Towards an Extensible Architecture for LLM-based Programming Assistants in IDEs

Albert Contreras, Esther Guerra, and Juan de Lara

Universidad Autónoma de Madrid, Spain
{albert.contreras, esther.guerra, juan.delara}@uam.es

Abstract. Large Language Models (LLMs) are the backbone of chatbots like ChatGPT, and are used to assist in all sort of domains. Following this trend, we are witnessing proposals of LLM-based assistants for coding tasks. However, current IDEs lack mechanisms tailored to facilitate the integration of such assistants, from how to interact with them to how to apply their suggestions without leaving the environment. To fill this gap, this short paper presents an extensible architecture for the definition of assistance tasks (e.g., method renaming) based on LLMs, and their binding to IDE commands and natural language prompts. We report on an ongoing effort to build a Java assistant within Eclipse based on this architecture, and illustrate its use.

Keywords: Programming assistant · Conversational agent · LLM · IDE

1 Introduction

Large Language Models (LLMs) are transformer-based neural networks trained on vast amounts of text data [8], able to generate sensible outputs upon user inputs. Many LLMs have appeared, either trained on general data (e.g., those of the GPT family, Llama, Claude) or on domain-specific data, like code (e.g., StarCoder, Code Llama) [8].

General LLMs are the backbone of chatbots such as ChatGPT or Gemini, and are used to build smart applications and assistants for all sorts of domains. Programming is no exception to this trend, with many emerging proposals of the use of LLMs to assist in programming tasks, like code generation from natural language [1,7], testing [6], or program repair [5]. Some of these assistants are integrated into popular IDEs. For example, Visual Studio and JetBrains integrate GitHub Copilot [4], and PyCharm [7] and Eclipse [3] also integrate specific assistants. However, these approaches lack an extensible architecture that facilitates the addition of LLM-based support for other programming tasks in an external way (i.e., without modifying the assistant code).

This short paper is a first step to fill this gap. It presents an extensible architecture whereby new LLM-based programming assistants can be added externally to an IDE, either bound to specific IDE commands or accessed in natural language. The approach is realised in Caret [2], an ongoing effort for a Java programming assistant within Eclipse. We illustrate the architecture by showing how to extend Caret via extension points to support new tasks.

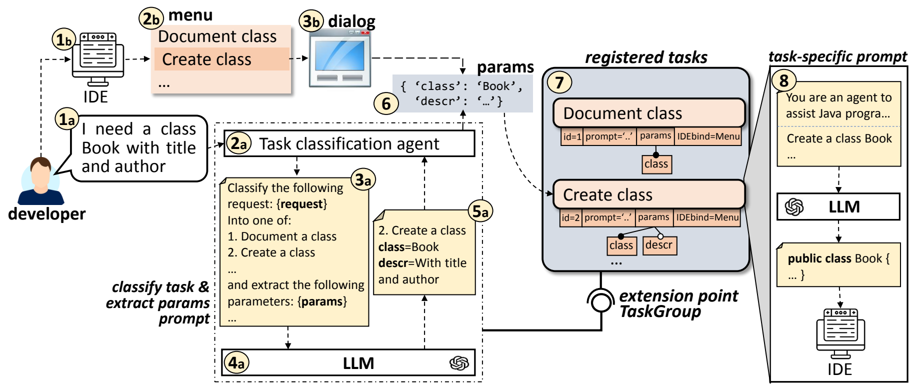


Fig. 1. Working scheme of the extensible assistant.

2 Extensible Architecture for Programming Assistants

Fig. 1 shows a scheme of our extensible architecture to integrate LLM-based assistants into IDEs. The idea is having a generic assistant that allows registering assistive tasks (e.g., document a class, create a class satisfying some domain requirements), which developers can activate on demand.

Registering an assistive task in the assistant requires specifying, among other things, how developers can request the assistance: in natural language (label 1a), or via commands of the IDE (e.g., a menu, label 1b). In the first case, the developer states to the IDE what he/she wants to achieve, and the assistant aims to deduce which registered task is suitable to accomplish it (label 2a). For this purpose, the assistant builds a prompt with the developer request and the set of registered tasks, and asking to classify the request into a task and to extract the relevant parameters from the request (label 3a). This prompt is sent to an LLM (label 4a), which outputs the classified task and the detected parameters (label 5a). For example, given the developer request "*I need a class Book with title and author*", the LLM classifies it into the task "*create a class*", and extracts the class name (*Book*) and description (*with title and author*) as parameters. The assistant can use these parameters as context for the requested task.

Then, the assistant activates the identified task, passing to it the parameters (label 6). The task creates a prompt with two parts (label 8): a general one ("*You are an agent to assist Java programming*"), and a specific one providing the task description and parameters. This prompt is sent to an LLM, which typically (though not necessarily) will output code. For instance, in the previous example, the LLM would return a Java class *Book* with attributes *title* and *author*. Finally, the assistant applies the necessary actions in the IDE to complete the task.

If the request comes from an IDE command (label 2b), the task gets uniquely determined. Hence, steps 1a to 5b are not necessary, but the developer is presented a dialog box to introduce the parameters needed for the task (label 3b), and the assistant proceeds from step 6.

Technically, assistive tasks are registered via an extension point, a mechanism present in many IDEs (e.g., Eclipse) and component-based frameworks. This allows adding arbitrary tasks in a non-intrusive way. Fig. 2 shows a conceptual model of the extension point. It defines the data needed to register assistive tasks by subclassing from TaskGroup. This class groups tasks, which can be bound to either menu commands, the usage context (e.g., when selecting a code snippet), or none (i.e., the task is activated by natural language). Each Task defines named Parameters that can be required or not, and have a type (either a programming concept such as class or attribute, or other type, like "class description"). Tasks have a description (used to build the prompt sent to the LLM Agent, cf. label 8 in Fig. 1) and perform Actions.

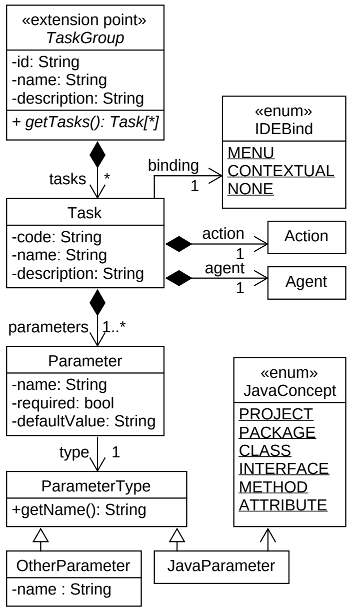


Fig. 2. Extension point.

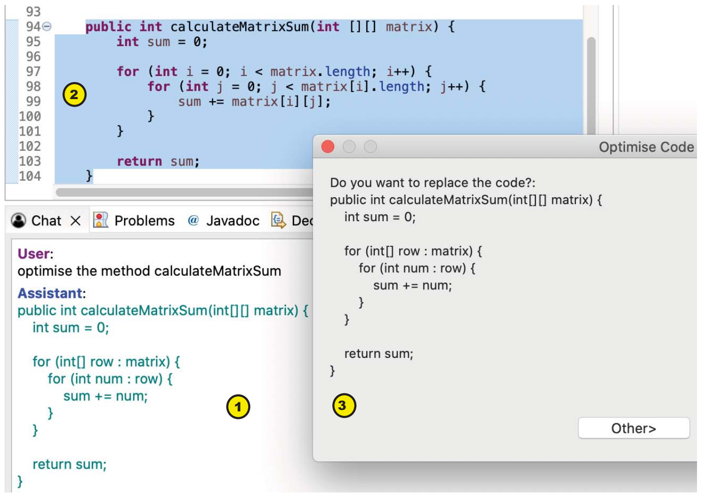
3 Tool Support

We have started to realise the proposed framework as an Eclipse plugin called Caret [2]. This is a conversational assistant that assists Java developers in programming tasks within Eclipse, based on the use of OpenAI's GPT-3.5 LLM and other intent-based conversational agents built with Rasa and Dialogflow (although it is possible to extend Caret to support other chatbot technologies). Caret provides an extension point to register assistive tasks and make them available from the IDE, either via natural language from a *Chat View*, or through commands from a context menu that is displayed by right-clicking on Java files. Currently, Caret assists with the following tasks: code completion, code documentation, unit testing, error detection, code optimisation, code comprehension and method renaming.

Fig. 3 illustrates Caret. It shows the request from a developer who writes "*optimise the method calculateMatrixSum*" in the *Chat View* (label 1). Internally, Caret classifies the task and extracts the method name *calculateMatrixSum* as a parameter of the request. Then, it searches the source code of the method in the active Java file (label 2). Next, it forwards the request to the registered optimisation task, which inserts the code into a prompt sent to GPT to perform the optimisation. Finally, a popup window displays the optimised code proposed by GPT (label 3), and the developer can apply it if desired.

4 Conclusions and Future Work

The advances in LLMs have prompted their use to support programming tasks. However, given their continuous growth and evolution, we argue that extensi-



```
    public int calculateMatrixSum(int [][] matrix) {
        int sum = 0;

        for (int i = 0; i < matrix.length; i++) {
            for (int j = 0; j < matrix[i].length; j++) {
                sum += matrix[i][j];
            }
        }

        return sum;
    }
```

User:
optimise the method calculateMatrixSum

Assistant:

```
public int calculateMatrixSum(int[][] matrix) {
    int sum = 0;

    for (int[] row : matrix) {
        for (int num : row) {
            sum += num;
        }
    }

    return sum;
}
```

Do you want to replace the code?:

```
public int calculateMatrixSum(int[][] matrix) {
    int sum = 0;

    for (int[] row : matrix) {
        for (int num : row) {
            sum += num;
        }
    }

    return sum;
}
```

Fig. 3. Screenshot of assistance request for optimising method *calculateMatrixSum*.

ble architectures enabling a flexible integration of the assistants into IDEs are required. We have presented our initial proposal, and an implementation atop Caret for assisting in Java programming in Eclipse. In the future, we plan to improve Caret regarding the user experience and support for other tasks. Our goal is to build a system to facilitate the construction of assistants for other (domain-specific) languages and software engineering activities (like modelling).

Acknowledgments. Work funded by the Spanish MICINN with projects TED2021-129381B-C21, PID2021-122270OB-I00 and RED2022-134647-T.

References

1. Barke, S., James, M.B., Polikarpova, N.: Grounded Copilot: How programmers interact with code-generating models. Proc. ACM Program. Lang. **7**, 85–111 (2023)
2. Contreras, A., Guerra, E., de Lara, J.: Conversational assistants for software development: Integration, traceability and coordination. In: ENASE (2024)
3. Eclipse Copilot: `https://www.genuitec.com/products/copilot4eclipse/` (last access in 2023)
4. GitHub Copilot: `https://github.com/features/copilot` (last access in 2024)
5. Jin, M., et al.: InferFix: End-to-end program repair with LLMs. In: ESEC/FSE. pp. 1646–1656. ACM (2023)
6. Lemieux, C., Inala, J.P., Lahiri, S.K., Sen, S.: CodaMosa: Escaping coverage plateaus in test generation with pre-trained large language models. In: ICSE. pp. 919–931. IEEE (2023)
7. Xu, F.F., Vasilescu, B., Neubig, G.: In-IDE code generation from natural language: Promise and challenges. ACM ToSEM **31**(2), 29:1–29:47 (2022)
8. Zhao, W.X., et al.: A survey of large language models. `https://arxiv.org/abs/2303.18223` (2023)