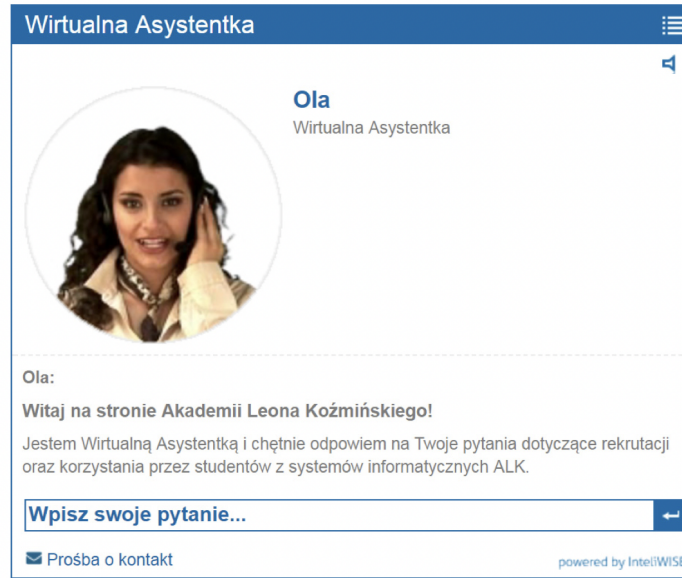


Figure 10: Screenshot of the AVATAR chatbot participants used. The visible avatar was animated (moved according to speech produced on the basis of its text responses presented on the screen) [17]

Jain et al. [18] evaluated the design of different chatbots by conducting a study with a group of first-time chatbot users and their experience with different kinds of chatbots. The study found that the participants expected that the chatbot should always outperform the "traditional" system (website, search engine, etc.). And that the chatbot should show social skills and a personality according to the participants.

2.5 Chatbot classifications

The word "chatbot" has become a container definition for a variety of systems. Chatbots can be classified in different ways, based on certain characteristics of the system [19]. Below is an example of a classification model for chatbots (Figure 11). This is a model presented by Hussain et al. [19]. One classification component that could be added to this model (concerning this system) is the initiator of a conversation. This could be the human user or the chatbot. This is further explained and discussed in section 3.2. Figure 12 shows an expanded and altered version of the model presented by Hussain et al., applied to the chatbot of this thesis. The blue elements are the ones that apply to this particular chatbot.



Figure 11: Broad classification of Chatbots [19]

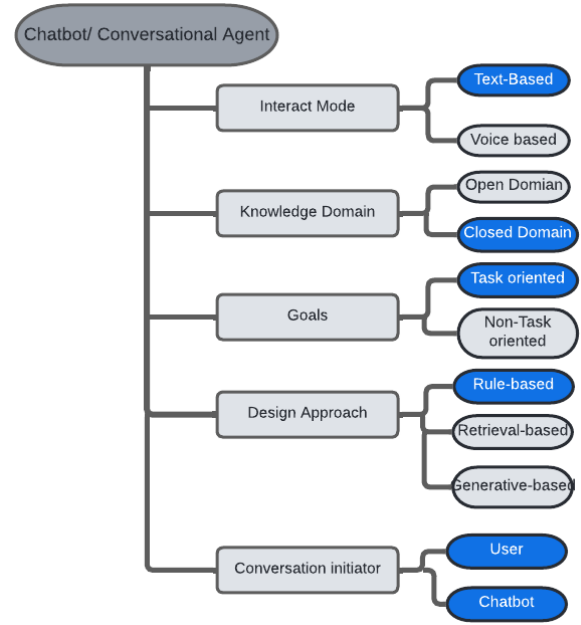


Figure 12: Applied classification model

Interaction mode

The interaction mode is the way a human user can communicate and interact with the chatbot. Hussain et al. [19] argue that a chatbot or conversational agent can interact via text or speech with a human user. The interaction mode of the chatbot of this thesis will be via text.

Another way of communicating with a conversational agent is via a holographic environment. This can be done (for example) with augmented reality (AR) or virtual reality (VR). A topic that is based on these techniques and receiving a lot of attention, is the metaverse. The metaverse could play an important role in the future of chatbots. Mystakidis describes the metaverse as: "The Metaverse is the post-reality universe, a perpetual and persistent multiuser environment merging physical reality with digital virtuality. It is based on the convergence of technologies that enable multisensory interactions with virtual environments, digital objects and people such as virtual reality (VR) and augmented reality (AR)." [20].

Chatbot application

According to Hussain et al. [19], there are 2 different types of chatbot applications: non-task-oriented and task-oriented. Dybala et al. [21] describe non-task-oriented chatbots as: "They are commonly known as "chatterbots" - and, in fact, the name perfectly describes what such systems do: they chat with users. This is why they are called "non-task-oriented" systems". These systems simulate a conversation with another human person. One could say that these systems are designed to test Turing's theorem [10], as discussed in section 2.4.

Task-oriented chatbots are the opposite of non-task-orientated chatbots. These systems are designed to have conversations with the human user, to achieve a pre-defined goal. These chatbots are usually of the domain-specific type (see next paragraph). Adamopoulou and Moussiades [22] describe task-oriented (task-based is an equivalent of task-oriented) chatbots as: "Task-based chatbots perform a specific task such as booking a flight or helping somebody. These chatbots are intelligent

in the context of asking for information and understanding the user's input. Restaurant booking bots and FAQ chatbots are examples of Task-based chatbots." The chatbot that is developed for this thesis is a task-oriented chatbot. Since it has clearly defined goals (see section 3.1).

Domain-specific or open-domain

Adiwardana et al. [23] characterize domain-specific chatbots as chatbots that respond to keywords or intents. Where open-domain chatbots can engage in conversations with the human user, on any topic. The classifications of chatbots based on application or domain are very similar. The difference is that a chatbot classification based on the application looks at a defined goal or the absence of a goal. A classification on domain looks if a domain is specified or not. But the two, in practice, share a lot of similarities. The chatbot that is developed for this thesis is domain-specific, it only responds to specific words and can perform specific tasks.

Design approaches

The design approach of the chatbot determines the way the system generates or selects a response, based on the input from the human user. The 3 different approaches in this classification model are explained below.

1. **Rule-based:** Thorat and Jadhav [24] describe rule-based chatbots as: "Here a Chatbot system works on basis of certain rules. However when the input pattern does not match with any predefined rule then this Chatbot system is inefficient to answer the question." So with this approach, the system uses pattern matching to generate a response. The rules represent the knowledge of the chatbot and are hand-coded by humans [22]. This knowledge is organized and represented in the set of rules using conversational patterns. This kind of system is not always robust against grammatical errors or incorrect input. Because this kind of system is using pattern matching on a predefined set of rules.
2. **Retrieval-based:** Retrieval-based chatbots select an answer from an existing repository of answers. The answer is selected based on the input of the user (certain words, structures, numbers, etc.) Thorat and Jadhav [24] describe a retrieval-based chatbot as: "These bots are trained for lot of inquiries and their possible answers. For each question, the bot can locate the most important answers from the set of every conceivable answer. Likewise, there is no issue with the language and sentence structure as the appropriate responses are pre-decided and it can't turn out badly in sentence structure way." A big advantage of a retrieval-based chatbot is that the answer is always a correct text or sentence since the answer is predefined. A disadvantage of this kind of chatbot is that there can be a situation where the chatbot does not have a good (predefined) answer or when the chatbot selects the wrong answer from the repository of answers [25].
3. **Generative-based:** A generative-based chatbot does not rely on a predefined set of responses. This kind of system generates a new response based on the input the human user provides [19]. A generative-based chatbot often generates the answers using machine learning algorithms. These algorithms are trained using big amounts of training data. This way the chatbot is able to respond to the human user on a variety of topics, depending on the set of training data. These chatbots are usually more fit for open-domain applications. Because they do not rely on a predefined set of responses, the system offers a lot more conversational flexibility

[26]. The downside often is that these kinds of chatbots are less success full when the user asks in-depth questions on a specific topic.

3 Logical system design

3.1 System requirements

The goals and requirements for the specific chatbot of this thesis are:

1. **Bridging the gap between the business user and the UML-models:** by providing a conversational component for the business (domain expert) user. This way a user-friendly environment is created to immediately fix problems in the model or requirements text. Without the business user needing to study UML models more.
2. **Provide a human-in-the-loop approach:** the idea is to combine the knowledge of the system and the business (domain expert) user. That is why the system generates the UML model. And the human user can then modify the model to create the best model, using his/her knowledge of the specific model or domain.
3. **Directly implement solutions:** if a developer or business user changes the stored model via the chatbot. The modification will be directly implemented, the model will be updated. For example, if the user chooses to give the attribute "price" the datatype integer (String by default), the generated UML model will show the price as an integer.
4. **Proactively reach out to the user:** the chatbot will send a message to the user when a new UML model is generated. The chatbot will ask if the user would like to change the datatype of some attributes in the generated UML model.
5. **Re-usability:** this is a key requirement since the ngUML project is an ongoing project of LIACS. The idea is to design and develop a framework. In such a way the chatbot's functionality can be expanded later to be able to fix more issues for the human user. To make this process easier, a manual for expanding the system has been added in appendix A. This manual can be used by (future) team members to expand the system. And the build framework can be reused for the construction of other chatbots.
6. **Robustness:** one of the requirements is that the system should be able to directly implement changes in the model. Therefore the system should be able to make changes to the model, that is stored in the database. This should be done in a safe and robust way. To guarantee that the user does not access parts of the databases, for which he/she has no permission. And to make sure that the model in the database remains correct at all times.
7. **Dynamic dialogues:** to make the system and the dialogues appear to be more "human", dialogues should be dynamic. This means that if a user goes through the same process twice (same path in the dialogue tree), the system should use different responses to achieve the same goal. By randomly using different responses with the same meaning and using dynamic responses, that change based on the user input.

3.2 Architecture overview

Conceptual architecture (Figure 13), in UML use-case format is shown below. What the most important components contain and how they work is explained in this section and the System technical design section.

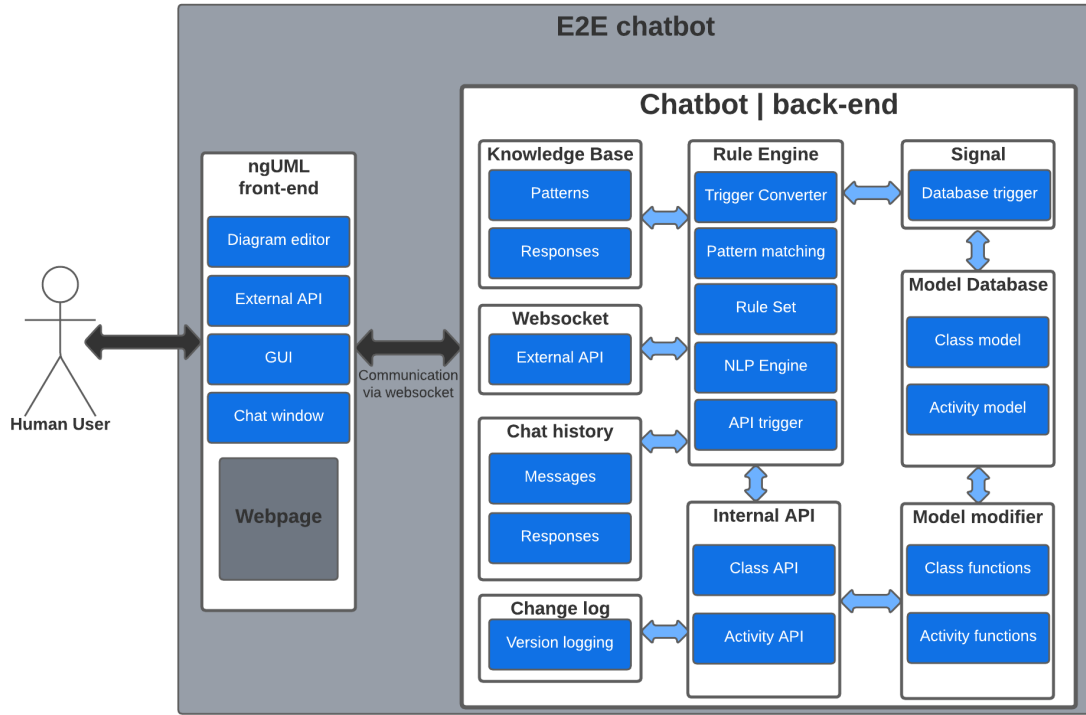


Figure 13: Schematic overview of the architecture

The components shown in Figure 13 all serve a separate purpose and they do so in a certain order. A user interacts with the diagram editor, where the chatbot is accessible. The front-end sends a message to the back-end, specifically to the rule engine. The rule engine selects a response from the knowledge base and sends that back to the front end. All send and received messages are saved in the chat history. If a stored model needs to be changed, the rule engine activates an internal API, which tracks all made changes in the change log. The internal API can make queries on the database. If a new UML model is saved in the database, a signal trigger sends a message to the rule engine.

With this design, a closed domain chatbot that is based on a rule-based back-end is realized (section 2.5). All the components of the system are further explained in specific sections. Next to the flow of the components in Figure 15, there is an example of how a response is generated for the chatbot. Since in this case the stored UML model is not changed, the only required architecture components from the back end are: Rule Engine, Chat History and Knowledge Base.

System components flow

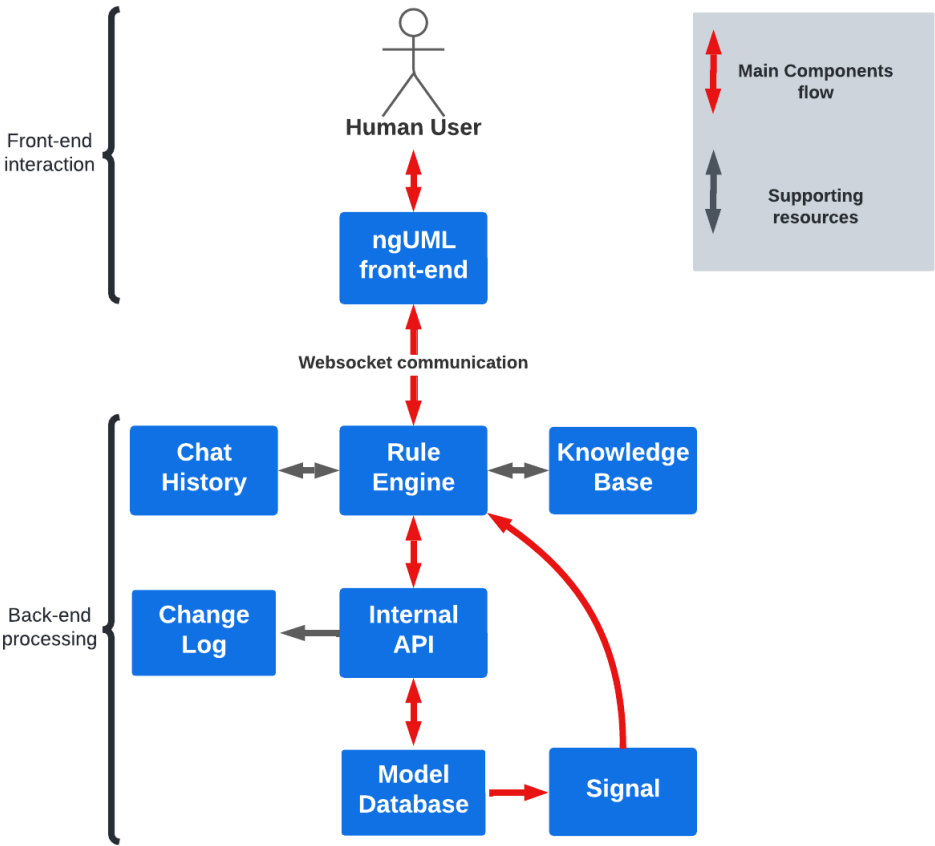


Figure 14: Flow and function of architecture components

Chatbot process example

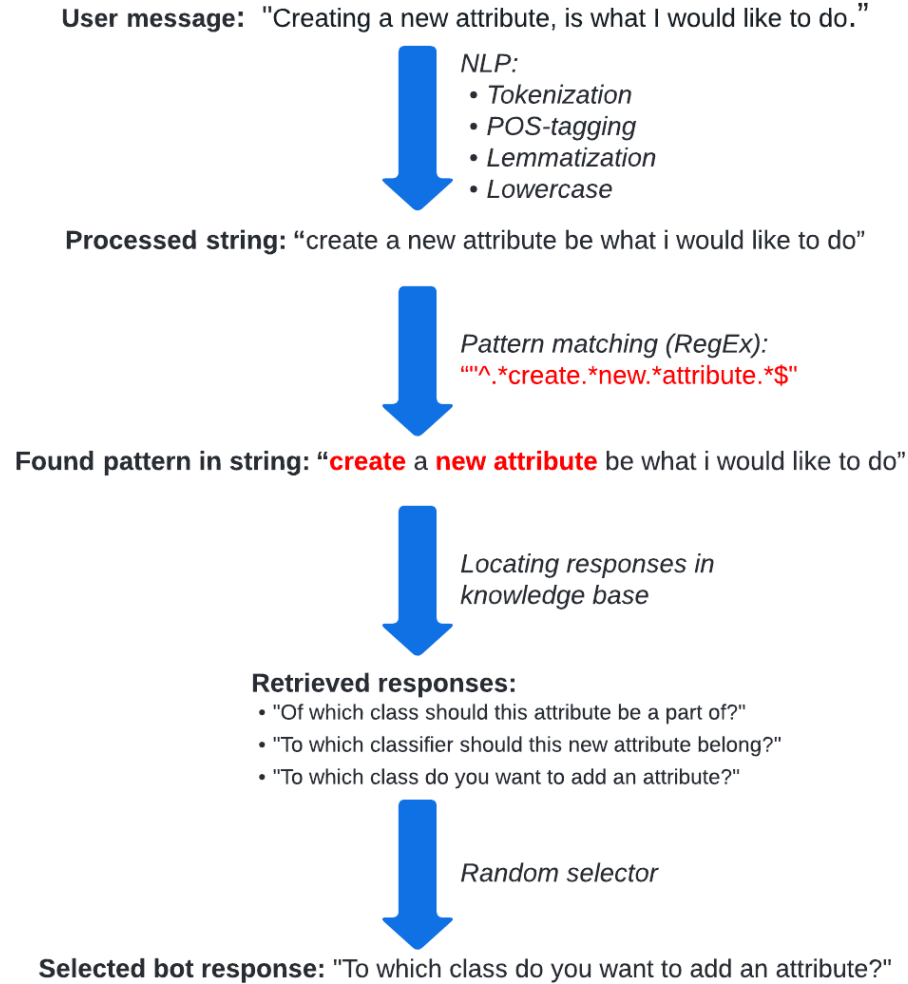


Figure 15: Example of the chatbot process

Figure 16 shows the flow of the chatbot process. The chatbot has two ways of interacting with the user:

1. **Proactive:** this way is activated when the user creates a new UML model and saves it in the database. This can be done either by supplying a requirements text or by manually building the model in the diagram editor. Once the model is saved, a trigger in the database is activated (this is further explained in the "Signal trigger" subsection). Once the trigger is activated, the bot will send a message to the user. The chatbot will ask if the user wants to change the datatype of attributes of the model. This way the user can the data types of a batch of attributes at once. This way the requirement of proactively reaching out to the user is satisfied. And the solution is directly implemented, also satisfying that requirement. The proactive approach allows the system to work (partially) with an event-driven architecture [27].

2. **Reactive:** this is the "standard" way of communicating with a chatbot. A user sends a message to the chatbot and the chatbot sends a response. The user can change the UML model via the chatbot or have a conversation with the chatbot. The user can for instance ask the chatbot for basic explanations of certain components of the model. By letting the user change the model in a conversational environment, the requirements of providing a human-in-the-loop approach and bridging the gap between the business user and the UML models are satisfied.

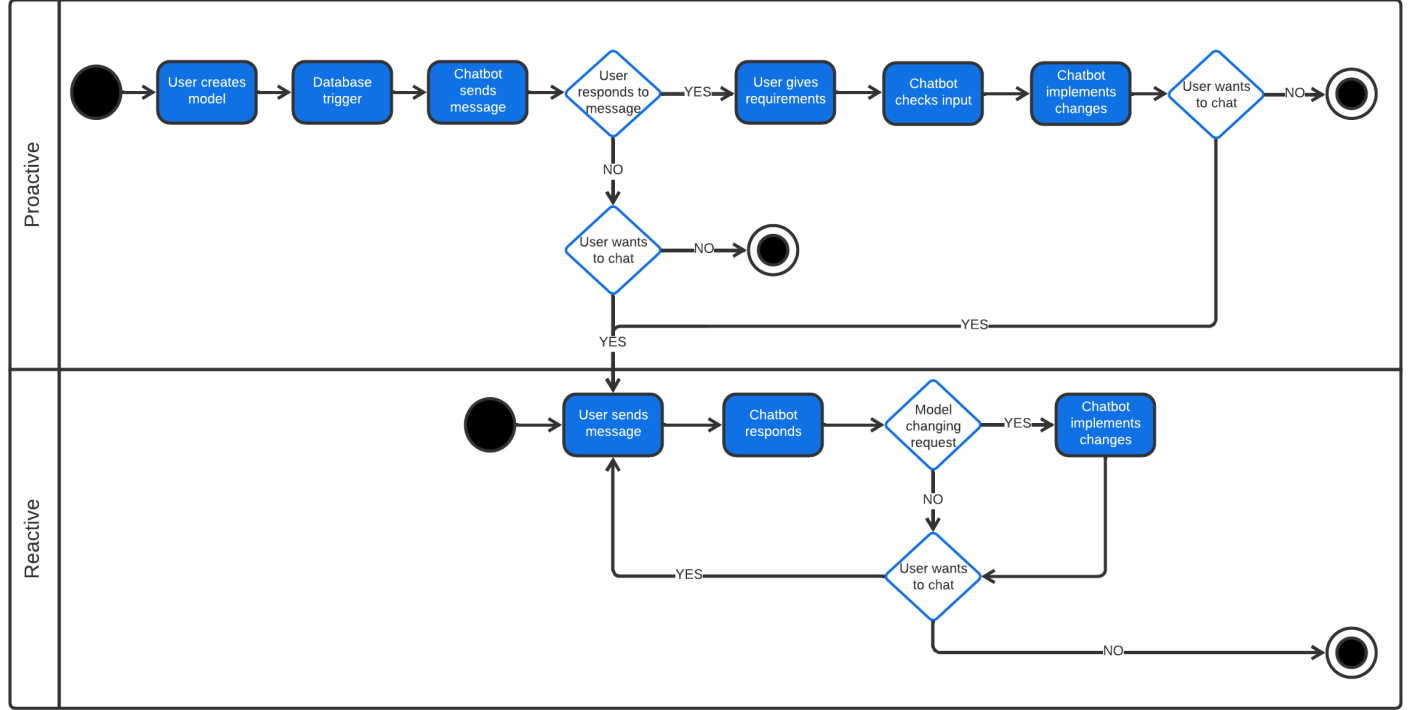


Figure 16: Activity diagram of process of the chatbot

3.3 Front-end

The front-end of the chatbot is a React application, located in the repository for the diagram editor. React was created at Facebook (now Meta) and is a library that can be used to create user interfaces. React is a library based on JavaScript and HTML [28].

A user can open the chatbot by clicking on the blue chat button on the bottom right of the diagram editor page (Figure 17). Once this button is pressed, a pop-up chat window appears (Figure 18). The user can type messages and click on the "send" button or hit the enter key, the message will be dynamically displayed in the chat window. The message is sent via a WebSocket to the back-end (see section 4.1.2). Once a response is received, it will be displayed in the chat window.

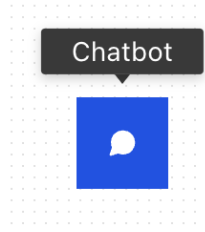


Figure 17: Button to open the chat window

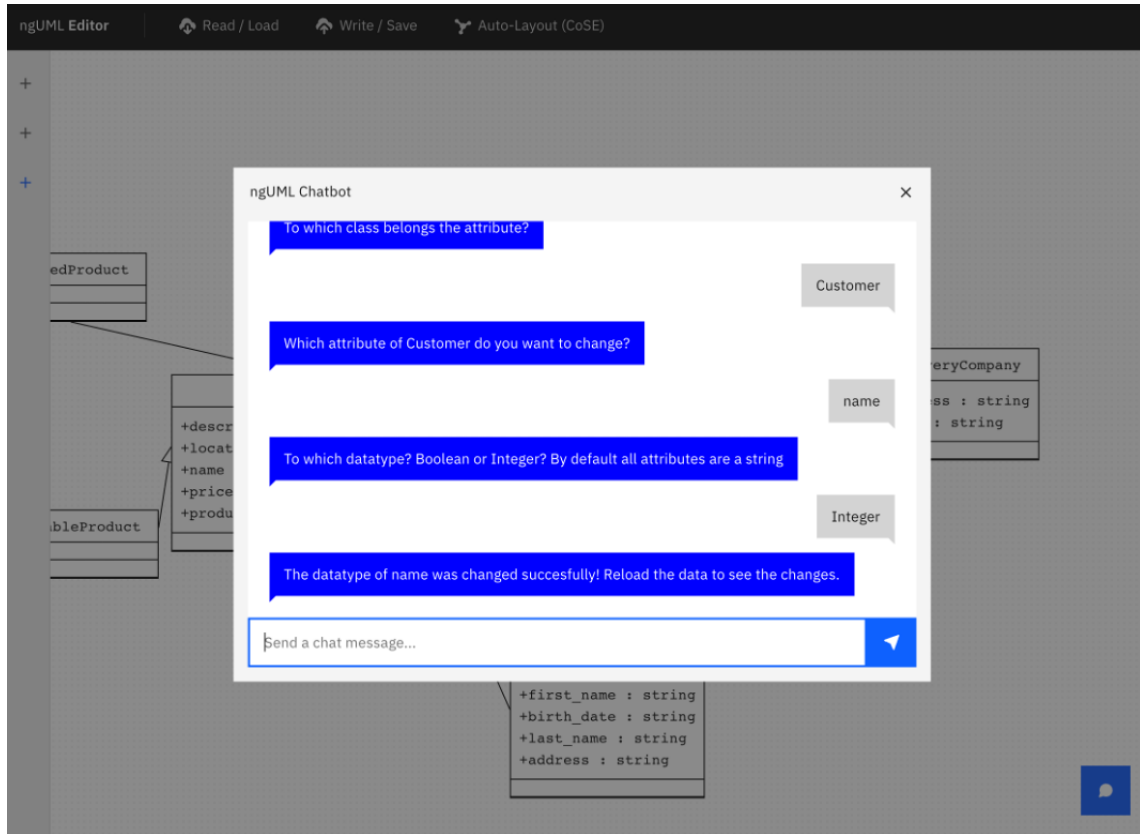


Figure 18: Screenshot of the chat window

3.4 Back-end

For the back-end of the chatbot, the same programming language and framework will be used as for the rest of the ngUML project, Python Django. For the design approach, the rule-based method was selected as the best approach for this chatbot. This decision was based on the following reasons:

- **Re-usability:** this is one of the earlier defined goals of the chatbot. The rule-based approach is the best fit for this goal. Expanding or changing the functionality of the system can be done relatively easily by altering the set of rules/ answers. Since the generative approach is AI-based, changing or adding functionality later can be a complex process. And the retrieval

approach uses a mathematical function to select an answer. This entails that for every addition this function should be reexamined, to check if it still delivers the desired result.

- **Shortage of training data:** the generative approach almost always uses machine learning algorithms. These algorithms need to be trained using training data, to achieve high accuracy. The chatbot that is developed for this thesis is operating in a very specific domain, in which not a lot of research has been done. Therefore almost no training data is available, making it hard to develop and train accurate AI-based chatbots for this thesis.
- **Human-in-the-loop:** one of the earlier defined goals of this chatbot. To achieve the best human-in-the-loop approach, a rule-based approach is the best option. Since there is no potential room for bias, which can be the case with one of the AI-based approaches. With rule-based, the developer can offer the business user a predefined set of options, to combine the knowledge of the system with the knowledge of the business user. With the rule-based approach, the user is able to "move" within the dialogue tree (defined by the rules). So a user has the freedom to choose which path he/she wants to take. But the boundaries and the options are set by the system. Therefore the knowledge of the system/ developer is combined with the input of the user, to achieve synergy. This leads to a robust system with a predefined degree of freedom.

The rules for this approach are located inside the rule engine. This is where the responses of the chatbot are generated and selected.

3.5 Rule engine

Pattern matching

The rule engine is used to determine which response must be used by the chatbot, based on the input from the user. To determine this, the function uses pattern matching. It compares the input of the user to predefined patterns. These patterns are paired with predefined responses (in a separate JSON file). The rule engine compares the (pre-processed) input to the patterns in the JSON file (the knowledge base). If there is a match, the paired response is sent back to the front end. If there is no match, a default message is shown. This matching is not just done with a "simple" direct comparison. The search function from the RegEx package is used, this package is explained later in this section. The responses are not always static but can have dynamic elements, that depend on the input or the model.

Knowledge base

The knowledge base is the place where the "intelligence" of the chatbot is stored. A human can add, delete or modify the knowledge base to change the behavior of the chatbot. This way the functionality of the chatbot is also easily expandable and reusable, which is one of the goals of the project. The knowledge is stored in a separate JSON file which is called from the rule engine. Every element of the knowledge base has 3 elements:

- pattern: (part) of the input from the user which will trigger this response. These are defined using the RegEx library.
- response: the response of the chatbot for this specific pattern

- tag: a tag that represents this pattern-response pair. This is used for easier and faster processing and matching.

Pre-processing

The rule engine does not use the "raw" input from the user. It first pre-processes the user's input. Firstly the input is transformed entirely to lowercase characters, to make the matching case insensitive. After that multiple functions from the NLTK library are used to perform lemmatization on the input. These functions are explained in the next few subsections. This way we make the "dumb" rule-based chatbot a little "smarter", by implementing some machine learning algorithms to pre-process the input. These functions are used to allow the chatbot to perform smarter and more efficient pattern matching. With this approach, not every pattern has to be hard-coded but only the generic cases have to be included in the knowledge base.

Database mutations & hierarchy

In the rule engine, the logic that determines the response of the chatbot is located. This logic can also trigger separate functions that are located in the rule engine. These functions can make modifications to the UML model stored in the database. So the user can use the chatbot to make changes to the model, without the need of using the diagram editor. The chatbot can also provide information about certain definitions and components of the model (in the case the user does not have this knowledge yet). By making the chatbot able to make changes to the model, the earlier defined goals of directly implementing solutions and fixing issues in the requirement text are met. The functions that can be used to change the stored UML model are called the Internal APIs. These are explained in section 4.2. With these functions, the chatbot can be used to delete, add or modify any component of the UML models.

In the database, the activity and class models are stored. The components and all the relationships are stored in a certain hierarchy. Figure 19 shows the database structure of the saved class and activity models. This hierarchy is important when one wants to make changes to the data stored in the database. For instance sub-class elements are only accessible via a foreign key, the parent class. Due to time constraints, the system is currently only able to change saved UML class models. But the structure and design for the same functionality with respect to UML activity models had already been developed. So the functionality can be expanded to activity models and other UML models in the future.

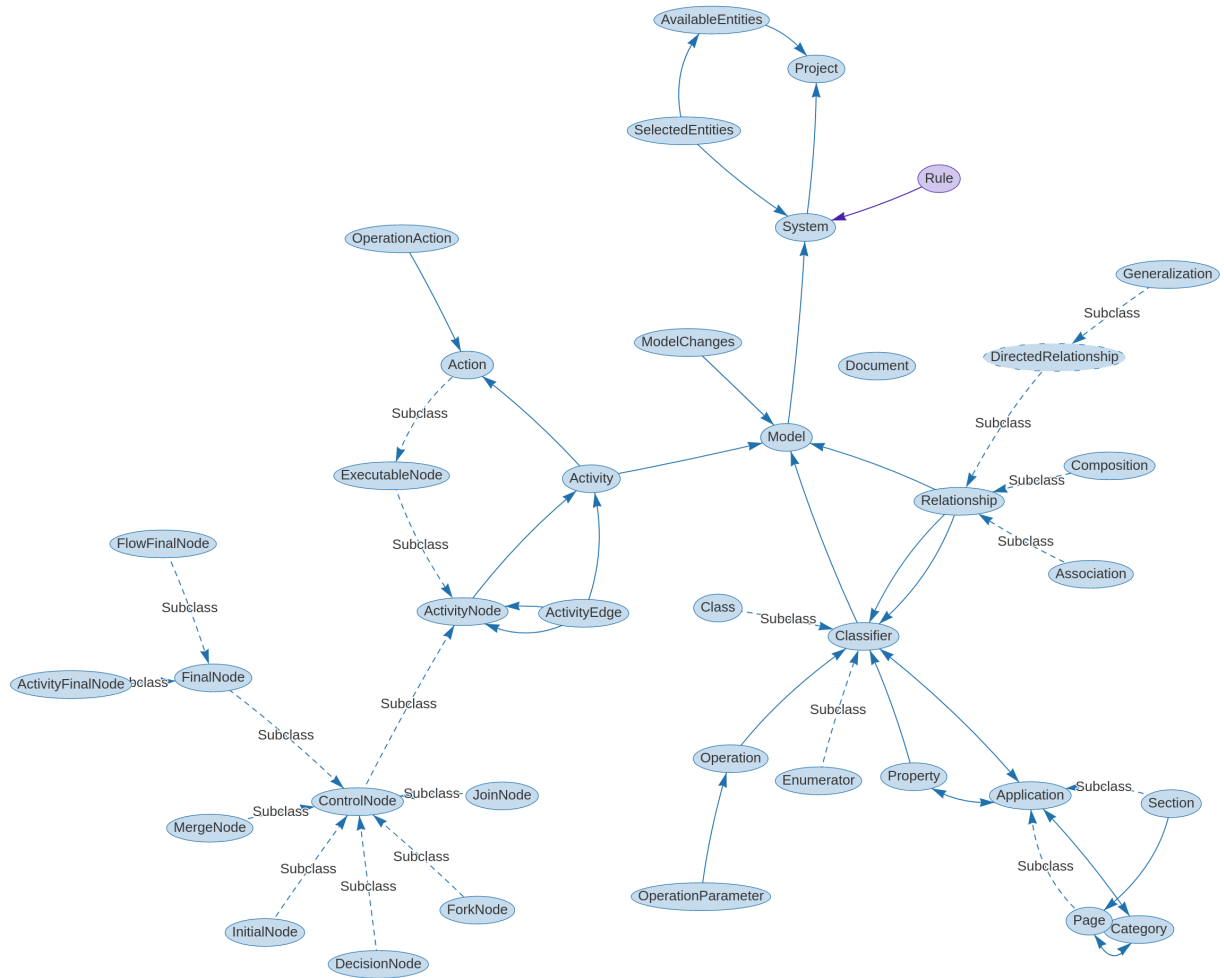


Figure 19: Overview of database structure for the models

Robustness

Robustness is in this system very important since the chatbot has the ability to alter, add and delete database objects. To make the chatbot robust, multiple checks are implemented within the rule engine. Every time a database model is changed or added a check is done if the object exists or if the addition or deletion is valid. To avoid the system from crashing due to database errors or making invalid mutations.

Random generic and dynamic responses

To make the system act more human and seem more competent, random generic and dynamic responses are introduced. The dynamic responses are predefined responses with a dynamic element. For instance, if a user wants to change the data type of an attribute of a classifier: the system will ask the user to provide the name of the classifier, for example, "Order". And then ask the customer which attribute of Order should be modified. So dynamically adding the previous message of the user in the response of the system. Random generic responses are another approach to provide variety in the dialogues between the user and the system. Here the system will randomly select one response out of a set of responses, which have the same meaning but in a different formulation. To make this possible a random selector was built in the Rule Engine. For instance, if a user wants to change the type of a relationship in a class model. The system randomly selects one of the following responses:

- "Which type is the relationship currently?"
- "Is it currently a composition, generalization or association?"
- "What is the current type of this relationship?"

4 Technical system design

4.1 System technical components

4.1.1 Django Framework

The code for the ngUML project for the backend repository is written using the Django framework for Python. Django is an open-source framework for developing web applications. As the Django Software Foundation describes it: "Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of web development, so you can focus on writing your app without needing to reinvent the wheel. It's free and open source." [29].

4.1.2 Websocket communication

For the best user experience, the decision was made to let the chatbot function asynchronously from the rest of the webpage. Using an asynchronous approach offers faster performance and is more user-friendly because the webpage doesn't have to refresh every time the user sends a message to the chatbot. Handling the requests asynchronously means that the interaction with the user happens independent of the communication with the server of the webpage [30]. The approach that was chosen is communication via a WebSocket. This was done based on the following reasons:

- **Latency:** communication via a WebSocket is faster (less latency) than HTTP polling [31].
- **Bidirectional communication:** in contrast to HTTP, Websockets are designed to support bidirectional communication [32]. This allows the chatbot to trigger communication in both directions. With the original approach, there could only be a single response to every single message of the user.
- **Functionality:** since the architecture requires communication between different repositories and applications. This limited the options to choose from significantly.

The WebSocket protocol consists of two parts: the handshake and the data transfer [31]. The handshake part consists of a handshake message from the client side to the server side and a response from the server side to the client side. The data transfer is a bidirectional communication channel where the client and server side can send data to the other side at any time, independently from each other [32]. Figure 20 shows the difference between AJAX and the WebSocket protocol. The blue arrows in the WebSocket visualization represent the handshake and the red arrows represent the data transfer. The WebSocket connection can be closed by the client or the server side. AJAX (Asynchronous Javascript + XML) is another option for letting the system work asynchronously. AJAX works on top of the normal HTTP requests [30]. This means that bidirectional communication is not possible. That is the reason why the choice was made to work with a WebSocket for the chatbot.

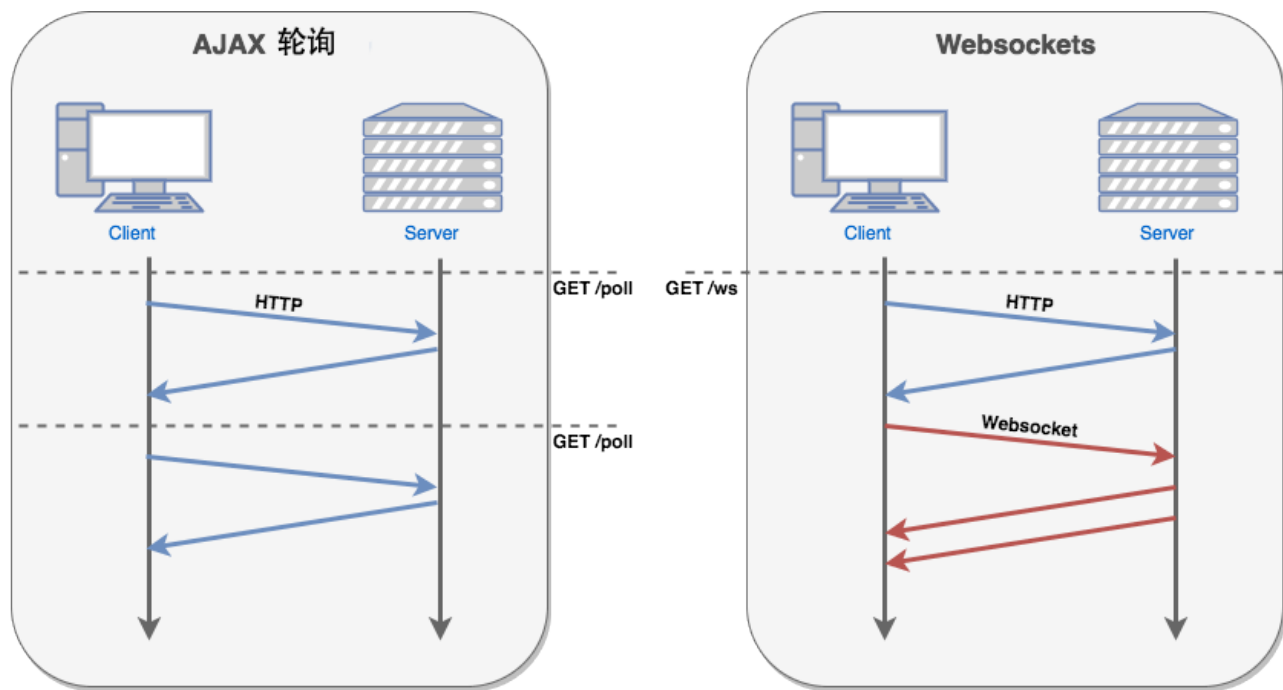


Figure 20: Comparison between AJAX and websocket protocol [33]

4.1.3 RegEx library

For the pattern matching in the rule engine, the search function from the RegEx (short for Regular Expression) library is used. Using this search function allows performing more advanced pattern matching than the standard Python "in" operation. Using a set of special characters the matching can be defined. For instance how the regular expression should start and end, repetitive elements, optional elements, the exact amount of occurrences and more [34]. This way the pattern matching of the rule engine can be more general and flexible and fewer patterns in the knowledge base are needed. An example of RegEx is the following email validator:

$$^[a-z0-9._\%+\-]+\@[a-z0-9.\-]+\.[a-z]{2,4}$$$

Which checks if the beginning is made of lowercase letters, numbers or some special characters. Followed by the @ sign and a domain check (no % and +). Then a backslash and a top-level domain of 2 to 4 letters.

4.1.4 NLTK library

NLTK, short for Natural Language Toolkit. Is a collection of functions and programs designed for symbolic and statistical NLP [35]. It was (originally) developed by Steven Bird and Edward Loper for the University of Pennsylvania in 2001. NLTK is designed specifically for Python and is open-source. Although NLTK originates from 2001, it is still being expanded and updated. The latest release note at this moment is from February 2022 [36]. There is a full NLTK team, working on the NLTK package.

Tokenization & POS tagging

The first function that is used from the NLTK library is tokenization. The tokenization function that is used is called: `word_tokenize`. This function separates a sentence into separate words, based on spaces and punctuation [37]. This function needs to be implemented because the POS tagger needs the tokenized version of a sentence to function correctly.

Next, we use the `pos_tag` function from NLTK. POS tagging stands for Part of Speech tagging. POS-tagging is the process of analyzing a sentence a determining what kind of word each word in the sentence is (noun, verb, adjective, etc.) [37]. This is important for the lemmatizer since the result of the lemmatizer depends on the tag given by the POS-tagger. You could for instance have the word "parking" which could be a noun and a verb. If parking is a noun, the stem is parking. But if parking is a verb, the stem is park.

Lemmatization

For lemmatization, the `WordNetLemmatizer` function is used. Lemmatization shares a lot of similarities with stemming. Stemming is the process of reducing a word to its stem. The difference with lemmatization is that here the word is reduced to a lemma [37]. With lemmatization the result is a normalized version of the original word [38]. If we look at the following words: computes, computing, computed. Using stemming, the stem would be "comput". With lemmatization, the lemma would be "compute". With lemmatization, the result is always a correct word, with stemming this is not necessarily the case. Since the system needs to process sentences or longer parts of a text, just implementing lemmatization would not work. The text first needs to be tokenized and POS-tagged, as explained in the previous subsections. Figure 21 shows an example of lemmatization for a sentence (the first sentence is in Dutch and the second in English). Including the tokenization and POS-tagging process for the sentence.

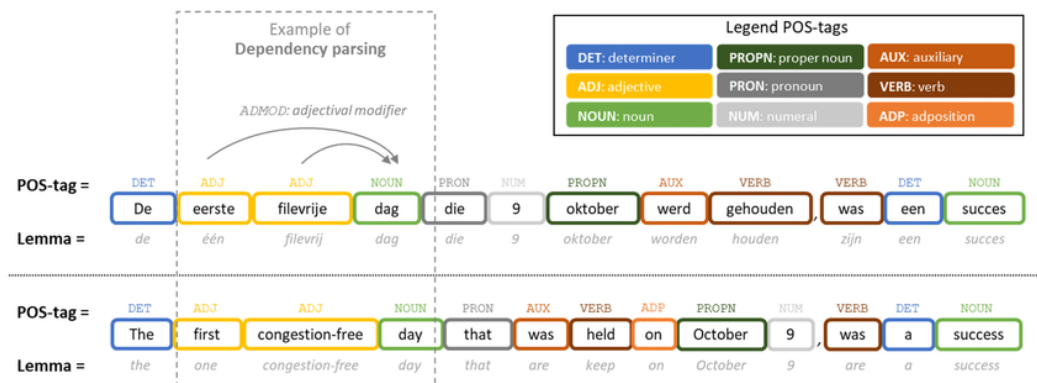


Figure 21: Example of lemmatization for a sentence with tokenization and POS-tagging [39]

4.2 Database storage & interaction

Chat history

To be able to have a longer conversation with the chatbot. The previous messages the user has sent, need to be stored. This way the chatbot can use a tree-like structure to have a conversation with the user. This tree structure is built in the rule engine using conditional statements. These

conditional statements check the tags of the previous messages. Every message that the user sends is stored in a Django model called "Chat_History", for more information about Django, models look at the Django documentation [40]. This model stores as long as the server runs. So even if the user refreshes the page, the conversation is saved and can be continued. The model consists of 4 attributes:

- user_message: stores the input from the user
- answer: stores the answer generated by the bot
- pattern_tag: stores the tag of the answer (from the knowledge base)
- time: a timestamp

Signal trigger

The chatbot should also be able to proactively start a conversation with the user. To achieve this a trigger is needed. The chatbot should reach out to the user when a UML model is saved for the first time. To do this a Django Signal post_save trigger is used. This trigger sends a signal every time (after) the database model is saved. The trigger is built in the Model instance in the database, as shown in Figure 19. Since this results in a signal being sent when (and only) when a UML model is created and saved for the first time in the database.

That signal is then received by a receiver function, which can call the trigger function in the rule engine. From the rule engine, a message can be sent to the front-end, via the WebSocket. For more information about Django signals and receiver functions, see the Django documentation [41].

Internal API

The modifications to the stored model in the database (explained in section 3.5) are executed via internal APIs. API is short for Application Programming Interface. APIs allow smooth communication between digital systems. By using these internal APIs the changes made via the chatbot can be tracked and stored in the version logging. So if a user reloads the requirements text, all the changes made before can be reloaded. This is done by creating a list of all the made changes that are made via the chatbot. The internal API takes JSON objects as arguments to apply the changes and to track all these changes. Figure 22 shows an example of such an object that could be used for calling the internal API, in this case for creating a new object.

```
{
  "type": "new-property",
  "to": {
    "name": "Test",
    "type": "string"
  },
  "key": {
    "name": "test",
    "type": "Class",
    "position": {
      "x": 624,
      "y": 360
    },
  },
  "data": {
    "properties": [{
      "name": "Test",
      "type": "string"
    }]
  },
  "instances": {}
}
```

Figure 22: Example of a JSON object for adding a new attribute

4.3 Reusability

This section described the design of the system in terms of technical components and interactions. This design was chosen to be able to make the system easily expandable and reusable. All communication with the front-end and preprocessing of input happens either separately or at the start of the back-end processing in the rule engine. This way the chatbot can be expanded easily by adding new dialogues directly in the rule engine. Without the need to alter the preprocessing of the input or communication via the WebSocket. That has already been developed and a developer can use the results of these processes, if he/she needs them for the addition to the system. This is also the reason, why the files are structured in the way that they are. So adding new components can happen seamlessly. For more in depth details about expanding the system, see appendix A.

5 System validation

5.1 Worked example

Model changing capabilities

As explained before an important part of the system is the model changing capabilities. This allows the user to modify a UML model in the user-friendly environment that is offered by this system. Figure 23 shows a simple UML class model that could have been generated with the ngUML software. A user could for instance want to add a new relationship to this model. Figure 24 shows (the end of) the conversation with the chatbot to add this new relationship to the model. Figure 25 shows the result of the chat in figure 24, in the visual editor. The newly generated relationship "works for" is clearly visible. The user can realize these changes by solely using the chatbot.

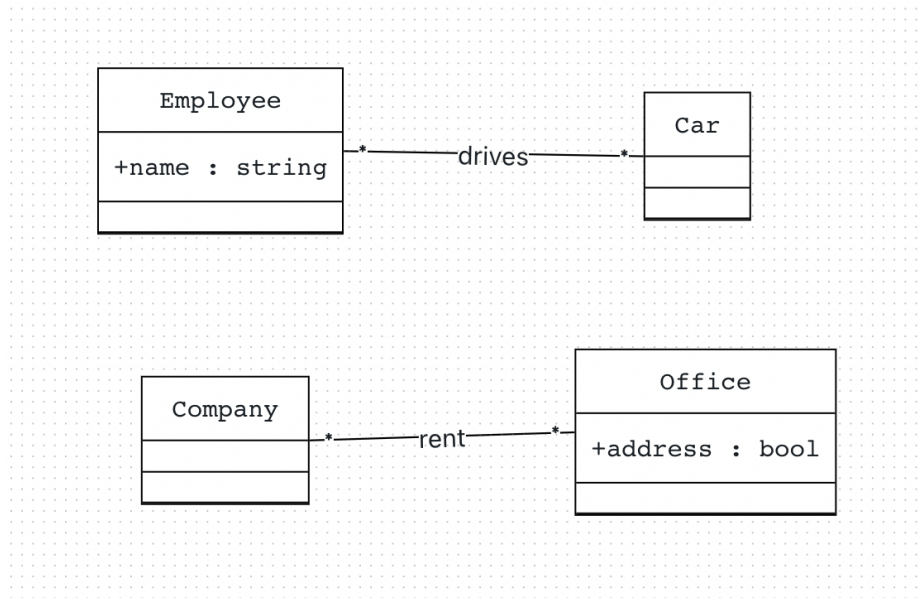


Figure 23: Example of a UML class model in the visual editor

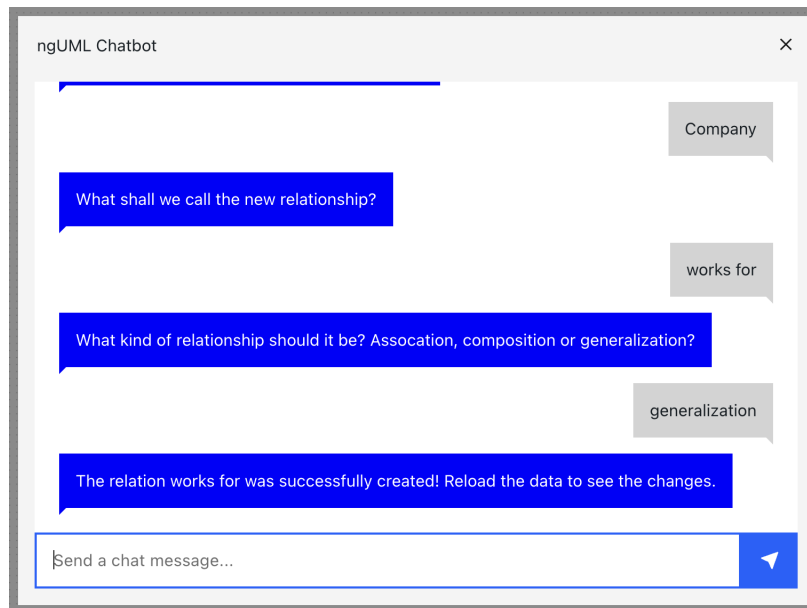


Figure 24: Example of the conversation for adding a new relationship

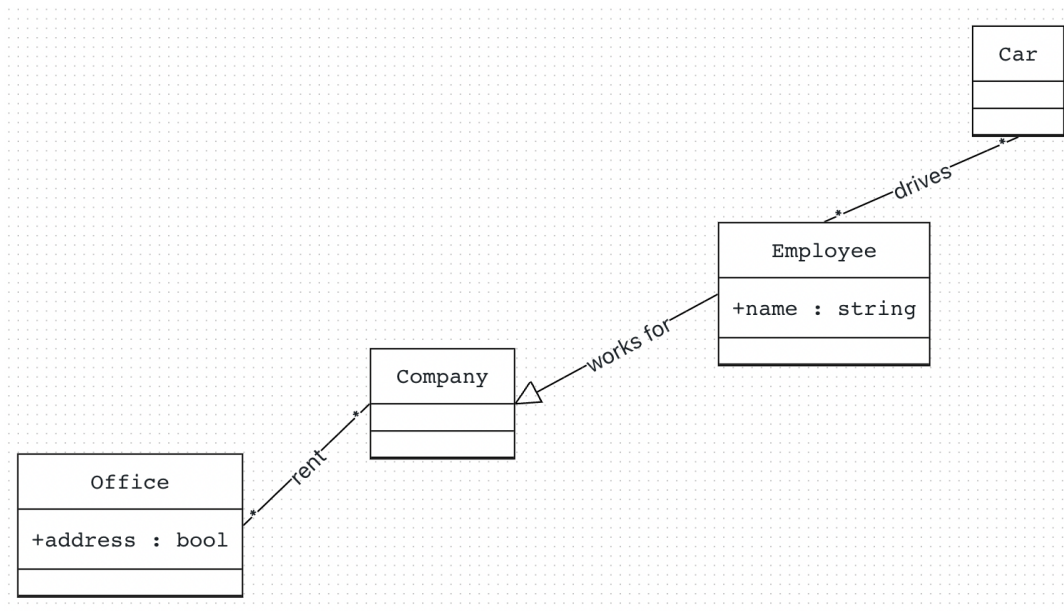


Figure 25: Example of an added relationship via the chatbot

Informative functionality

To allow users without extensive UML knowledge to use the system. It is possible to ask for definitions of UML, diagrams, and their components. These definitions are excluded from the random and dynamic responses, explained in section 3.5. Because this could lead to confusion for the business user. Figure 26 shows an example of how such a conversation could go.

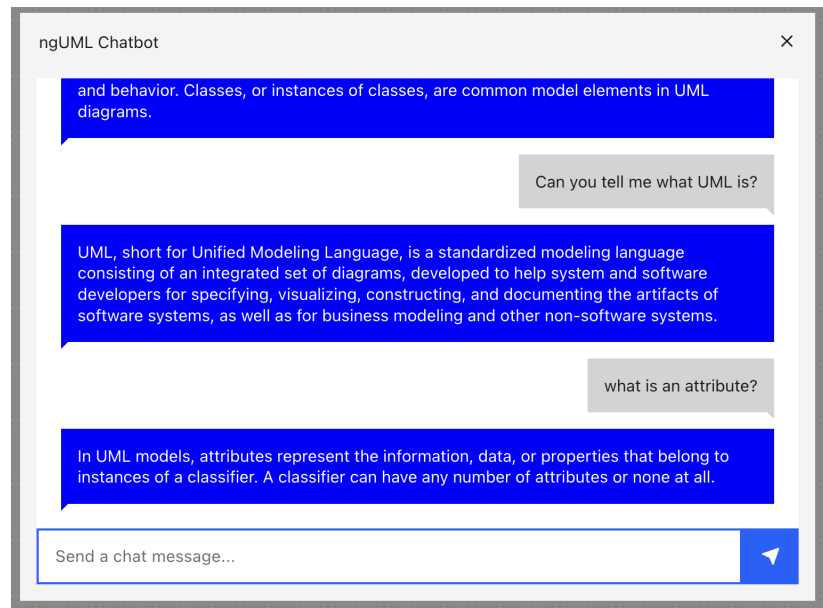


Figure 26: Example of the informative functionality of the system

Proactive approach

The system will reach out to the user if a new model is created and saved in the database. This is done with the signal trigger, explained in a previous section. This would not be possible with a "traditional" HTTP communication, which is why a Websocket is used for this design. The system will ask the user if a datatype has to be changed (since all are set to string as default). The user is then able to change as many data types as he wants. The system will keep asking the user and changing them until the user answers "no" to the question in figure 27, ending the loop. With this approach the predefined goal (section 3.1) of fixing incompleteness in the provided requirements text is satisfied. Since the NLP of the ngUML project is not able to set data types automatically. With this system, it is now possible to do so via the chatbot. With the chatbot reaching out to the user, instead of only acting in a reactive manner.

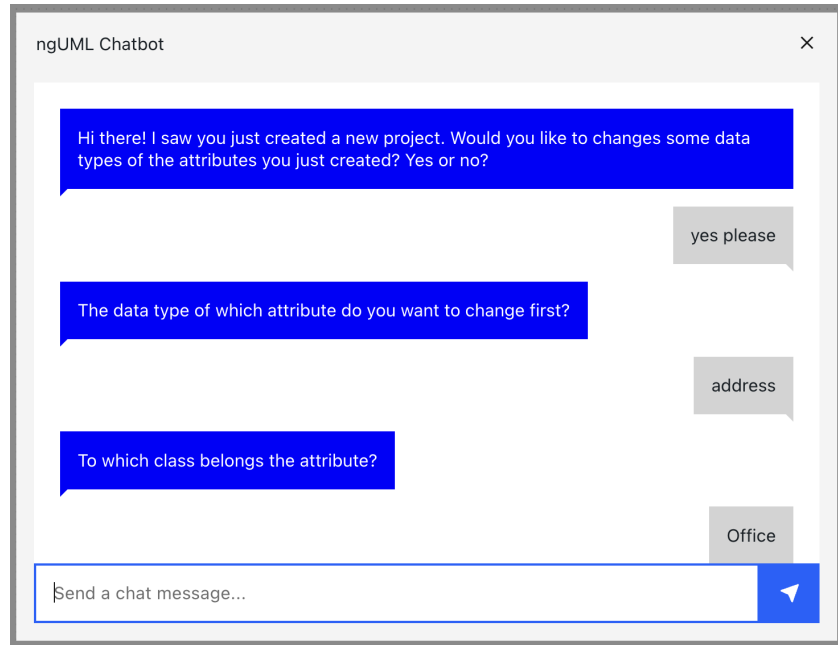


Figure 27: Example of the proactive functionality of the system

5.2 Prototype testing & experiments

Method

The worked example that is described in the previous section, has been tested and validated. A group of project members has used the chatbot to execute a set of test cases. These test cases and the instruction for them can be found in appendix B. The choice was made to only let project members take part in the validation. Since access to the project is required. And without extensive knowledge about the existing code and programming in general, testing would not have been possible. This however meant that the potential group of testers was rather small.

After executing the test cases, the participants completed a short survey. In this survey their experience with the system was measured, to be able to measure if the system is full filling its goals. No questions about the background or demographic of the participants were asked since that serves no purpose for the goal of this validation process. And this validation process and the survey are additional verifications. This is not the main goal of this thesis, which is the design and development of the system. The used survey can be found in appendix C. The survey was conducted using the Qualtrics software package license of Leiden University.

Results

The results of the survey are shown below. The first 12 questions offer 4 options for answering, following a 4-point Likert scale [42]. 1 representing "Strongly disagree", 2 "Somewhat disagree", 3 "Somewhat agree" and 4 "Strongly agree". The table below shows the results of the survey. The question number in the table refers to the number (in terms of order) of the question, not the number before the letter 'Q' in the survey. A short description is added after the question number for clarification.

Question	Mean	Std Deviation	Variance
1 (instruction document)	3.50	0.50	0.25
2 (creating a project)	3.50	0.50	0.25
3 (test case 1)	2.75	0.83	0.69
4 (test case 2)	3.25	0.43	0.19
5 (test case 3)	3.00	0.71	0.50
6 (test case 4)	4.00	0.00	0.00
7 (testing overall)	3.50	0.50	0.25
8 (accessibility)	4.00	0.00	0.00
9 (user-friendly)	3.50	0.50	0.25
10 (relevant addition)	4.00	0.00	0.00
11 (recommendation)	3.50	0.50	0.25

Table 1: Results of validation survey

Only question 3 has a mean below 3. So for all the other questions, the mean was 3 or higher than 3. Meaning that the participants (on average) answered "Somewhat agreed" or "Strongly agree" on the other questions. These questions stated positive stands on the system, so the overall feedback was positive (on average). Especially question 10, which stated that the system is a relevant addition to the existing package, can be viewed as a vital one. This question was answered with "Strongly agree" by all the participants.

Furthermore, the questions regarding test cases 1 and 3 received less positive feedback than the other two test cases (on average). So these test cases were executed less successfully, according to the survey. With this feedback, the functionality of these test cases should be further tested. To spot and fix any potential errors. The last question (12) was an open question in which the participants could leave any comments they had after executing the test cases.

Constraints

In the current validation and testing process, only the reactive approach of the chatbot is tested. This was done because the proactive approach was not ready for testing at the time of the test cases and the survey. This however means that during the validation, not every capability of the system was tested. So the validation could be extended for also testing these capabilities.

For the validation, all participants were ngUML team members. This was necessary since they are the only ones who have access to the software. And they are the only ones with the knowledge to operate and start the software. But it would give a better representation of how users experience the chatbot, to also test with people outside the ngUML project. And a bigger group of participants could also improve the quality of the validation process. Adding some negative statements to the survey could also improve the fairness and accuracy of the results, currently, those were all positive statements.

6 Conclusion

6.1 Conclusion

The system presented in this thesis proposes solutions for the predefined goals, presented in section 3.1. The gap between the business user and the UML models is bridged. This is done via the conversational environment that is part of the system. And with the informative function that the system has, for instance providing definitions and explanations about the models. The human-in-the-loop approach is achieved by involving the business user in the changes that are made to the model. The user is asked specific questions about the modifications, this way the process is easy and user-friendly. The system will proactively reach out to the user to fix incompleteness issues from the requirements text. The process can be viewed in Figure 16. The re-usability goal is satisfied with the structure of the system. This allows for easy expansion in the future for new functionalities. To show how the system can be expanded, an expansion manual has been added to this thesis. This can be found in the appendices.

The system is able to directly implement the solutions that are made in cooperation with the business user, which was also one of the goals. The interactions with the database that can be made via the system are:

1. Changing the datatype of an attribute (UML class model)
2. Changing the name of an attribute (UML class model)
3. Checking the datatype of an attribute (UML class model)
4. Changing the name of a relationship (UML class model)
5. Adding a new relationship (UML class model)
6. Adding a new operation (UML class model)
7. Adding a new classifier (UML class model)
8. Mirroring a relationship (UML class model)

With these solutions, this system tries to offer a user-friendly and effective conversational component. Which tries to have a positive impact and contribution to the ngUML project. The system offers more efficiency, fixing issues, and more user-friendliness. And tries to help with achieving the bigger goal of the research group: changing the way we practice Software Engineering.

With the validation and testing (explained in section 5) the claims made above can be verified. The results of the questionnaire showed an overall positive outcome. Only one question showed a mean below 3. This means that for all the other questions the participants expressed that they (at least) somewhat agreed with the presented statements (on average). And the only question with a mean lower than 3 had a mean of 2.75. Which is still closer to "Somewhat agree" (3) than "Somewhat disagree" (2). Since all presented (closed) questions in the survey presented a positive statement. If the participants (on average) agreed with those statements, the testing can be marked as successful. Since all the participants answered with "Strongly agree" to the question of whether this system is a relevant addition to the software package. The bigger goal of this thesis can be considered verified, namely designing and developing a relevant conversational component for ngUML.

6.2 Constraints

The system proposed in this thesis has its constraints. The rule-based approach that was chosen has its limitations. This design approach only has correct responses for the predefined patterns that it can find. So if a user sends a message without any of the patterns located in the knowledge base, there will not be a desired response. Although there will always be a default response if no pattern is found by the rule engine. The same holds for if the user misspells a word in the pattern. The rule engine will not find the correct response in that case. But if the spelling error is made in a word that is not in the pattern but in another part of the sentence, the system will still find the correct response. So the system is partially robust against spelling errors.

The system is now designed to react to a single message coming from the user. If a user sends multiple commands in one message. The system will only pick up one, the first one it will find in the knowledge base (based on the patterns).

The system is currently only able to change a stored UML class model. The structure and design for UML activity models have been developed and can be included. The same structure can be expanded and used for other types of UML models in the future.

6.3 Future work

Interaction mode

For this thesis, the chatbot that is developed communicates via text messages only. For further development adding communication via speech with the chatbot could be introduced. To expand the functionality and enhance a user-friendly experience. Keep in mind that these kinds of additions have to be executed in a good way. To avoid the issues we saw in the paper of Ciechanowski et al [17], when implementing "human-like" features.

Problem-solving capabilities

One of the goals of the chatbot is that the system should be reusable and expandable. Because the system's capabilities should be expanded in the future. Not only the number of responses but also the functionality and robustness should be expanded.

Design approach

The current design approach of the chatbot is rule-based. With the current developments in AI and Machine Learning, switching to an AI-based approach could be wise. Therefore looking into switching to an AI-based system could be a good option. Since AI-based systems are more suitable to respond to any input of the user. And more robust against spelling errors in the user's input [19].

Communication via speech

Currently, the only way to interact with the chatbot is via a text message. Looking at the current trend for chatbots and virtual assistants. Adding an interaction mode via speech could be a wise addition. This would make the interaction with the system easier and more accessible.

Validation from the user

The system now only proactively reaches out to the user to fix data types (incompleteness) in

the provided requirements text. For a more complete and robust approach, the system could also validate the other generated metadata. So the user could modify this using the chatbot before the model is generated.

References

- [1] I. J. James Rumbaugh and G. Booch, *Unified Modeling Language Reference Manual*. Addison-Wesley Professional, 1998.
- [2] VISUAL_PARADIGM. (2022) <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/>.
- [3] Edraw. (2022) <https://www.edrawsoft.com/example-uml-class-diagram.html>.
- [4] M. Dumas and A. H. M. ter Hofstede, “Uml activity diagrams as a workflow specification language,” in *UML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, M. Gogolla and C. Kobryn, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 76–90.
- [5] Creately. (2022) <https://creately.com/blog/diagrams/activity-diagram-tutorial/>.
- [6] P. G. G. Ramackers, M. Schouten and M. Chaudron, “From prose to prototype: Synthesising executable uml models from natural language,” *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pp. 380–389, 2021.
- [7] D. Khurana, A. Koli, K. Khatter, and S. Singh, “Natural language processing: State of the art, current trends and challenges,” *ArXiv*, vol. abs/1708.05148, 2017.
- [8] P. M. Nadkarni, L. Ohno-Machado, and W. W. Chapman, “Natural language processing: an introduction,” *Journal of the American Medical Informatics Association*, vol. 18, no. 5, pp. 544–551, 09 2011. [Online]. Available: <https://doi.org/10.1136/amiajnl-2011-000464>
- [9] A. S. Lokman and M. A. Ameen, “Modern chatbot systems: A technical review,” in *Proceedings of the Future Technologies Conference (FTC) 2018*, K. Arai, R. Bhatia, and S. Kapoor, Eds. Cham: Springer International Publishing, 2019, pp. 1012–1023.
- [10] A. M. TURING, “I.—COMPUTING MACHINERY AND INTELLIGENCE,” *Mind*, vol. LIX, no. 236, pp. 433–460, 10 1950. [Online]. Available: <https://doi.org/10.1093/mind/LIX.236.433>
- [11] J. H. Moor, “An analysis of the turing test,” *Philosophical Studies: An International Journal for Philosophy in the Analytic Tradition*, vol. 30, no. 4, pp. 249–257, 1976. [Online]. Available: <http://www.jstor.org/stable/4319091>
- [12] A. P. Saygin and I. Cicekli, “Turing test: 50 years later,” *Minds and Machines*, vol. 10, no. 4, pp. 463–518, 2000.
- [13] L. M. Eleni Adamopoulou, “Chatbots: History, technology, and applications,” *Machine Learning with Applications*, vol. 2, p. 100006, 2020.
- [14] I. Dokukina and J. Gumanova, “The rise of chatbots – new personal assistants in foreign language learning,” *Procedia Computer Science*, vol. 169, pp. 542–546, 2020, postproceedings of the 10th Annual International Conference on Biologically

Inspired Cognitive Architectures, BICA 2019 (Tenth Annual Meeting of the BICA Society), held August 15-19, 2019 in Seattle, Washington, USA. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1877050920303355>

- [15] Elsevier-B.V. (2022) <https://www.scopus.com>.
- [16] P. Brandtzaeg and A. Følstad, “Chatbots: changing user needs and motivations,” *Interactions*, vol. 25, pp. 38–43, 08 2018.
- [17] L. Ciechanowski, A. Przegalinska, M. Magnuski, and P. Gloor, “In the shades of the uncanny valley: An experimental study of human–chatbot interaction,” *Future Generation Computer Systems*, vol. 92, pp. 539–548, 2019. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167739X17312268>
- [18] M. Jain, P. Kumar, R. Kota, and S. N. Patel, “Evaluating and informing the design of chatbots,” in *Proceedings of the 2018 Designing Interactive Systems Conference*, ser. DIS ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 895–906. [Online]. Available: <https://doi.org/10.1145/3196709.3196735>
- [19] S. Hussain, O. Ameri Sianaki, and N. Ababneh, “A survey on conversational agents/chatbots classification and design techniques,” in *Web, Artificial Intelligence and Network Applications*, L. Barolli, M. Takizawa, F. Xhafa, and T. Enokido, Eds. Cham: Springer International Publishing, 2019, pp. 946–956.
- [20] S. Mystakidis, “Metaverse,” *Interference*, 2019.
- [21] P. Dybala, M. Ptaszynski, R. Rzepka, and K. Araki, “Evaluating subjective aspects of hci on an example of a non-task oriented conversational system,” *International Journal of Artificial Intelligence Tools*, vol. 19, p. 819, 12 2010.
- [22] E. Adamopoulou and L. Moussiades, “An overview of chatbot technology,” in *Artificial Intelligence Applications and Innovations*, I. Maglogiannis, L. Iliadis, and E. Pimenidis, Eds. Cham: Springer International Publishing, 2020, pp. 373–383.
- [23] D. Adiwardana, M.-T. Luong, D. R. So, J. Hall, N. Fiedel, R. Thoppilan, Z. Yang, A. Kulshreshtha, G. Nemade, Y. Lu, and Q. V. Le, “Towards a human-like open-domain chatbot,” 2020.
- [24] S. A. Thorat and V. Jadhav, “A review on implementation issues of rule-based chatbot systems,” *Social Science Research Network*, 2020.
- [25] Y. Wu, W. Wu, M. Zhou, and Z. Li, “Sequential match network: A new architecture for multi-turn response selection in retrieval-based chatbots,” *CoRR*, vol. abs/1612.01627, 2016. [Online]. Available: <http://arxiv.org/abs/1612.01627>
- [26] L. Wang, M. I. Mujib, J. R. Williams, G. Demiris, and J. Huh-Yoo, “An evaluation of generative pre-training model-based therapy chatbot for caregivers,” *CoRR*, vol. abs/2107.13115, 2021. [Online]. Available: <https://arxiv.org/abs/2107.13115>

- [27] B. Michelson, “Event-driven architecture overview,” *Patricia Seybold Group*, Feb, 2006.
- [28] A. Banks and E. Porcello, *Learning React: Functional Web Development with React and Redux*, 1st ed. O’Reilly Media, Inc., 2017.
- [29] Django-Software-Foundation. (2022) <https://www.djangoproject.com>.
- [30] J. J. Garrett, “Ajax: A New Approach to Web Applications,” 2005, adaptive Path LLC, <http://www.adaptivepath.com/publications/essays/archives/000385.php>.
- [31] V. Pimentel and B. G. Nickerson, “Communicating and displaying real-time data with websocket,” *IEEE Internet Computing*, vol. 16, no. 4, pp. 45–53, 2012.
- [32] I. Fette and A. Melnikov, “The websocket protocol,” Internet Requests for Comments, RFC Editor, RFC 6455, December 2011, <http://www.rfc-editor.org/rfc/rfc6455.txt>. [Online]. Available: <http://www.rfc-editor.org/rfc/rfc6455.txt>
- [33] Programmer_Group. (2022) <https://programmer.group/websocket-usage-and-how-to-use-it-in-vue.html>.
- [34] The-Python-Software-Foundation. (2022) <https://docs.python.org/3/library/re.html>.
- [35] E. Loper and S. Bird, “Nltk: The natural language toolkit,” *CoRR*, vol. cs.CL/0205028, 2002. [Online]. Available: <http://dblp.uni-trier.de/db/journals/corr/corr0205.html#cs-CL-0205028>
- [36] NLTK-Team. (2022) <https://www.nltk.org/news.html>.
- [37] J. Perkins, *Python Text Processing with Nltk 2.0 Cookbook: Lite*. Packt Publishing, 2011. [Online]. Available: <https://books.google.cl/books?id=XjXXnWPkd-AC>
- [38] J. Plisson, N. Lavrac, and D. Mladenic, “A rule based approach to word lemmatization,” in *Proceedings of IS04*, 2004.
- [39] T. Manders and E. Klaassen, “Unpacking the smart mobility concept in the dutch context based on a text mining approach,” *Sustainability*, vol. 11, no. 23, 2019. [Online]. Available: <https://www.mdpi.com/2071-1050/11/23/6583>
- [40] Django_Software_Foundation. (2022) <https://docs.djangoproject.com/en/4.0/topics/db/models/>.
- [41] Django-Software-Foundation. (2022) <https://docs.djangoproject.com/en/4.0/topics/signals/>.
- [42] A. Joshi, S. Kale, S. Chandel, and D. Pal, “Likert scale: Explored and explained,” *British Journal of Applied Science Technology*, vol. 7, pp. 396–403, 01 2015.

A System expansion manual

One of the predefined goals of the system was that it should be reusable and easily expandable in the future. To realize this a certain structure in the system was designed (see the architecture overview). The system can be expanded in the following way:

1. **Add new responses:** if a new input response pair needs to be added. The knowledge base has to be expanded. The knowledge base is located in the `knowledge_base.json` file. Here a new input response pair can be added by giving the expression that will be searched for (pattern matching), the response, and the tag for this pair (all in JSON format). Figure 28 shows an example in the knowledge base for adding a new attribute to a class model. The patterns should be stated according to the RegEx format [34]. Multiple responses can be stated here, then the random response selector can be used to randomly pick one.

```
{
  "patterns": [
    "^.*add.*attribute.*$",
    "^.*create.*attribute.*$",
    "^.*add.*attr.*$",
    "^.*add.*attr.*$",
    "^.*add.*property.*$",
    "^.*new.*attribute.*$",
    "^.*new.*property.*$",
    "^.*create.*new.*attribute.*$"
  ],
  "responses": [
    "Of which class should this attribute be a part of?",
    "To which classifier should this new attribute belong?",
    "To which class do you want to add an attribute?"
  ],
  "tag": "add_new_attr"
},
```

Figure 28: Example of an instance in the knowledge base

2. **Add new dialogue:** if a dialogue consisting of more than 1 input and 1 response needs to be added. A logical tree should be built inside of the `chatbot_main` function within the Rule Engine. This function has the input message of the user as input. Based on the tags of the previous message the logic can be applied, creating a tree-like structure for the dialogue. Figure 29 shows a simple example of logic in the Rule Engine. This piece of logic is built for checking the datatype of an already existing attribute of a classifier in a class model. The first response of the chatbot is inside the knowledge base, after that the `"ask_attr_datatype"` tag is given to the dialogue and the logic is activated. The user stated the classifier of which the attribute is a component in a previous message. This message is no longer directly accessible, therefore it is retrieved from the database, storing all the input messages and responses. The object is stored in a variable called `"atr_msg"` (attribute message) and the specific user input from this object is then stored in `"atr_str"` (attribute string). Then there is a check if the requested attribute and classifier exist within the database (of the model) and if so the datatype of the attribute is retrieved. In this dialogue there are no random answers (for more info, see section 3.5), often there are. In that case, the answers should be posted in the knowledge base and the `"random_select_response"` function should be called with the tag of the dialogue phase.

```

# Conditional statements for checking what the datatype of an attr is
elif latest_tag == "ask_attr_datatype":
    antwoord = "To which classifier does " + raw_input + " belong?"
    current_tag = "ask_attr_class"
elif latest_tag == "ask_attr_class":
    atr_msg = Chat_history.objects.values("user_message").order_by("-id")[0]
    atr_str = atr_msg["user_message"]
    if (
        Classifier.objects.filter(name=raw_input).exists()
        and Property.objects.filter(
            name=atr_str, classifier__name__contains=raw_input
        ).exists()
    ):
        cobj = Property.objects.get(
            name=atr_str, classifier__name__contains=raw_input
        )
        data_type = cobj.type
        antwoord = "The datatype of " + atr_str + " is: " + data_type
    else:
        antwoord = "Something went wrong, are you sure the relationship exists?"

```

Figure 29: Example of logic in the Rule Engine

3. **Add new database mutation:** making changes to a saved UML model, usually happens after a conversation with the chatbot. So firstly a dialogue would need to be constructed, this can be done using the structure that is explained in the previous paragraph. For actually changing the model, the internal APIs have to be used. They can be found in the "tools" folder on the GitHub repository for the back end. Using the `make_changes` function, they can be used. This function needs the model id and an object for the actual changes. These objects are created using special functions, that can generate objects for specific tasks. For an overview of what these objects (should) look like, see the documentation for the internal APIs. Figure 30 shows a simple example of a function that creates such an object. In this case, the object that can be generated is for adding a new attribute to a classifier. Figure 31 shows how the function from figure 30 can be used within the rule engine. The result of the function is stored in a variable, that is passed to the internal API (`make_changes` function). And after that via a database check, the system checks if there is indeed a new attribute in the database (if the addition was successful or not). And then communicates that back to the user.

```

# Function for generating an object for adding an attr (internal API)
# Input = 1. new datatype for attr
#         2. the name that has to be used
#         3. class of which the attr is a part
#         4. id of the current model
# Output = object for adding a attr
def generate_new_attr(dtype, model_id, new_name, class_str):
    node = find_node_by_key(model_id, {"name": class_str})
    curr_id = getattr(node, "id")
    new_attr = [
        {
            "type": "new-property",
            "to": {"name": new_name, "type": dtype},
            "key": {
                "type": "Class",
                "name": class_str,
                "data": {
                    "properties": [],
                    "methods": [],
                },
                "instances": {},
                "id": curr_id,
                "position": {"x": 0, "y": 0},
            },
        },
    ]
    return new_attr

```

Figure 30: Example of function for generating an object for the internal API's

```

new_attr = generate_new_attr(dtype, model_id, new_name, class_str)
make_changes(get_class_model(model_id), new_attr)
if Property.objects.filter(
    name=new_name, classifier__name__contains=class_str
).exists():
    antwoord = "The attribute was added successfully! Reload the data to see the changes."
else:
    antwoord = "Something went wrong, are you sure the classifier exists? And the datatype is correct?"

```

Figure 31: Example of code for calling internal API

B Test cases & instruction

Chatbot validation - Test cases

Max Vogt

July 2022

Contents

1	Introduction	1
2	Practical requirements	1
3	Creating a project	2
4	Test cases	6
4.1	Remarks	6
4.2	Adding a new relationship	7
4.3	Adding a new attribute	8
4.4	Changing data type	9
4.5	Informative functionality	10

1 Introduction

This document gives instructions for the validation and testing of the chatbot system of the NG-UML project. The tests should in total take 20-30 minutes, that is including the survey afterward. Many thanks for taking the time to take part in the testing process.

2 Practical requirements

To be able to test the system, some practical requirements apply. The requirements are listed below. These are written under the following assumptions: you have access to the Github repositories of the NG-UML project and you are familiar with the project.

- Docker installed with an assigned memory (RAM) of at least 4GB.
- You have downloaded git and are signed in with your credentials.
- For the tests two repositories are required: ngUML.backend and ngUML.editor

- For the back end: you should use the latest version of the branch "chatbot".
- For the front end: you should use the latest version of the branch: "chatbot".

3 Creating a project

For the testing of the system, a project needs to be created. This can be done with the following steps:

1. Start the back end, using "docker compose up" command. If you need to build the container first, use the command "docker compose build" and then "docker compose up". **NOTE:** before running "docker compose up" delete all the current data with "bash delete_demo_data.sh" (if you used the back end before).
2. Start the front end, with the "yarn start" command.
3. Browse to the address "http://localhost:3000/", now should see the following web page:

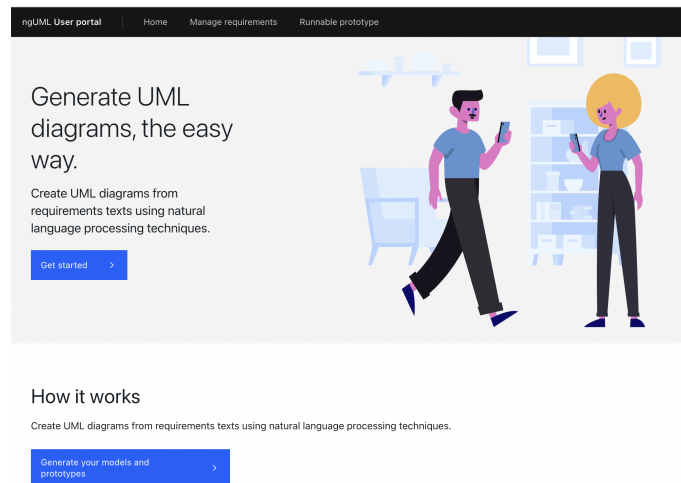


Figure 1: Landing page of the front end

4. Now click on the "Get started" button. You should now reach the following page:

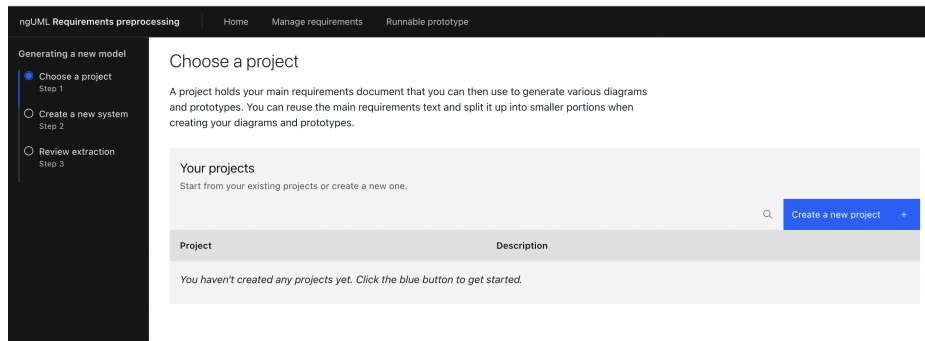


Figure 2: Page for creating projects

5. There should be no existing projects on this page. If there are existing project: stop running the back end container, use the command "bash delete_demo_data.sh" (to clear existing projects) and restart the back end container (start again with step 1).
If there are **no** existing projects: Click on the "Create a new project" button. You should reach the following page:

Figure 3: Page for submitting project data

6. On this page you need to submit the data for the project. You can submit the following data:
 - Project name: Test project
 - Project description: Project for testing the chatbot
 - Write requirements: An employee has a name and drives a car.
An office has an address and is rented by a company.

Now click on the "Save and continue" button on the bottom right of the

page.
Now you should reach the following page:

ngUML: Requirements preprocessing | Home | Manage requirements | Runnable prototype

Generating a new model

- Choose a project Step 1
- Create a new system Step 2
- Review extraction Step 3

Create new system

Systems are a subset of your requirements document, focused on specific entities that are of interest for modelling purposes, and with a specific type of UML model. Working with systems means that you can compartmentalize your requirements, allowing for incremental development and better intermediate feedback.

System name

Test system

Select entities of interest

- ☒ Employee
- ☒ Car
- ☒ Office

Select UML types

Class model Activity model Use case model

Back Save and continue

Figure 4: Page for defining the system

7. On this page, execute the following actions:

- In "System name" write: Test system
- For select entities of interest: mark all the options
- For select UML_types: select Class model

Now your page should look like the one in the figure above. Now click on the "Save and continue" button, on the bottom right.

8. No should have reached the page shown in the figure below, with the metadata shown in the figure. If all the data is the same as shown in the figure, click on the "Confirm and generate diagrams" button, on the bottom right.

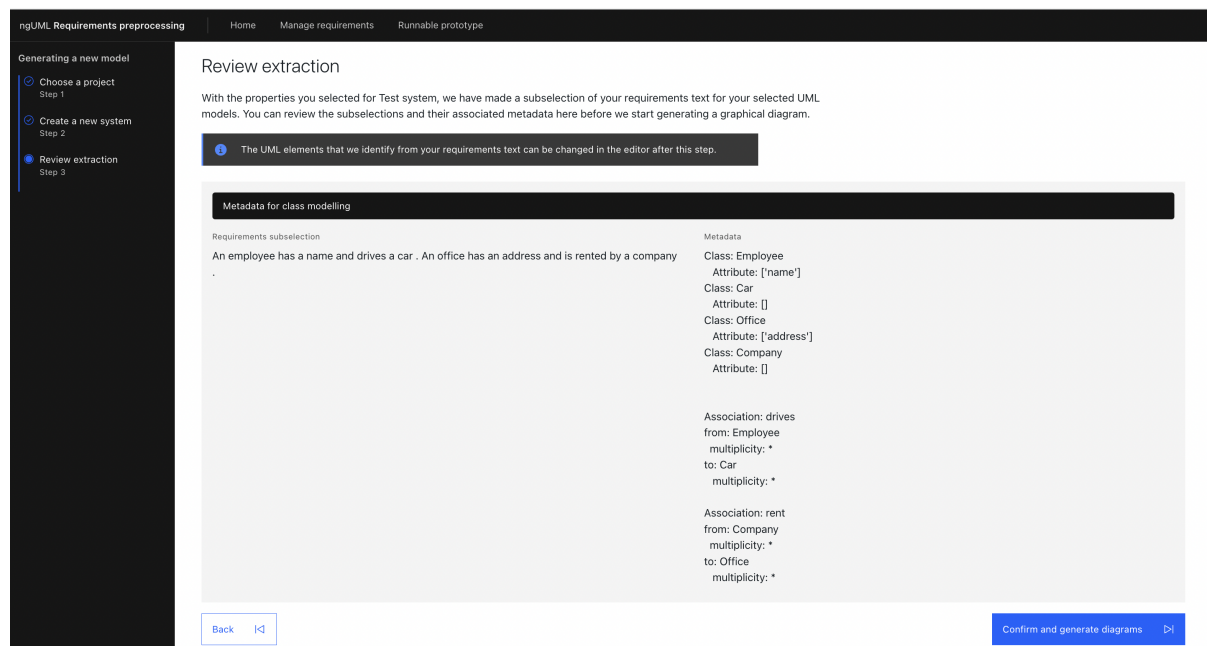


Figure 5: Page for reviewing the metadata

9. After this you should be redirected into the diagram editor. After waiting for a few moments, the generated model should appear.
10. Click on the "Auto-Layout (CoSe)" button on the top of the page, now your screen should like this:

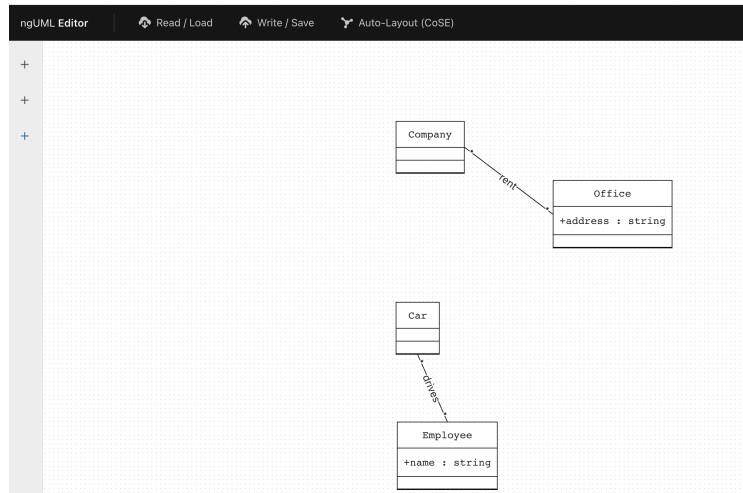


Figure 6: Diagram editor

11. You have now successfully submitted a project and generated a UML class model. You can now go to the next section of this document to start with the test cases.

4 Test cases

4.1 Remarks

This system is still being developed and therefore you might experience trouble/ bugs while executing the test cases. If you do, you can always reach out to Max Vogt, contact details will be shared with all participants. Some remarks about using this version of the chatbot:

1. If you want to exit/ break a conversation tree, use one of the following commands (by sending them in the chat window): cancel, quit, exit, break or escape. This will reset the conversation to the default, without deleting the previous messages. This can be useful if you for instance make a spelling error and want to start over.
2. If the whole chatbot stops responding (for instance due to a DB issue), use the "DELETE99" command by sending it in the chat window. This will

delete all previous messages from the database and allows you to start over. It is wise to use the "cancel" command after the "DELETE99" command to completely start over.

3. The system uses dynamic and random responses, this entails that the chatbot will use different responses for the same conversation (if you retry it). These responses should all serve the same purpose, but you can expect different articulates if you retry a test case.
4. The system is not fully robust against spelling errors and lower/upper case errors. Especially if you are entering a name of a UML component (classifiers, relationship etc.), make sure the spelling is correct. For "normal" sentences, the system can handle spelling error up to a certain degree.

4.2 Adding a new relationship

For this test case a new relationship will be generated for the generated UML model, using the chatbot.

1. Open the chatbot from the diagram editor, using the blue button on the bottom right of the page. You should now see the chat window:

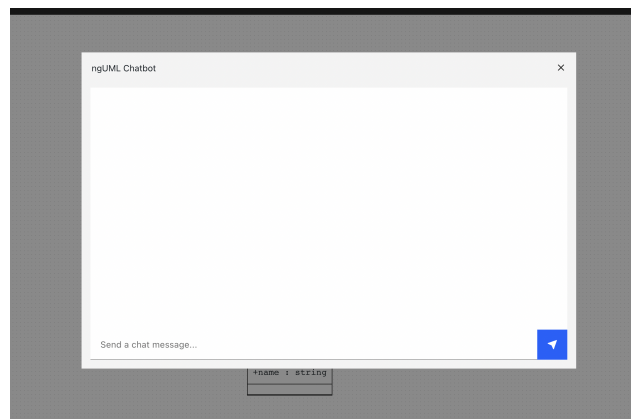


Figure 7: Chat window of the chatbot

2. To test the connection type "Hi!" and send the message. The bot should respond with some kind of greeting.
3. Now type and send "I want to add a new relationship".
4. The bot will ask you for a starting class of the relationship. Type and send "Employee", make sure that the spelling is correct.

5. The bot will now ask for an endpoint of the relationship. Type and send "Company".
6. The bot will now ask for the name of the relationship. Type and send "works for".
7. The bot will ask for the type of the relationship. Type and send "generalization".
8. The bot will now tell you if it successfully created the new relationship. If this is not the case, type and send "cancel" and start over. If this is the case: close the chat window and reload the data (click on the "Read/Load" on the top, click on "classes" and then on "Yes" in the pop-up). If you now use the "Auto-Layout (CoSe)" button again, you should see something like this:

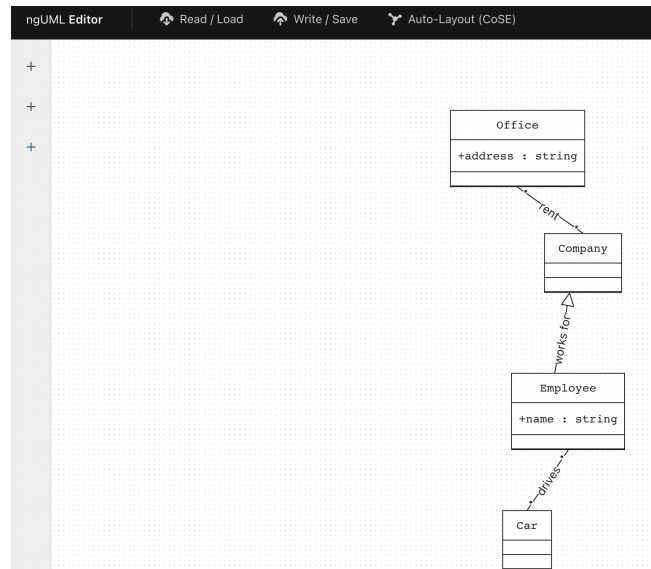


Figure 8: Diagram editor with the new relationship

4.3 Adding a new attribute

For this test case a new attribute will be added to an existing classifier, via the chatbot:

1. Open the chat window.
2. Type and send "Creating a new attribute, is what I would like to do".

3. The bot will ask to which class you want to add an attribute, type and send "Company".
4. Now you will be asked for the name of the new attribute, type and send "workforce".
5. The bot will ask you for a datatype, type and send "Integer please".
6. The bot will now announce if the addition was success full. If this is not the case, type and send "cancel" and retry. If this is the case reload the data, with the same steps as for the previous test case. Your model should now have the new attribute in the class Company, with the correct datatype:

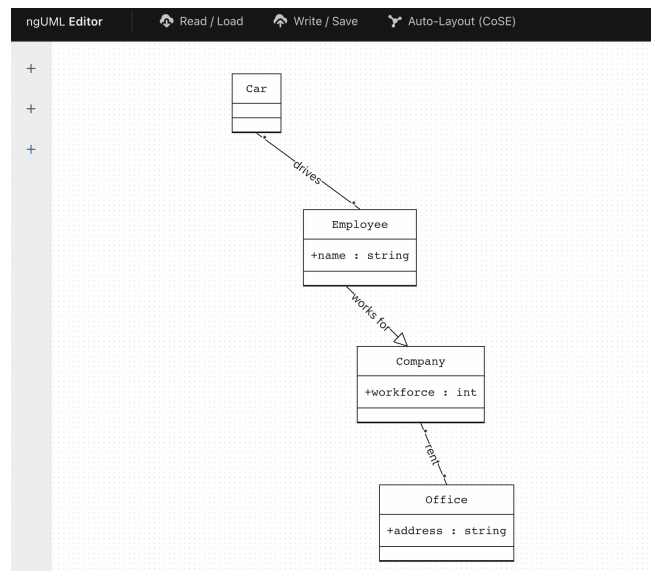


Figure 9: Diagram editor with the new attribute

4.4 Changing data type

For this test case the data type of the attribute created in the previous test case will be modified, using the chatbot:

1. Open the chat window.
2. Type and send "I want to change the data type of an attribute".
3. The bot will ask for the "parent" classifier, type and send "Company"
4. The bot will now ask which attribute of Company you want to change, type and send "workforce".

5. Now you need to provide a data type, type and send "bool".
6. The bot will now announce if the modification was success full. If this is not the case, type and send "cancel" and retry. If this is the case reload the data, with the same steps as for the first test case. Your model should now have the changed data type for the attribute in the class Company:

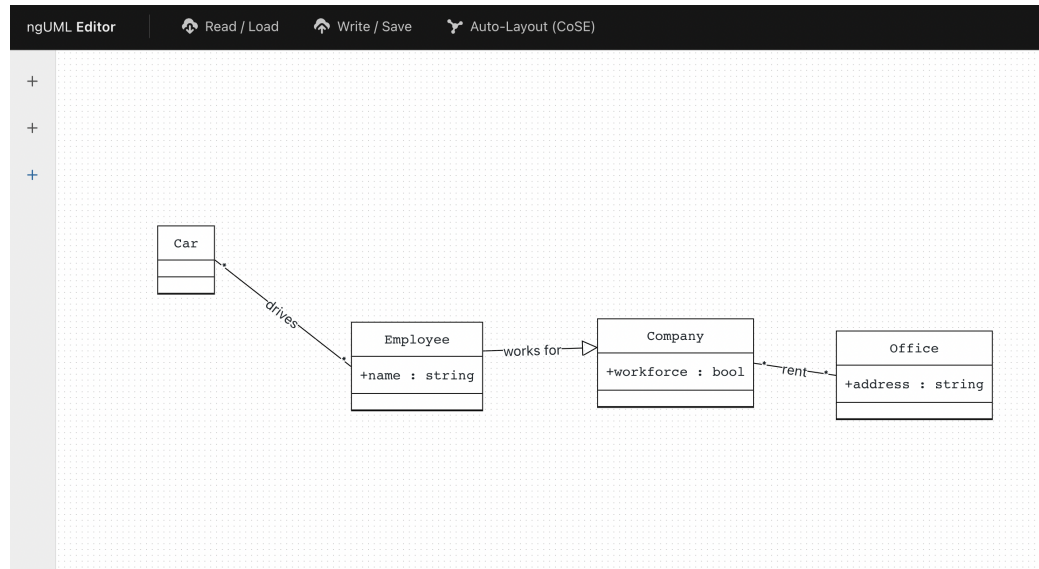


Figure 10: Diagram editor with the changed data type of the attribute

4.5 Informative functionality

For the requirements of the system it is important that the chatbot can help the user, also if the user has limited knowledge of UML diagrams. That is why the chatbot can provide definitions of UML and its components, that will be tested in this test case:

1. Open the chat window.
2. Type and send "Can you tell me what UML is?".
3. The bot should give a short definition/ explanation of UML. The figure below shows what the conversation could look like:

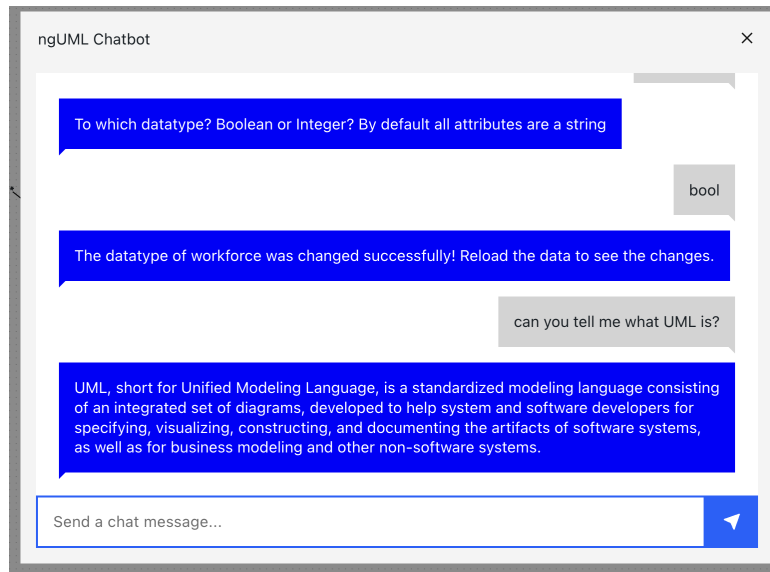


Figure 11: Asking the chatbot about the definition of UML

4. Type and send "I want to know what a classifier is"
5. The bot should now give a short definition/ explanation of classifiers for UML class models.

That was the last test case. Many thanks for taking the time to participate in this validation process. The only thing that you have to do now is answer a short questionnaire of 11 questions about your experience with the chatbot. The link to this survey will be shared with you.

C Questionnaire

Chatbot_survey

Start of Block: Default Question Block

Q1 The instructions in the document were sufficient for the testing

- ☐ Strongly disagree (1)
 - ☐ Somewhat disagree (2)
 - ☐ Somewhat agree (3)
 - ☐ Strongly agree (4)
-

Q2 Creating the new project was successful

- ☐ Strongly disagree (1)
 - ☐ Somewhat disagree (2)
 - ☐ Somewhat agree (3)
 - ☐ Strongly agree (4)
-

Q4 Testcase 1 (adding a new relationship) was successful

- ☐ Strongly disagree (1)
- ☐ Somewhat disagree (2)
- ☐ Somewhat agree (3)
- ☐ Strongly agree (4)

Q5 Testcase 2 (adding a new attribute) was successful

- ☐ Strongly disagree (1)
 - ☐ Somewhat disagree (2)
 - ☐ Somewhat agree (3)
 - ☐ Strongly agree (4)
-

Q6 Testcase 3 (changing a data type) was successful

- ☐ Strongly disagree (1)
 - ☐ Somewhat disagree (2)
 - ☐ Somewhat agree (3)
 - ☐ Strongly agree (4)
-

Q7 Testcase 4 (informative function) was successful

- ☐ Strongly disagree (1)
 - ☐ Somewhat disagree (2)
 - ☐ Somewhat agree (3)
 - ☐ Strongly agree (4)
-

Q8 Overall I think the testing was successful

- ☐ Strongly disagree (1)
 - ☐ Somewhat disagree (2)
 - ☐ Somewhat agree (3)
 - ☐ Strongly agree (4)
-

Q12 This system can help make the project more accessible for people with less UML knowledge

- ☐ Strongly disagree (1)
 - ☐ Somewhat disagree (2)
 - ☐ Somewhat agree (3)
 - ☐ Strongly agree (4)
-

Q13 This system can help make the project more user-friendly

- ☐ Strongly disagree (1)
 - ☐ Somewhat disagree (2)
 - ☐ Somewhat agree (3)
 - ☐ Strongly agree (4)
-

Q9 This system is a relevant addition to the project

- ☐ Strongly disagree (1)
 - ☐ Somewhat disagree (2)
 - ☐ Somewhat agree (3)
 - ☐ Strongly agree (4)
-

Q10 I would recommend using the chatbot

- ☐ Strongly disagree (1)
 - ☐ Somewhat disagree (2)
 - ☐ Somewhat agree (3)
 - ☐ Strongly agree (4)
-

Q11 If you have any comments/ additions please fill them in below:

End of Block: Default Question Block
