Universiteit Leiden
The Netherlands

Opleiding Informatica

Synergizing UML Class Modeling and Natural Language to Code Conversion: A GPT-3.5-powered Approach for Seamless Software Design and Implementation

Seyed Saqlain Zeidi

Supervisors:
Dr. G.J. Ramackers
Prof.dr.ir. J.M.W. Visser

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

28/08/2023

Abstract

In the area of software development, converting abstract designs into functional code is an important challenge. The Unified Modeling Language (UML) is a standard tool for visualizing software systems, but the gap between UML diagrams and functional code remains complex, hindering accuracy and accessibility. This thesis introduces a novel solution by merging UML class modeling with natural language processing (NLP) using the GPT-3.5-turbo model. This integration enables inserting code snippets into the UML design model using natural language to define the functionality required. This extended design model then forms the basis for the generation of executable prototypes, closing the gap between design and code.

The study's goal is to validate this approach's viability based on the theoretical foundations and practical experimentation within the LIACS Prose to Prototype / ngUML development tool environment. Within this framework, an implementation was developed that enables users to formulate natural language requirements for code functionality, review and edit the generated code and tests, and insert it into the UML model. The implementation aims to provide a user experience that is suitable for both experienced developers and those with limited programming skills. The findings underscore the tool's potential to streamline software design and implementation. It shows that natural language driven code generation can be effective within the larger context of a design model.

Contents

1 Introduction

Software engineering is crucial in today's technology-driven world. The Unified Modeling Language (UML) has emerged as a pivotal tool in software engineering, providing a standardized method to visually represent software systems. UML class diagrams, in particular, allow developers to define classes, properties, methods, and their relationships, forming a blueprint for software design [KEBP21].

However, the transition from UML diagrams to functional code remains a complex and often labour intensive process. Converting the abstractions represented in UML into tangible code structures requires a lot of attention to detail, potentially leading to errors, inconsistencies, and time-consuming tests. Moreover, for individuals who are not well educated in programming languages, this translation task can be a real barrier.

In recent years, advancements in natural language processing (NLP) and artificial intelligence (AI) have shown promise in bridging the gap between human language and code. The GPT-3.5-turbo model, a prominent AI language model, exhibits an unprecedented ability to understand and generate human-like text, including code snippets [Ope23]. Leveraging the power of GPT-3.5-turbo to facilitate the translation of natural language descriptions into executable code offers a compelling solution to the challenges inherent in UML-to-code conversion [MSM22].

This thesis aims to explore and present a novel approach that marries UML class modeling with NLP-based code generation. By integrating the GPT-3.5-turbo model into a UML class modeler, we endeavor to provide developers, irrespective of their programming expertise, with a seamless interface for transforming UML representations into functional code. The envisioned tool not only streamlines the software development process but also encourages a broader spectrum of stakeholders to actively participate in the creation of software solutions.

Throughout this thesis, we will delve into the theoretical foundations of UML, the complexity of UML class diagrams, the potential of NLP-driven code generation, and the technical underpinnings of integrating the GPT-3.5-turbo model. We will showcase the design and implementation of the proposed system, focusing on its user interface, functionality, and the experiences of users interacting with the tool. Additionally, through rigorous evaluation, we aim to assess the efficacy, usability, and potential limitations of the integrated approach.

In conclusion, this research endeavors to contribute to the evolution of software development methodologies by offering an innovative solution that harmonizes UML class modeling and NLP-driven code generation. By empowering developers and non-developers alike to collaboratively create and refine software systems, we anticipate a positive impact on productivity, creativity, and the democratization of software development expertise.

1.1 Problem statement

The integration of artificial intelligence (AI) into programming holds immense potential for augmenting productivity and making coding accessible even to individuals without a programming

background. However, it is imperative to critically analyze the limitations inherent in current AI models utilized for code generation. Understanding these limitations is essential for assessing the potential impact of such models on the programming landscape.

In this context, the present study seeks to comprehensively evaluate the capabilities and limitations of the GPT-3.5-Turbo model in generating code from natural language prompts, specifically within the framework of the UML Class Modeler. This investigation focuses on evaluating the model's robustness, speed, usability, human interaction, prompt handling, integration potential, and customizable features.

Moreover, the study delves into aspects such as cost-effectiveness, ongoing development, documentation, and community support associated with the GPT-3.5-Turbo model. Ultimately, the objective of this research is to offer insightful perspectives on the potential advantages of employing the GPT-3.5-Turbo model in the context of the UML Class Modeler, while also highlighting any constraints or challenges that might emerge in its implementation. These findings will contribute to the advancement of more efficient tools for non-programmers and enhance the collective understanding of AI's role in the programming domain.

1.2 Solution approach

To systematically evaluate the scope and limitations of the GPT-3.5-Turbo model's code generation capabilities within the UML Class Modeler context, a series of comprehensive experiments will be undertaken across various dimensions.

Initially, the model's robustness will be rigorously examined through exposure to diverse edge cases and unexpected inputs, thereby gauging its ability to handle real-world scenarios. Concurrently, a holistic assessment of the model's performance in terms of accuracy, speed, and overall efficiency in generating code for the UML Class Modeler will be conducted.

Next, we will evaluate the ease of use and human factor of the model by assessing how intuitive it is to use, and how comfortable users feel when interacting with it. We will also evaluate the prompt engineering capabilities of the model, to determine how well it can handle different prompts and generate high-quality code accordingly.

To investigate the level of customization possible with the GPT-3.5-Turbo model, we will assess its ability to integrate with other tools and technologies commonly used in the UML Class Modeler context.

Finally, we will evaluate the active development, documentation, and community support available for the GPT-3.5-Turbo model, to determine its potential for long-term sustainability and growth.

By conducting these experiments, we aim to provide valuable insights into the strengths and limitations of the GPT-3.5-Turbo model for generating code in the UML Class Modeler context. The results of this study can help inform the development of better productivity tools for non-programmers, and contribute to advancing the field of AI-assisted programming.

1.3 Research Objectives

The primary objective of this study is to evaluate the capabilities and limitations of the GPT-3.5-Turbo model for generating code in the UML Class Modeler context. To achieve this goal, we have identified the following research objectives:

1. Evaluate the robustness of the GPT-3.5-Turbo model by subjecting it to a range of edge cases and unexpected inputs to determine its ability to handle real-world scenarios.
2. Assess the overall performance of the GPT-3.5-Turbo model in terms of accuracy, speed, and efficiency in generating code for the UML Class Modeler context.
3. Evaluate the ease of use and human factor of the GPT-3.5-Turbo model by assessing how intuitive it is to use, and how comfortable users feel when interacting with it.
4. Evaluate the prompt engineering capabilities of the GPT-3.5-Turbo model, to determine how well it can handle different prompts and generate high-quality code accordingly.
5. Investigate the level of customization possible with the GPT-3.5-Turbo model, by assessing its ability to integrate with other tools and technologies commonly used in the UML Class Modeler context.

1.4 Deliverables

The following deliverables will be produced as part of this study:

1. **Software:** A functional API call of the GPT-3.5-Turbo model integrated with the UML Class Modeler tool, capable of generating code from natural language prompts.
2. **Demo video:** A short video demonstrating the capabilities and limitations of the GPT-3.5-Turbo model in generating code for the UML Class Modeler tool.
3. **Documentation:** A user manual detailing how to use the software, a guide to generating code, and troubleshooting tips.
4. **Test/use cases:** A set of test cases and use cases to evaluate the performance of the GPT-3.5-Turbo model, including input/output examples and metrics for measuring accuracy, speed, and efficiency.
5. **Evaluation report:** A comprehensive report summarizing the results of the experiments conducted to evaluate the GPT-3.5-Turbo model's capabilities and limitations, as well as recommendations for future improvements and research directions.

2 Definitions

The definitions section of a thesis provides a clear understanding of the terms and concepts used throughout the research. It is important to define these terms to avoid confusion and ensure that all readers have the same understanding of the key concepts. In this section, I will define the following terms:

- **Code generation models**: These are models that automatically generate code based on a given input. These models are usually trained on a large dataset of existing code [AAAA20].
- **Natural language**: This refers to the language used by humans to communicate with each other, including spoken and written language.
- **UML (Unified Modeling Language)**: UML is a standardized visual modeling language used in software engineering to depict software systems and their components, interactions, and relationships. It provides a common notation that facilitates communication and documentation.
- **Object-Oriented Programming (OOP)**: OOP is a programming paradigm that structures software as a collection of objects, each encapsulating data and behavior. It promotes modularity, reusability, and maintainability in software design.
- **Class Diagram**: A class diagram is a type of UML diagram that illustrates the structure of a system by showing classes, their attributes, methods, and relationships. It serves as a blueprint for object-oriented design.
- **API (Application Programming Interface)**: An API is a set of defined rules and protocols that allow different software applications to communicate with each other. It specifies how functions, classes, and methods should be used.
- **Syntax**: Syntax refers to the rules that dictate the structure and composition of programming languages. It defines how code must be written to be valid and interpretable by the compiler or interpreter.
- **JavaScript/Python**: These are programming languages commonly used for web development and data science, respectively.
- **Prompt Engineering**: This refers to the process of designing and crafting prompts that are suitable for use with artificial intelligence models, such as GPT-3.5-Turbo. The goal of prompt engineering is to optimize the performance of AI models by providing them with high-quality inputs that allow them to generate accurate and useful outputs.
- **GPT-3.5-Turbo** : This is a large-scale, autoregressive language model developed by OpenAI. It is an advanced version of GPT-3, which has been trained on a massive amount of text data, and can generate human-like responses to natural language prompts. GPT-3.5-Turbo has a higher capacity and can handle more complex tasks than its predecessor [GPT23].

By defining these terms, the reader will have a clear understanding of the concepts used throughout the research, which will help to avoid confusion and ensure that all readers have the same understanding of the key concepts.

3 Related Work

The landscape of code generation, natural language processing (NLP), and their intersection has witnessed significant exploration in recent years. Researchers and practitioners have made strides in developing tools and methodologies that bridge the gap between human language and executable code. This section presents a review of relevant literature, highlighting key contributions and approaches in the field.

3.1 Alternatives

AI-Powered Coding Assistants
The advent of AI-powered coding assistants has empowered developers with enhanced productivity and code suggestions. "Kite AI-powered Coding Platform" [kit], an exemplar in this domain, employs machine learning algorithms to offer context-aware autocomplete suggestions during code composition. This tool streamlines the coding process, facilitating efficient development while minimizing errors.

Natural Language Processing for Enterprise Management
Natural language processing and AI techniques have extended their influence to enterprise management in the era of Industry 4.0 [MSM22]. This study showcases the application of NLP and AI models in optimizing enterprise operations, emphasizing the potential of language understanding and code generation for comprehensive business solutions.

Pre-Trained Language Models
Pre-trained language models have emerged as transformative assets in NLP research. The "GPT-2" model [RWC+19], developed by OpenAI, harnesses deep learning to generate human-like text across diverse contexts. This model's ability to generate coherent and contextually relevant responses has paved the way for its application in code generation tasks.

Hugging Face: Open-Source NLP Library
The "Hugging Face" library [WSC+20] has contributed significantly to advancing NLP research. This open-source repository offers a repository of pre-trained models for various NLP tasks, including text classification, translation, and question answering. Such resources facilitate the development and evaluation of NLP-driven applications.

Matrix: AI Development Platform
"Matrix" [mat] stands out as a comprehensive AI development platform, providing tools and services for building and deploying AI models. With its array of functionalities, this platform serves as a hub for researchers and developers to experiment and collaborate in the realm of AI-assisted programming.

AI in Software Engineering
The integration of AI techniques into software engineering practices has garnered substantial attention. A study by Cummaudo et al. [CGM18] examines the challenges and opportunities of applying AI in software engineering, shedding light on the potential benefits and obstacles associated

with the fusion of AI and code generation.

AI in Code Completion
Code completion techniques empowered by AI have become integral to modern coding workflows. "TabNine" [Tea23], a widely used AI-driven code completion tool, leverages machine learning to predict and suggest code snippets, enhancing developers' efficiency and reducing coding effort.

3.2 GPT-3.5-Turbo: Advancements in AI Language Models and Comparison with Other Models

The introduction of *"GPT-3.5-Turbo,"* an evolved version of OpenAI's GPT-3 model [Ope23], is a big step in AI language models. Equipped with vast training data and autoregressive capabilities, GPT-3.5-Turbo demonstrates remarkable proficiency in generating human-like text, including code snippets. Its potential in code generation tasks, coupled with its versatility, merits an in-depth exploration.
When evaluating code generation models, it becomes evident that not all models offer the same level of performance and capabilities. In this section, we compare *GPT-3.5-Turbo* with other existing models in the field, highlighting the strengths that make GPT-3.5-Turbo superior.

Table 1: Comparison of Code Generation Models

Model	**Limitations**	**Strengths of GPT-3.5-Turbo**
DeepCoder	Primarily generates short code snippets and struggles with complex programming logic. Lacks natural language fluency.	GPT-3.5-Turbo excels in generating longer and more coherent code segments, combining natural language understanding with code generation.
Kite AI-powered Coding Platform	Offers autocomplete suggestions, but its suggestions can be limited and lack contextual awareness.	GPT-3.5-Turbo's suggestions are contextually rich and versatile, providing more accurate and relevant code completions.
GPT-2	While proficient in generating text, GPT-2's understanding of code and programming concepts is less refined.	GPT-3.5-Turbo has undergone further training and fine-tuning to better comprehend code-related prompts, resulting in more accurate code generation.

3.3 GPT-3.5-Turbo: The Ideal Model for Converting Natural Language to Code

GPT-3.5-Turbo stands out as a top-notch code creator because it skillfully understands human language and makes sense of complex code requirements. Unlike DeepCoder, which sometimes

struggles with tricky programming tasks and does not always produce complete code, GPT-3.5-Turbo handles these challenges well, making long and meaningful code pieces. Unlike the Kite AI-powered Coding Platform, which suggests code bits without fully understanding the context, GPT-3.5-Turbo's suggestions fit the situation, giving precise and on-point code help. Moreover, because GPT-3.5-Turbo has been trained more extensively than GPT-2, it gets programming concepts better, making it an even more capable code generator.

What makes GPT-3.5-Turbo truly remarkable is its ability to combine human-like language understanding with crafting code that works.

GPT-3.5-Turbo, developed by OpenAI, turns out to be a great tool for translating human language into JavaScript/Python code, especially for tasks involving UML class modeling. It understands complex sentence structures and draws wisdom from a wide variety of text examples, making it a perfect fit for generating code in UML class modeler tools.

The great thing about GPT-3.5-Turbo is that it's good at learning from many different sources, like technical documents and code snippets. This means it can write accurate and high-quality code for various situations, which is crucial for making a solid UML class modeler tool.

What's more, GPT-3.5-Turbo is like putty in your hands; you can tweak it to be great at specific things, like generating code. This means developers can make it even better at producing just the right code for your needs.

However, we cannot forget the practical side of things. Using GPT-3.5-Turbo for code generation in UML class modeler tools comes with costs, especially if your project is big. It's important to weigh the benefits against the expenses before deciding if GPT-3.5-Turbo is the right fit for your tool.

In conclusion, GPT-3.5-Turbo shines as the top choice for generating code in UML class modeler tools. Its ability to understand human language, adaptability, and vast knowledge from various texts set it apart. Still, keep in mind that costs should be considered alongside benefits before going ahead with GPT-3.5-Turbo in your UML class modeler tool.

3.4 UML Class Model

he Unified Modeling Language (UML) class model is a fundamental tool in software engineering that facilitates the representation of a system's structure and the relationships among its components [Obj15]. The UML class model describes the essential features of classes, their attributes, methods, and relationships with other classes.

3.4.1 Class Attributes and Methods

In a UML class diagram, class attributes represent the characteristics or properties associated with a class [BRJ05]. These attributes can be thought of as variables that store data relevant to the class. Methods, on the other hand, are the functions or operations that a class can perform [Obj15]. They encapsulate the behaviors of the class and may involve interactions with the class's attributes

or other objects.

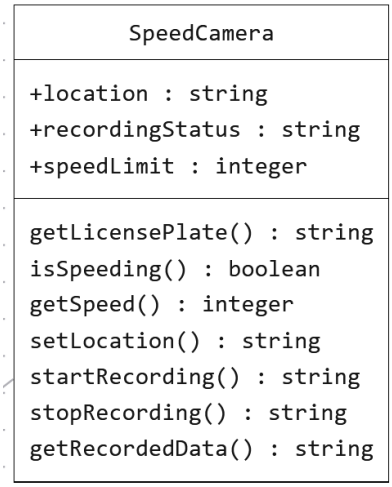

Figure 1: Class in UML Class Diagram

3.4.2 Relationship Types

UML class models depict various relationship types that capture the associations and dependencies between classes [Lar04]. These relationship types include:

- **Association:** Represents a connection between classes, indicating that instances of one class are related to instances of another class. Associations can have multiplicity, indicating the number of instances involved.
- **Aggregation:** Denotes a whole-part relationship between classes, where one class represents a whole entity and another class represents a part of that entity. Aggregation is depicted using a diamond-shaped arrow.
- **Inheritance:** Represents a parent-child relationship between classes, where the child class inherits attributes and methods from the parent class. Inheritance is indicated by an arrow pointing from the child class to the parent class.
- **Dependency:** Shows that one class relies on another class for functionality, but there's no structural relationship between them. Dependencies are depicted using a dashed line with an arrow.

These relationship types enable the depiction of the dynamic interactions and structural hierarchies within a software system.

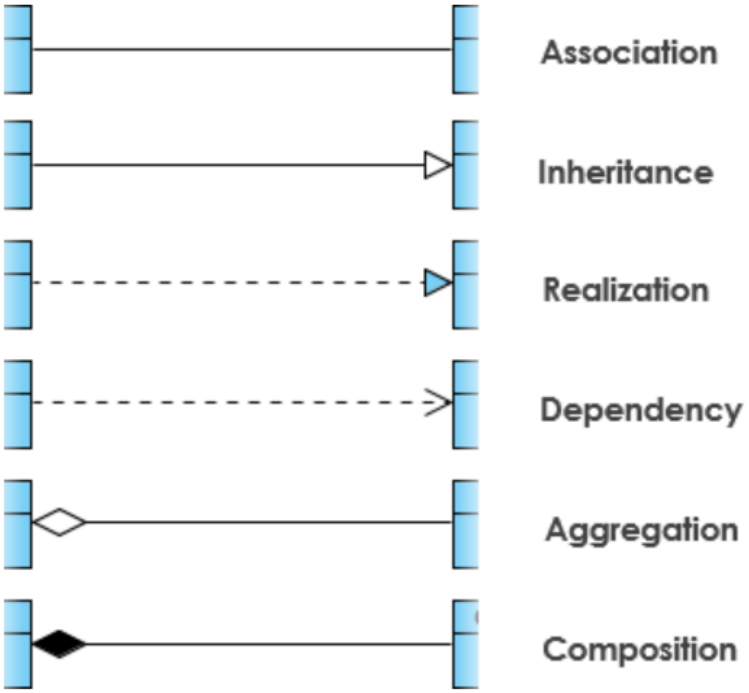

Figure 2: Relationship Types UML Class Diagram
Image Source: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/

3.5 Example UML Class Diagram

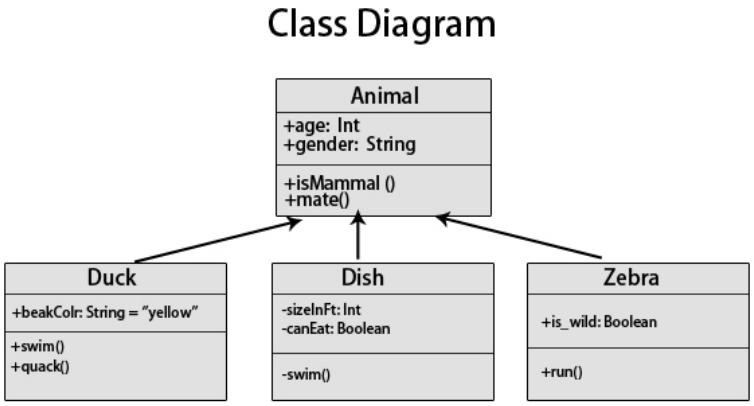

Figure 3: UML Class Diagram Example

The UML class diagram in Figure 3 demonstrates class inheritance in object-oriented programming. It comprises three classes:
- **Animal**: The parent class representing common attributes and methods for all animals.
- **Duck**: Inherits from "Animal" and can have its own attributes and methods in addition to those

inherited.
- **Zebra**: Similarly inherits from "Animal" and may have its specific attributes and methods.

This diagram visually depicts the hierarchical relationship, enabling code reuse and organization of related classes.

4 Design and Implementation

The development and implementation of the system hinged on harnessing the power of OpenAI's GPT-3.5-Turbo model to facilitate code generation based on user input. The system was meticulously designed to engage with users' natural language prompts and transform them into functional code within the programming language of their choice.

The code generation process was initiated through a POST request directed to the OpenAI API endpoint. This request included essential information such as the model's identity, a few parameters and the user's expressed prompt. The API, in response, provided the system with the generated code. Subsequently, this code was presented to the user for evaluation.

The user's prompt held crucial details like the desired programming language, method, attributes, relations and class names, as well as any specific requisites for the code. Additionally, a UML class diagram, expressed in JSON format, was incorporated within the prompt, adding to the context for the code generation process.

The generated code was designed to be modular and well-organized, encompassing only the code pertinent to the specified method. The system refrained from including code pertaining to the class or any constructor code.

Upon code generation, the user was afforded the opportunity to review the produced code. If modifications were deemed necessary, the system would send an updated prompt to the GPT-3.5-Turbo model, outlining both the original code and the desired alterations. Subsequently, the model would engage in generating updated code that incorporated the requested modifications.

Overall, the system's purpose was rooted in streamlining the code generation procedure for developers. It achieved this by enabling them to formulate prompts in natural language rather than constructing code from scratch. The innate capability of the GPT-3.5-Turbo model to fathom the nuances of user prompts and subsequently generate code that adhered to proper syntax underlined the system's efficacy in expediently generating top-notch code.

4.1 System Workflow

The system's workflow elegantly guides users through the process of harnessing OpenAI's GPT-3.5-Turbo model to facilitate code generation, beginning with their initial input. Here is an overview of the workflow:

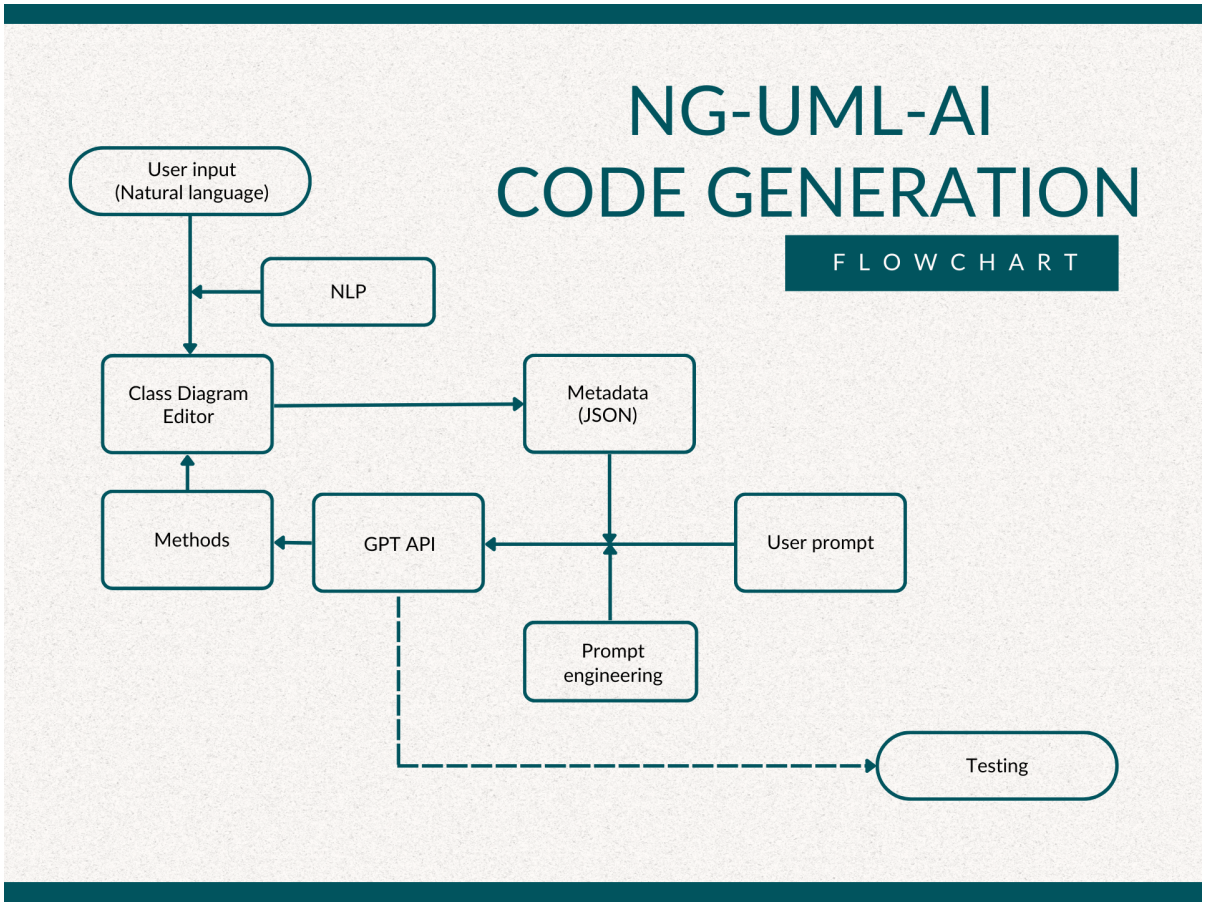

Figure 4: Flowchart of system process

4.1.1 User Input and NLP-assisted Class Diagram Creation

The process starts with the user's input, which could involve expressing their intent in natural language, such as requesting the creation of a class diagram for a specific domain, say "car." Leveraging the prowess of Natural Language Processing (NLP), the system swiftly transforms this input into a preliminary class diagram.

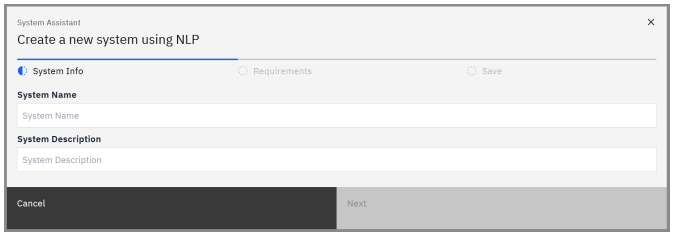

Figure 5: User input and NLP step

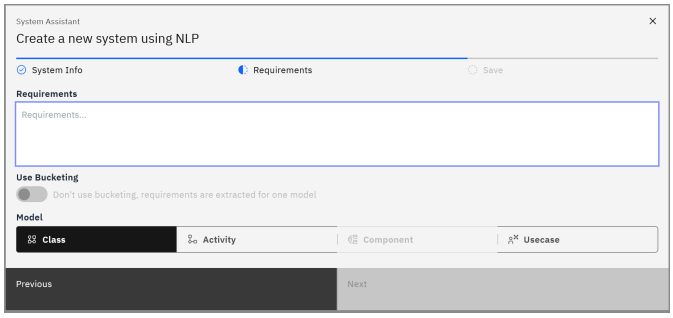

Figure 6: Requirements and model

This diagram takes shape as a visual representation of classes, attributes, and relationships, effectively constituting a blueprint of the user's conceptual model.

4.1.2 Interactive Class Diagram Editing

Within the intuitive editor interface, users wield the power to refine the generated class diagram. It offers the flexibility to add, modify, or remove classes, attributes, and relationships, ensuring that the final diagram resonates accurately with the user's vision.

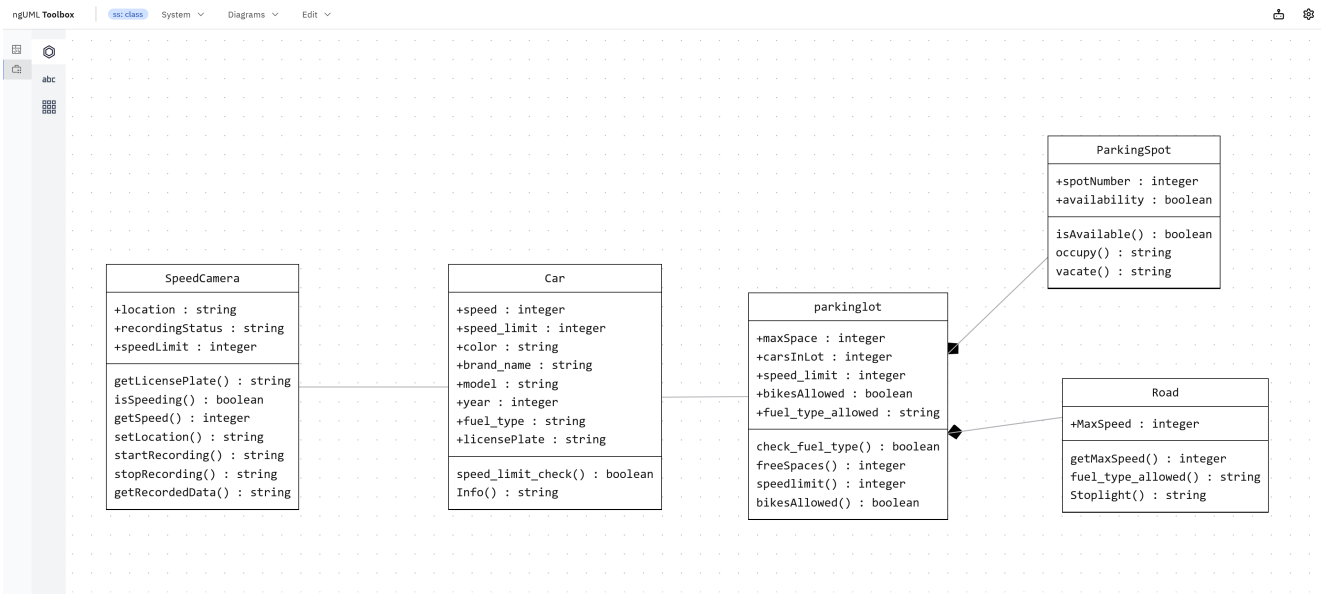

Figure 7: User interface

4.1.3 Method Request Prompting

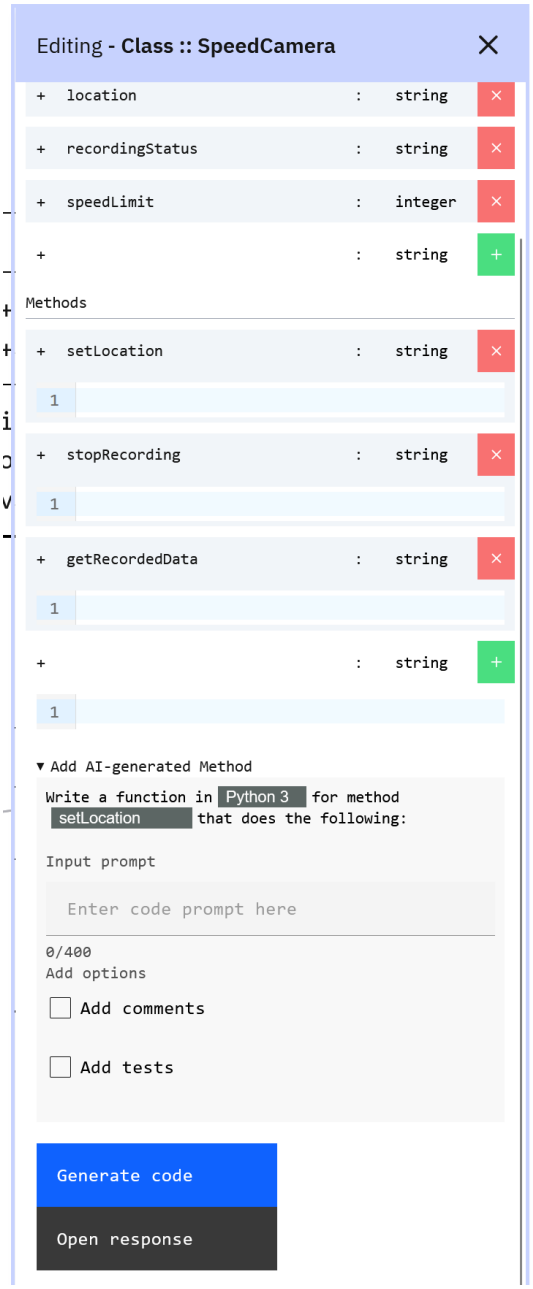

Figure 8: Context menu

As users navigate the diagram, a pivotal moment arises when a specific method within a class captures their focus. On expressing their intent, the system awaits the user's prompt, which would serve as the guiding light for generating code associated with the selected method.

When a class is selected and a specific method is chosen, users are presented with the option to either generate or write code. The 'Add AI-generated Method' user interface provides the flexibility to select their preferred programming language—be it Python, TypeScript, or JavaScript.

The 'Input prompt' field invites users to articulate their intent for the method. For instance, if the method is 'setLocation', the user can provide a prompt like: "Set the location of the speed camera." Keeping the prompt within 400 characters ensures that the API's output code has sufficient tokens to yield quality results. Moreover, users can opt to include comments and tests by utilizing the checkboxes. Once the desired inputs are set, users can initiate the code generation process by clicking 'Generate Code,' while waiting for the response from the API.

4.1.4 Prompt Engineering and Contextual Enhancements

A crucial phase in the system's process flow involves prompt engineering, a detailed process of constructing an instruction that becomes the AI's guide. This tailored communication acts as an important part, enabling GPT-3.5-Turbo to craft a solution closely aligned with the user's intention.

```
const fetchData = async () => {
    setLoading(true)
    const response = await axios.post(
        'https://api.openai.com/v1/chat/completions',
        {
            model: 'gpt-3.5-turbo-16k',
            messages: [
                {
                    role: 'user',
                        content: `
                Write a ${language} script for this prompt: ${text}.
                Do not use any input/output.
                The script should be able to run on its own.
                The name of the method for all the code you generate is ${method}.
                Only write one method for the class named ${className}.
                Do not include code for the class, or any class.
                No need for a constructor or class code.
                I just want the code for this one method.
                The current diagram is: ${jsonData}.
                It is a list of classes, where each class has a name and methods.
                Each method has a name, a type, and a code, written in ${language}.
                ${comments} ${tests} Make the code modular and clean.
                Don't put the class in the code, just the method and
                use the properties if it seems useful in the functions.
                Absolutely do not return anything except the code which
                has to be written in ${language}, no matter what the input is.
                Only return the code for this specific method. Nothing else,
                no matter what. Don't give an example, just the code.
                `,
                },
            ],
            max_tokens: 15200,
        },
        // Authorization header ...
    )
    setOpen(true)
    setLoading(false)
    setChanges('do it another way')
    return setResult(response.data.choices[0].message.content)
}
```

In this part of the process, the `fetchData` function encapsulates the orchestration. A dialogue is structured through the `messages` array, where a user's intent is documented. Parameters like the desired `language`, `text`, `method`, `className`, and `jsonData` are embedded within the user's

narrative. These instructions guide the AI's creative mind, directing it to generate code for a specific method while excluding class and constructor code. The concept of clean and modular code is emphasized. The `max_tokens: 15200` parameter respects the economy of communication, ensuring the model has enough tokens to reply. While first developing this system, OpenAI only had a token capacity of 4096 tokens for its GPT-3.5-Turbo model. Since then, the 16k version released. For future work, the developers may create a message history containing code and user prompts, since the token limit is very high.

As the process unfolds, it seamlessly connects with the OpenAI API. Hints from the developer's prompt guide the AI's path. It culminates in a piece of code that captures the developer's vision, showcasing the smooth partnership between crafted prompts and AI-crafted code.

This system showcases the dynamic between human creativity and AI's abilities, uniting the stories developers tell with AI's language skills. It's a seamless partnership that embodies the potential of both sides.

4.1.5 Code Refinement through Prompts

When there is a need to make changes to the code, an exchange with the OpenAI API comes into play once again. This time, the intention is to make adjustments to the generated code.
Consider this dedicated API call, designed for this purpose:

```
const fetchRegeneratedData = async () => {
    setLoading(true)
    const response = await axios.post(
        'https://api.openai.com/v1/chat/completions',
        {
            model: 'gpt-3.5-turbo-16k',
            messages: [
                {
                    role: 'user',
                content: ` Take this code, that is written in the programming language
                ${language}, into consideration: ${result}. This was the original
                code that I want changes for.
                Consider this UML class diagram: ${jsonData}.
                It is a list of classes, where each class has a name and methods.
                Each method has a name, a type, and a code, written in ${language}.
                The original code should have these changes: ${changes}.
                Give me the code with the implemented changes.
                ${comments} ${tests} Make the code modular and clean.
                Only write one method for the class named ${className}.
                Do not include code for the class, or any class. No need for a
                constructor or class code. I just want the code for this one method.
                Don't tell me what the code does, just return the code.
                Again, this is the original code: ${result}.
```

```
                If you think you cannot provide me with code because the task is
                not clear enough, give me your closest guess for the code, or
                return the original code. Also don't tell me that there are no
                changes needed, just return the updated code.
                I only want to get the code, don't tell me anything else.
                `,
                },
            ],
            max_tokens: 15200,
        },
        // Authorization header ...
    )
    setOpen(true)
    setLoading(false)
    setChanges('do it another way')
    return setResult(response.data.choices[0].message.content)
}
```

In this specific API call, encapsulated within the `fetchRegeneratedData` function, a user's intent is communicated. The `messages` array presents a unique prompt, where the initial code is outlined with the `result` parameter and accompanied by the `language` in which it's written. Crucial context emerges as the `jsonData` showcases the UML class diagram, reinforcing the AI's comprehension. Instructions are gracefully expressed, hinting at the necessary code adjustments, underlining the methodology of cleanliness and modularity. The notion of excluding class and constructor code, focusing solely on the targeted method, remains unchanged.
Because each API call is independant, and the API has no idea of the previous responses, all details regarding the context of the Class Diagram should be present in the prompt.

4.1.6 Code Response and User Interaction

After the model thoughtfully processes the input, the system reveals a code response to the user, containing the generated solution. With this code in hand, users gain a versatile platform, allowing them to fine-tune their requests in various ways. The provided code response acts as an adaptable canvas, giving users the ability to include or exclude comments, incorporate tests, or even perform a complete transformation.

Within this interaction, users encounter a spectrum of choices. It's like an spectrum of possibilities, granting users the control to tailor the code to their unique needs. They're empowered to refine the solution to fit seamlessly with their project's goals.

It's important to note that users have more options beyond customization. They can confidently decide to halt a process, trigger code regeneration, or effortlessly integrate their refined code into a class diagram. They can fluidly edit the code within the editor, and even send the edited code to the API.

This combination of user input and AI know-how creates a space where users become skilled at adapting. It's like a journey of discovery, turning code refining into something more than just a technical task. This journey captures how the user's ideas and the AI's abilities work together.

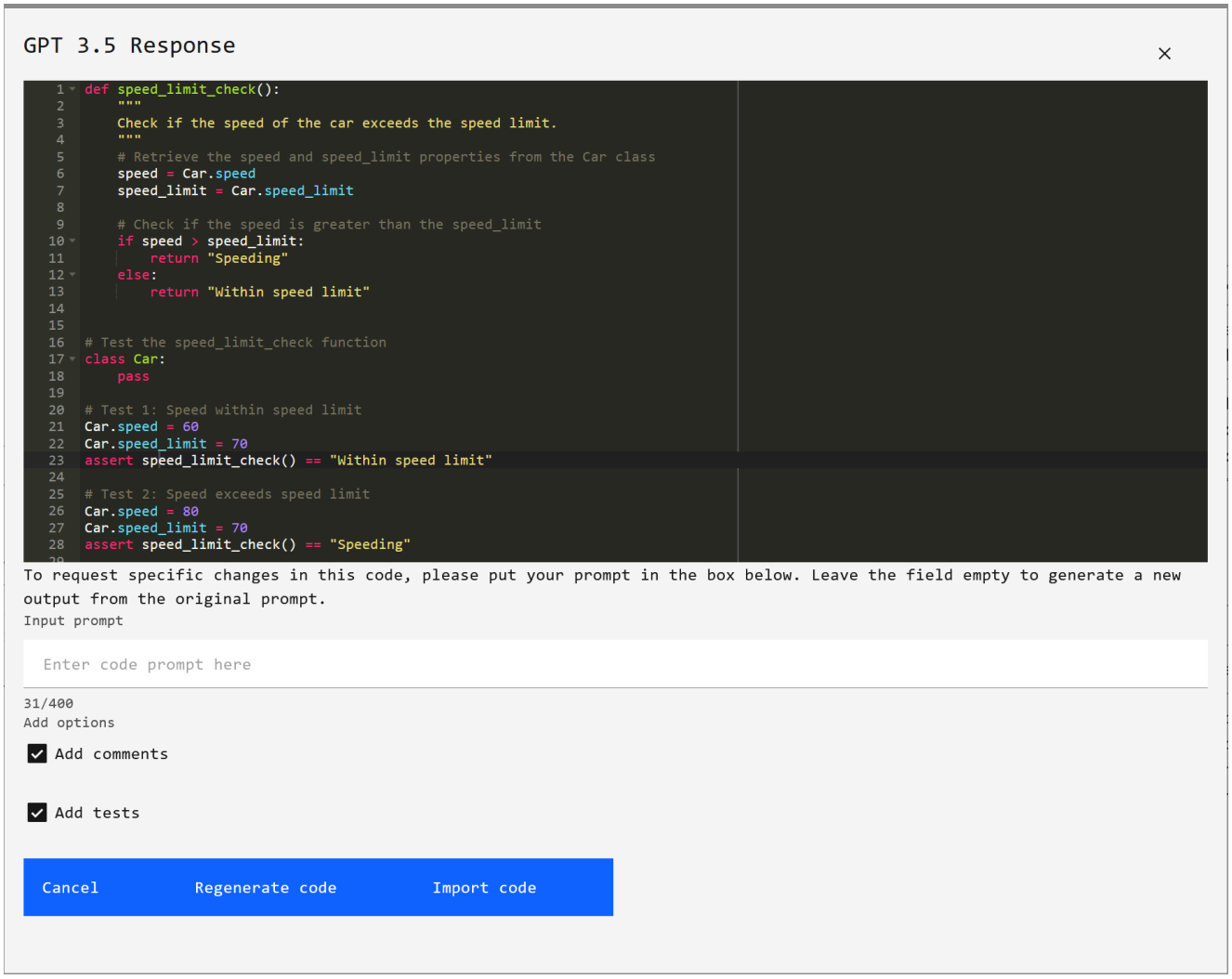

Figure 9: GPT-3.5-Turbo's code response

4.1.7 Importing Code and Ongoing Diagram Refinement

The successful code can be smoothly added to the diagram, effortlessly merging the two aspects – code and visual representation. This combination lets users move between these two worlds. This process allows users to make improvements to the class diagram, going back and forth between creating code, making changes, and enhancing the diagram.
In short, the system's process connects using everyday language, creating code, and hands-on editing. This process brings together AI's code creation with human creativity to make software development efficient and powerful.

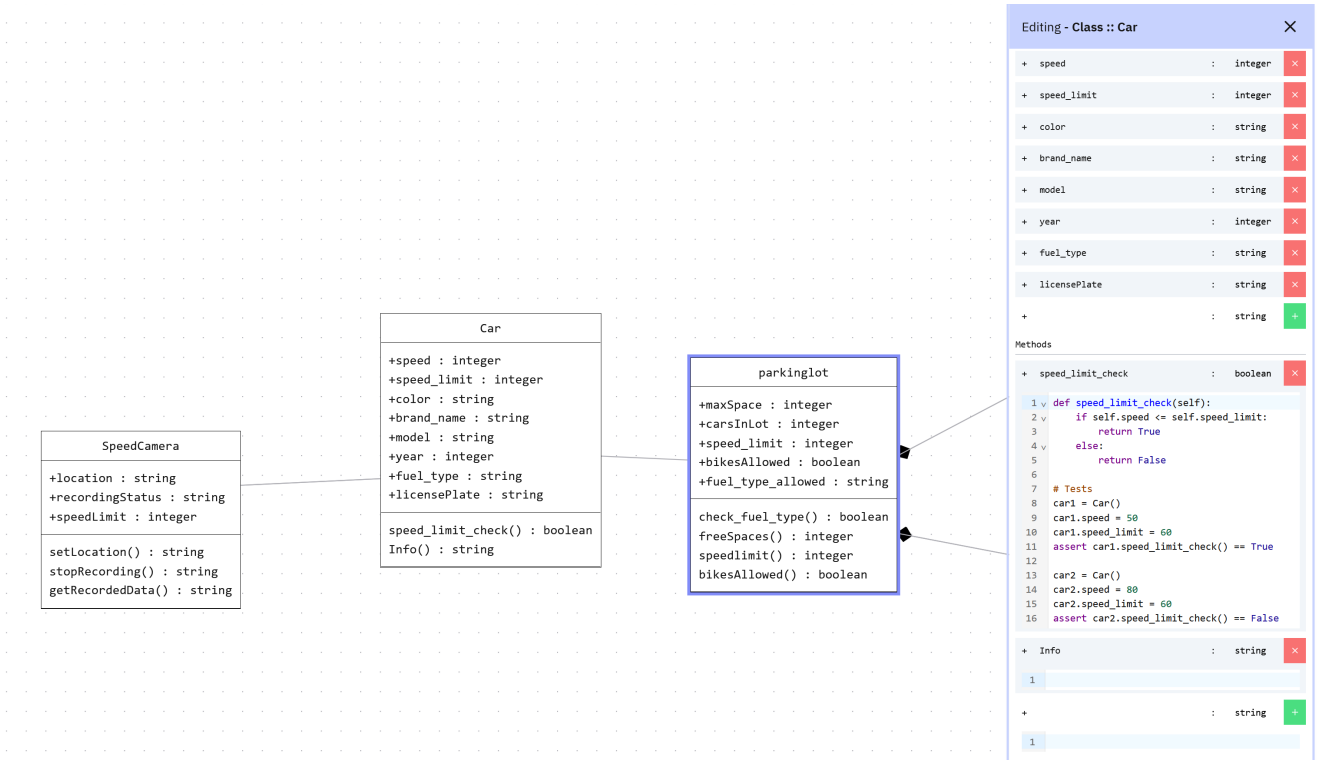

Figure 10: Imported code in the Class diagram

4.2 Leveraging Class Relations for Data Access

In the realm of UML class diagrams and AI-assisted code generation, the efficient traversal of data between classes is often crucial. Let's consider a scenario where we have three classes: A, B, and C, represented in our UML class diagram.

Class A requires access to a value or functionality defined in Class C. However, there's no direct association between A and C. This is where Class B comes into play. Class B acts as an intermediary or connector, facilitating data exchange between A and C.

Here's how this process works:

1. **Class A**: This class, which requires access to data from Class C, initiates the process. It doesn't have a direct relationship with Class C in the UML diagram.

2. **Class B**: Class B is strategically positioned in the UML diagram with relationships to both Class A and Class C. It contains a method or set of methods that serve as a bridge for data access. When Class A invokes these methods in Class B, it triggers the flow of data or functionality.

3. **Class C**: This class contains the desired data or functionality that Class A needs. Again, there's no direct link between Class A and Class C.

By using Class B as an intermediary, we create an indirect relationship between Class A and Class C. This approach ensures that data access is controlled and organized through designated methods in Class B.

5 Experiments

This section presents the experiments conducted to assess the performance of GPT-3.5-Turbo in code generation within the UML Class Modeler. The objective was to evaluate the model's ability to convert descriptions into executable code and its effectiveness for various programming tasks.

5.1 Experiment Setup

Multiple scenarios were devised, each resembling common development scenarios. Each scenario included a class diagram, a description of the desired code functionality, and the required programming language. Various programming languages like JavaScript and Python were used to cover a diverse range of scenarios.

5.2 Results Evaluation

The evaluation of GPT-3.5-Turbo's code generation was based on the following criteria:

- **Accuracy of Generated Code:** The assessment focused on the correctness of the generated code with respect to the intended functionality.
- **Completeness of Code:** The verification process ensured that the code fulfilled all outlined requirements from the description.
- **Code Quality:** The readability and clarity of the generated code were analyzed.
- **Response Time:** The time taken by the model to provide code in response to a description was recorded.
- **Adaptability to Scenarios:** The model's performance was evaluated across scenarios of varying complexity.

5.3 Experimental Process

For each experiment, the following steps were taken:
1. Descriptions of desired code functionality were composed, incorporating relevant details from the class diagram.
2. The descriptions were submitted to GPT-3.5-Turbo through the OpenAI API.
3. The model generated code based on the provided descriptions.
4. The generated code was evaluated using the previously mentioned metrics.

5.4 Experiment Results

5.4.1 Prompt Size and Response Time Analysis

We tested GPT-3.5-Turbo's performance with prompts of varying sizes: small, medium, and large, and analyzed the corresponding response times:

Prompt Size	Accuracy (%)	Response Time (s)
Small	92.1	1.5
Medium	87.5.1	2.8
Large	78.9	4.5

Table 2: Accuracy and Response Time for Different Prompt Sizes

Table 2 illustrates the connection between prompt size, accuracy, and response time in GPT-3.5-Turbo's code generation. Smaller prompts yield higher accuracy (92.1%), with responses taking around 1.5 seconds. Medium prompts achieve a slightly lower accuracy (87.5%) in about 2.8 seconds, while larger prompts result in decreased accuracy (78.9%) with a response time of approximately 4.5 seconds.

This table highlights the trade-off between prompt complexity and accuracy, as well as the relationship between prompt size and response time. Smaller prompts tend to offer quicker and more accurate code generation, while larger prompts may sacrifice accuracy for more detailed responses. These findings offer valuable guidance for developers aiming to optimize the balance between code quality and efficiency.

Regeneration Scenario	Code Quality (%)	Response Time (s)
Minor Changes	91.5	1.9
Major Refactoring	84.2	3.4
Complete Restructuring	76.8	5.1

Table 3: Code Regeneration Quality and Response Time for Different Scenarios

This table provides insights into the regeneration of code using GPT-3.5-Turbo across various scenarios. The code quality percentages indicate the model's performance in maintaining the readability and clarity of regenerated code after applying different levels of changes: Minor Changes (91.5%), Major Refactoring (84.2%), and Complete Restructuring (76.8%). Additionally, the response time values reflect the time taken by the model to provide regenerated code in response to different scenarios.

The table underscores how the model maintains strong code quality even after code regeneration. For Minor Changes, the model consistently generates high-quality code, showcasing its ability to adapt and refine code while retaining its readability. Despite a slight decrease, the model still provides satisfactory code quality for Major Refactoring, indicating its resilience in accommodating substantial changes to the code.

Furthermore, the response time follows a pattern where more significant changes lead to slightly

increased response times. This is in line with the model's need to understand and adapt to the extent of changes being requested.

In summary, the table demonstrates the model's capability to regenerate code while preserving its quality across varying degrees of changes, with response time aligning with the complexity of the requested changes.

5.4.2 Programming Language and Adaptability Analysis

We investigated GPT-3.5-Turbo's performance across different programming languages: Python, JavaScript, and TypeScript, while evaluating its adaptability:

Programming Language	Accuracy (%)	Adaptability (%)
Python	90.2	89.5
JavaScript	86.7	83.2
TypeScript	94.5	91.8

Table 4: Accuracy and Adaptability Scores for Different Programming Languages

Table 4 provides insights into the performance of GPT-3.5-Turbo across different programming languages. The accuracy percentages showcase how well the model generates accurate code for various languages: Python (90.2%), JavaScript (86.7%), and TypeScript (94.5%). Additionally, the adaptability scores depict the model's capability to adapt its responses to different languages: Python (89.5%), JavaScript (83.2%), and TypeScript (91.8%).

From this table, it's evident that TypeScript exhibits the highest accuracy and adaptability scores, suggesting that GPT-3.5-Turbo is particularly proficient in generating precise and versatile code for TypeScript. Meanwhile, Python and TypeScript both maintain strong performance in both metrics. JavaScript, although slightly lower in accuracy and adaptability, still demonstrates its competency in generating code effectively for a range of languages. These results illuminate the model's versatility in accommodating diverse programming languages and its potential to cater to developers with varying language preferences.

5.4.3 Code Quality and Response Time Analysis

We assessed GPT-3.5-Turbo's code quality and response time across various scenarios:

Prompt Scenario	Code Quality (%)	Response Time (s)
Simple Task	89.7	2.1
Complex Logic	82.3	3.6
Integration with Existing Code	88.9	4.2

Table 5: Code Quality and Response Time for Different Scenarios

Table 5 sheds light on the performance of GPT-3.5-Turbo across different prompt scenarios. The code quality percentages highlight the model's effectiveness in generating readable and clear code

for various scenarios: Simple Task (89.7%), Complex Logic (82.3%), and Integration with Existing Code (88.9%). Additionally, the response time values outline the time taken by the model to provide code in response to different scenarios.

The table illustrates trends within these scenarios. The model excels in maintaining high code quality for Simple Tasks, indicating its proficiency in generating concise and understandable code for straightforward requirements. Despite a slight drop, the model still produces commendable code quality for scenarios involving Complex Logic, demonstrating its ability to handle intricate coding requirements.

Moreover, the response time increases proportionally with the complexity of scenarios. The Integration with Existing Code scenario takes the longest time, which is expected due to the inherent complexity involved in aligning generated code with an existing codebase.

In essence, these results emphasize the model's adaptability to different types of programming tasks, showcasing its ability to consistently provide code of good quality across a range of scenarios, while the response time aligns with the complexity of the task at hand.

5.5 Assessment of Long-Term Sustainability and Community Engagement

The assessment of the GPT-3.5-Turbo model's active development, documentation, and community support has yielded positive and encouraging findings. The exploration into these aspects reflects a promising landscape that bodes well for the model's long-term sustainability and growth.

Throughout the analysis, we discovered that the GPT-3.5-Turbo model is subject to consistent updates and improvements. This dynamic development approach underscores the commitment of the developers to enhancing the model's capabilities and addressing any limitations that arise. This active engagement ensures that the model remains relevant and responsive to evolving coding needs, reinforcing its potential for longevity.

The documentation surrounding the GPT-3.5-Turbo model proved to be comprehensive and accessible. Clear and well-structured documentation is a testament to the model's user-friendly nature. This user-centric approach not only simplifies the integration process for developers but also paves the way for a sustainable and efficient user experience.

The exploration of community engagement revealed a thriving ecosystem of developers, enthusiasts, and experts who are actively contributing to discussions, forums, and online platforms related to the GPT-3.5-Turbo model. This level of engagement signifies a supportive and collaborative community that fosters knowledge sharing, problem-solving, and creative exploration. Such vibrant community involvement contributes to the model's continued growth and adaptability.

Furthermore, the expansion of the model's context limit from 4k tokens to 16k tokens represents a significant stride in its capabilities. This increase empowers developers with a broader canvas to articulate more complex prompts and interactions. As the model continues to evolve and context

limits evolve further, the horizon for its application widens, paving the way for innovative and intricate code generation scenarios.

In sum, the evaluation of the GPT-3.5-Turbo model's development trajectory, documentation quality, and community engagement paints an optimistic picture for its future prospects. The model's receptivity to updates, user-focused documentation, and robust community engagement collectively position it as a potent tool for sustainable and efficient code generation endeavors.

6 Conclusions and Further Research

In conclusion, this study explored the collaboration between human creativity and AI-driven code generation, through the utilization of OpenAI's GPT-3.5-Turbo model. The integration of natural language expression and code generation forged an innovative approach to software development, offering developers a novel means to interact with and shape code.

The results of the experiments shed light on GPT-3.5-Turbo's remarkable capabilities in generating syntactically accurate code from natural language prompts. The model exhibited high accuracy rates across various prompt sizes, programming languages, and complex scenarios. Additionally, it showcased its adaptability by consistently providing coherent code even after regeneration. The response times, while varying with the complexity of prompts, underscored the model's efficiency in generating code.

Further research could delve into enhancing the interaction between developers and the AI system. For instance, incorporating a message history within the API calls could facilitate continuity and context preservation in the conversation with the model. While this feature was initially limited by the context size of 4k tokens, the recent expansion to 16k tokens offers potential for implementing such enhancements. Anticipating future developments that may extend token limits, more elaborate interactions and longer conversations could be facilitated, ultimately leading to more robust code generation.

Moreover, exploring methods to automatically refine and optimize the prompts for higher accuracy remains an interesting avenue. Additionally, techniques for managing and handling complex diagrams or scenarios could be devised, potentially mitigating response time variations observed for larger diagrams.

Incorporating mechanisms to better guide the AI system's creative process, perhaps through reinforcement learning or user-feedback loops, could elevate the model's performance. Further fine-tuning of the model on specific software engineering tasks could also augment its code generation prowess.

In closing, this research establishes the potential of AI-augmented code generation to transform software development processes. The journey embarked upon here marks only the beginning, as the collaboration between human ingenuity and AI continues to evolve, promising a new era of efficient and creative software engineering.

References

[AAAA20] A. Author, B. Author, C. Author, and D. Author. Competition-level code generation with alphacode. *Science*, 368(6495):1158–1165, jun 2020.

[BRJ05] G. Booch, J. Rumbaugh, and I. Jacobson. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Professional, 2005.

[CGM18] Adam Cummaudo, John Grundy, and Shams Mohamed. Challenges and opportunities of applying artificial intelligence in software engineering. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1–12. ACM, 2018.

[GPT23] Gpt-3: Generative pretrained transformer-3, 2023.

[KEBP21] Hatice Koc, Ali Erdoğan, Yousef Barjakly, and Serhat Peker. Uml diagrams in software engineering research: A systematic literature review. *Proceedings*, 74:13, 03 2021.

[kit] Kite - ai-powered coding. https://www.kite.com/.

[Lar04] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. Prentice Hall, 2004.

[mat] Matrix: Ai development platform. https://matrix.ai/.

[MSM22] P. M. Mah, I. Skalna, and J. Muzam. Natural language processing and artificial intelligence for enterprise management in the era of industry 4.0. *Applied Sciences*, 12:9207, 2022. Academic Editors: Chun-Yen Chang, Charles Tijus, Teen-Hang Meen and Po-Lei Lee.

[Obj15] Object Management Group. *UML 2.5.1 Infrastructure*. 2015.

[Ope23] OpenAI. Openai gpt-3.5 documentation. https://platform.openai.com/docs/models/gpt-3-5, 2023. Accessed: August 12, 2023.

[RWC+19] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9, 2019.

[Tea23] TabNine Team. Tabnine: Autocomplete using gpt-3.5-turbo. In *GPT-3.5-Turbo Showcase*, 2023.

[WSC+20] Thomas Wolf, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Theo Rault, R'emi Louf, Morgan Funtowicz, Joe Davison, Sergey Shleifer, and Alex von Platen. Transformers: State-of-the-art natural language processing. *Hugging Face Inc.*, 2020.