Issues in Information Systems
Volume 24, Issue 4, pp. 66-81, 2023

DOI: https://doi.org/10.48009/4_iis_2023_106

Exploring GitHub Copilot assistance for working with classes in a programming course

Pratibha Menon, *Pennsylvania Western University, menon@pennwest.edu*

Abstract

Object-oriented programming (OOP) is one of the most used programming styles and is present in most of the Computing and Information Systems curricula. Teaching and assessing an introductory OOP course require students to write code to create and use class specifications to solve problems described by the text. Recent advances in AI assistants like GitHub Copilot, which uses a natural language machine learning model trained on billions of lines of code, became available as a free IDE plugin for use by students. Termed by GitHub as 'your AI-Pair Programmer,' Copilot can provide coding suggestions and generate correct and readable code in common programming languages. The use of Copilot in programming courses prompts educators to know about the capabilities of Copilot and how they could adapt their teaching by incorporating AI assistants that could generate code. This paper qualitatively evaluates the interactions with Copilot that provide code solutions and suggestions to use or create classes in an application. The result of this study is used to discuss the implications of students using Copilot, a free IDE plugin, and some possible measures that educators could adopt in their programming courses in response to the use of AI assistance for code completion and code exploration.

Keywords: GitHub Copilot, AI-assistance, IDE, object-oriented programming, programming

Introduction

Object Oriented Programming (OOP) paradigm that is taught in most computing programs teaches good software development practices such as writing well-structured programs and efficient reuse of code and design. One of the critical challenges of teaching an introductory OOP course is due to its abstract nature and after having previously learned using an imperative programming method (Medeiros et al., 2019; Gutierrez et al., 2022). Students who would have struggled with the abstractness of imperative programming (Medeiros et al., 2019) might find it even harder to work with Object-oriented programming that requires them to abstract the entities into classes and model the entity relationships using object-oriented principles. The basic Object-oriented programming concepts and structures may be intuitive for an experienced programmer. However, students new to programming have yet to encounter enough problems to appreciate good design practices and common coding patterns (Chibizova, 2018). Due to students' challenges while learning OOP, Yu et al. (2021) indicated that it is vital that the teacher tries to maintain an interactive learning environment and use a problem-based learning approach in which students learn the concepts by solving meaningful problems. When the students face a problem, they google the solution or ask the teacher, and they essentially learn by doing instead of being lectured (Chis et al., 2018).

Recent developments in Large Language Models have opened immense possibilities for students to obtain coding assistance. One of the recent developments include OpenAI's Codex (Chen et al., 2021) that can generate multiple lines of code in response to a textual problem description. GitHub's Copilot, which is studied in this paper, is a programming assistant released by GitHub on June 2021 (Friedman, 2021), and it is currently integrated into several development environments, including Visual Studio Code, JetBrains, and Neovim. Copilot is made possible by the OpenAI Codex family of models, which is derived by fine-tuning GPT-3 (Brown et al., 2022) on publicly available GitHub code repositories. Copilot makes the programmer's interaction with Codex more seamless because it allows obtaining suggestions as they type in the IDE, provides ways to prompt for alternative suggestions, and accepts Copilot's suggestions with one keystroke. GitHub dubs Copilot as "your AI pair programmer." Given the widespread use of IDEs to support problem-based learning and the capability to easily attach a free Copilot plugin that is available to students, it is unavoidable for educators to consider the impacts of AI assistance on teaching and learning programming (Becker, 2023). There could be ethical and academic integrity concerns in using AI assistants to generate solutions in coursework. Nevertheless, educators need to adapt to the reality that Copilot is a free IDE plugin and that penalizing the use of Copilot is practically impossible. Instead, a much more productive approach is to educate students and instructors about the advantages and challenges of using the AI-assistant feature of Copilot.

This study presents two concrete scenarios of using Copilot to generate code inspired by basic tasks that students should learn in any object-oriented programming class. These tasks are: 1) Using a given class (file) to meet the requirements of an application; 2) Defining /Creating a class based on a given application's requirements stated in an application class (file). This study qualitatively explores Copilot's interaction capabilities to complete the two basic task scenarios to identify the interaction modes that students may encounter as they use Copilot associated with an IDE to get assistance with their programming assignments.

Background Literature

Machine learning advances, especially deep learning, have successfully generated code for general-purpose programming languages. For example, deep learning models using transformer architectures could recognize code structures using abstract syntax trees (Kim et al., 2021) and structural language models have remove any restriction on the vocabulary or structure so that code generation could be made possible for a general purpose programming language (Alon et al., 2020). While these methods had shown promising results, they still suffered from low accuracy and reliability (Tufano et al., 2019; Ciniselli et al., 2021). For instance, the RoBERTa-based only produces correct solutions for 7% of the tasks listed under a code search benchmark for retrieving relevant code given a natural language query (Husain. et al., 2019). Large language model (LLM) like ChatGPT3 is a game changer for automatic code generation (OpenAI & Pilipiszyn, 2021). Codex, a fine-tuned version of ChatGPT3, can solve almost 29% of unseen problems (164 problems with tests) using the first round of suggestions (without the programmer tweaking the solution). A fine-tuned model of Codex trained only on correct Python code solved 38% of the problems on its first attempt. If that model is allowed 100 attempts per problem, then at least one of the attempts was correct for 78% of the problems (Chen et al., 2021).

The performance of a Codex model called Davinci was tested using 23 problems from a CS1 Python course, and it could solve 10 of them on the first attempt (Finnie-Ansley et al., 2022). Another study on the performance of Codex on CS2 tasks using Python showed that Codex performs as well as students in the top quartile of the class (Finnie-Ansley et al., 2023). In both these studies, OpenAI’s Codex capabilities include fixing a bug, generating documentation strings, step-by-step explanations, and code summaries. Finally, a study by Sarsa et al. showed that Davinci could create exercises, sample solutions, explanations, and tests (Sarsa et al., 2022). In this study, researchers submitted a Python docstring containing keywords describing the problem’s application domain and the programming constructs. Out of the 240 generated exercises, 75% seemed reasonable as they had included the programming construct, but 30% had no solutions or tests. In summary, Codex solves a good percentage of problems on the first attempt but requires more than one attempt for most problems. This study uses the findings of these prior research to approach the AI-assistance as an imperfect tool and that the response generated by Copilot would depend on the kind of prompts that users provide.

AI-based code generation tools suffer from inherent uncertainty and imperfection common to any AI tool. They may generate code with errors or ones that may differ from users' expectations. A user study found no statistically significant difference in task completion time or task correctness scores when experienced programmers were using or not using a natural language-to-code plugin (Xu et al., 2021). Another older study showed that when people observed an automated system make errors, they began to distrust the system unless an explanation was provided about the nature of errors, but also that such explanations may lead to unwarranted over-reliance on the system (Dzindolet et al., 2003). These studies were done on the older models and the users involved in the study were experienced programmers, which makes it essential to conduct studies using Copilot and for users who could be students learning to write code for very basic programming tasks.

While prior studies on the use of Codex and Copilot have focused on programs taught in CS1 and CS2 courses that teach imperative and advanced programming using Python, no studies have explored how Copilot could be used to write programs that need to refer to pre-written Java classes. Prior research on the use of Copilot to solve CS1 and CS2 problems were quantitative in nature did not explicitly detail what could have been the user's experience while working with AI-assistance. This result of this study provides an important perspective for instructors that could inform teaching and learning by knowing about the capabilities of Copilot to aid coding assistance for novice learners in a Java OOP course.

The Study

Methodology

The research questions that have guided this study are formulated as follows:

- **Research Question 1:** *How could code auto-completion by GitHub Copilot assist novice programmers in using and creating code associated with classes in an introductory object-oriented programming course?*

- **Research Question 2:** *How do instructors and students with prior programming experience perceive the capabilities and usability of GitHub Copilot autocompletion in an introductory Object-Oriented Programming course.*

This paper presents a descriptive case study that involves a contextual analysis and a thick description of the use of GitHub Copilot to solve fundamental problems that are typically encountered in an introductory programming course. While this study subjectively analyzes the experience of using GitHub Copilot to solve introductory object-oriented programming problems, it does so by framing the use of Copilot into two distinct usage scenarios. These usage scenarios were selected based on the researcher's experience as a course instructor for introductory programming course. The researcher's subjective experience as an instructor implies an unavoidable selection bias for picking the usage scenarios. Therefore, this study tries to provide an accurate and comprehensive description of the data collection procedures and documentation of every piece of information in order to achieve reliability of a case study. The researcher bias is also addressed by applying reflexivity and by validating the results through student surveys.

This study considers the responses of Copilot to generate code and suggestions for two main types of programming problem scenarios that occur in the early part of an introductory OOP course.

1. **Use-task scenario:** The programmer writes textual (question) prompts as comments on the IDE, and the Copilot responds by writing code. The use-tasks problem scenario requires students to understand the code specifications written in the pre-written class and apply their knowledge of the class structure to create a desired application as required by a step-by-step statement of the problem.

2. **Create-task scenario:** The programmer writes code statements on the IDE, and the Copilot generates code and suggestions to complete, fix, or extend the code. The create-type exercise aims to generate a class code whose methods and constructors will meet the requirements of an application. Each create-type scenario problem is also broken into ten steps, but each step, in this case, is a code statement that is either a constructor call or a method calls to the intended class. In addition, an additional question prompt will be provided to explain what each method and constructor should accomplish. Finally, the programming task is to let Copilot create the methods and constructors in the class to fulfill the method and constructor calls stated in the application.

This study uses the JetBrains IDE and Java Programming language. Copilot can be downloaded and installed onto the IDE using a GitHub account. Copilot attempts to provide an inline suggestion on the IDE interface when the user pauses typing or presses Alt-\ or Enter. The suggestion is provided in italic grey font and can be accepted by pressing the tab key. For example, Copilot could provide suggestions to complete the current statement or generate code containing several statements.

Generating the Question Prompts

Question prompts and pre-written code are created to explore the interaction with Copilot. The question prompts are generated for application contexts familiar to students. Two problem scenarios, the use-task, and the create-task, are created for each application context. For each problem scenario, a file with pre-written class code is created that will be referenced by an associated application. Each application context also includes an application class that uses a class code specifying an entity. If the class code specifies an entity used in the application, then the name of the class (and the file) will be the same as the entity name. If the class contains an executable application (with a main method), the name of the class will have the word "App" as the suffix. For example, Pizza.java is a file that specifies the Pizza class (or entity), and PizzaApp1.java is the class file for a Pizza application.

In this study, classes were to model entities such as - Pizza, Coffee, Fish, Pet, Ice Cream, and Loan. Each of these entities were modeled as a distinct class in files Pizza.java, Coffee.java, fish.java, Pet.java, IceCream.java and Loan.java. The constructors and methods of described in these files are used in applications called PizzaApp1.java, CoffeeApp1.java, FishApp1.java, PetApp1.java, IceCreamApp1.java and LoanApp1.java, respectively. The same applications are re-written to run the create-task scenario on a different set of files – PizzaApp2.java, CoffeeApp2.java, FishApp2.java, PetApp2.java, IceCreamApp2.java and LoanApp2.java, respectively.

Each use-type scenario associated with an application context uses a pre-written code that specifies the attributes, constructors, and methods of an entity. For example, a Pizza application has a pre-written Pizza class code saved as a file called Pizza.java. Another class with a main method that contains ten steps of textual instructions to call a method or a constructor from the pre-written class is associated with the same application. For example, PizzaApp1 class has a main method consisting of ten prompts, all written as textual comments. For the create-type scenario, the class that contains a main method has ten Java statements that will either instantiate objects or call methods from an intended class. This intended class is written in a separate file that declares all the data fields as instance variables and lets Copilot write any required methods and constructors. For example, PizzaApp2.java will contain a class with a main method. The Pizza2App class will use the Pizza class that initially only defines the instance variables but not the methods and constructors. Copilot will complete the instance methods and the constructors of the Pizza class based on the constructor and method calls stated in the PizzaApp2 class.

Running Experiments

The question prompts for the use-type and the create-type tasks for each scenario were run many times until Copilot generated the desired solutions. All the input prompts and codes were entered during the initial run. Then each line containing the prompt was run by pressing the Enter key to allow Copilot to respond to that prompt. After Copilot responds to all the prompts, responses are noted, and a screenshot of the result is saved for future analysis. If Copilot produces incorrect, inaccurate, or undesired responses, the prompts are reworded to make them more specific before the next run. If the second run still does not provide desired solutions, the prompts are tweaked again, repeated at most five times until Copilot generates the desired and expected code. Every method, including the main method, gets a maximum of five runs.

Applying Reflexivity

Qualitative studies depend on subjectivity (Barrett, 2020), so researchers need to account for how subjectivity shapes their inquiry. Reflexivity is tied to the researcher's ability to make and communicate nuanced decisions as they generate data that tends to be influenced and intertwined by participants' real-world experiences and practices (Finlay, 2002). In this study, the researcher evaluates the use of Copilot based on their subject perspective, which is shaped by their role as an instructor with prior experience developing and teaching object-oriented programs. As an instructor for multiple sections of OOP courses for a Computer Information Systems undergraduate program, the researcher's experiences as an instructor shaped the perspective that the abstractions and coding standards applied to an OOP course are new to most students.

While students in an OOP class find it relatively easy to answer short-answer questions or write code snippets with only one solution possibility, having them design an application by figuring out the required entities leaves many design possibilities. Students often need help to generate and compare design choices, explain their choice, or even consider different ways to fix an error. The open-ended and context-specific nature of such problems requires students to obtain considerable assistance from tutors and instructors and, in some cases, even to validate their solutions. This teaching experience is one reason why the researcher was motivated to study the AI assistance provided by Copilot for OOP tasks that requires referring to a pre-written code to generate an application for a given context.

The researcher's prior experience as a programmer affords the ability to verify the code solutions generated by Copilot. However, a novice programmer may need help to verify the Copilot-generated solutions thoroughly and for the correctness or adequacy of the solution. The trust that novice programmers may develop for Copilot based on their experiences may dictate how exactly they might use the assistance provided by Copilot. It may not be ethical for the researcher, who is also an instructor, to test the use of Copilot by novice programmers when they have just begun to learn OOP. At this point, it needs to be clarified how much Copilot will help or hinder the learning process of novices. Therefore, the inability to involve novice programmers could impact how Copilot's interactions with the user are tested. For example, the researcher deliberately made the initial prompts vague to emulate how a novice would write the prompts, although the researcher knew how to write more specific prompts. Compared to the attempts the researcher needed to produce the correct code, the number of re-attempts to obtain the correct or desired code could have been different for a novice. Therefore, without really involving students who have just begun to learn OOP, it was impossible for the researcher to quantitatively estimate the number of re-attempts novice learners would need to get Copilot to produce the required code. Therefore, the researcher decided to take the qualitative approach and provide a detailed narrative of the experience of interacting with Copilot so that the readers could understand the capabilities and limitations of the AI assistance and its possible implications on teaching and learning.

Validating Observations

To validate the researcher's observations about Copilot's interaction patterns and responses, six undergraduate students with senior level standing volunteered to run and tweak the question prompts. As part of a usability study assignment intended for a Capstone course, students from that course were asked to evaluate different types of user interfaces. The assignment samples were de-identified and used for this study. Six senior students from the Capstone class attempted to study Copilot by running the application scenarios created for this study. They all had attended the object-oriented programming class during their sophomore year and programmed using at least two languages. The students picked two application scenarios that were tested by the researcher and tried to get Copilot to generate code. The students wrote comments about their experiences of interacting with Copilot. Since this study aims to explore the interaction patterns and experiences of using Copilot, only three open-ended questions were asked. Students were asked to answer the following questions in about one paragraph: 1) Please comment on your experience of interacting with Copilot to complete the tasks provided to you. 2) Explain how you would like to use Copilot from now on. 3)Explain why you would have/have not used Copilot in your introductory programming class had it been freely available to you then.

Results

Use-Tasks

Generating the first round of code using Copilot: The use-task prompts were entered as comments on the IDE. After entering a comment, the Enter key had to be pressed for Copilot to generate a code underneath the comment. After Copilot generated the first line of code, the next prompt was entered as a comment on the IDE to generate the next required line of code. In some cases, a single prompt could generate multiple lines of code. It was observed that Copilot generated a Java statement in response to each textual prompt and did not respond to a prompt by predicting the textual prompt. Figure 1 shows an example of a set of initial prompts and their responses during the first round of entering all the prompts under the main method. There are errors in lines 5, 11, 13, 15, and 19. In this initial run, the error rate, which is a ratio of the correct responses to the number of prompts, is 5/10 (or 50%).

In the initial prompt shown in Figure 1, there is no explicit mention of the type of constructor to create the object. The Pizza.java class only mentions one type of full-argument constructor with five parameters. Additionally, there are no default constructors specified in this class. Therefore, instantiation of the Pizza object in line 5 of Figure 1 has an error because it generates a default constructor not specified in the Pizza class. Copilot does not further explain why the instantiation has an error in it. Instead, IDE’s quick-fix feature (that is not powered by AI) provides several other suggestions to fix the error, which were not feasible for the given problem. For example, one suggestion includes creating a default constructor, which is not allowed in this problem and requires explicitly using the Pizza.java class (exactly as it is, without any further modifications, such as adding an extra default constructor). Meanwhile, the error report (which is not powered by Copilot) provided by the IDE informs that "*'Pizza(java.lang.String, int, int, int)' in 'Pizza' cannot be applied to '()'*" , which indicates that the number of arguments required for the constructor compared to what was actually provided. Line number 15 also has another instantiation error. Other errors in lines 11, 12, and 19 result from the use of incorrect method names in the method call. Copilot generated these method names based on the prompt that needed more specificity to provide the exact method name.

Respecifying the prompts to generate code using Copilot: During the next run, the prompts are rewritten with more specificity, such as providing the class name, type of constructor, and exact names of the attributes. After inspecting and respecifying each prompt that had generated an incorrect answer, all the prompts were rerun to see if the change in one of the prompts would have improved the solution. Figure 2 above shows the correct responses produced by Copilot after the fifth prompting attempt for the Pizza application. Even after getting rid of all the errors, the construction of the Pizza object in line number 7 does not obtain the parameter values for cheese toppings, pepperoni toppings, and ham toppings from the users, as intended in the prompt. For Copilot to write the code to obtain user inputs, the prompts may need to instantiate, for example, a Scanner or a JOptionPane class and then specify appropriate methods from this class to obtain and parse the user inputs. Copilot will also need to be directed to output comments for the user to provide the required input values corresponding to the data fields in a Pizza object. As a test, when, when Copilot was prompted to get the value of the size parameter from the user, it responded with another textual prompt that was repetition, but this time, to get the value of the next set of parameters. Although it did not generate any code, it seemed as if Copilot could predict the next course of actions that would be highly probable. It could have generated an appropriate code, instead of more textual prompts, if the prompts were specified step-by-step and in more detail.

```
3   public static void main(String[] args) {
4       /*Use Pizza.java class to generate a Pizza object*/
5       Pizza pizza1 = new Pizza();
6       //print the description of the Pizza object.
7       System.out.println(pizza1.getDescription());
8       //change the size of the Pizza object to a "large"
9       pizza1.setSize("large");
10      //set the number of cheese toppings to  2
11       pizza1.setNumCheeseToppings(2);
12      //set the number of pepperoni toppings to  1
13      pizza1.setNumPepperoniToppings(1);
14      //Use Pizza.java class to generate a Pizza object
15      Pizza pizza2 = new Pizza( size: "medium",  cheeseToppings: 1,  pepperoniToppings: 1);
16      //compare the two pizza objects
17      System.out.println(pizza1.equals(pizza2));
18      //compare the size of the two pizza objects
19      System.out.println(pizza1.equalsSize(pizza2));
20      //change the size of the Pizza object to a "medium"
21      pizza1.setSize("medium");
22      //get the description of the Pizza object
23      System.out.println(pizza1.getDescription());
24  }
```

Figure 1: Use-type task prompts at the first attempt for a Pizza Application

After repeating the prompting experiments on ten different application contexts, each with its user-type scenario, it was observed that Copilot could generate, after repeated attempts, many of the required statements without errors. However, even if the code generated by Copilot is error-free, it still did not meet all the expected requirements of the task.

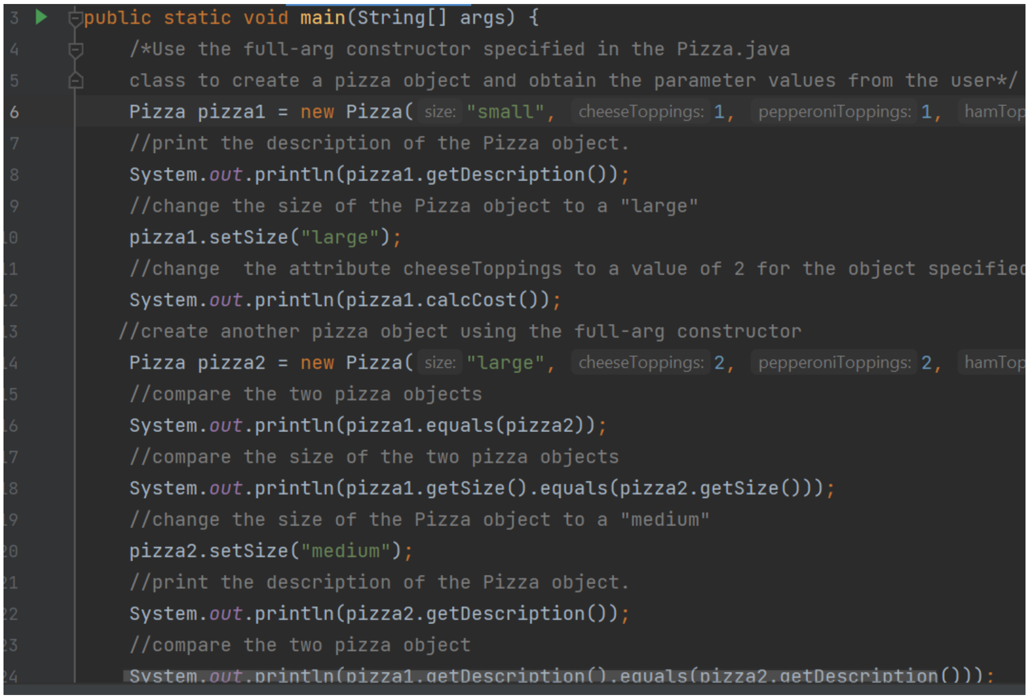


Figure 2: Use-type task for the Pizza application after the 5th attempt.

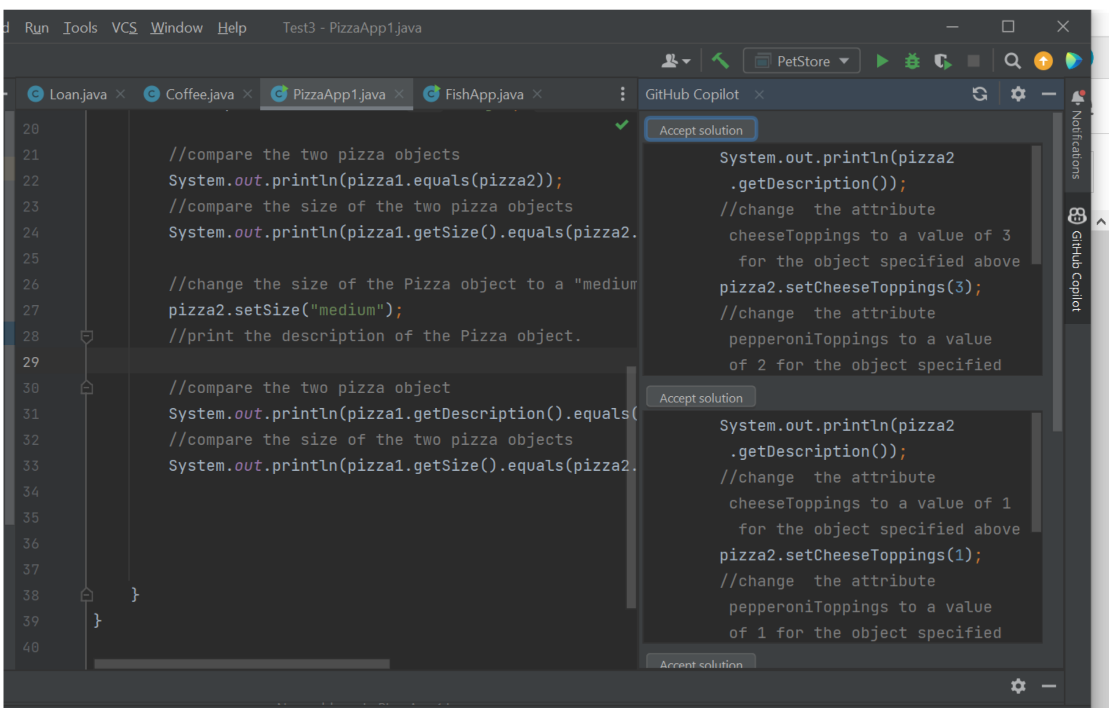


Figure. 3: Multiple Code options provided by Copilot

In addition to generating the code in response to a prompt, Copilot also allows the exploration of multiple auto-completion options for one or more tasks associated with the required line of code. Figure 3 shows the IDE interface with Copilot code suggestions pane on the right side. These suggestions contain several different statements that the user can choose. In the example above, however, the suggestions included more lines of code than required, and this is because Copilot, just like Codex, generates responses based on the probability of occurrence of tokens, and it could anticipate the next couple of lines of code that the user may request. For example, in Figure 3, in addition to calling the getDescription method, Copilot suggests setting the instance variables, which is not required for the given problem.

Create-Task Response

<u>Suggestion for actions by the IDE:</u> Figure 4 shows the prompts for a create-task problem, which in this case will be Java statements provided as prompts in the main method of an application, PizzaApp2.java, that calls the constructors and methods of the Pizza class. As expected, there are errors in the statements in the PizzaApp2 class because, for the create-task problem scenario, the constructors and the methods of the modified Pizza class will need to be specified. This because, before running the create-task prompts in the PizzaApp2 class, all the methods and constructors of the Pizza class used in the use-task scenario were removed to test if Copilot could generate them. The IDE detects the errors in the application code), as in Figure 4) and provides one or more quick fixes to resolve the error, as seen by the red-colored 'light bulb' on the left side of the IDE screen. Please note that these quick fixes are part of the IDE and unrelated to the Copilot actions. Out of the different types of suggestions provided by Copilot, one relevant suggestion is to create a constructor, although it did not specify the type of constructor. Even though the suggestion of a 'Dummy' constructor is misleading, clicking this suggestion causes the IDE to insert a constructor template in Pizza.java, as shown in Figure 5.

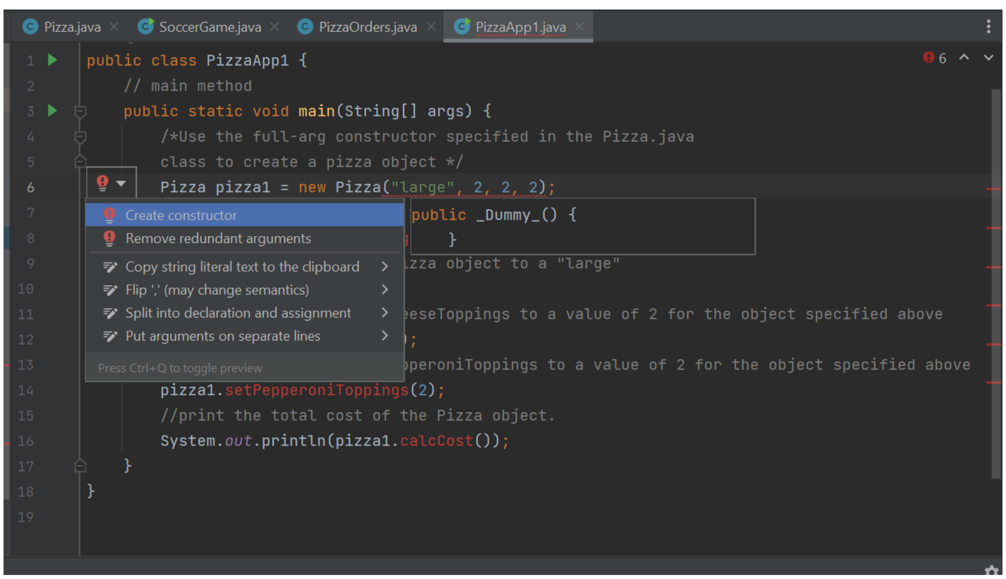


Figure 4: Create-type task prompts provided as code and quick-fix by the IDE in PizzaApp2.java

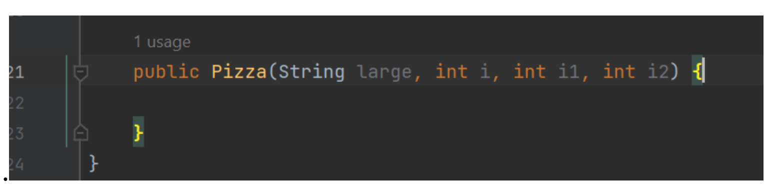
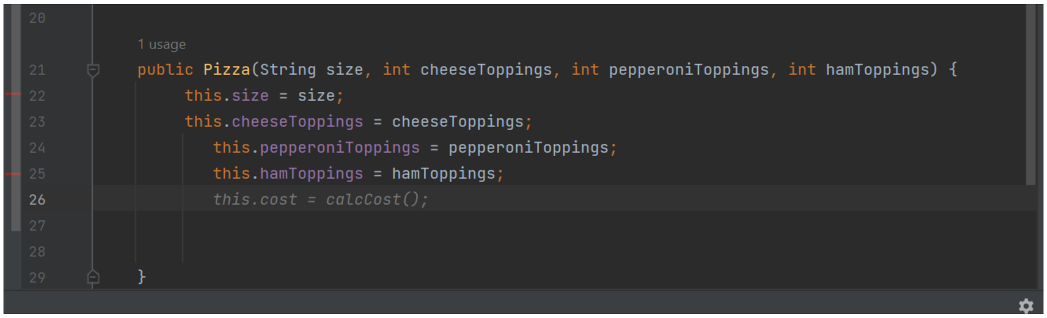
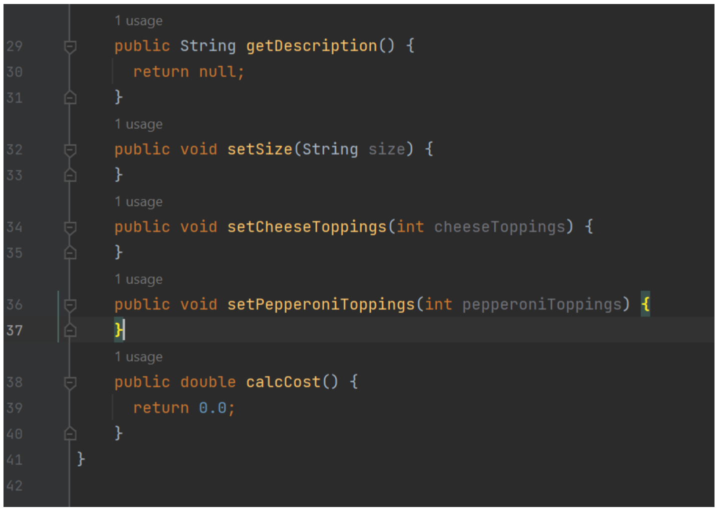


Figure 5.: Constructor generated by the IDE following the quick-fix



```
20
     1 usage
21   public Pizza(String size, int cheeseToppings, int pepperoniToppings, int hamToppings) {
22       this.size = size;
23       this.cheeseToppings = cheeseToppings;
24         this.pepperoniToppings = pepperoniToppings;
25         this.hamToppings = hamToppings;
26         this.cost = calcCost();
27
28
29   }
```

Figure 6: Constructor generated after the 2nd attempt



```
    1 usage
    public String getDescription() {
        return null;
    }
    1 usage
    public void setSize(String size) {
    }
    1 usage
    public void setCheeseToppings(int cheeseToppings) {
    }
    1 usage
    public void setPepperoniToppings(int pepperoniToppings) {
    }
    1 usage
    public double calcCost() {
        return 0.0;
    }
}
```

Figure 7: Method stubs generated by Copilot.

<u>Generating a Constructor:</u> Figure 5 shows the IDE's constructor (and not Copilot's constructor) generated in the Pizza class. While this constructor has the required number and types of parameters, it does not insert proper names (which should have been the names of the instance variables) to make the code more readable and to meet the coding best practice. The constructor stub also lacks the assignment of the parameters to the instance variables. Instead of using the quick-fix feature of the IDE to generate the method stub, we could have directly prompted Copilot to generate the required constructor in the Pizza class. However, the IDE was used in this case to see if Copilot interfered with the quick-fix feature of the IDE. From this experience of using the quick-fix feature and many more instances that used this feature to generate method stubs, it is evident that the traditional assistance features that were always available in the IDE prior to the Copilot plugin were not modified by the introduction of Copilot features.

Figure 6 shows the results after manually renaming the parameter variables of the constructor. The next step was to prompt code generation by the Copilot by manually typing the first statement inside the constructor that assigns the value to the instance variable size. After that, Copilot predicted the rest of the statements to assign values to other instance variables and autocompleted lines 23,24,25, and 26. While lines 23,24 and 25 were the expected responses; line 26 was syntactically correct but different from what was desired. Copilot had assigned the instance variable called cost with a value returned by calling the calcCost() method. While this does not throw an error, calling an instance method inside a constructor is not a good practice. So far, the prompts have been tweaked two times- once to pick the constructor stub and next to generate the statements inside the constructor body. During the third run, the prompt is more explicit by specifying that the cost must be assigned the default value of 0.0.

<u>Generating the method stubs</u>: In response to the method calls for the setters in the Pizza2.java application class, as shown in Figure 4, the quick-fix feature of the IDE generated the method stubs for the setters in Pizza.java. In the initial run for each setter, the setter method stubs generated by the IDE, as shown in Figure 7, do not specifically describe any details of the algorithm inside the methods. The IDE will only generate syntactically correct method stubs based on what it infers about the application's method signatures. In some cases, the default return types were incorrect, and the names of the parameters did not follow the naming conventions. However, the IDE will allow the user to choose the correct return type and the parameter name before accepting the method stub. Until this point, Copilot was not involved in generating the method stub. Copilot will be used only to generate the method body. Instead of using the quick-fix feature of the IDE, a programmer could also directly write a textual prompt in the Pizza class to generate a required setter. However, the IDE was used to see if Copilot could moderate the limited assistance provided by the IDE.

```
52  public double calcCost() {
53      // return the total cost of the Pizza object
54      // a small pizza cost $10 plus the cost of toppings which is $2 per topping
55         if(size.equalsIgnoreCase( anotherString: "small")){
56             cost = 10 + (2 * (cheeseToppings + pepperoniToppings + hamToppings));
57         }
58
59      // a medium pizza cost $12 plus the cost of toppings which is $2 per topping
60          else if(size.equalsIgnoreCase( anotherString: "medium")){
61              cost = 12 + (2 * (cheeseToppings + pepperoniToppings + hamToppings));
62          }
63      // a large pizza cost $14 plus the cost of toppings which is $2 per topping
64            else if(size.equalsIgnoreCase( anotherString: "large")){
65                cost = 14 + (2 * (cheeseToppings + pepperoniToppings + hamToppings));
66            }
67      return cost;
68
69  }
70 }
```

Figure 8: Code for a method called calcCost() after 4 attempts (and specifying 4 prompts)

<u>Prompting Copilot to generate the method body</u>: Copilot is prompted to generate the code that should go inside a method's body. In some cases, even before writing a textual prompt, or the first line of code, Copilot can infer the possible code purely based on the method name that follows a standard naming convention. If the code cannot be inferred by Copilot, helping it by adding the first line of code could generate an auto-completion of the rest of the required statements. However, in all cases, the user must verify Copilot's suggestions before accepting and, if needed, alter the code to fit the problem description. Figure 8 shows the textual prompts (as comments) for the calcCost() method and the code generated as a response to the prompts. This method required two runs to complete the method body. The initial prompt, "return the total cost of the Pizza object," did not generate any code suggestions. The second run required writing a portion of the initial part of the first 'if' statement and Copilot generated the rest of the code without any further prompt. The textual prompt in line 54 in Figure 8 generated the required code in lines 55- 57. Specifying the rest of the prompts (lines 59 and 63) also generated the required code without error. Each instance of providing a prompt (textual or the initial portion of a statement) is considered a run. The calcCost() method needed four runs to complete the method body. All the methods written for this study's application scenarios were simple. They had at most eight lines of code consisting of arithmetic and logical operators, simple decision structures, simple loops, the use of Math class, and print statements.

Task Response Analysis

Six different application scenarios were tested for the use-type and create-type tasks. Table below shows the application scenarios and final correctness of the generated code along with the number of times the prompts were re-written. The results show that the total number of runs required to obtain or improve the solutions of use-type tasks varied even though the initial prompts were similar in complexity for every application domain. Prompts had to be rewritten for most cases by making them more specific such as using the same method name and giving instructions resembling a pseudo code. In addition, the applications needed to learn how to obtain input values from the user based on a very high-level specification, so the final accuracy is 90% (9 out of 10 prompts yielded a correct answer).

The create-type tasks require writing a constructor and multiple methods. The number of attempts to obtain the code for a constructor or method's body depended on the prompts' specificity. For example, the constructor's code that Copilot inserted was syntactically correct but did not always follow the best practice and, therefore, needed additional prompting. In the case of setters, even if the method body had just one clear-cut statement that assigned the parameter value to the instance variable, the directions had to be specified during the second run. Therefore, even if Copilot generated syntactically correct code, programmers must clearly understand the program's functionalities and naming conventions to produce valuable, readable code. Even more important is that the programmers should know what the desired code should look like based on prior exposure so that they can validate the results or tweak their prompts to produce the desired response.

Table 1: Results – accuracy and number of attempts for the tasks

Accuracy of Use-type Tasks						
Attempt for each application with 10 prompts leading to method/ constructor calls	**PetApp**	**PizzaApp**	**CoffeeApp**	**FishApp**	**LoanApp**	**IceCream App**
1st	50%	50%	50%	90%	30%	100%
5th	50%	90%	50%	90%	30%	90%
Number of attempts for Create-type Tasks						
	Pet class	**Pizza Class**	**Coffee Class**	**Fish Class**	**Loan Class**	**IceCream Class**
Number of Attempts to generate a required constructor	2	5	3	2	4	2
Number of Attempts to generate a getter	1	1	1	1	1	1
Number of Attempts to generate a correct setter	2	2	2	2	2	2
Number of Attempts to generate a correct method	2	4	3	2	3	4

Student Volunteer Perceptions

Student responses to the question- *"1) Please comment on your experience of interacting with Copilot to complete the tasks provided to you*" indicated that students were "*surprised by the fact that it was so easy to generate code*" but also mentioned that "*prompts have to be very precise if one were to expect reliable answers*." They all reported that it took more than one round of re-writing a prompt or, sometimes, even *"starting to write the code"* to generate the correct response from Copilot. A student mentioned that they "*tinkered with the suggestions*" that Copilot had provided, although they had ended up with a "*messier code."* than what was intended. One of the students found this exercise to be a "*good review*" of what they had forgotten, and the Copilot helped them "*jog their memory*." Four out of six students mentioned that they found the autosuggestion and autocompletion of one or more sentences "handy." However, students also mentioned that the autosuggestions sometimes seemed "*confusing*." However, many agreed they had to have "*prior programming experience*" to verify if the solutions produced by Copilot were correct. They mentioned that Copilot is a good "*productivity tool"* to *save time figuring out the syntax*.

Student responses to the question – "*2) Explain how you would like to use Copilot from now on."* All students mentioned that they would continue to explore the use of Copilot and would use it in their jobs if needed. One student expressed the intention of "*practicing coding skills"* by "*exploring the suggestions*" provided by Copilot. They all felt that AI assistants could be part of their daily jobs in the future.

Student responses to the question – *"3) Explain why you would have/have not used Copilot in your introductory object-oriented programming class had it been freely available to you then."* Five out of six students responded that they would have used Copilot somewhat. Four out of six students had mentioned that they had all used Google/Stack Overflow to find answers or get sample codes and that having Copilot give suggestions right through the IDE would have been beneficial. Five students mentioned that they were unsure how using Copilot would have impacted their learning and problem-solving ability.

Discussion

An imperfect AI-assistant

The problems used in this study were elementary applications of object-oriented concepts. They did not include advanced concepts such as object-oriented design patterns or relationships such as inheritance, composition, or aggregation that may need an application to reference multiple classes. An analysis of the re-attempts at obtaining the desired code shows that the Copilot actions can be classified into two main categories: 1) code autocompletion and 2) providing multiple options for code completion. Autocompletion of code requires the user to think critically before accepting the suggested completion. Selecting from multiple options for code completion requires the user to compare multiple ways of completing the program before picking or picking one of the options. Not all autocompletion or suggestions may not always lead to correct or desirable results, and the user may need further actions such as re-prompting Copilot or manually writing code as part of further actions. This observation suggests that while Copilot could autocomplete code and save students from making syntax errors, the user will still need to learn to verify the results using their programming knowledge of OOP and an accurate understanding of the problem's requirements. Even though Copilot can generate comments to explain what a line of code does, it can't explain the choices it provides, or the code it completes, like an actual human could.

Implications for Teaching and Learning

AI assistant has substantial implications for how educators teach and assesses programming knowledge. Educators should stop using trivial problems that have only one solution. Minor problems with straightforward and specific prompts could quickly generate correct solutions within the first attempt. More comprehensive, multi-step problem statements that require students to explain their code will help students develop their ability to critically analyze their choices and decompose a problem to generate the prompts. Object-oriented programming requires students to focus on program structure in addition to making algorithmic choices to produce clean, readable, reusable code.

Any good programmer should be able to explain design choices, including the choices made to call a particular method or constructor, specify methods and fields, use a particular algorithm, and name a variable or a method. Copilot provides autocompletion based on the probability it learned from training a large code repository. On the other hand, a programmer chooses the coding statements based on their understanding and could explain their rationale. Therefore, including the need to explain one's code and design choices should become an essential part of programming pedagogy that uses Copilot. Furthermore, the problems used to assess a student's programming abilities should extend beyond their ability to solve 'toy problems' with only one clear-cut solution. This study shows how Copilot can generate solutions for simple problems with just one or two prompts. Instead, the pedagogy could include open-ended problems with many possible solutions. One assessment criterion should be the ability to explain algorithms or design choices for a given context.

Conclusions

This study investigates the interactions with Copilot that could generate and autocomplete code and provide programming suggestions with very few attempts. The programming tasks used for this study were very context-specific and required referring to pre-written codes and the problems used to test the interactions were very simple compared to the problems that students would solve when they complete a semester of OOP course. The results show that even for solutions to minor problems, the programmer needs to verify the code and multiple suggestions generated by Copilot. Both the student volunteers and the researcher who interacted with Copilot in this study have prior programming experiences and know what kind of solutions to accept from Copilot and how to tweak the prompts with more specific information about the programming task.

Observing novice students interact with Copilot is necessary to know how much they would trust it, how well they will write the prompts, or even if they will know how to recognize a correct solution. Nevertheless, this study shows how Copilot may not be a perfect AI assistant and may not explain the rationale behind its choices for autocompleting the code. Therefore, students could be instructed about the pitfalls of relying too much on Copilot, especially during the initial stages of learning how to write code. At the same time, Copilot's ability to provide coding suggestions is worth exploring since its ability to provide multiple suggestions could teach students to evaluate the possible design choices and coding actions critically. With appropriate scaffolding and pedagogical approaches, computer programming courses could facilitate learning by experimentation.

References

Alon, U., et.al. (2020). Structural Language Models of Code. In Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119), Hal Daumé III and Aarti Singh (Eds.). PMLR, 245–256.

Barrett A, Kajamaa A, Johnston J. (2020). How to be reflexive when conducting qualitative research. Clin Teach. 17(1):9–12.

Becker, B. A., et.al. (2023). *Programming Is Hard - Or at Least It Used to Be*. https://doi.org/10.1145/3545945.3569759

Brown, T., et. al. (2020). Language models are few-shot learners. Advances in neural information processing systems 33 (2020), 1877–1901.

Ciniselli, M. et.al, 2021. An Empirical Study on the Usage of BERT Models for Code Completion. arXiv preprint arXiv:2103.07115 (2021).

Chibizova, N. (2018). The problems of programming teaching. In 2018 iv international conference on information technologies in engineering education (inforino) (p. 1-4). doi: 10.1109/INFORINO.2018.8581834

Chis, A., Moldovan, A.-N., Murphy, L., & Muntean, C. (2018). Investigating flipped classroom and problem-based learning in a programming module for computing conversion course. Educational Technology and Society, 21 , 232-247.

Chen, M., et. al., 2021. Evaluating Large Language Models Trained on Code. https://doi.org/10.48550/ARXIV.2107.03374

Dzindolet, M. T., et.al. (2003). The role of trust in automation reliance. International journal of human-computer studies 58, 6 (2003), 697–718

Finnie-Ansley, J., et. al. (2022). The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming. In Australasian Computing Education Conference (Virtual Event, Australia) (ACE ’22). Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/3511861.3511863

Finnie-Ansley, J., et.al. (2023). My AI Wants to Know if This Will Be on the Exam: Testing OpenAI’s Codex on CS2 Programming Exercises. In Australasian Computing Education Conference. https://doi.org/10.1145/3576123.3576134

Finlay L. (2002). Negotiating the swamp: the opportunity and challenge of reflexivity in research practice. Qual Res. 2(2):209–230.

Friedman, N. 2021. https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer

Gutierrez, L., Guerrero, C., & L´opez-Ospina, H. (2022). Ranking of problems and solutions in the teaching and learning of object-oriented programming. Education and Information Technologies, 27 , 1-35. doi:10.1007/s10639-022-10929-5

Husain, H., et. al. (2019). Codesearchnet challenge: Evaluating the state of semantic code search. arXiv preprint arXiv:1909.09436 (2019).

Kim, S., et. al. (2021). Code prediction by feeding trees to transformers. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 150–162.

Medeiros, R. P., Ramalho, G. L., & Falc˜ao, T. P. (2019). A systematic literature review on teaching and learning introductory programming in higher education. IEEE Transactions on Education, 62 (2), 77-90.

OpenAI and Pilipiszyn. A. (2021). GPT-3 Powers the Next Generation of Apps. https://openai.com/blog/gpt-3-apps/.

Rees C.E, Crampton P.E.S, Monrouxe L.V. 2020. Re-visioning academic medicine through a constructionist lens. Acad Med. 95(6): 846–850

Reid, A.M, et. al., (2018). Ethical dilemmas and reflexivity in qualitative research. Perspect Med Educ. 7(2):69–75.

Sarsa, S., et.al. (2022) Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models. In Proceedings of the 2022 ACM ICER Conference - Volume 1 (Lugano and Virtual Event, Switzerland) (ICER '22).

Tufano, M., et.al., (2019). An empirical study on learning bug-fixing patches in the wild via neural machine translation. ACM Transactions on Software Engineering and Methodology (TOSEM) 28, 4, 1–29

Yu, X., Yang, Y., & Wu, X. (2021). Exploration and practice of object-oriented programming in the mode of "internet +" education. In 2021 2nd international conference on computers, information processing and advanced education (p. 709–711). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3456887.3457047

Xu, F. F., Vasilescu, B., & Neubig, G. (2021). In-IDE Code Generation from Natural Language: Promise and Challenges. arXiv preprint arXiv:2101.11149