Investigating student use of Copilot for object-oriented programming

Author: Mieke Maarse, 5750032
Supervisor: Dr. Hieke Keuning
Second supervisor: Prof. dr. Johan Jeuring

A thesis presented for the degree of
MSc Artificial Intelligence

Universiteit Utrecht

15-7-2024

Abstract

Generative artificial intelligence is gaining popularity and becoming more accessible to students in higher education. Students learning programming can use code-recommender systems such as GitHub Copilot to help them code. Educators worry students might become over-reliant on these types of systems. To gain more insight into how GitHub Copilot is used in higher education this study aims to find typical behavior of students while they use Copilot to solve an object-oriented programming problem. A think-aloud study was conducted to observe students while using Copilot. Observations focused on identifying whether students used fundamental programming skills and how this affected their performance. I found that when students planned their solution well they used Copilot as a beneficial tool. While students who did not plan their solution did not benefit from using Copilot.

Contents

1 Introduction

Generative artificial intelligence models have improved and become more accessible over the last few years and will continue to do so in the future [5]. Essays can be written with GPT-3, a large language model by OpenAI, without being detected as such [26]. Also, GPT-4, which is an improved version of GPT-3, was recently able to pass the bar exam, a notoriously difficult exam all US lawyers must pass [15]. Another significant advancement is ChatGPT, a conversational agent developed by OpenAI, which uses mentioned GPT models to generate human-like text responses [20]. With these models becoming more accessible, students are discovering how to use them in their education. Students can do this in a way that helps them learn, but they can also have the models do their work for them. Students' use of these models worries teachers, which makes it necessary to adapt education to the availability of these models [24, 5, 21].
Besides text, these generative large language models can also produce code. OpenAI's Codex model can solve many small programming problems [9]. This model has been used to create GitHub Copilot, which integrates Codex's code generation abilities into popular code editors and IDEs [13]. Since programming is seen as a difficult skill to learn [6] and novice programming students often do not feel confident in their skills [14], students can use these tools to help them with programming. While the use of learning tools can be of help to programming students [8, 2, 4], tools such as Copilot are not developed specifically for education and can give incorrect suggestions. That does not mean they cannot be used in education at all. For example, Codex can be used to generate programming problems [25] and examples of code created with Codex can help students learn [12, 5]. The Codex model has also been used to create a helpful tool to show more understandable error messages [17]. While educators worry about students using these systems, there is not yet a lot of research on how students typically use them in practice for larger programming problems. Therefore, this thesis aims to answer the following questions:

1. How do students typically use GitHub Copilot while working on object-oriented programming problems?

2. What fundamental programming skills do students use while working on object-oriented programming problems with Github Copilot?

In the related work, I will highlight several fundamental programming skills, focusing on skills needed when first learning (object-oriented) programming. I will also highlight related work on Copilots abilities and which programming problems it can solve. Subsequently, the methods section will explain the think-aloud study on the use of Copilot by university students. In the results section, I answer the research questions by showing common participant behavior from the think-aloud study and the skills students use to achieve this behaviour. Finally, in the discussion, future research is discussed, and a conclusion is composed.

2 Background and related work

In this section I first explore related work on how students learn object-oriented programming, and how tools can help them with this. Next I explore the abilities of GitHub Copilot by looking at OpenAI's large language model Codex, which is responsible for Copilot's code generation. I also explore Copilot's usability and how Copilot can be used in education.

2.1 Learning programming

2.1.1 Knowledge and skills

When students first learn to program, Yuen found that their knowledge and skills go through different stages [31]. He says that students start with automatic knowledge: they know what their instructor told them, but do not truly understand what it means. When they progress, they connect programming concepts such as arrays or loops. However the students do not fully understand the relationships between them. For example: students realise arrays and loops are often used together in their exercises, but do not understand what the exact relationship between them is.
After working with these concepts more, students will develop a deeper understanding, and use the concepts more effectively in practice. The type of knowledge a student has of a particular programming concept will show once they implement it.
Yuen links the mentioned stages of knowledge to stages of skills [31]. The first stage is "need to code". When students have only automatic knowledge of the concepts they are working with they feel like they need to start writing code immediately after reading an exercise. In the second stage they will focus on the syntax before thinking about the structure of their code. As students start connecting concepts, they start generalizing; they might not know how to solve an exercise fully, but are able to think of what type of loop they want to use, or whether to use recursion or not. This means students can abstract from the exercise and find the general idea of the solution, even though they might not understand how to fill in the details. The final stage of learning to program according to Yuen is when students are able to plan out a solution fully before they start coding. They will also be able to think about the efficiency of their solution. Knowing and understanding what stages novice programmers go through while learning can help prevent misconceptions, and help instructors give students better guidance in applying their theoretical knowledge in their programming exercises.
McCracken et al. suspected that many students do not learn some of the fundamental programming skills in their first year of programming education [18]. They made a framework of first-year learning objectives to clarify what they would expect from students after a year. This framework consists of a list of five steps needed to solve a programming problem: abstracting a problem from a description, dividing a problem into sub-problems, solving these sub-problems,

combining them to solve the complete problem, and finally, testing and revising the solution. When first-year students were tested on these skills, they did not meet these expectations. McCracken et al. found three reasons why students failed: 1) students handed in an empty program file, 2) students started programming right away without a plan, or 3) students started with a plan and structure but still failed to find a solution to the problem. Students of type 1 and 2 have a problem finding out where to start. These students are not able to abstract the problem from the description and divide the problem up into sub-problems. The students of type 3 are planning and structuring their code but have problems with creating the sub-solutions, combining these, or testing and revising. In the end, the most common problem for students is their lack of skill in abstracting the problem from the description.

2.1.2 Abstraction and generalization

Both McCracken et al. and Yuen emphasize the importance of abstraction and generalization in solving programming problems. Teaching students to abstract and generalize can be done with the help of examples, according to Zander et al. [32]. When instructors want students to solve a programming problem with a certain technique, they often use code examples that solve a similar problem with that same technique. Students will then have to work out which parts of the example code they can reuse to solve their problem. This forces students to look at code and problem descriptions in a more general way, and think about how to solve a problem before starting to code. Examples can also be used at other stages of learning. For example, when students are learning a new programming language, example code can help them learn the syntax [5]. When students are learning a difficult concept, or are making a lot of mistakes with certain concepts, giving students well-written example code to play around with can give them a deeper understanding of these concepts [32].

2.1.3 Object-oriented programming

The previously described skills are needed for programming in general, independent of the programming language or paradigm being used. However, when learning object-oriented programming, additional skills are needed. Some fundamental concepts for object-oriented programming are objects and classes. According to Xinogalos, students often misunderstand the relationship between these concepts [30]. Some misconceptions students might have are that there is no difference between objects and classes, or that classes are collections of objects. Xinogalos found that when students do not have these misconceptions, it does not mean they have a full understanding of classes and objects. For example, students often understand a class as an entity in the code and part of its structure, and an object as a piece of code. Although this is not exactly wrong, these students do not understand the concepts, or their relationship, fully. These students are focused on the static code and are not looking at the bigger picture of how parts of the code interact. Xinogalos shows that almost

half of his students have a good understanding of classes and objects. They see objects as entities of a real-world phenomena or objects, and a class as a model describing kinds of objects. Almost all students recognise that classes play an important part in the structure of the code.
From interviewing students, Stamouli et al. [28] confirm that when students start learning object-oriented programming, students are mostly focused on the syntax of the object-oriented language they are learning. At this point, students believe that learning the syntax of that language is what constitutes learning object-oriented programming. The more students learn, the more their view changes. They start to realise learning programming constructs, writing structured programs, learning to think a certain way, learning problem solving, and finally, learning a new skill, is what constitutes learning object-oriented programming.

2.1.4 Programming tools

Programming tools, such as tools for static code analysis (linting), syntax highlighting, debugger tools, code completion, and so on have all become standard, or at least common as integrated tools in IDEs. These tools can be of great use to novice programmers. In addition, tools are specifically being designed for novice programmers, such as those showing clearer error messages. All these tools can be used to address and prevent misconceptions for students according to Qian et al. [23]. For example, programming environments that highlight syntax errors or use understandable error messages can reduce cognitive load for students. This will help them focus on the problem at hand and understand and learn from their mistakes.
For tools to be helpful to novice programmers, they should be well integrated into an IDE [16]. This makes it is easier to learn how the tool works, and the use of the integrated tool causes less cognitive load than a separate tool. The integration of a tool will also make it a possibility for the tool to have access to the code the student is working on.

2.2 GitHub Copilot

Generative large language models are machine learning models trained on very large amounts of unlabeled text. They are able to predict what text would most likely be produced based on their training data. These models are able to produce any text, including code, as long as this type of text was included in the training data.
With the release of GPT-3 to the public, this technology has become better and much more accessible. Numerous tools have started to appear using this technology. As large language model based tools keep becoming better at predicting and generating language and code, students start using them more to help them with their work. GitHub Copilot is one of these tools. It is an extension for IDEs or code editors that suggests lines of code, or entire functions to you as you work. As seen in Figure 1, it suggests code while typing, which can be

accepted pressing tab. The multi-selection pane is shown in Figure 2, where you can view up to ten code suggestions for where you are currently working in the document and choose if you want to accept one. The multi-selection pane does not open by itself, only when prompted.

GitHub Copilot uses OpenAI's Codex, which is a fine-tuned GPT-3 (Third-

```
static int factorial(n)
{
    if (n == 0)
    {
        return 1;
    }
    else
    {
        return n * factorial(n - 1);
    }
}
```

Figure 1: A suggestion from Copilot. All gray code was suggested at once.

generation Generative Pre-trained Transformer) model. Codex has been trained on 159GB of Python code from public GitHub repositories (preprint)[9]. In the Copilot tool, Codex is responsible for the code generation, while Copilot extends its usability by integrating into IDEs and code editors. This integration makes Copilot easy to use while working, as there is no need to switch to a separate tool. Copilot also adapts code suggestions to the file you are working in, and to neighbouring files [13].

2.2.1 Codex

To evaluate Codex, Chen et al. have created their own evaluation set, which they called HumanEval (Preprint)[9]. They did this because Codex was trained on publicly available GitHub code, and so, commonly used benchmark problems could have very well been included in the training data. HumanEval is a set of 164 programming problems handwritten for this evaluation. Every problem comes with a function signature, docstring, body, and several unit tests. Figure 3 shows two examples of problems from HumanEval. To evaluate Codex, the model was given each HumanEval problem one by one, where the models output was tested on the problems' unit tests. A problem was counted as solved when at least one of Codex's 100 code samples passed the unit tests. With this method, Codex solves 70.2% of the HumanEval problems successfully.

Sobania et al. compared the performance of Copilot's code generation to that of tools with a genetic programming approach [27]. When tested on standard program synthesis benchmarks, GitHub Copilot could solve the same type and amount of problems as tools with a genetic programming approach. However, when analysing the produced solutions, Sobania et al. found a big difference between the code produced by Copilot and that produced with a genetic programming approach. They decided the purely functional evaluation was insufficient in this comparison, as Copilot's code had better readability. Its failed

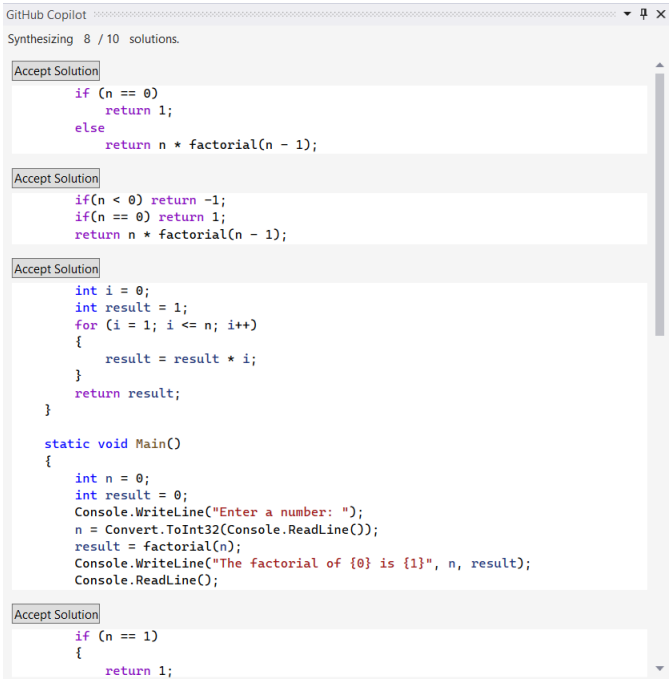


Figure 2: The Copilot multi-suggestion pane

attempts were also often easy to adjust into a correct answer. With genetic programming approaches, code was often not easy to comprehend. This makes Copilot far more helpful to programmers, even though the functionality of produced code was similar with both approaches.

Naser Al Madi confirms that Copilot's code has good readability; he found it comparable to the readability of human-written code [1]. In the same study, he compared how well students read the code written by other students when pair programming to how well they read the code written by using Copilot to pair program. He found that students read other students' code more carefully than they read Copilot's code. Some students were observed accepting large pieces of Copilot's code at once, which would take a long time to read well. Not reading Copilot's code carefully could indicate over-reliance on Copilot. It would also cause problems because Codex has some weaknesses. According to Chen et al. Codex has trouble with more high-level programming problems and with longer text prompts. Its problem with longer prompts remains when Codex can solve the subproblems of the exercise separately. Another weakness of Codex is keeping track of many different variables in one prompt. Overall, Codex does better when given step-by-step prompts [9].

Finnie-Ansley et al. found that Codex outperforms novice computer science students on novice programming problems. However, they noted that the formulation of a programming problem can significantly affect how well Codex

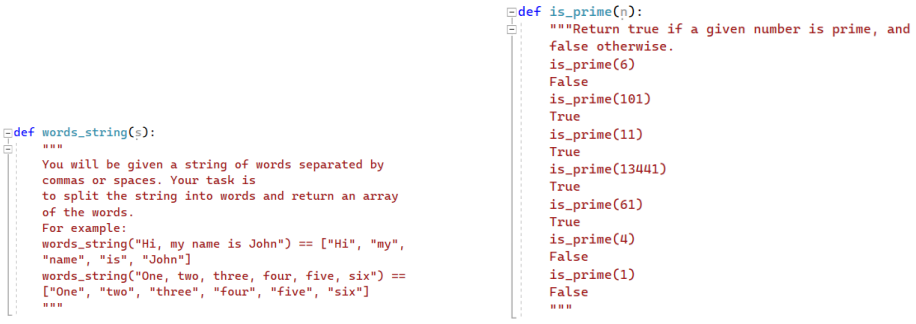


Figure 3: two problems from the HumanEval evaluation set (Preprint)[9]

solves it. Specifically, Codex struggles when an exercise does not explicitly state what to do with edge cases [12].

2.2.2 Copilot use

GitHub markets Copilot as an AI pair programmer [13], and taking Codex's weaknesses into account, Copilot's abilities greatly depend on how it is used. Cipriano and Alves (2023) directly address limitations in the use of GPT-3 for object oriented programming. They raised issues such as failing to apply inheritance best practices and suggesting solutions with code duplication [10]. Their findings reinforce the idea that Copilot's abilities could greatly depend on the person using it.
Barke et al. found that when professional programmers use Copilot their interactions with it can be categorised into two modes [3]. Programmers either use Copilot to *accelerate* their work, meaning they know how to proceed, and Copilot suggests just that, saving them the time to type it out themselves. The other mode is *exploration*, where a programmer needs inspiration on how to proceed. In exploration mode, they often use the multi-suggestion pane (Figure 2), where they might take some lines out of different suggestions or accept one and adjust it. Programmers also use the multi-suggestion pane to give them a general idea of how to solve a problem and write their own solution.

Mozannar et al. extended this work by developing a taxonomy of code-recommender user programming states (CUPS) [19]. This CUPS taxonomy includes twelve programming states in a hierarchical structure, as seen in Figure 4. In this study, twenty-one participants were screen-recorded while they each worked on one out of eight different programming tasks. Subsequently, they labeled this recording with the CUPS labels. This labeling was done in a very specific way. First, the whole recording was divided into *telemetry segments*. In Figure 4, the telemetry segments can be seen in the timeline. One telemetry segment happens from t_0 to t_1, one happens from t_1 to t_2, and so on. There are two types of telemetry segments: one consists of the moment between a suggestion

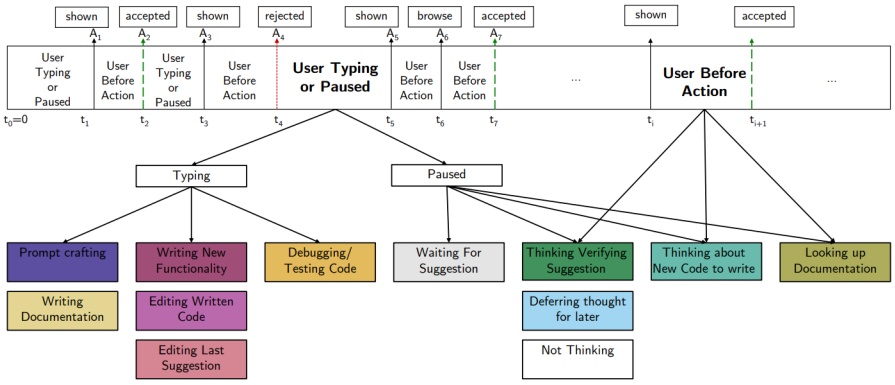


Figure 4: The Cups taxonomy and telemetry segments [19]

being shown by Copilot and it being rejected/accepted (the *user before action*), the other consists of the moment between rejection/acceptance and a new suggestion being shown (the *user typing or paused*). Once the recording is divided into telemetry segments, the participants labeled their own segments according to Figure 4: in a *user typing or paused* segment, the labels for typing or paused can be used. In a *user before action*, all but one of the paused labels can be used (not *waiting for suggestion*). Every individual telemetry segment was given one CUPS label by the participant. In these labeled recordings, Mozzanar et al. found some common patterns of programming states for developers. One such pattern was programmers rejecting many suggestions while prompt crafting and while writing new functionalities. Another was verifying a suggestion both before and after accepting it. Mozzanar et al. also found many instances of developers accepting suggestions before verifying them. Developers likely do this because they want to look at the suggestion with syntax highlighting or verify multiple suggestions as a whole instead of as separate lines of code. Mozzanar et al. revealed that developers might spend half their time on Copilot-related activities when programming with Copilot. Because developers spend a significant amount of time waiting for suggestions, Mozzanar et al. suggest improving Copilot by reducing latency. They also suggest that if users can indicate their current programming state, Copilot can more effectively meet their needs. For example, it could avoid providing distracting suggestions when a user is actively typing.

Vaithilingam et al. compared how programmers used Copilot to how they used Intellisense, a code completion tool [29]. They found that even though their participants solved fewer exercises using Copilot than they did using Intellisense, they still found Copilot more helpful than Intellisense. Participants used Copilot suggestions as a starting point when they did not know how to start by themselves, or they used Copilot for suggestions instead of looking something up online. They found Copilot untrustworthy when it suggested large pieces of code at once and did not want to read, test, and debug that code.

2.2.3 Codex in programming education

Most research evaluating Codex and Copilot uses programming problems similar to those used in computer science education, but does not go further into the implications of Copilot on computer science education.
Becker et al. believe that we must urgently consider these implications and address the opportunities and challenges of tools like Codex in computer science education [5]. They emphasise that tools like Codex will only improve and become more easily available in the future.
Becker et al. see opportunities in using these tools to provide example code: solutions to problems when students get stuck, multiple solutions that use different methods to solve a problem, and examples of specific concepts for an instructor to use. They also see opportunities in using Codex for generating programming exercises and code explanations. The Codex model can be used to power other tools and students can start learning higher-level concepts earlier, using Codex to fill in the simpler parts of code.
Becker et al. see challenges as well: students can use these tools to cheat without being detected and they can become over-reliant on these tools. They also worry that the style of Codex's code is inappropriate for novice programmers, as it is based on mostly professional code. Other challenges they see are in establishing the owner of the code produced by these tools, their sustainability, harmful biases in generated code, and insecure generated code.
Finnie-Ansley et al. address the challenges and opportunities of Codex in programming education [12]. They see some of the same opportunities and challenges as described in Becker et al.'s paper: the opportunity of providing students with many example solutions and the challenge of students becoming over-reliant on Codex. Although Finnie-Ansley et al. see examples generated by Codex as an opportunity, they fear that Codex will provide buggy code, causing confusion and misconceptions in students. They agree with Becker et al. that students using Codex to do their work is a challenge, and add to it that it will remain a challenge even when Codex cannot successfully solve a student's programming problem. They fear that even though Codex's code may contain mistakes or not solve the entire problem, students might still get a passing grade for the partial solution. Finnie-Ansley et al. believe we cannot ignore tools such as Codex and must adjust computer science education to these challenges.
Prather et al. [22] wanted to know how students use Copilot in practice and how students perceive the use of Copilot. They observed students working with Copilot on a typical introductory programming assignment and subsequently interviewed them about their experiences. They discuss their results based on the four themes they saw emerge during observation and interviews: *Interactions, Cognitive, Purpose,* and *Speculation.* The *interactions* theme is about participants using Copilot: coding with it, accepting/rejecting, and their general experience interacting with Copilot. The second theme, *cognitive*, is about the cognitive state the user has: confused, thinking about their thought process, positive, and negative. Next, *purpose* is about why users are using Copilot. This could be Copilot guiding the user, the user outsourcing work to Copilot, or

Copilot helping speed up the coding process. Lastly, the theme of *speculation* is about users thinking about Copilot in a broader sense, like whether Copilot has intelligence or knowledge, or what the future will look like with tools such as Copilot.
The theme of purpose also came up in other research discussed in this section. Prather et al. themselves note that accelerations v.s. exploration from Barke et al. [3] is similar to the theme of purpose, where there is speeding up (acceleration), and guiding (similar to exploration). They find that students use both of these modes while using Copilot.
From the interaction theme, Prather et al. found two new modes of Copilot use: shepherding and drifting. In shepherding, students type slowly to try and get a (specific) suggestion from Copilot. In drifting, students accept copilot code when they are not sure about it, maybe change it a bit, before deleting the code and starting again.
In terms of student perception of Copilot, Prather et al. look mostly to the cognitive and speculation themes. While students generally found copilot helpful, its suggestions could be confusing for students, sometimes leading them away from their solution. This could cause students to be frustrated. Students did find it helpful to describe their problems to Copilot and found it less intimidating to approach a problem when they knew they had Copilot's help. When students talk about Copilot in general, they can be excited, but also fearful about how Copilot seems to know what you want from it. Some students took it further and expressed a fear of Copilot writing complete programs by itself in the future.

3 Methods

This section describes the methodologies used in the study, including a think-aloud study to observe how students interact with GitHub Copilot while solving programming problems.
To analyze the sequencing of participant actions and decision-making processes during the think-aloud study, we used n-grams. Here follows a detailed explanation of what n-grams are and how they contribute to the understanding of participant interactions with GitHub Copilot.

3.1 Think-aloud study

To find how students commonly use Copilot, and if there are any common patterns in their use, a think-aloud study was carried out. This think-aloud study was approved by the ethics review board of Utrecht University.
A think-aloud study is a study in which participants verbalize their thoughts in real-time while they perform a task [7]. This research method used to capture and analyze the participants' thinking process. It is often used to research students' knowledge and skills. Participants thoughts during the task may be supplemented with an interview or questionnaire after the task.

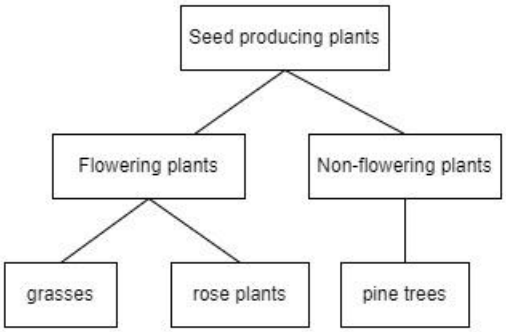


Figure 5: A small part of the plant taxonomy

During the think-aloud study, participants worked on an object-oriented programming problem in Visual Studio code with access to GitHub Copilot. Participants were instructed to verbalise their thoughts during the study. Because this verbalizing increases cognitive load for novice programmers, participants practiced on a small simple exercise for a maximum of 10 minutes before starting the main exercise of the study. All exercises can be found in appendix A and B.
After this practice round, participants began with the main programming problem. During this part of the study, the participants' voices and screens were recorded. They worked on this problem for up to 30 minutes.

3.1.1 Exercise

Participants were asked to implement a small part of the plant taxonomy as seen in Figure 5 using different classes. This exercise focuses on the use of classes and objects and was based on a classic object-oriented programming exercise, in this case from "Java, how to program early objects" by Paul Deitel and Harvey Deitel [11]. In this classic programming exercise, the shape hierarchy from Figure 6 must be implemented and specific methods need to be added to different types of shape. Since variations of this exercise are used very often, it might be included in Codex's training data. Because of this, I decided to adapt it to a different context: that of plant taxonomy. Plant taxonomy is a hierarchical structure that is obscure enough to not be commonly used in programming exercises, which makes it a great substitution for shapes in this exercise. I then simplified the exercise for the think-aloud study. After the first think-aloud session, I rewrote the exercise slightly to make the description clearer, and once again after the third participant. The different versions of the exercise can be found in appendix B.
The final version of the exercise focuses on the architecture of the code and requires only very simple functionalities to be added. This invites participants

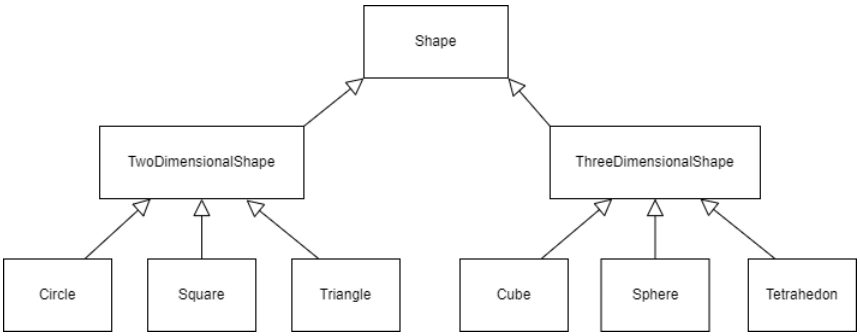


Figure 6: The class structure for the original shapes exercise [11]

to plan how to structure their code before they start writing and makes it possible to test how helpful Copilot is to students when working specifically on the planning and architecture part of object-oriented programming. Participants programmed their solution in C# since is the language used most in the computer science program at Utrecht University, especially for object-oriented programming. They had a maximum of half an hour to work on this problem, after which they were asked to fill out a questionnaire.

3.1.2 Questionnaire

Participants answered some questions about their experience in programming and using GitHub Copilot, so that this could be linked to differences in Copilot use. Participants were also asked to reflect on their experience during the study, to both give feedback on the exercise and to give more information on how they felt about using Copilot. The complete questionnaire is included in appendix C.

3.2 Participants

Participants were recruited at Utrecht University through announcements at mandatory classes for first-year computer science students. Participants were offered refreshments during the study, but no other reward was provided. There were six participants in total. Table 2 shows details on each participants background in programming, as well as the difficulty rating (one to nine) they gave to the exercise after completing it.
Three participants, participant one, three, and four, were bachelor computer science students, all in their first or second year. Two participants, participant two and six were bachelor artificial intelligence students. One at the start, and one at the end of their bachelor's.
Participant five was a master student who had finished their bachelor of artificial intelligence.
Of all participants, three had earlier experience with GitHub Copilot: partici-

pant one and four had worked on projects with Copilot and participant five had tried it out. Participant two did not have experience working with Copilot, but had used ChatGPT for programming before.

Table 1: Participants overview

Participant	Study	Year	Programming experience
1	Computer Science	Second	Secondary school, hobby
2	Artificial Intelligence	Third	Side job, hobby
3	Computer Science	First	Secondary school
4	Computer Science	First	Side job, hobby, former education
5	Artificial Intelligence	Other	Former education
6	Artificial Intelligence	Other	

Table 2: Background of participants

3.3 Data processing

The data collected during the think-aloud study was processed similarly to how Mozannar et al. did in their similar study as explained in Section 2.2.2 [19]. All timestamps of suggestions appearing, being accepted, and being rejected were collected. The transcription was then divided up for every telemetry segment so that participants quotes could be shared without sharing their voices. Next, the CUPS taxonomy was used to tag the telemetry segments using the hierarchy seen in Figure 4.
After the data had been fully processed and tagged as described, n-gram sequences of the CUPS were computed, from bigrams up to 10-grams, as explained below. This shows how common every unique sequence of CUPS is in the data.

3.3.1 N-grams

N-grams are contiguous sequences of n items from a given sample of text or speech. In the context of this study, an "item" refers to a programming state identified during the think-aloud sessions. N-grams are used to capture and analyze the patterns in sequences of these states, helping us understand the typical behavior of participants over multiple steps. For instance, a bigram (2-gram) would involve sequences of two consecutive programming states (e.g., from "writing new functionality" to "verifying suggestion"), while a trigram (3-gram) involves three, and so on. Analyzing these patterns allows us to identify common sequences in participant behavior that might indicate typical usage strategies or common responses to Copilot's interventions.

4 Results

This section discusses the results of the think-aloud study illustrating typical individual and common behavior in programming with Copilot. These typical behaviors of the participants are shown both by the CUPS N-grams and by participants' quotes and behaviors during the study. The section starts with a discussion of shared behaviors all participants show, after which the participants are divided into two groups based on their current programming experience. This is done because participant one, two, three, and four were all in a bachelor program where they program every week, and all had programming experience outside of their current bachelor program. On the other hand, participant five and six were working on a thesis and had not programmed in a while, and did not have programming experience outside of their higher education. They also show different typical behaviors as will be discussed in this section.
In this section, the heatmap of bigrams of CUPS for all participants combined can be seen, along with tables of the most occurring trigrams and 4-grams. Heatmaps of bigrams of CUPS from all individual participants and tables of trigrams, 4-grams, 5-grams, and 6-grams from both individual participants and all participants combined can be found in appendix E and D.

Table 3: Most Occurring CUPS 3-grams for All Participants

3-gram	Count
('writing new functionality', 'writing new functionality', 'writing new functionality')	40
('prompt crafting', 'prompt crafting', 'prompt crafting')	20
('writing new functionality', 'writing new functionality', 'thinking verifying suggestion')	14

Table 4: Most Occurring CUPS 4-grams for All Participants

4-gram	Count
('writing new functionality', 'writing new functionality', 'writing new functionality', 'writing new functionality')	17
('prompt crafting', 'prompt crafting', 'prompt crafting', 'prompt crafting')	11
('writing new functionality', 'writing new functionality', 'writing new functionality', 'thinking verifying suggestion')	9
('writing new functionality', 'writing new functionality', 'thinking verifying suggestion', 'writing new functionality')	8

4.1 Common participant behaviors

The behaviors contained in this section apply to all participants.

4.1.1 Lack of note taking

Although some participants planned out a solution before starting to code, none took notes on the exercise despite explicitly being given paper for that purpose. All participants came back to reading the exercise multiple times for details on how sub-problems had to be implemented. To do this, they had to search for details in the exercise, taking up more time than necessary.

4.1.2 Object-oriented exercise

All participants except participant five commented on the fact that the exercise was quite easy, and that most work was in structuring the classes. The exercise was designed with this in mind, so that student use on Copilot could be tested explicitly on object-oriented programming. All participants immediately understood that this was what the exercise was about and all but one tried to plan the structure themselves. Participants four and five did not know how to start. Participant six started the exercise by opening Google and searching for information on class structures in C#. However, participant five started by prompt crafting in order for Copilot to generate a class structure.

4.1.3 Distracted by suggestions

When participants were writing code, whether they had a plan in mind at that time or not, Copilot suggesting code at an unexpected moment was often distracting to participants. They lost their focus and it took time to get back to what they were writing. Participant two rejected six suggestions in thirty seconds, saying "What is it saying?" and "Copilots suggestions are wrong" all while continuing to write the code they already had in mind. This participant also accepted a suggestion, spending a lot of time reading and working on the code, but then removing large chunks of this code later. When seeing the suggestion this participant said: "do I need this? No, I don't... I'll accept it just to check". This participant spent a lot of time on code they knew they didn't need beforehand.

4.1.4 Helpful assistant

The survey conducted in this study found that all participants thought Copilot was helpful, although some also found it annoying besides finding it helpful. Despite this, participants two and three would not like to continue using Copilot. Both mentioning they like working with the code suggestions in their IDE (integrated development environment) more as they only recommend lines of code which they find more helpful. Participants mostly mentioned Copilot being helpful in speeding up the programming process and helping with syntax.

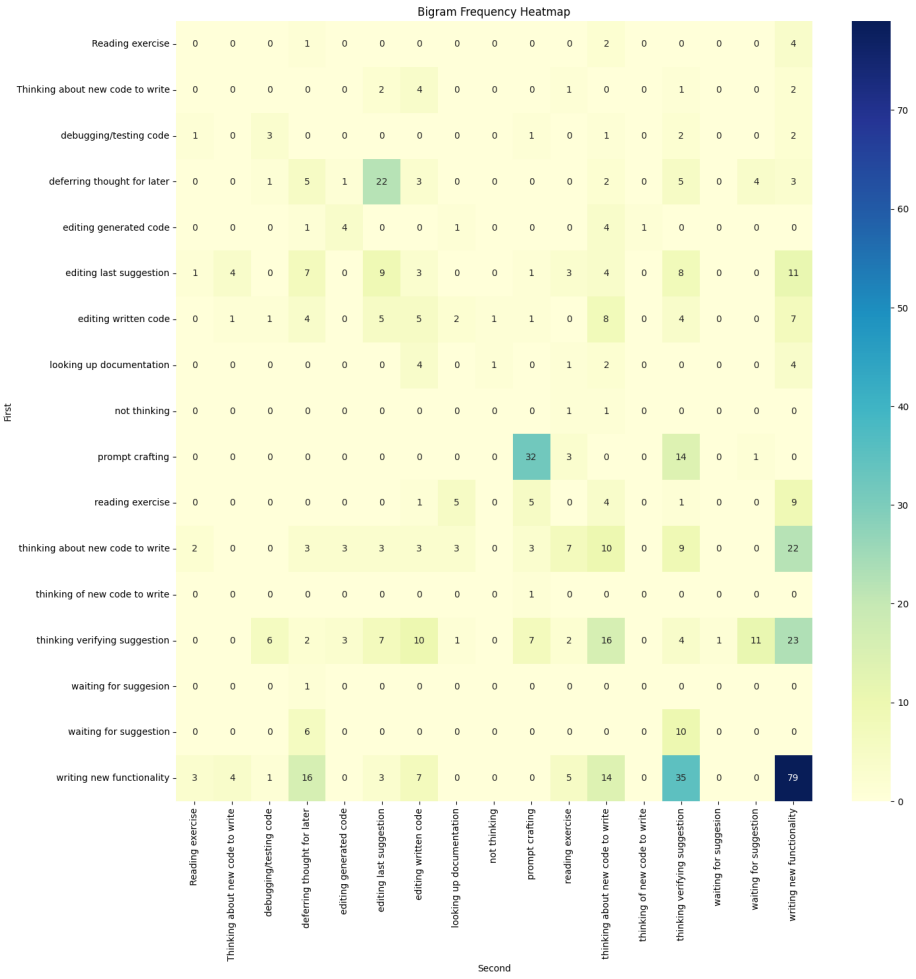


Figure 7: CUPS Bigram Frequency for all Participants

4.2 Actively programming participant behaviors

This subsection contains common behaviors from the group of participants that program regularly: participant one, two, three, and four. All these participants had programmed for a job or as a course in secondary school as well. Participant one and four had extensive experience with Copilot.

4.2.1 Following a plan

A common behavior in the group of participants who programmed regularly was that they wanted to write the code they had in mind already. They clearly planned a solution, or an outline of a solution, while reading the exercise and started programming with a plan in mind. While reading the exercise, participant one said: "Ok, so the first thing I see is that I will need to implement different classes. I'll make a plant mother class with two daughters and one of those will also have two daughters and the other one will have one daughter. So I'll start with a superclass." The other participants from this group started planning similarly while reading the exercise. It is clear that these participants already abstract the problem from the description while they read it, thinking about how they would implement the plant taxonomy the moment they see the tree structure: "Ok, I'll start with the graph", "ok, looking at the picture...". This leads to these participants wanting to write the code they have in mind, and rejecting Copilots suggestions. This is supported by the common bigrams, trigrams, and 4-grams seen in Figure 7 and Table 3 and 4 where *writing new functionality* is followed by itself as this means participants ignored Copilot's suggestions and kept writing their own code.

4.3 Behavior from participants with past experience

This subsection contains behavior from participants who had not programmed in a while: participant five and six. Participant five wrote in the questionnaire that they were not very familiar with C#. Both had no other experience with Copilot than simply trying it out.

4.3.1 Syntax examples

Both participants who had not programmed in a while commented that Copilot could be helpful when using language with complex syntax requirements such as C#. These participants were not or no longer very familiar with C#. Participant six commented on Copilot in the questionnaire: "[Copilot] makes the whole syntax thing easier if you know what you want and where you want it.". During the think-aloud, participant six said: "I'm fiddling around a lot with the syntax, so I'm removing a lot. It's so long ago that I don't really know what I'm doing.". Since there are no specific cases in both participants think-aloud studies where Copilot helped with syntax, it could mean that these participants were using Copilot's suggestions as general examples of C# code.

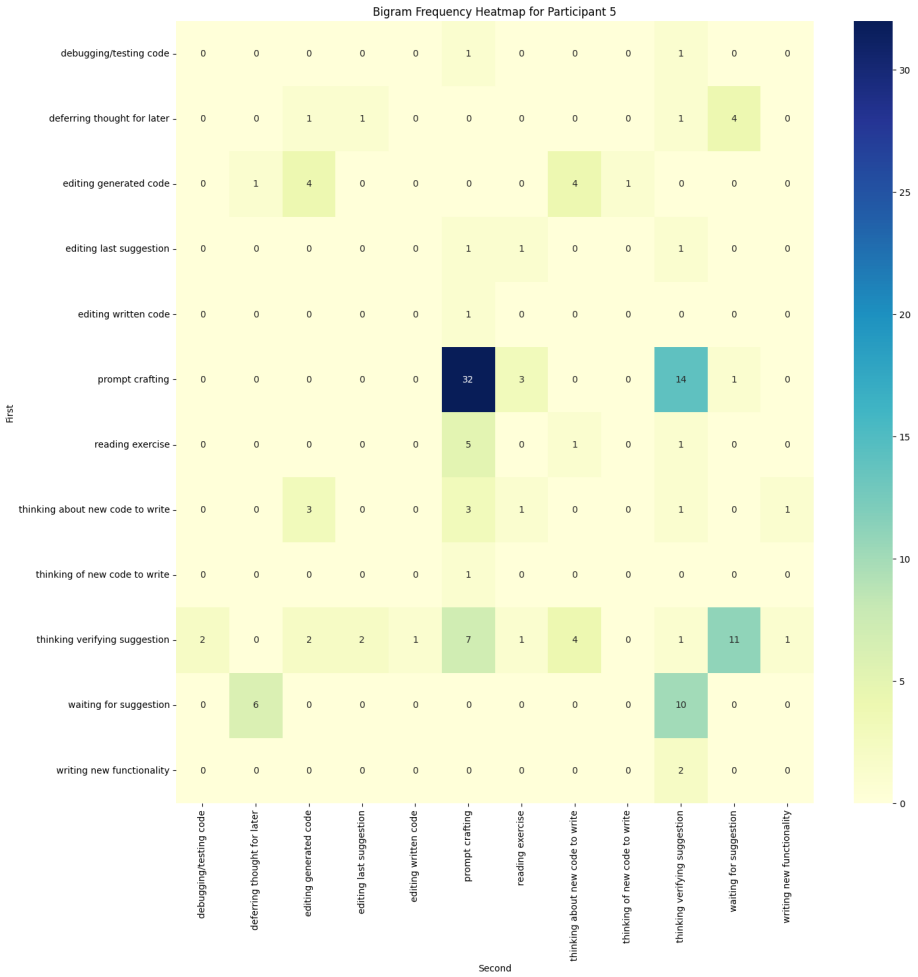


Figure 8: CUPS Bigram Frequency Heatmap for Participant 5

Table 5: Most Occurring CUPS 3-grams for Participant 5

3-gram	Count
('prompt crafting', 'prompt crafting', 'prompt crafting')	20
('thinking verifying suggestion', 'waiting for suggestion', 'thinking verifying suggestion')	9
('prompt crafting', 'prompt crafting', 'thinking verifying suggestion')	8

Table 6: Most Occurring CUPS 4-grams for Participant 5

4-gram	Count
('prompt crafting', 'prompt crafting', 'prompt crafting', 'prompt crafting')	11
('prompt crafting', 'prompt crafting', 'prompt crafting', 'thinking verifying suggestion')	7
('thinking verifying suggestion', 'waiting for suggestion', 'thinking verifying suggestion', 'waiting for suggestion')	5
('prompt crafting', 'thinking verifying suggestion', 'waiting for suggestion', 'thinking verifying suggestion')	5

4.3.2 Prompt crafting

Participant five had a different approach to using Copilot than the other participants. From this participants bigram heat map and sequence tables (Figure 8, Table 5, 6, and 7) it is clear that they tried to get Copilot to solve this exercise by using prompts. This lead to this participant leaning on Copilot heavily while not understanding the suggested code. During the study they said after prompt crafting, but not getting the desired result: "I will make it a bit nicer so maybe Copilot understands me better", which was then followed by more prompt crafting.

5 Discussion and future work

5.1 Discussion

Students who use fundamental programming skills such as planning a solution before starting to code, abstracting a problem from a description and dividing that problem up into sub-problems, use Copilot to accelerate their programming. They typically accept suggestions that fit into their plan. Students who do not use these fundamental programming skills try to use Copilot for exploration, but were mostly confused by Copilot suggestions. Using Copilot for acceleration and exploration have previously been found by Barke et al. as explained in the background section citebarke2022grounded. If students plan their solution out before starting to code, their code will be better, and written faster. Planning out code is also an important step in learning to code. They are the first steps in McCracken et al. their framework: abstracting a problem from a description,

Table 7: Most Occurring CUPS 5-grams for Participant 5

5-gram	Count
('prompt crafting', 'prompt crafting', 'prompt crafting', 'prompt crafting', 'prompt crafting')	7
('prompt crafting', 'prompt crafting', 'prompt crafting', 'prompt crafting', 'thinking verifying suggestion')	4
('prompt crafting', 'prompt crafting', 'prompt crafting', 'thinking verifying suggestion', 'prompt crafting')	3
('prompt crafting', 'prompt crafting', 'thinking verifying suggestion', 'prompt crafting', 'thinking verifying suggestion')	3
('thinking verifying suggestion', 'waiting for suggestion', 'thinking verifying suggestion', 'waiting for suggestion', 'thinking verifying suggestion')	3

dividing a problem up into sub-problems [18]. These steps were also found to be very beneficial in programming with Copilot as seen in the common behavior of **following a plan**. When participants were following a plan, they used Copilot as an assistant that had to conform to their plan. Participants only accepted code that fit into their plan and so, deciding whether or not to accept a suggestion from Copilot did not take up too much time. While some participants did form a plan before starting to code, none took notes. If students were allowed to use Copilot for a object oriented programming exercise, it would be helpful to ensure they take notes on how they plan to solve the exercise before they start to code. This will save students time, as they will have to look up details in the exercise less often, and will help students use Copilot in a productive way.
When participants had less current experience writing C#, they started coding without a clear plan in mind. This is described as a *need to code* by Yuen[31]. This *need to code* without having a clear idea of what to write, drove participants to using Copilot early in the problem solving process. This caused Copilot to give many unfitting suggestions as it did not have a lot of input to base its suggestions on. Participants using Copilot in this way seemed to follow its lead in stead of letting Copilot follow their lead. This caused these participants confused on what to do.
It seems that students who have not mastered the skills of abstracting a problem from a description and dividing a problem up into sub-problems, needs to learn those before starting to use Copilot as a programming assistant. However, Copilot can be helpful to these students in providing examples of code in an unfamiliar language, as it can clearly show the right syntax quicker than Google. Using examples can be very beneficial in learning programming according to Zander et al. [32].

5.2 Limitations

When looking at the participant selection and sample size there are some clear limitations to this study. All participants that were selected came from the same university (Utrecht University). For the sample size, the study was conducted with six people, of which only two fit into the demographic of first year computer science students. This means that the results and conclusion are more fit to support future, more extensive research on the same topic, than to stand alone.

One of the limitations of this study is the sole use of GitHub Copilot for AI assisted programming, as currently there are multiple other examples such as Amazon CodeWhisperer[1] and Tabnine[2]. Besides other programming assistants, GitHub Copilot has recently been updated with more features, meaning this study does not fully cover Copilot's current features. If more programming assistants were tested in this study, more features could have been tested for typical student use.

5.3 Future work

Future work is needed on if code-recommender systems should be used as a tool for learning in higher education and if so, how such tools could be implemented to be used as successfully as possible. This could differ for age and experience level of students, and also for different courses. Research is also needed on how widespread Copilot use is among different categories of students.
In terms of using generative AI for programming education in a wider sense, research is needed on which type of generative AI is most helpful for learning. Would tools generating natural language, only code, or a combination work best?
For example, comparing the use of Copilot with that of ChatGPT [20]. This path could research which method is more beneficial when learning to program.
Another interesting field of research is in how tools such as GitHub Copilot can be used to give students reliable personalised feedback. This can be greatly beneficial to their learning but decreases the workload of educators significantly.

6 Conclusion

This thesis presents a think-aloud study on how computer science students solve an object oriented programming problem using Github Copilot, a state of the art code-recommender system based on a large-language model. The study identified several typical behaviors among participants, which were linked to fundamental programming skills described in the literature. To find these behaviors,

[1] `https://aws.amazon.com/codewhisperer/`
[2] `https://www.tabnine.com/`

the CUPS taxonomy was used, a taxonomy of code-recommender system user states, specifically designed to tag behavior of programmers using systems such as Copilot. Through this study, it was found that code-recommender systems are percieved to be helpful to students in reducing the time they spend on programming exercises while students still use fundamental (object-oriented) programming skills, specificaly those related to planning out solutions to object oriented programming problems. Students who do not posses those skills were not helped by using code-recommender systems as assistants, but were confused by unhelpful suggestions. However code-recommender sytems might be helpful in providing these students with examples of correct code. If code-recommender systems are to be used by students, an emphasis must be placed by educators on planning out a solution before starting to code.

References

[1] Naser Al Madi. "How Readable is Model-generated Code? Examining Readability and Visual Inspection of GitHub Copilot". In: *37th IEEE/ACM International Conference on Automated Software Engineering*. 2022, pp. 1–5.

[2] Michal Armoni, Orni Meerbaum-Salant, and Mordechai Ben-Ari. "From scratch to "real" programming". In: *ACM Transactions on Computing Education (TOCE)* 14.4 (2015), pp. 1–15.

[3] Shraddha Barke, Michael B. James, and Nadia Polikarpova. "Grounded Copilot: How Programmers Interact with Code-Generating Models". In: *Proc. ACM Program. Lang.* 7.OOPSLA1 (Apr. 2023). DOI: 10.1145/3586030. URL: https://doi.org/10.1145/3586030.

[4] Brett A Becker et al. "Compiler error messages considered unhelpful: The landscape of text-based programming error message research". In: *Proceedings of the working group reports on innovation and technology in computer science education* (2019), pp. 177–210.

[5] Brett A. Becker et al. "Programming Is Hard - Or at Least It Used to Be: Educational Opportunities and Challenges of AI Code Generation". In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. SIGCSE 2023. Toronto ON, Canada: Association for Computing Machinery, 2023, pp. 500–506. ISBN: 9781450394314. DOI: 10.1145/3545945.3569759. URL: https://doi.org/10.1145/3545945.3569759.

[6] Yorah Bosse and Marco Aurélio Gerosa. "Why is programming so difficult to learn? Patterns of Difficulties Related to Programming Learning Mid-Stage". In: *ACM SIGSOFT Software Engineering Notes* 41.6 (2017), pp. 1–6.

[7] Elizabeth Charters. "The use of think-aloud methods in Qualitative research An introduction to think-aloud methods". In: *Brock education* 12.2 (July 2003). DOI: 10.26522/brocked.v12i2.38. URL: https://journals.library.brocku.ca/brocked/index.php/home/article/view/38.

[8] Chin Soon Cheah. "Factors contributing to the difficulties in teaching and learning of computer programming: A literature review". In: *Contemporary Educational Technology* 12.2 (2020), ep272.

[9] Mark Chen et al. "Evaluating large language models trained on code". In: *arXiv preprint arXiv:2107.03374* (2021).

[10] Bruno Pereira Cipriano and Pedro Alves. "GPT-3 vs Object Oriented Programming Assignments: An Experience report". In: *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1* 9798400701382 (June 2023), pp. 61–67. DOI: 10.1145/3587102.3588814. URL: https://doi.org/10.1145/3587102.3588814.

[11] Harvey Deitel. "Java How to Program (early objects)". In: (2014).

[12] James Finnie-Ansley et al. "The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming". In: *Australasian Computing Education Conference*. 2022, pp. 10–19.

[13] Github. *Your AI pair programmer*. URL: https://github.com/features/copilot. (accessed: 21.11.2022).

[14] Jamie Gorson and Eleanor O'Rourke. "Why do cs1 students think they're bad at programming? investigating self-efficacy and self-assessments at three universities". In: *Proceedings of the 2020 ACM Conference on International Computing Education Research*. 2020, pp. 170–181.

[15] Daniel Katz et al. "GPT-4 passes the bar exam". In: *Philosophical Transactions of the Royal Society A* 382 (Feb. 2024). DOI: 10.1098/rsta.2023.0254.

[16] Michael Kölling. "The problem of teaching object-oriented programming, Part 2: Environments". In: *Journal of Object-Oriented Programming* 11.9 (1999), pp. 6–12.

[17] Juho Leinonen et al. "Using Large Language Models to Enhance Programming Error Messages". In: *arXiv preprint arXiv:2210.11630* (2022).

[18] Michael McCracken et al. "A multi-national, multi-institutional study of assessment of programming skills of first-year CS students". In: *Working group reports from ITiCSE on Innovation and technology in computer science education*. 2001, pp. 125–180.

[19] Hussein Mozannar et al. "Reading between the lines: Modeling user behavior and costs in AI-assisted programming". In: *arXiv preprint arXiv:2210.14306* (2022).

[20] OpenAI. *ChatGPT*. Nov. 2022. URL: chat.openai.com.

[21] Mike Perkins. "Academic Integrity considerations of AI Large Language Models in the post-pandemic era: ChatGPT and beyond". In: *Journal of University Teaching & Learning Practice* 20.2 (2023), p. 07.

[22] James Prather et al. "" It's Weird That it Knows What I Want": Usability and Interactions with Copilot for Novice Programmers". In: *arXiv preprint arXiv:2304.02491* (2023).

[23] Yizhou Qian and James Lehman. "Students' misconceptions and other difficulties in introductory programming: A literature review". In: *ACM Transactions on Computing Education (TOCE)* 18.1 (2017), pp. 1–24.

[24] Jürgen Rudolph, Samson Tan, and Shannon Tan. "ChatGPT: Bullshit spewer or the end of traditional assessments in higher education?" In: *Journal of Applied Learning and Teaching* 6.1 (2023).

[25] Sami Sarsa et al. "Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models". In: *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*. 2022, pp. 27–43.

[26] Mike Sharples. "Automated essay writing: an AIED opinion". In: *International Journal of Artificial Intelligence in Education* 32.4 (2022), pp. 1119–1126.

[27] Dominik Sobania, Martin Briesch, and Franz Rothlauf. "Choose your programming copilot: a comparison of the program synthesis performance of github copilot and genetic programming". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. 2022, pp. 1019–1027.

[28] Ioanna Stamouli and Meriel Huggard. "Object oriented programming and program correctness: the students' perspective". In: *Proceedings of the second international workshop on Computing education research*. 2006, pp. 109–118.

[29] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models". In: *Chi conference on human factors in computing systems extended abstracts*. 2022, pp. 1–7.

[30] Stelios Xinogalos. "Object-oriented design and programming: an investigation of novices' conceptions on objects and classes". In: *ACM Transactions on Computing Education (TOCE)* 15.3 (2015), pp. 1–21.

[31] Timothy T. Yuen. "Novices' Knowledge Construction of Difficult Concepts in CS1". In: *SIGCSE Bull.* 39.4 (Dec. 2007), pp. 49–53. ISSN: 0097-8418. DOI: 10.1145/1345375.1345413. URL: https://doi.org/10.1145/1345375.1345413.

[32] Carol Zander et al. “Copying Can Be Good: How Instructors Use Imitation in Teaching Programming”. In: *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’19. Aberdeen, Scotland Uk: Association for Computing Machinery, 2019, pp. 450–456. ISBN: 9781450368957. DOI: 10.1145/3304221.3319783. URL: https://doi.org/10.1145/3304221.3319783.

A Appendix A: Practice exercise

Given a string and a non-negative int n, return a larger string that is n copies of the original string.

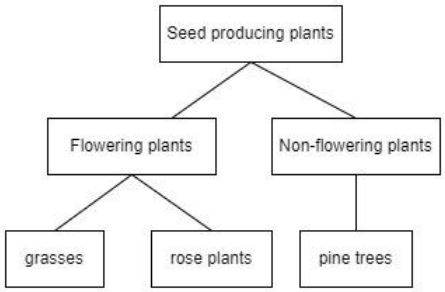
stringTimes("Whoop!", 2) → "Whoop!Whoop!"

stringTimes("Hi", 3) → "HiHiHi"

stringTimes("Hi", 1) → "Hi"

B Appendix B: Exercise versions

B.1 First version



In the diagram you see part of the plant taxonomy. Non-flowering seed producers such as pine trees produce seeds through cones. Flowering plants have their seeds enclosed in a fruit that comes from a pollinated flower.
Implement the plant taxonomy in the diagram using different classes, with appropriate methods and attributes.

All plants should have a *name* that you can make up yourself.

All plants should be able to produce seeds with seedProduction() which prints a string describing how this plant produces seeds. Non-flowering and flowering plants produce seeds through cones and fruits respectively. Since all plants produce seeds in their own way, the string this method prints is different for every plant.

Plants that produce cones should have a coneProduction() method that, in the case of a pine tree, prints: "Pine tree *name* created *nrOfCones* cones!" when called. Here, *nrOfCones* is the number of cones this specific pine tree produces. Every non-flowering plant should have this attribute that specifies how many

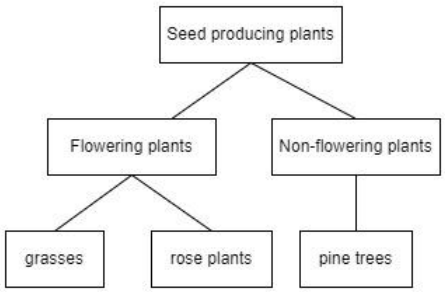
cones it produces when coneProduction() is called.

Flowering plants have to be pollinated before they can produce fruits that contain seeds. They should have a bool *pollinated* that starts out as false. When the method pollination() is called, *pollinated* turns true. Now the plant can create fruit with the method fruitFormation(). When *pollinated* is false when this method is called, the following string should be printed: "Plant *name* was not able to produce any fruit, since it is not pollinated". When *pollinated* is true, the string printed by this method depends on the type of plant that it belongs to:

- Rose plants create rosebuds: "Rose *name* produced many rosebuds!"
- Grasses create grains: "Grass *name* produced many grains!"

Create a grass, a rose plant, and a pine tree and have them all produce seeds in their own ways.

B.2 Second version



In the diagram you see part of the plant taxonomy. Non-flowering seed producers such as pine trees produce seeds through cones. Flowering plants have their seeds enclosed in a fruit that comes from a pollinated flower.
Implement the plant taxonomy in the diagram using different classes, with appropriate methods and attributes.

All plants should have a *name* that you can make up yourself.

All plants should be able to produce seeds with seedProduction() which prints a string describing how this plant produces seeds. Non-flowering plants produce seeds inside cones and flowering plants produce seeds inside fruits. Since all plants produce seeds in their own way, the string this method prints is different for every plant: for a pine tree, which is a non-flowering plant, it should print: "Pine tree *name* created *nrOfCones* cones!". Here, *nrOfCones* is the number

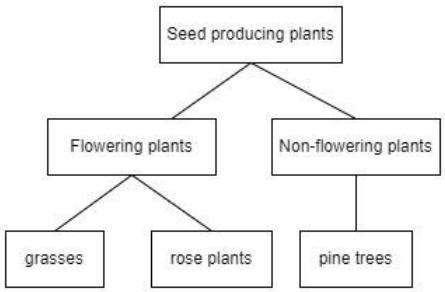
of cones this specific pine tree produces. Every non-flowering plant should have this attribute that specifies how many cones it produces when seedProduction() is called.

Flowering plants have to be pollinated before they can produce fruits that contain seeds. They should have a bool *pollinated* that starts out as false. When the method pollination() is called, *pollinated* turns true. Now the plant can create fruit. When a flowering plant tries to create a fruit when *pollinated* is false, the following string should be printed: "Plant *name* was not able to produce any fruit, since it is not pollinated". When *pollinated* is true, the string printed depends on the type of plant that it belongs to:

- Rose plants create rosebuds: "Rose *name* produced many rosebuds!"
- Grasses create grains: "Grass *name* produced many grains!"

Create a grass, a rose plant, and a pine tree and have them all produce seeds in their own ways.

B.3 Final version



In the diagram you see part of the plant taxonomy. Non-flowering seed producers such as pine trees produce seeds through cones. Flowering plants have their seeds enclosed in a fruit that comes from a pollinated flower.
Implement the plant taxonomy in the diagram using different classes, with appropriate methods and attributes.

All plants should be able to produce seeds with seedProduction() which prints a string. Since plants produce seeds in their own way (cones or fruits), the string this method prints is different for every plant: for a pine tree, which is a non-flowering plant, it should print: "Pine tree *name* produced *nrOfCones* cones!". Here, *name* is a name you can choose for this specific pine tree. Every plant object should have a name. *nrOfCones* is the number of cones this specific pine tree produces. Every non-flowering plant should have this attribute

that specifies how many cones it produces when seedProduction() is called. For example, if you name a pine tree "western bristlecone" and say it produces 30 cones, seedProduction should print: "Pine tree western bristlecone produced 30 cones".

Flowering plants have to be pollinated before they can produce fruits that contain seeds. They should have a bool *pollinated* that starts out as false. When the method pollination() is called, *pollinated* turns true. Now the plant can create fruit. When a flowering plant tries to create a fruit when *pollinated* is false, the following string should be printed: "Plant *name* was not able to produce any fruit, since it is not pollinated". When *pollinated* is true, the string printed depends on the type of plant that it belongs to:

- Rose plants create rosebuds: "Rose *name* produced many rosebuds!"
- Grasses create grains: "Grass *name* produced many grains!"

Create a grass, a rose plant, and a pine tree and have them all produce seeds in their own ways.

C Appendix C: Questionnaire

Participant Identification

Please fill out your participant ID:
Participant ID (1) _____

Survey Questions

Exercise Difficulty

How difficult did you find the exercise?

Open-Ended Feedback

Q21. Do you have any comments/feedback on the main exercise (plant taxonomy)?

_____

_____

Study Programme

Which study programme are you following?

Computer Science (Informatica) (1)

Artificial Intelligence (kunstmatige intelligentie) (2)

Other: (3) _____

Year of Study

In what year of that study programme are you?

first year (1)

second year (2)

third year (3)

other: (4) _____

Outside Experience

What programming experience do you have besides your bachelor programme? (multiple answers possible)

none (1)

computer science in secondary school (middelbare school) (2)

from a job (bijbaan) (3)

from other formal education (4)

from programming as a hobby (5)

other: (6) _____

Experience with GitHub Copilot

Q13. Did you have any experience working with GitHub Copilot before participating in this study?

yes, I have tried it once/a few times (1)

yes, I work with it on some projects (2)

yes, I always have copilot activated while programming (3)

no (4)

other: (5) _____

Further GitHub Copilot Experience

Q14. How long have you been using GitHub Copilot?

_____

_____

Usage Context of GitHub Copilot

Q15. On which types of projects do you use Copilot? (hobby, uni work, etc.)

_____

_____

Usage of Additional AI Tools

Q16. What other AI tools do you use while programming?

none (1)

ChatGPT (2)

GPT-4 (3)

Other: (4) _____

D Appendix D: CUPS N-gram tables

3-gram	Count
('writing new functionality', 'writing new functionality', 'writing new functionality')	11
('writing new functionality', 'writing new functionality', 'thinking verifying suggestion')	4
('writing new functionality', 'thinking verifying suggestion', 'writing new functionality')	4

Table 8: Most Occurring 3-grams for Participant 1

4-gram	Count
('writing new functionality', 'writing new functionality', 'writing new functionality', 'writing new functionality')	5
('writing new functionality', 'writing new functionality', 'thinking verifying suggestion', 'writing new functionality')	3
('writing new functionality', 'thinking verifying suggestion', 'writing new functionality', 'writing new functionality')	2
('writing new functionality', 'writing new functionality', 'deferring thought for later', 'editing last suggestion')	2

Table 9: Most Occurring 4-grams for Participant 1

5-gram	Count
('writing new functionality', 'writing new functionality', 'thinking verifying suggestion', 'writing new functionality', 'writing new functionality')	2
('writing new functionality', 'writing new functionality', 'writing new functionality', 'thinking verifying suggestion', 'writing new functionality')	2
('reading exercise', 'writing new functionality', 'writing new functionality', 'writing new functionality', 'writing new functionality')	2
('writing new functionality', 'writing new functionality', 'writing new functionality', 'writing new functionality', 'Thinking about new code to write')	2
('writing new functionality', 'writing new functionality', 'writing new functionality', 'deferring thought for later', 'editing last suggestion')	2

Table 10: Most Occurring 5-grams for Participant 1

6-gram	Count
('writing new functionality', 'writing new functionality', 'writing new functionality', 'writing new functionality', 'thinking verifying suggestion', 'writing new functionality')	2
('writing new functionality', 'writing new functionality', 'thinking verifying suggestion', 'writing new functionality', 'writing new functionality', 'writing new functionality')	2
('writing new functionality', 'writing new functionality', 'writing new functionality', 'thinking verifying suggestion', 'writing new functionality', 'writing new functionality')	1
('Thinking about new code to write', 'writing new functionality', 'reading exercise', 'writing new functionality', 'writing new functionality', 'writing new functionality')	1
('writing new functionality', 'reading exercise', 'writing new functionality', 'writing new functionality', 'writing new functionality', 'writing new functionality')	1
('reading exercise', 'writing new functionality', 'writing new functionality', 'writing new functionality', 'writing new functionality', 'Thinking about new code to write')	1

Table 11: Most Occurring 6-grams for Participant 1

3-gram	Count
('writing new functionality', 'writing new functionality', 'writing new functionality')	20
('thinking about new code to write', 'writing new functionality', 'writing new functionality')	7
('writing new functionality', 'thinking verifying suggestion', 'writing new functionality')	7

Table 12: Most Occurring 3-grams for Participant 2

4-gram	Count
('writing new functionality', 'writing new functionality', 'writing new functionality', 'writing new functionality')	10
('writing new functionality', 'writing new functionality', 'thinking verifying suggestion', 'writing new functionality')	5
('writing new functionality', 'writing new functionality', 'writing new functionality', 'thinking verifying suggestion')	5
('thinking about new code to write', 'writing new functionality', 'writing new functionality', 'writing new functionality')	4

Table 13: Most Occurring 4-grams for Participant 2

5-gram	Count
('writing new functionality', 'writing new functionality', 'writing new functionality', 'writing new functionality', 'writing new functionality')	5
('writing new functionality', 'writing new functionality', 'writing new functionality', 'thinking verifying suggestion', 'writing new functionality')	4
('writing new functionality', 'writing new functionality', 'Reading exercise', 'writing new functionality', 'writing new functionality')	3
('writing new functionality', 'writing new functionality', 'thinking verifying suggestion', 'writing new functionality', 'writing new functionality')	2
('writing new functionality', 'writing new functionality', 'writing new functionality', 'writing new functionality', 'thinking verifying suggestion')	2

Table 14: Most Occurring 5-grams for Participant 2

6-gram	Count
('writing new functionality', 'writing new functionality', 'writing new functionality', 'writing new functionality', 'writing new functionality', 'writing new functionality')	3
('thinking about new code to write', 'writing new functionality', 'writing new functionality', 'writing new functionality', 'writing new functionality', 'writing new functionality')	2
('deferring thought for later', 'editing last suggestion', 'thinking about new code to write', 'writing new functionality', 'writing new functionality', 'writing new functionality')	2
('Reading exercise', 'writing new functionality', 'writing new functionality', 'writing new functionality', 'writing new functionality', 'thinking verifying suggestion')	2
('writing new functionality', 'writing new functionality', 'writing new functionality', 'writing new functionality', 'thinking verifying suggestion', 'writing new functionality')	2
('writing new functionality', 'writing new functionality', 'Reading exercise', 'writing new functionality', 'writing new functionality', 'writing new functionality')	2

Table 15: Most Occurring 6-grams for Participant 2

3-gram	Count
('writing new functionality', 'writing new functionality', 'writing new functionality')	6
('thinking verifying suggestion', 'thinking about new code to write', 'thinking verifying suggestion')	4
('thinking about new code to write', 'writing new functionality', 'thinking verifying suggestion')	3

Table 16: Most Occurring 3-grams for Participant 3

4-gram	Count
('writing new functionality', 'writing new functionality', 'writing new functionality', 'deferring thought for later')	3
('writing new functionality', 'thinking verifying suggestion', 'thinking about new code to write', 'thinking verifying suggestion')	2
('editing last suggestion', 'thinking verifying suggestion', 'writing new functionality', 'thinking verifying suggestion')	2
('thinking about new code to write', 'writing new functionality', 'thinking verifying suggestion', 'thinking about new code to write')	2

Table 17: Most Occurring 4-grams for Participant 3

5-gram	Count
('thinking verifying suggestion', 'thinking about new code to write', 'thinking verifying suggestion', 'thinking about new code to write', 'thinking verifying suggestion')	2
('thinking about new code to write', 'thinking about new code to write', 'writing new functionality', 'writing new functionality', 'writing new functionality')	2
('writing new functionality', 'writing new functionality', 'writing new functionality', 'deferring thought for later', 'thinking verifying suggestion')	2
('thinking about new code to write', 'writing new functionality', 'writing new functionality', 'writing new functionality', 'writing new functionality')	2
('thinking about new code to write', 'thinking about new code to write', 'thinking verifying suggestion', 'editing written code', 'debugging/testing code')	1

Table 18: Most Occurring 5-grams for Participant 3

6-gram	Count
('thinking about new code to write', 'thinking about new code to write', 'writing new functionality', 'writing new functionality', 'writing new functionality', 'writing new functionality')	2
('thinking verifying suggestion', 'writing new functionality', 'thinking verifying suggestion', 'editing last suggestion', 'writing new functionality', 'editing last suggestion')	1
('thinking about new code to write', 'thinking verifying suggestion', 'thinking about new code to write', 'thinking about new code to write', 'reading exercise', 'writing new functionality')	1
('thinking verifying suggestion', 'thinking about new code to write', 'thinking about new code to write', 'reading exercise', 'writing new functionality', 'editing written code')	1
('thinking about new code to write', 'thinking about new code to write', 'reading exercise', 'writing new functionality', 'editing written code', 'thinking about new code to write')	1
('thinking about new code to write', 'reading exercise', 'writing new functionality', 'editing written code', 'thinking about new code to write', 'writing new functionality')	1

Table 19: Most Occurring 6-grams for Participant 3

3-gram	Count
('deferring thought for later', 'deferring thought for later', 'deferring thought for later')	4
('writing new functionality', 'deferring thought for later', 'editing last suggestion')	3
('editing last suggestion', 'deferring thought for later', 'editing last suggestion')	3

Table 20: Most Occurring 3-grams for Participant 4

4-gram	Count
('deferring thought for later', 'deferring thought for later', 'deferring thought for later', 'deferring thought for later')	3
('deferring thought for later', 'editing last suggestion', 'deferring thought for later', 'editing last suggestion')	2
('editing last suggestion', 'writing new functionality', 'deferring thought for later', 'editing last suggestion')	2
('writing new functionality', 'thinking verifying suggestion', 'thinking about new code to write', 'reading exercise')	2

Table 21: Most Occurring 4-grams for Participant 4

5-gram	Count
('deferring thought for later', 'deferring thought for later', 'deferring thought for later', 'deferring thought for later', 'deferring thought for later')	2
('editing written code', 'writing new functionality', 'deferring thought for later', 'editing last suggestion', 'writing new functionality')	1
('writing new functionality', 'thinking verifying suggestion', 'thinking about new code to write', 'reading exercise', 'writing new functionality')	1
('thinking verifying suggestion', 'thinking about new code to write', 'reading exercise', 'writing new functionality', 'thinking verifying suggestion')	1
('thinking about new code to write', 'reading exercise', 'writing new functionality', 'thinking verifying suggestion', 'writing new functionality')	1

Table 22: Most Occurring 5-grams for Participant 4

6-gram	Count
('writing new functionality', 'reading exercise', 'writing new functionality', 'thinking verifying suggestion', 'thinking about new code to write', 'reading exercise')	1
('editing written code', 'writing new functionality', 'deferring thought for later', 'editing last suggestion', 'writing new functionality', 'deferring thought for later')	1
('writing new functionality', 'thinking verifying suggestion', 'thinking about new code to write', 'reading exercise', 'writing new functionality', 'thinking verifying suggestion')	1
('thinking verifying suggestion', 'thinking about new code to write', 'reading exercise', 'writing new functionality', 'thinking verifying suggestion', 'writing new functionality')	1
('thinking about new code to write', 'reading exercise', 'writing new functionality', 'thinking verifying suggestion', 'writing new functionality', 'writing new functionality')	1
('reading exercise', 'writing new functionality', 'thinking verifying suggestion', 'writing new functionality', 'writing new functionality', 'writing new functionality')	1

Table 23: Most Occurring 6-grams for Participant 4

3-gram	Count
('prompt crafting', 'prompt crafting', 'prompt crafting')	20
('thinking verifying suggestion', 'waiting for suggestion', 'thinking verifying suggestion')	9
('prompt crafting', 'prompt crafting', 'thinking verifying suggestion')	8

Table 24: Most Occurring 3-grams for Participant 5

4-gram	Count
('prompt crafting', 'prompt crafting', 'prompt crafting', 'prompt crafting')	11
('prompt crafting', 'prompt crafting', 'prompt crafting', 'thinking verifying suggestion')	7
('thinking verifying suggestion', 'waiting for suggestion', 'thinking verifying suggestion', 'waiting for suggestion')	5
('prompt crafting', 'thinking verifying suggestion', 'waiting for suggestion', 'thinking verifying suggestion')	5

Table 25: Most Occurring 4-grams for Participant 5

5-gram	Count
('prompt crafting', 'prompt crafting', 'prompt crafting', 'prompt crafting', 'prompt crafting')	7
('prompt crafting', 'prompt crafting', 'prompt crafting', 'prompt crafting', 'thinking verifying suggestion')	4
('prompt crafting', 'prompt crafting', 'prompt crafting', 'thinking verifying suggestion', 'prompt crafting')	3
('prompt crafting', 'prompt crafting', 'thinking verifying suggestion', 'prompt crafting', 'thinking verifying suggestion')	3
('thinking verifying suggestion', 'waiting for suggestion', 'thinking verifying suggestion', 'waiting for suggestion', 'thinking verifying suggestion')	3

Table 26: Most Occurring 5-grams for Participant 5

6-gram	Count
('prompt crafting', 'prompt crafting', 'prompt crafting', 'prompt crafting', 'prompt crafting', 'thinking verifying suggestion')	4
('prompt crafting', 'prompt crafting', 'prompt crafting', 'prompt crafting', 'prompt crafting', 'prompt crafting')	3
('prompt crafting', 'prompt crafting', 'prompt crafting', 'thinking verifying suggestion', 'waiting for suggestion', 'thinking verifying suggestion')	3
('prompt crafting', 'prompt crafting', 'thinking verifying suggestion', 'waiting for suggestion', 'thinking verifying suggestion', 'waiting for suggestion')	2
('reading exercise', 'prompt crafting', 'prompt crafting', 'prompt crafting', 'reading exercise', 'prompt crafting')	2
('thinking verifying suggestion', 'waiting for suggestion', 'thinking verifying suggestion', 'waiting for suggestion', 'deferring thought for later', 'waiting for suggestion')	2

Table 27: Most Occurring 6-grams for Participant 5

3-gram	Count
('thinking about new code to write', 'thinking about new code to write', 'thinking about new code to write')	3
('editing written code', 'thinking about new code to write', 'thinking about new code to write')	2
('looking up documentation', 'editing written code', 'thinking about new code to write')	2

Table 28: Most Occurring 3-grams for Participant 6

4-gram	Count
('looking up documentation', 'editing written code', 'thinking about new code to write', 'thinking about new code to write')	2
('writing new functionality', 'writing new functionality', 'writing new functionality', 'reading exercise')	2
('editing written code', 'thinking about new code to write', 'thinking about new code to write', 'thinking about new code to write')	2
('thinking about new code to write', 'looking up documentation', 'thinking about new code to write', 'reading exercise')	1

Table 29: Most Occurring 4-grams for Participant 6

5-gram	Count
('looking up documentation', 'editing written code', 'thinking about new code to write', 'thinking about new code to write', 'thinking about new code to write')	2
('thinking about new code to write', 'looking up documentation', 'thinking about new code to write', 'reading exercise', 'writing new functionality')	1
('writing new functionality', 'editing written code', 'writing new functionality', 'writing new functionality', 'writing new functionality')	1
('editing written code', 'writing new functionality', 'writing new functionality', 'writing new functionality', 'reading exercise')	1
('writing new functionality', 'writing new functionality', 'writing new functionality', 'reading exercise', 'writing new functionality')	1

Table 30: Most Occurring 5-grams for Participant 6

6-gram	Count
('reading exercise', 'looking up documentation', 'writing new functionality', 'editing written code', 'writing new functionality', 'writing new functionality')	1
('looking up documentation', 'writing new functionality', 'editing written code', 'writing new functionality', 'writing new functionality', 'writing new functionality')	1
('writing new functionality', 'editing written code', 'writing new functionality', 'writing new functionality', 'writing new functionality', 'reading exercise')	1
('editing written code', 'writing new functionality', 'writing new functionality', 'writing new functionality', 'reading exercise', 'writing new functionality')	1
('writing new functionality', 'writing new functionality', 'writing new functionality', 'reading exercise', 'writing new functionality', 'writing new functionality')	1
('writing new functionality', 'writing new functionality', 'reading exercise', 'writing new functionality', 'writing new functionality', 'writing new functionality')	1

Table 31: Most Occurring 6-grams for Participant 6

3-gram	Count
('writing new functionality', 'writing new functionality', 'writing new functionality')	40
('prompt crafting', 'prompt crafting', 'prompt crafting')	20
('writing new functionality', 'writing new functionality', 'thinking verifying suggestion')	14

Table 32: Most Occurring 3-grams for All Participants

4-gram	Count
('writing new functionality', 'writing new functionality', 'writing new functionality', 'writing new functionality')	17
('prompt crafting', 'prompt crafting', 'prompt crafting', 'prompt crafting')	11
('writing new functionality', 'writing new functionality', 'writing new functionality', 'thinking verifying suggestion')	9
('writing new functionality', 'writing new functionality', 'thinking verifying suggestion', 'writing new functionality')	8

Table 33: Most Occurring 4-grams for All Participants

5-gram	Count
('prompt crafting', 'prompt crafting', 'prompt crafting', 'prompt crafting', 'prompt crafting')	7
('writing new functionality', 'writing new functionality', 'writing new functionality', 'writing new functionality', 'writing new functionality')	6
('writing new functionality', 'writing new functionality', 'writing new functionality', 'thinking verifying suggestion', 'writing new functionality')	6
('writing new functionality', 'writing new functionality', 'writing new functionality', 'writing new functionality', 'thinking verifying suggestion')	5
('writing new functionality', 'thinking verifying suggestion', 'writing new functionality', 'writing new functionality', 'writing new functionality')	5

Table 34: Most Occurring 5-grams for All Participants

6-gram	Count
('writing new functionality', 'writing new functionality', 'thinking verifying suggestion', 'writing new functionality', 'writing new functionality', 'writing new functionality')	4
('prompt crafting', 'prompt crafting', 'prompt crafting', 'prompt crafting', 'prompt crafting', 'thinking verifying suggestion')	4
('writing new functionality', 'writing new functionality', 'writing new functionality', 'writing new functionality', 'thinking verifying suggestion', 'writing new functionality')	4
('prompt crafting', 'prompt crafting', 'prompt crafting', 'thinking verifying suggestion', 'waiting for suggestion', 'thinking verifying suggestion')	3
('writing new functionality', 'writing new functionality', 'writing new functionality', 'writing new functionality', 'writing new functionality', 'writing new functionality')	3
('prompt crafting', 'prompt crafting', 'prompt crafting', 'prompt crafting', 'prompt crafting', 'prompt crafting')	3

Table 35: Most Occurring 6-grams for All Participants

E Appendix E: Bigram Heatmaps

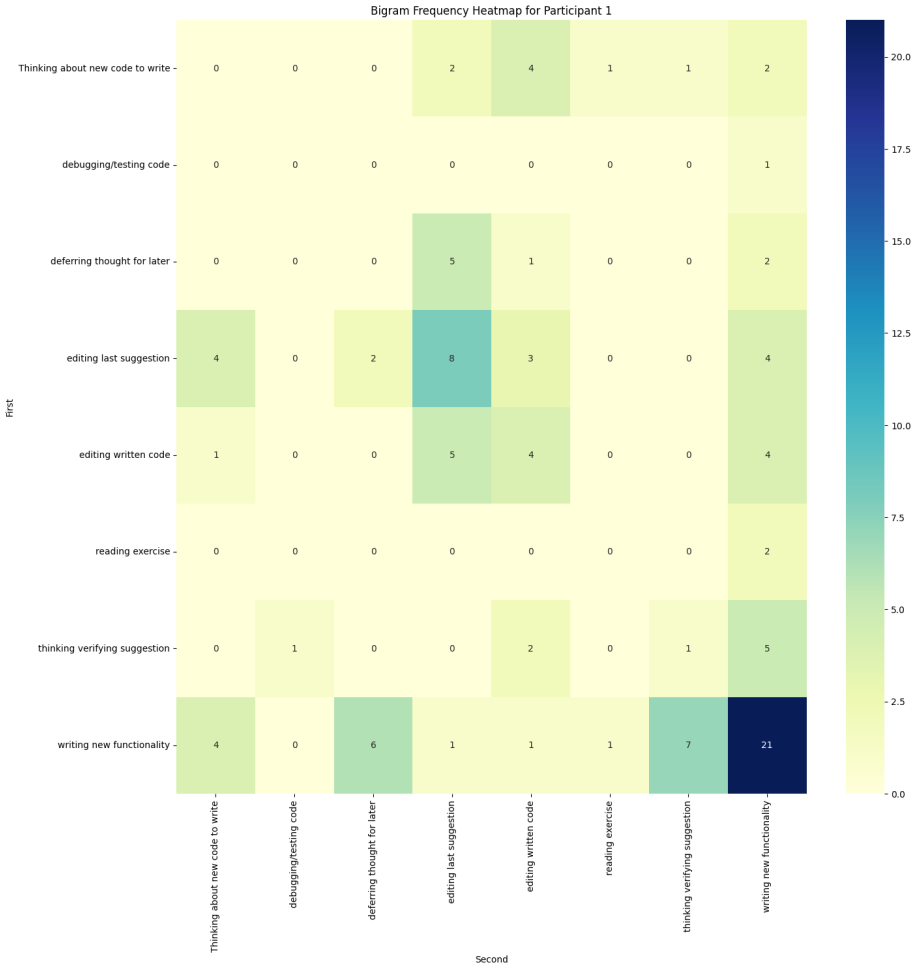


Figure 9: Bigram Frequency Heatmap for Participant 1

Figure 10: Bigram Frequency Heatmap for Participant 2

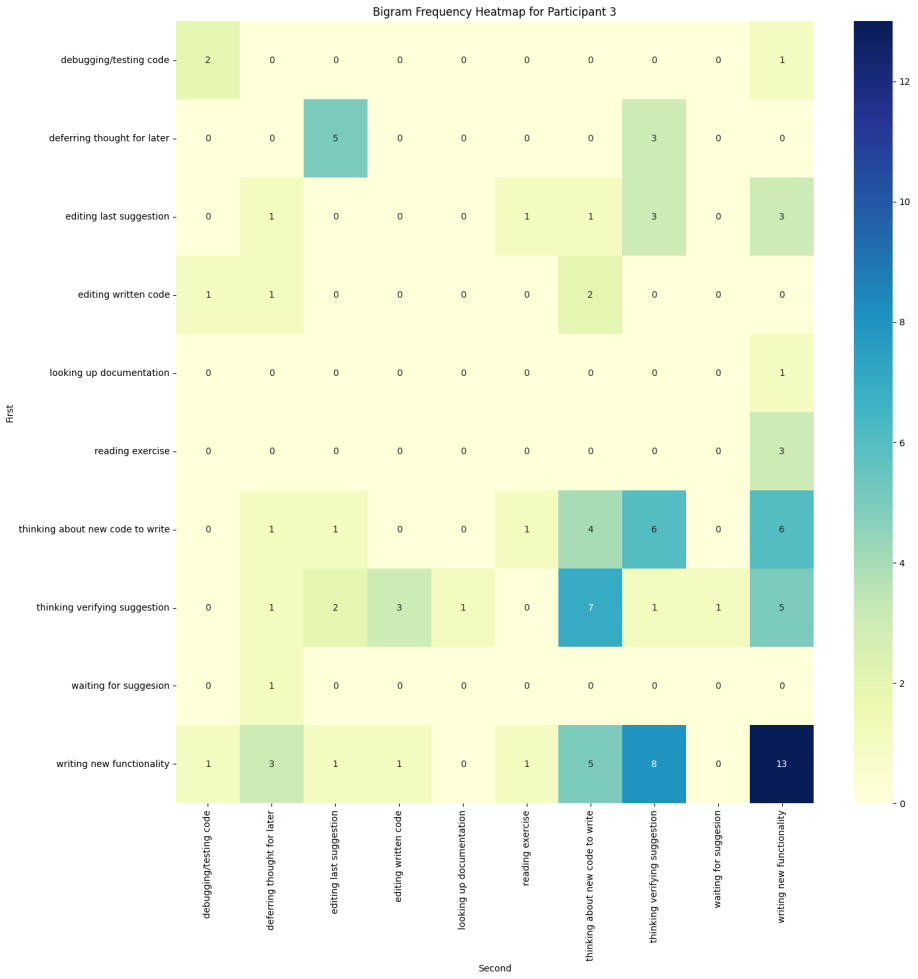


Figure 11: Bigram Frequency Heatmap for Participant 3

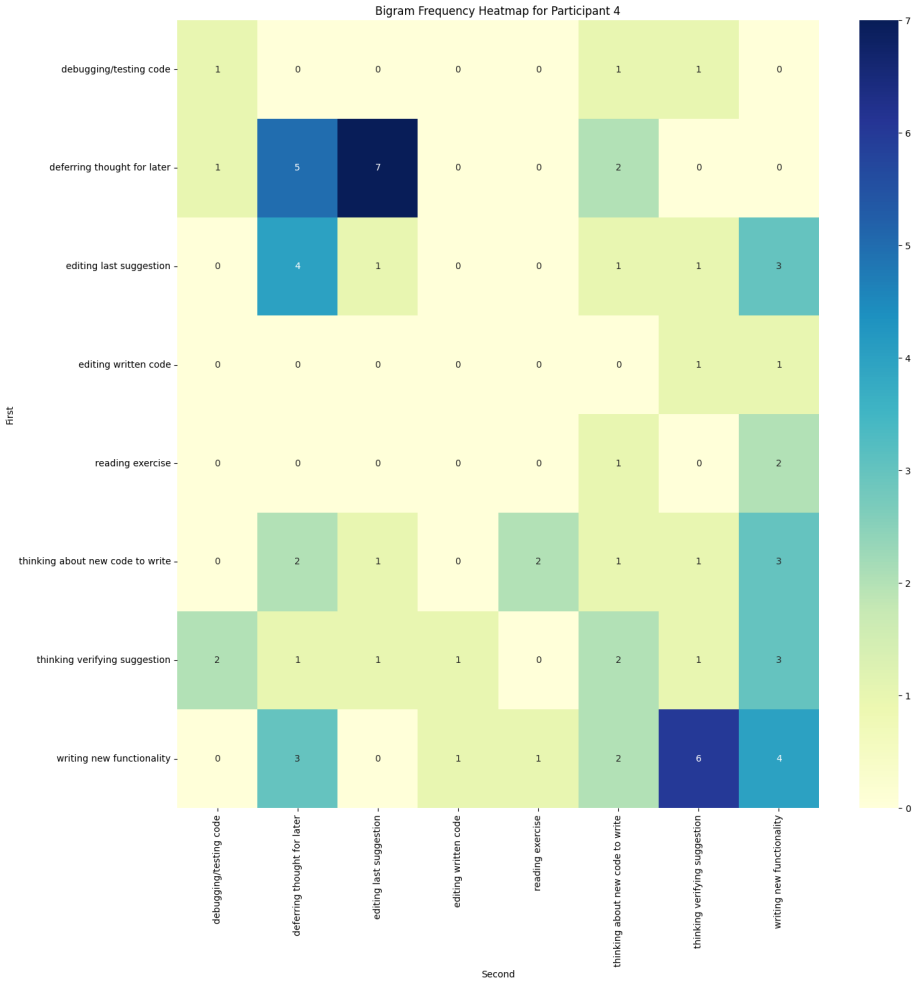


Figure 12: Bigram Frequency Heatmap for Participant 4

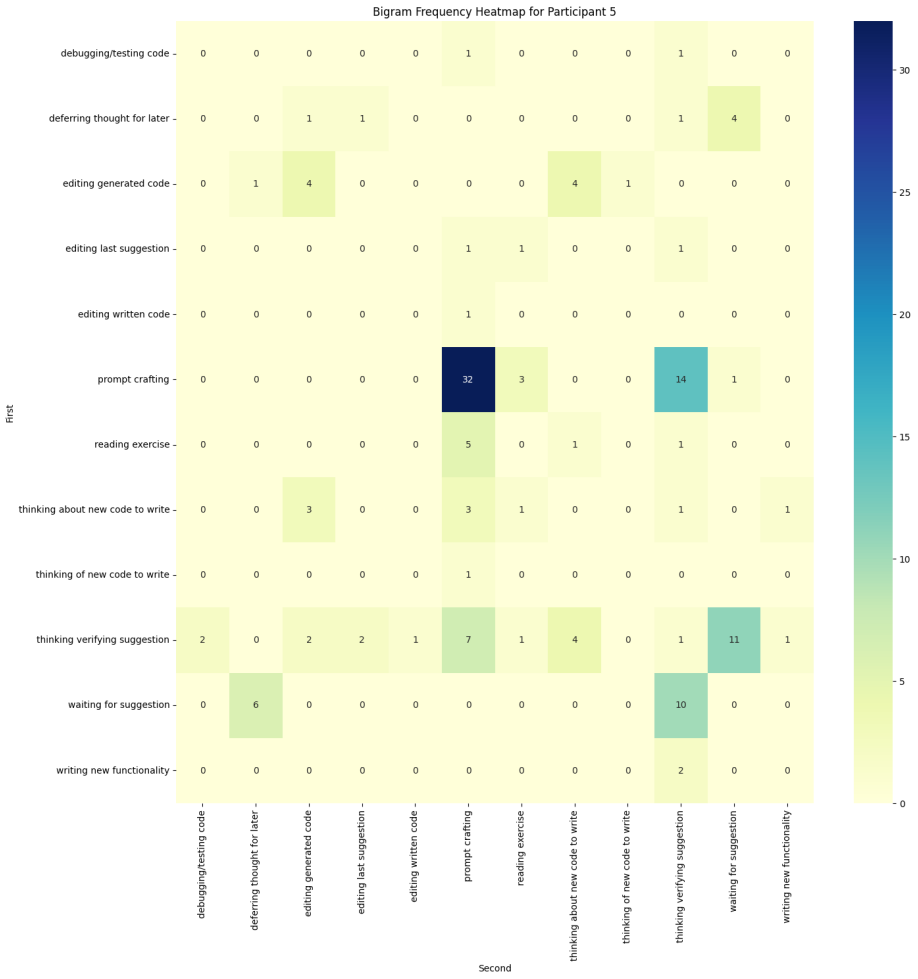


Figure 13: Bigram Frequency Heatmap for Participant 5

Figure 14: Bigram Frequency Heatmap for Participant 6