UNIVERSITY OF PITTSBURGH

INFSCI 2970 / ISSP 2990
INDEPENDENT STUDY
FALL 2023

Worked Example Authoring Tool: Code Explanation Generation with ChatGPT

Author:
Mohammad HASSANY,
Arun-Balajiee
LEKSHMI-NARAYANAN

Supervisor:
Dr. Peter BRUSILOVSKY

A report submitted in fulfillment of the requirements for Independent study

in the

PAWS Lab
Information Science Program & Intelligent Systems Program
School of Computing and Information

December 17, 2023

UNIVERSITY OF PITTSBURGH

Abstract

Information Science Program & Intelligent Systems Program
School of Computing and Information

Worked Example Authoring Tool: Code Explanation Generation with ChatGPT

by Mohammad HASSANY,
Arun-Balajiee LEKSHMI-NARAYANAN

Worked examples (solutions to typical programming problems presented as a source code in a certain language and are used to explain the topics from a programming class) are among the most popular types of learning content in programming classes. Most approaches and tools for presenting these examples to students are based on line-by-line explanations of the example code. However, instructors rarely have time to provide line-by-line explanations for a large number of examples typically used in a programming class. In this paper, we explore and assess a human-AI collaboration approach to authoring worked examples for Java programming. We introduce an authoring system for creating Java worked examples that generates a starting version of code explanations and presents it to the instructor to edit if necessary. We also present a study that assesses the quality of explanations created with this approach.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Program code examples play a crucial role in learning how to program (Linn and Clancy, 1992). Instructors use examples extensively to demonstrate the semantics of the programming language being taught and to highlight the fundamental coding patterns. Programming textbooks also pay a lot of attention to examples, with a considerable textbook space allocated to program examples and associated comments.

Through this practice, worked code examples emerged as an important type of learning content in programming classes. Following the tradition established by a number of programming textbooks (Deitel and Deitel, 1994; Kelley and Pohl, 1995), a typical worked example presents a code for solving a specific programming problem and explains the role and function of code lines or code chunks. In textbooks, these explanations are usually presented as comments in the code or as explanations on the margins. While informative, this approach focused on passive learning, which is known for its low efficiency. Recognizing this problem, several research teams developed learning tools that offered more interactive and engaging ways to learn from examples (Brusilovsky, Yudelson, and Hsiao, 2009; Sharrock et al., 2017; Khandwala and Guo, 2018; Park et al., 2018; Hosseini et al., 2020). These tools demonstrated their effectiveness in classroom studies, but their practical impact, i.e., broader use by programming instructors was limited due to the *authoring bottleneck*. Although the authors of example-focused learning tools usually provide a good set of worked examples that can be presented through their tools, many instructors prefer to use their own favorite code examples. The instructors are usually happy to broadly share the code of examples they created (usually providing it on the course Web page), but they rarely have time or patience to augment examples with explanations and add their examples to an example-focused interactive system. Indeed, producing a single explained example could take 30 minutes or more, since it requires typing an explanation for each code line (Brusilovsky, Yudelson, and Hsiao, 2009; Hosseini et al., 2020) or creating a screencast in a specific format (Sharrock et al., 2017; Park et al., 2018).

The authoring bottleneck has been recognized by several research teams, which have offered several ways to address it. Among the approaches explored are learner-sourcing, that is, engaging students in creating and reviewing explanations for instructor-provided code (Hsiao and Brusilovsky, 2011) and automatic extraction of information content from available sources, such as lecture recordings (Khandwala and Guo, 2018). In this work, we present an alternative approach to address the authoring bottleneck based on human-AI collaboration. With this approach, the instructor provides the code of one of their favorite examples along with the statement of the programming problem it is solving. The AI engine based on large language models (LLM) examines the code and generates explanations for each code line. The explanations could be reviewed and edited by the instructor. To support and explore this authoring approach, we created an authoring system, which radically

decreases the time to create a new interactive worked example. The examples created by the system could be uploaded to an example-exploration system such as WebEx (Brusilovsky, Yudelson, and Hsiao, 2009) or PCEX (Hosseini et al., 2020) or exported in a reusable format. To assess the quality of the resulting examples, we performed a user study in which TAs and students compared code explanations created by experts through a traditional process with examples created by AI to contribute to the human-AI collaborative process.

The remainder of the work is structured as follows. We start by reviewing related work, introduce the example authoring system that implements the proposed collaborative approach, and explain how specific design decisions were made through several rounds of internal evaluation. Next, we explain the design of our user study and review its results. We conclude with a summary of the work and plans for future research.

Chapter 2

Worked Examples

Code examples are important pedagogical tools for learning programming. Not surprisingly, considerable efforts have been devoted to the development of learning materials and tools to support students in studying code examples. Hosseini (Hosseini et al., 2020) classified program examples that have been used in teaching and learning to program into two groups, according to their primary instructional goal: *program behavior examples* and *program construction examples*. Program behavior examples demonstrate the semantics (i.e., behavior) of various programming constructs (i.e., what is happening inside a program or an algorithm when it is executed). Program construction examples attempt to communicate important programming patterns and practices by demonstrating the construction of a program that achieves various meaningful purposes. (e.g., summing an array). This distinction might not be clear-cut for code-only examples, since the same code could be used for both purposes. However, attempts to augment examples with learning technologies to increase their instructional value (i.e., adding code animation or explanations) usually focus on one of these goals.

Program behavior examples have been extensively studied. While textbooks still explain program behavior by using textual comments attached to lines of program code, a more advanced method for this purpose — *program visualization*, which visually illustrates the runtime behavior of computer programs — is now considered state-of-the-art. Over the past three decades, several specialized educational tools for observing and exploring program execution in a visual form have been built and assessed (Sorva, Karavirta, and Malmi, 2013).

Computer-based technologies for presenting program construction examples are less explored. For many years, the state-of-the-art approach for presenting worked code examples in online tools was simply code text with comments (Linn and Clancy, 1992; Davidovic, Warren, and Trichina, 2003; Morrison et al., 2016). More recently, this approach has been enhanced with multimedia by adding audio narrations to explain the code (Ericson, Guzdial, and Morrison, 2015) or by showing video fragments of code screencasts with the instructor's narration being heard while watching code in slides or an editor window (Sharrock et al., 2017; Khandwala and Guo, 2018). Both ways, however, support *passive* learning, which is the least efficient approach from the prospect of the ICAP framework (Chi et al., 2018)[1]

An attempt to make learning from program construction examples *active* was made in the WebEx system, which allowed students to interactively explore instructor-provided line-by-line comments for program examples via a web-based interface (Brusilovsky, Yudelson, and Hsiao, 2009). More recently, several projects (Khandwala and Guo, 2018; Park et al., 2018; Hosseini et al., 2020) augmented examples with simple problems and other constructive activities to elevate the example study

[1]The ICAP framework differentiates four modes of engagement, behaviorally exhibited by learners: *passive, active, constructive and interactive.*

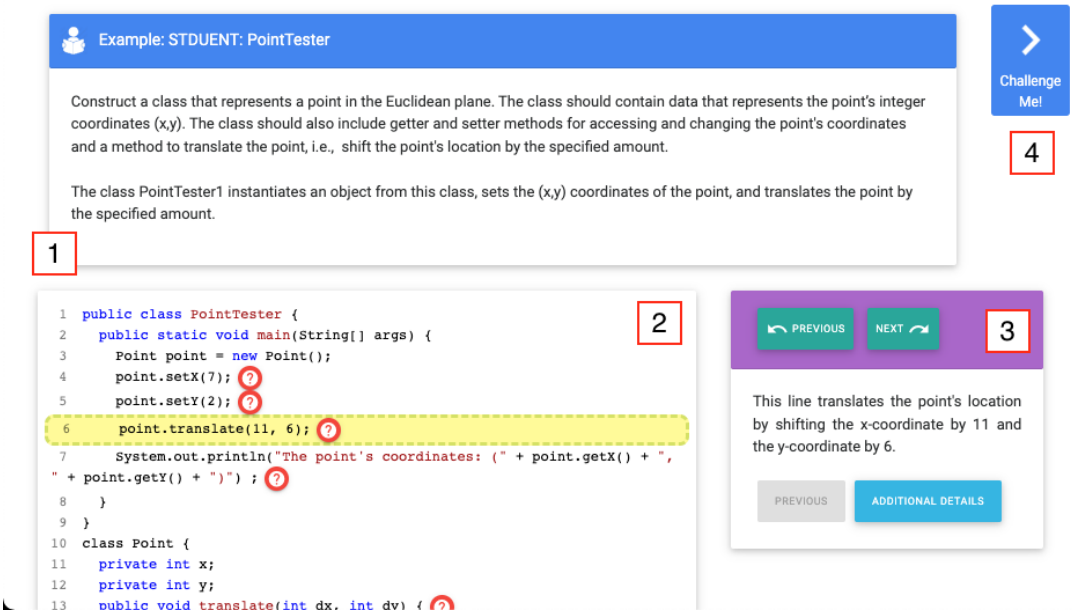

FIGURE 2.1: Studying a code example in the PCEX system: 1) title and program description, 2) program source code with lines annotated with explanations, 3) explanations for the highlighted line, 4) link to a "challenge" - a small problem related to the example.

process to the *interactive and constructive* levels of the ICAP framework, known as the most pedagogically efficient.

A good example of a modern interactive tool for studying code examples is the PCEX system Hosseini et al., 2020. PCEX (Program Construction EXamples) was created in the context of an NSF Infrastructure project (https://cssplice.org) with a focus on broad reuse and has been used by several universities in the US and Europe in the context of Java, Python, and SQL courses. PCEX interface (Figure 2.1) provides interactive access to traditionally organized worked examples, i.e., code lines augmented with instructor's explanations. Separating explanations (Figure 2.1-3) from the code (Figure 2.1-2), allows students to study explanations for code lines they want selectively. Explanations are provided on several levels of detail, so more details could be requested if the brief explanation is insufficient (Figure 2.1-3).

Since line-by-line multi-level example explanations offered by PCEX is currently the most detailed approach for explaining worked examples, we selected the code example structure implemented by PCEX as the target model for our authoring tool introduced in the next section. The tool produces code augmented with line-by-line explanations on several levels of detail. The resulting example could be directly uploaded to PCEX or exported in a system-independent format to be uploaded to other example exploration systems like WebEx(Brusilovsky, Yudelson, and Hsiao, 2009).

Chapter 3

Worked Example Authoring Tool

Although PCEX proved to be beneficial for students, it suffers from authoring bottleneck. Previously, the author should have annotated a source file using special tags. The author should add the required metadata to the source file and lines of code in the form language-specific comments. Not only these tags have their learning curve, they are prone to errors. There was no tool support, no highlighting, auto-completion, syntax error highlighting, or any other type of support. This made the authoring process cumbersome. Instructors with limited time would not spend time learning the PCEX annotation language, and create worked example with it. There should be a solution to this.

Worked Example Authoring Tool - WEAT (Figure 3.1) is a graphical user interface to address authoring bottleneck in PCEX. WEAT provide a GUI so that author - anyone including an instructor could create their worked examples and challenges, bundle them up as an activity and share them with others.

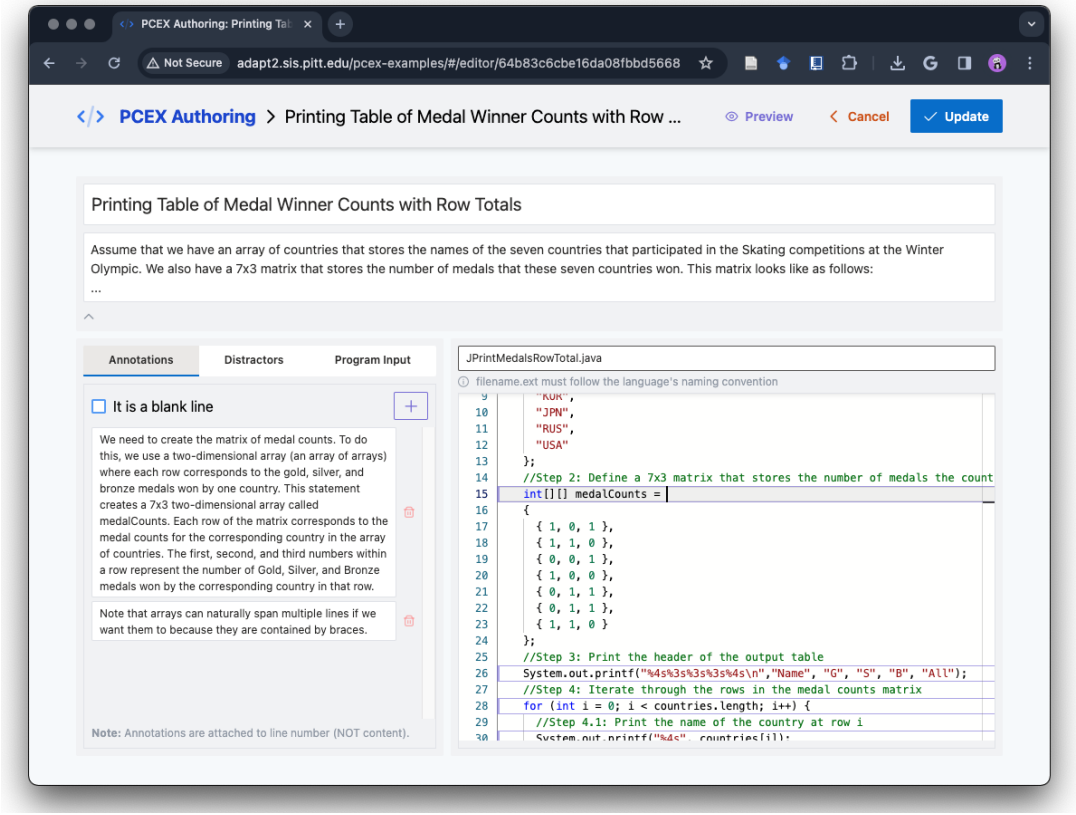

FIGURE 3.1: WEAT: A GUI authoring tool for PCEX.

In the PCEX, there are two important type of resources: Activities and Sources (Figure 3.2). A *source* is a worked example - an annotated Python or Java program. Each worked example has a title, problem statement, and a source code with its line being annotated with explanations. PCEX also has program challenges, allowing students to test their understanding of the concept being presented in the worked example. In the PCEX, a program challenge is created by annotating a line of code as a "blank line" and then providing "distractors" for them. Distractors are possible solutions for the marked empty line. Student will be able to drag these distractors into the blank line as the solution, and check if they were correct. An *activity* is the way to bundle multiple sources to create a learning activity. In the context of PCEX, a learning activity is a number of worked examples and program challenges, that students can use to learn the presented concepts and test their understanding of those concepts.

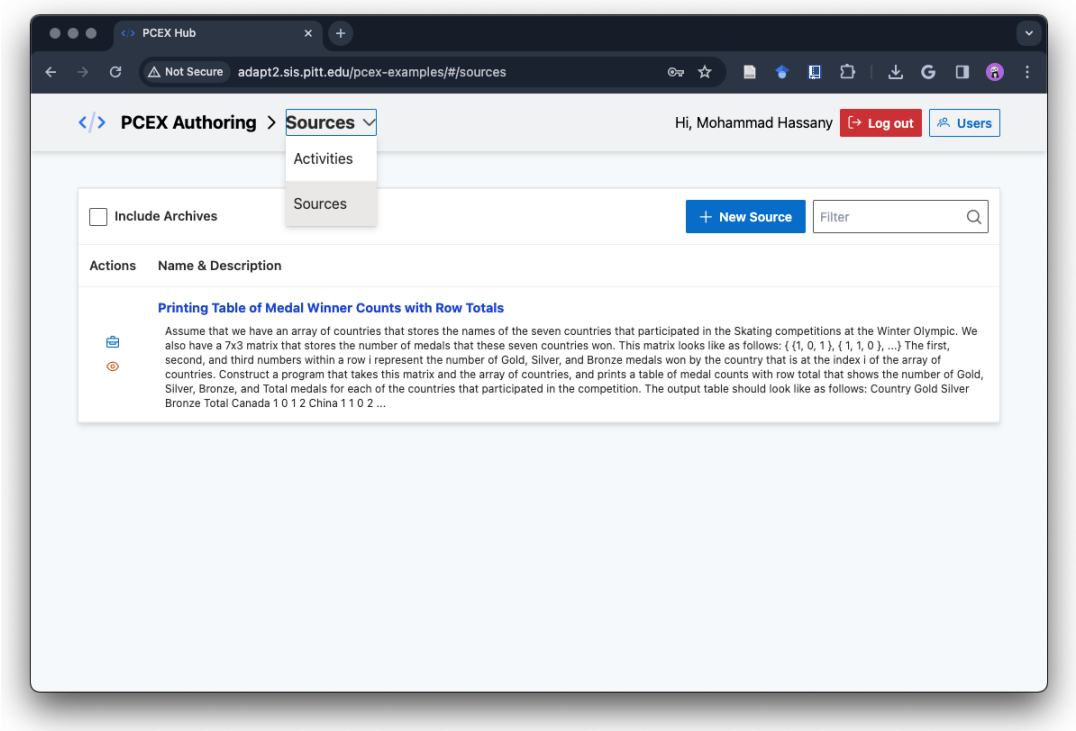

FIGURE 3.2: WEAT Home Page: List of sources authored by instructor.

Figure 3.1 shows the authoring window for a worked example. Title is an identifier given by the author. It is suggested that this title be informative, so that author and student can identify its purpose. Problem statement comes after the title. It describes the intent of the example. In the bottom right, the source for this example should be added. It is important to note that the filename of the source code should match the class name if the example is in Java. In the bottom left, the "Annotations" tab house the explanations provided for a line. To provide explanation for a line, author must put the cursor in the interested line, then click the add button on the left to create a explanation placeholder which explanation should be typed into. Every line can be annotated with multiple explanations. By checking the "It is a blank line" option, the selected line will be marked as a blank line which in the "Distractors" tab, author should provide possible solutions for these blank lines. Distractors are one-line code snippets that students can place into the blank line when using the PCEX

challenges. At anytime, author can click on the "Preview" on the top-right to view the current example as a PCEX example. Later, author can bundle these sources as a single activity - in the "Activities" page.

After authoring their examples, authors have the option to share their activities with others. WEAT features a Hub (Figure 3.3), where the publicly shared activities by authors are available. Everyone can search through them and use them.

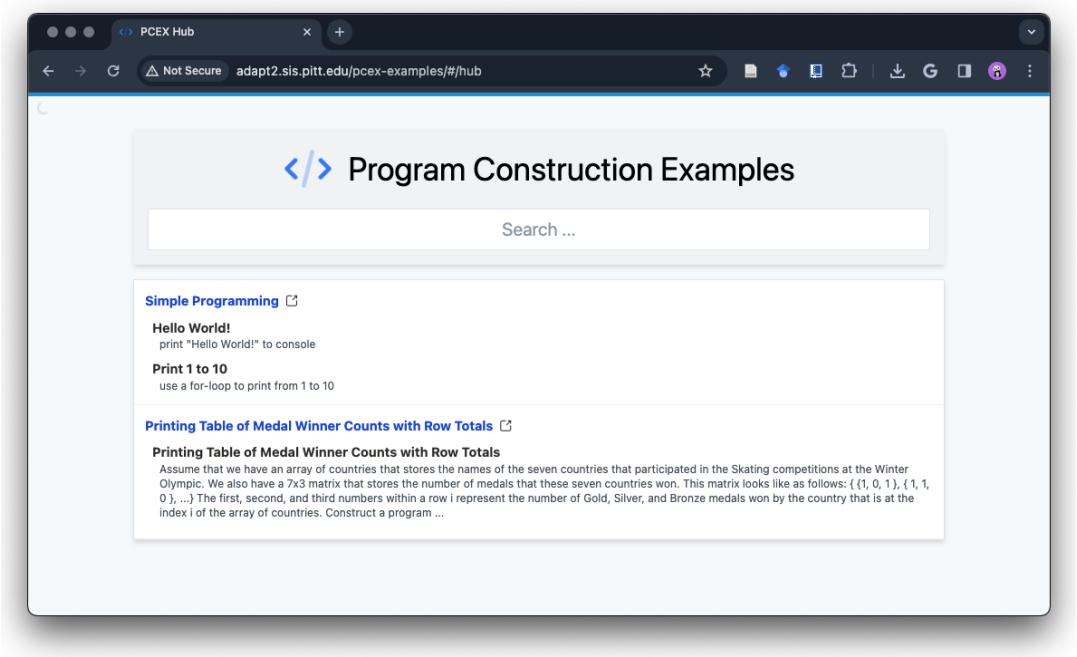

FIGURE 3.3: WEAT Hub: Publicly shared activities are accessible by the public.

Chapter 4

ChatGPT Generated Explanations

4.1 Use of LLMs for Code Explanations

Multiple researchers have explored code summarization (Phillips et al., 2022) and explanations using transformer models (Choi et al., 2023; Peng et al., 2022), abstract syntax trees (Shi et al., 2022), and Tree-LSTM (Tian et al., 2023). With the announcement of ChatGPT, several research teams explored the use of LLMs for code explanations using GPT 3 (Zamfirescu-Pereira et al., 2023; MacNeil et al., 2023; Leinonen et al., 2023), GPT 3.5 (MacNeil et al., 2023; Li et al., 2023; Chen et al., 2023), GPT 4 (Li et al., 2023), OpenAI Codex (Sarsa et al., 2022; Tian et al., 2023; MacNeil et al., 2023), and GitHub Copilot (Chen et al., 2023). Table 4.1 presents a brief summary of the most important prior work.

In the prior work, LLMs were used to generate explanations at different levels of abstraction (line-by-line, step-by-step, and high-level summary). Sarsa et al. Sarsa et al., 2022 observed that ChatGPT can generate better explanations at low-level (lines). Li et al. (2023) used the result of specific-to-general generated explanations as one of the inputs to their LLM solver, trying to solve competitive-level programming problems more efficiently. A novel research tried to understand how non-experts approach LLMs (Zamfirescu-Pereira et al., 2023). They have identified common mistakes and provided advice for tool designers.

Explanations and summaries generated by these LLMs were mostly evaluated by authors (Sarsa et al., 2022), students (MacNeil et al., 2023; Leinonen et al., 2023), and tool users (Chen et al., 2023). Sarsa et al. (2022) reported a high correct ratio for generated explanations with minor mistakes that can be resolved by the instructor or teaching assistant. Students rated LLM-generated explanations as being useful, easier, and more accurate than learner-sourced explanations (Leinonen et al., 2023).

Prompt, as an essential part of communication, directly influences the LLM's performance. A verbose prompt will limit the LLM's ability to utilize its knowledge (Tian et al., 2023). Iterative prompts are proven to perform well (Zamfirescu-Pereira et al., 2023). In terms of code explanation, providing the source code and expected outcome is essential. Adding input/output examples can help generate better explanations. Although LLMs like ChatGPT can understand the natural language very well, researchers suggested writing the prompt as writing a code: following a structure and marking different parts of the prompt (Zamfirescu-Pereira et al., 2023). If possible, it is better to control the randomness of LLMs responses (for instance, adjusting the temperature to a lower value, perhaps 0). A temptation to allow non-expert form input prompts will not be any good, as Zamfirescu-Pereira et al. (2023) observed non-experts have misconceptions about LLMs and will struggle to come up with a well-formed prompt. Researchers believe that LLMs can be beneficial in environments where humans and AI can work together, where the human

Source	Goal	LLM(s)	Type of Explanations	Evaluation
Chen et al., 2023	Provide explanations for a code fragment selected in the IDE	GPT 3.5	Explain the selected code	Interview with students, teachers, and bootcamp tutors
Leinonen et al., 2023	Scaffold student's ability to understand and explain code	GPT 3	Explain the intended purpose of a function	Compare ChatGPT explanations with student/peer explanations
Li et al., 2023	Given the problem description and expert solution, ChatGPT is prompted to generate explanations	GPT 3.5 vs GPT 4	Program summary, used algorithm, step-by-step solution description, time complexity, etc	Generated explanations were evaluated by the human programming expert who authored the "oracle" solution
MacNeil et al., 2023	Generate specific explanation, summary, and concepts for a given code snippet	GPT 3 and Codex	line-by-line explanations, list of important concepts, high-level summary of the code	Students' ratings of explanations, and their utility time/count
Sarsa et al., 2022	Help introductory programming course teachers by creating programming exercises + test cases, and code explanations	Codex	step-by-step explanation, problem-statement-like description, high-level description	Internal evaluation, measuring the percentage of code being explained

TABLE 4.1: Prior works in using LLMs (ChatGPT/Codex) to generate code explanations.

can perform the expert evaluation and tune the responses generated by the AI while the AI performs the time-consuming manual tasks (White et al., 2023).

4.2 Human-AI Collaborative Authoring

Creating a new worked example for an interactive example-focused system such as PCEX is a time-consuming task even in systems that provide some authoring support. Practical instructors who need to create code examples rarely have this time, which results in the authoring bottleneck mentioned earlier. Our Worked Example Authoring Tool (WEAT) attempts to reduce this bottleneck by engaging ChatGPT in the human-AI collaborative authoring process. In this collaboration, the main task of a human author is to provide the code of the example and the statement of the problem that the code solves. The main task of ChatGPT is to generate the bulk of code line explanations on several levels of detail. As an option, a human author could edit and refine the text produced by ChatGPT to adapt it to the class goals and target students. As in any productive collaboration, each side does what it is best suited to do, leaving the challenging work to the partner. In the main part of the WEAT interface the problem (Figure 4.1-1) and the code (Figure 4.1-2) have to be provided by the instructor, while the explanations (Figure 4.1-3) are generated by ChatGPT. All generated explanations could be edited by the instructor, who could also turn a regular example into a challenge by marking some lines as blank (within a challenge, these lines can be replaced with distractors, one-line code snippets, by the student). Note that the WEAT interface allows the author to completely replace generated explanations or even create the whole example from scratch, without the help of AI, however, we do not expect that this option will be used frequently.

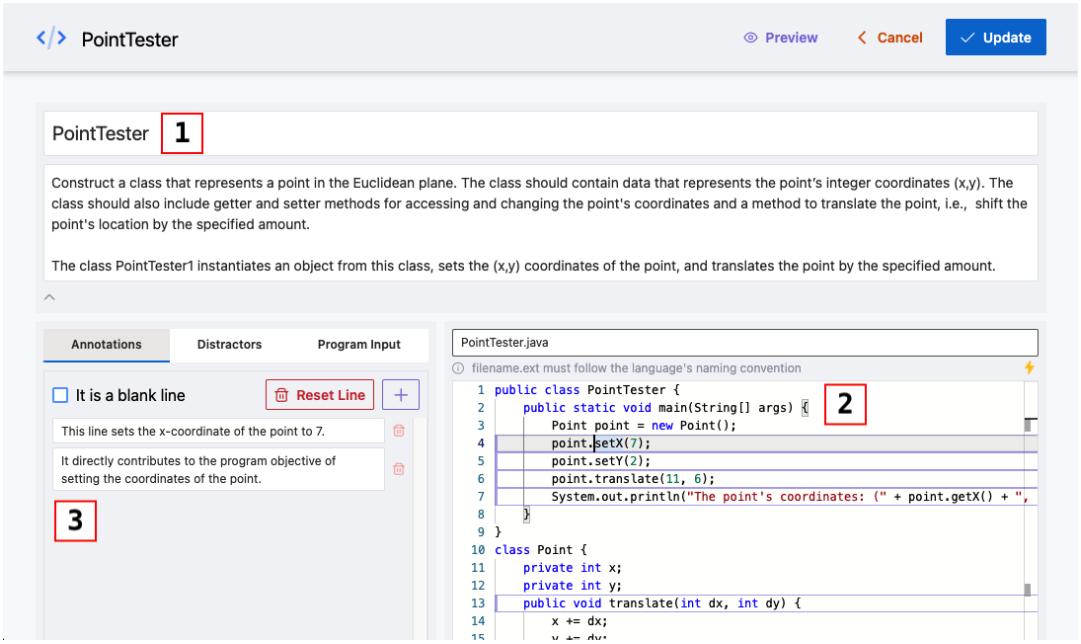

FIGURE 4.1: WEAT Authoring, 1) program title and description, 2) program source code (lines with explanations are marked with purple border), 3) explanations for the selected line (line 7 in the screenshot).

To generate ChatGPT explanations for the provided example code and problem description, the author has to click the small-yellow-bolt icon to open the ChatGPT dialog (Figure 4.2). In this dialog, the author should click "Generate" to generate the

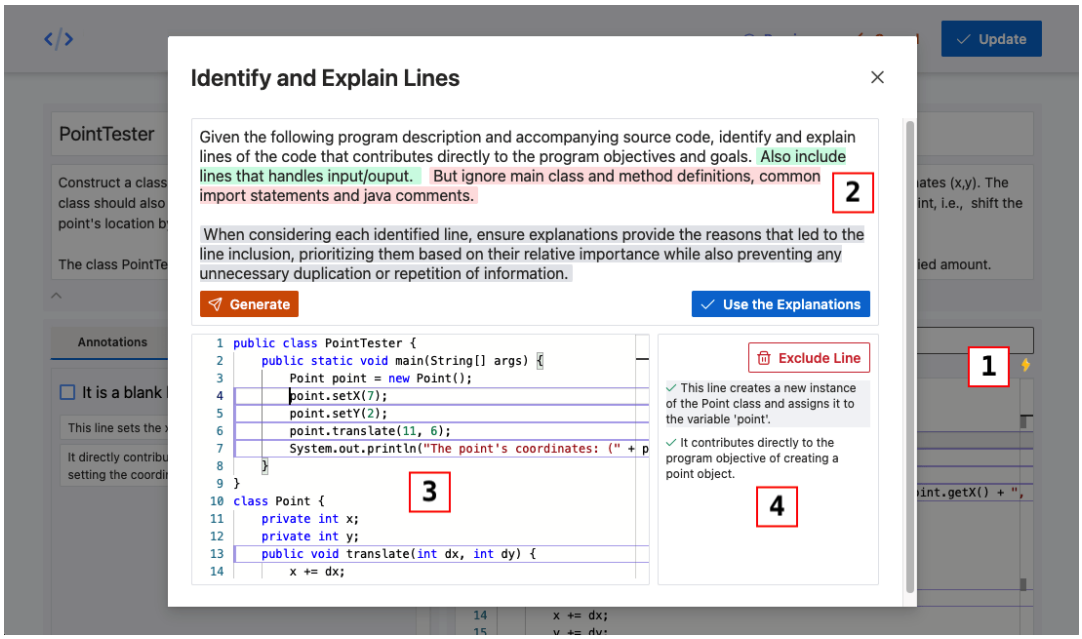

FIGURE 4.2: Human-AI Collaborative Worked Example Authoring, 1) open dialog button (small-yellow-bolt icon), 2) default prompt (author can tune the prompt - optional), 3) program source preview (lines with explanations are marked with a purple border), 4) generated explanations for the selected line.

explanations, and then click "Use the Explanations" to add them to the example. The WEAT provides the human author several opportunities to control the outcome of the explanation generation process: 1) the author can tune the prompt to their needs, 2) the author can decide whether to include or exclude a generated explanation and 3) the author can edit or remove the explanation after it is edited.

Chapter 5

Evaluation

5.1 Internal Evaluation: Prompt Tuning

Following the majority of recent work on generating code explanations, we chose ChatGPT as the target LLM to generate code explanations. ChatGPT provides an easy-to-use API and an affordable pricing model. Adding ChatGPT to an application is not a straightforward process and requires careful planning. The key part of this process is crafting a prompt, which requires multiple iterative trials. Following the suggestions in the previous work (Zhou et al., 2022; Chen et al., 2023), the authors used an internal evaluation process to engineer a prompt that produces high-quality explanations.

To shorten the prompt design process, we adopted several design decisions that were shown to be effective in previous work: assigning a role to ChatGPT (White et al., 2023), avoiding verbosity (Tian et al., 2023), repetition (Zamfirescu-Pereira et al., 2023), prompt that looks like code (Zamfirescu-Pereira et al., 2023), and defining the expected output format (Zamfirescu-Pereira et al., 2023). However, a few design decisions not evaluated previously were not evident, so we had to use an internal evaluation process to select the best-performing option. The questions answered through the evaluation included the following: 1) Does the presence of a program description in the prompt result in better explanations? 2) Does iterative prompting perform better than a single prompt, and if so, how many iterations are sufficient to have a good explanation? 3) Does adding line inclusion/exclusion criteria in the prompt help ChatGPT to select or ignore lines in generating an explanation? To answer these questions, we formally compared ChatGPT-generated explanations through an independent rating performed by the authors of this work and a colleague.

Since we started from previously explored prompting techniques, the first version of our prompt was reasonably close to our final prompt. At the first stage of the process, we made a few small corrections of the prompt based on observations. First, we observed that ChatGPT cannot associate the line number with the line correctly. To address this issue, we marked each line with its line number. We also observed that sometimes with iterative prompting ChatGPT generates duplicate explanations. Hence in our iterative prompts, we asked ChatGPT to generate explanations that are new. Figure 5.1 shows the final version of the prompt that we used with *ChatGPT gpt-3.5-turbo/16k* model (*temperature=0*) through OpenAI API for our internal and external evaluations.

Selecting Examples for Evaluation: We randomly selected eight Java examples with different difficulty levels (string operation, array, loop, and object-oriented programming) from the PCEX repository for the study. Selected examples include:

- *Initials*: Extracting initials from full name.

Role (System):

You are a professor who teaches computer programming.

Role (User) - Iteration #1:

Given the following program description and accompanying source code, identify and explain lines of the code that contributes directly to the program objectives and goals. **[inclusion criteria]** **[exclusion criteria]**

When considering each identified line, ensure explanations provide the reasons that led to the line inclusion, prioritizing them based on their relative importance while also preventing any unnecessary duplication or repetition of information.

Program Description:
[program description]

Program Source Code:
The line number is defined as /*line_num*/ at the start of each line.

'''java
/*1*/[program lines]
'''

Output format:
Reply ONLY with a JSON array where each element, representing a "line of code," includes "line_num" and an "explanations" array. For example:
'''json
[{ "line_num": "2", "explanations": ["explanation ...", "explanation ...", ...] }, ...]
'''

Role (User) - Iteration #2:

Update your explanations with more insightful and complementary YET COMPLETELY new explanations. If you missed a line, this is the time to include them.

Role (User) - Iteration #nth:

Please repeat that once more.

FIGURE 5.1: ChatGPT Prompt Template. ChatGPT (is given the "professor" role) is prompted iteratively.

- *JAdjacentDuplicates*: Checks whether a sequence of numbers contains adjacent duplicates.
- *JArrayIncrementElements*: Increments all elements of the array by 1.
- *JArrayMax*: Finds the maximum value in an array.
- *JPrintDigitsReverse*: Prints the digits of an integer from right to left.
- *JSearchArrayValues*: Search for values from one array in another.
- *JSmallestDivisor*: Smallest divisor of a positive number.
- *PointTester*: Translate 2-dimensional coordinates.

Including/Excluding Program Description: We hypothesized that adding a program description for the prompt adds information for ChatGPT to produce better explanations, but we were also concerned that it could confuse ChatGPT. To compare the quality of the explanation with and without description, the evaluators checked the explanations for the following: 1) correctness, 2) relevance to the given program description (when present), 3) presence of new information in the 2^{nd} round compared to the 1^{st} round, 4) presence of hallucinations when the program description is not present, 5) whether the 2^{nd} round with program description in the prompt had more information than without description. Both Correctness and Relevance were binary ratings. For example, given an explanation *This line initializes a variable 'fullName' and assigns it the value 'John Smith'. The 'fullName' variable stores the full name of the person whose initials are to be printed.* for the line of code *String fullName = "John Smith";* was rated as "correct" and "relevant" by one rater.

We, as internal evaluators, rated higher correctness in explanations for both rounds 1 and 2 of generation by ChatGPT ($R_1 = 99.23\%, R_2 = 98.77\%$) as summarized in the Table 5.1. As an interesting example, when observing the ratings that we used to compare the amount of new information generated in round 2 from round 1, we observed that more information is generated without program description (48.24%) than with description ((35.80%) as summarized in Table 5.2. The generated explanations when the program description is not present had additional information compared to when it is present as shown in Figure 5.2. This validates prior work that comprehensive prompts limit LLMs' ability to utilize their knowledge Tian et al., 2023. Additionally, when the program description is present, the authors selected the 2^{nd} round of explanations for the external evaluation because students relate better with the program and it is also rated higher for correctness. In the conditions that we did not include the program description in prompts, we were interested to know to what extent ChatGPT may hallucinate. We observed hallucinations 2.94% *on average* when considering prompts without program description, which could be attributed to greater information generation in round 2. Given this tendency to hallucinate when generating explanations with prompts that exclude problem descriptions, we considered using the explanations that are generated with prompts that include program descriptions.

Assessing Multi-Round Prompting: In this step, we assessed whether iterative prompting ChatGPT to explain (see Figure 5.1), results in additional explanations. When the program description was present in the prompt, only 3 out of 9 examples had additional explanations compared to none when not included (Table 5.3). Explanations generated in the 3^{rd} round were either minor wording changes (high cosine similarity) or included explanations for unnecessary lines (closing bracket for

	Round 1		Round 2		
	*C	**R	*C	**R	***A
Min	98.46%	44.62%	97.53%	37.04%	32.10%
Max	100.00%	70.77%	100.00%	68.75%	39.51%
Average*	99.23%	55.38%	98.77%	46.91%	35.80%

TABLE 5.1: Internal evaluators rating, when program description is present in the prompt, *Correctness, **Relevance to program description, ***Additional information compared to 1st round.

	Round 1		Round 2		
	*C	**H	*C	**H	***A
Min	93.98%	0.00%	92.94%	0.00%	41.18%
Max	100.00%	4.82%	100.00%	5.88%	55.29%
Average*	96.99%	2.41%	96.47%	2.94%	48.24%

TABLE 5.2: Internal evaluators rating, when program description is not present in the prompt, *Correctness, **Explanation contains hallucinations, ***Additional information compared to 1st round.

main method and class). Qualitatively assessing explanations generated in the 2^{nd} round, they included additional explanations or improved wordings. The number of additional explanations or improvements was not consistent among the examples in the 2^{nd} round, but on average, in 35.80% of lines (Figure 5.1) when the program description is present in the prompt, and in 48.24% of lines (Figure 5.2) when not, additional information was reported by the evaluators. Based on these findings, we decided to adopt a two-round prompting option for WEAT and used this option in the external evaluation process. We summarize our results in Table 5.3.

	with desc		without desc
Examples	R_2	R_3	R_2
Initials	88.8%	96.0%	90.7%
JAdjacentDuplicates	93.6%	99.0%	86.7%
JArrayIncrementElements	40.0%	–	93.3%
JArrayMax	85.7%	–	83.8%
JPrintDigitsReverse	57.1%	–	–
JSearchArrayValues	90.0%	71.4%	93.0%
JSmallestDivisor	46.3%	–	86.3%
PointTester	91.6%	–	73.3%

TABLE 5.3: Cosine similarity between rounds of explanations ($R_{n=2} = cosine\_sim(R_n, R_{n-1})$) with and without including program description.

Assessing Inclusion/Exclusion Criteria: A program description can provide a rich context for identifying and explaining lines of code. However, ChatGPT may sometimes include an unnecessary line or exclude a necessary one from the explanation. Initially, we assumed that directly adding inclusion/exclusion criteria in the ChatGPT prompt can address this issue. However, evaluating this option internally, the authors observed that it resulted in less than 1% new lines inclusion and around

Program description: Write a program that finds the maximum value in an array.

Line #	Line Content	With Description 2nd Round	Without Description 2nd Round
3	int[] values = { 5, -4, 78, 95, 12 };	This line initializes an array called 'values' with the given values. The array 'values' represents the set of numbers from which the maximum value needs to be found.	This line initializes an array called 'values' with the given values. The array 'values' holds the numbers to be evaluated to find the maximum value. **The values in the array can be modified or expanded to include any set of numbers.**

FIGURE 5.2: Example of line explanations in which the 2^{nd} round of explanation when the program description is not present had additional information compared to when present.

$4-6\%$ of lines exclusion. When these criteria are present in the prompt, ChatGPT ends up having unnecessary rounds of explanations. Sometimes ChatGPT falls into a loop where it flips wordings between each round. Since the author can review and ignore the explanations for a specific line in the authoring interface, we decided not to use Inclusion/Exclusion criteria in the prompt.

5.2 External Evaluation: ChatGPT vs Expert Explanation

To assess the quality of the explanations generated by ChatGPT using the best-performing prompt with options tuned through the internal evaluation, we performed a user study. In this study, we compared the explanations generated by ChatGPT with explanations created by experts for the same PCEX examples. Unlike some earlier studies that used beginner students to evaluate ChatGPT explanations, we used more experienced users – advanced undergraduate and graduate students. The reason for this difference is that in our authoring system, the direct users of the ChatGPT explanation are not *consumers* of explanations, but prospective *authors*. In the implemented human-AI collaborative authoring approach, authors have the option to edit the generated explanation. Thus, it is up to the prospective authors to decide how good the explanations are since their perception of quality impacts the amount of their work: poor explanations will require a lot of editing, while good explanations could be accepted as-is or with minimal changes.

To support these evaluation needs, we recruited 15 evaluators, of which 6 were graduate students researching computing education and 9 were undergraduate students who just completed an advanced Java programming class. Graduate students selected for the study usually serve as assistants or instructors in programming classes where supplementary content development is their major responsibility. For brevity, we refer to them as *authors* in our analysis. Advanced undergraduate students are frequently involved in learning content production through "learnersourcing" (Hsiao and Brusilovsky, 2011; Williams et al., 2016). To distinguish them from the true authors, we refer to them as *students*. Participants had to provide their responses through an evaluation form. The evaluation was estimated to take one hour to complete. Participants received a gift card of US $20 as compensation.

The evaluation form included 8 examples introduced above. For each example, the form included a program description and the example code. For each line of each code example, it listed an explanation generated for this line by ChatGPT and by an expert. The participants had to rate both explanations for a given line of code and compare them. The order of ChatGPT and expert explanations for a given line

of code was randomized, and the evaluators did not know which explanation was generated by ChatGPT or the expert. "Expert" explanations were extracted from real worked examples in PCEX system (Hosseini et al., 2018). These explanations were authored by instructors and TAs and polished through several years of classroom use.

To evaluate the explanations, the participants had to rate to what extent each explanation is *complete* and which is *better*. We defined a *better* explanation as "providing more information, going deeper, better connecting to programming concepts". However, we did not provide the definition to a *complete* explanation.

More specifically, participants had to rate the two explanations with the following metrics:

1. *Explanation 1 is sufficiently complete*: Not complete (0), Complete (1), Very complete (2)

2. *Explanation 2 is sufficiently complete*: Not complete (0), Complete (1), Very complete (2)

3. *Which explanation is better?* Both are the same (0), Explanation 1 is better (1), Explanation 2 is better (2)

From the collected responses, we excluded lines that had either ChatGPT or expert explanations but not both. In these cases, the evaluators generally rated the explanations as better without comparison with a missing counterpart explanation. Altogether, 18 lines were explained by ChatGPT but not by the expert, and 5 lines were explained by the expert but not ChatGPT. Looking closer at these lines, we observed that 4 out of 5 missing lines by ChatGPT were in the *PointTester* example, which included class definition, object instantiation, and instance variable definition. We are not aware of the reason why the expert didn't explain these lines, but we assume these lines are either mentioned in explanations generated for other lines or they don't provide important information toward understanding the program. Although the program description had related wordings, there were missed by ChatGPT: "*Construct a class* that represents... The class should contain *data that represents the point's integer coordinates(x, y)*. ... The class *PointTester instantiates an object from this class*, sets the (x, y) coordinates of the ...". Conversely, in 14 out of 18 of these lines, ChatGPT unnecessarily explained class, main method definition, and closing brackets (class, method, loop, and condition). The other 4 lines were informative and useful. This can support the importance of having inclusion criteria in the prompt.

For the remaining 45 lines of code, we observed from the evaluators' ratings for the question "Explanation 1 is sufficiently complete?" or "Explanation 2 is sufficiently complete?" that ChatGPT explanations were rated as 0.59% (not complete), 21.04% (complete) and 78.37% (very complete) compared to Expert explanations as 6.96% (not complete), 56.44% (complete), and 36.59% (very complete). In response to the question "Which explanation is better?", evaluators selected ChatGPT as the better explanation in 53.93% of lines, compared to experts (20.59%); and in the rest of the lines (25.48%) both were rated as the same. Our calculations of the inter-rater reliability for the ratings of the question "Which explanation is better?" using Fleiss-Kappa gave us 0.182, $p < 0.01$ score of agreement. This can be interpreted as "slight agreement" based on the 2-raters/2-categories table. Given that Fleiss-Kappa is a chance-corrected coefficient, it can be interpreted as a better agreement due to the high number of subjects (45 lines of code by 15 evaluators) Sim and Wright, 2005.

We observe that the students did not rate ChatGPT explanations incomplete at all with their 13.33% and 86.67% ratings being that ChatGPT explanations are complete and very complete respectively. Authors also rated ChatGPT explanations as complete (32.59%) or very complete (65.93%). Hence, a majority of authors and students find ChatGPT explanations complete, as shown in Table 5.4. In terms of comparing the explanations for which is better, 51.11% and 58.15% of students and authors, respectively, find that the explanations of ChatGPT are better for the given lines of code. On average, the authors rated the ChatGPT explanations more complete than students and students preferred ChatGPT explanations more than the authors, as summarized in Table 5.6. A direct comparison of two options, based on the question "Which explanation is better (ChatGPT vs Expert)?", is presented in Table 5.5. Given that the assessment was performed using blind rating, this is an encouraging result for the use of generative AI for authoring tools.

	Not complete=0	Complete=1	Very complete=2
ChatGPT			
Students	0.00%	13.33%	86.67%
Authors	1.48%	32.59%	65.93%
Overall	0.59%	21.04%	78.37%
Expert			
Students	2.22%	55.56%	42.22%
Authors	14.07%	57.78%	28.15%
Overall	6.96%	56.44%	36.59%

TABLE 5.4: Percentage of Ratings for different items on the scale for "Explanation 1 / 2 is sufficiently complete?"

	Explanation		
Rating	Students	Authors	Overall
Both are the same = 0	32.84%	14.44%	25.48%
Expert is better = 1	16.05%	27.41%	20.59%
ChatGPT is better = 2	51.11%	58.15%	53.93%

TABLE 5.5: Percentage of Ratings for the different items on the scale for "Which explanation is better?"

	All	Students	Authors
ChatGPT*	1.867 (0.133)	1.644 (0.258)	1.778 (0.163)
Expert*	1.400 (0.388)	1.141 (0.465)	1.296 (0.408)
Which is better?	1.183 (0.510)	1.437 (0.373)	1.284 (0.427)

TABLE 5.6: Average (Stdev) Ratings - *Completeness

Chapter 6

Conclusion and Future Work

In this paper, we introduce a worked example authoring tool that utilizes ChatGPT for the automatic generation of line-by-line code explanations. The tool is designed to allow human and AI to collaborate in the process of authoring such examples. To the best of our knowledge, this is the first attempt to produce worked example through human-AI collaboration. Our work supports findings by other researchers and provides empirical evidence on the value of using ChatGPT for generating line-by-line code explanations. Through an external evaluation, this work also compared the generated explanations and human expert explanations.

As the first step towards this important goal, our work has several limitations. First, the scale of our evaluation was relatively small. Since we targeted prospective authors as users in our evaluation process, we were able to recruit only 15 qualified subjects. Furthermore, within the time allocated for the study, the subjects were able to process only eight worked examples. Although we attempted to broadly vary the topics and difficulty of selected examples to achieve sufficient generalizability of results, a larger-scale study with a broader variety of examples might be necessary to obtain deeper insights. We plan to carry out such a study in our future work.

Although the use of the same best-performing prompt to generate explanations for examples of different difficulties was an important design decision to explore the generalizability of the approach, it might be possible that different prompts will perform best for examples of different difficulties. We will explore this opportunity in the next round of our work.

We also observed that for some lines of code in our dataset, experts, ChatGPT, or both choose to provide no explanations. In the current study, these lines were excluded from the evaluation since a meaningful comparison was not possible. However, choosing whether to explain a specific line or not is an important decision and the current study did not assess who is making better decisions about skipping lines, ChatGPT or experts. This aspect requires further investigation. In our next study, we plan to ask participant evaluators to specify whether each line of code needs an explanation or not.

Another potential limitation of the study was the lack of a formal definition of what a "complete" explanation means during external evaluation. We let the participants decide how to rate completeness since it is a personal decision, which editors should make when deciding whether to update generated explanation or not. While it was a natural thing to do, it might have decreased the agreement between evaluators. In our future work, we will see whether the agreement could be increased by defending correctness and completeness ratings more formally.

Finally, one aspect of human-AI collaboration not explored in this study is the value of keeping our engineered prompt open to the authors to change. Existing research reviewed above demonstrates that users unfamiliar with LLM are not able to produce well-performing prompts Zamfirescu-Pereira et al., 2023. However, most

instructors and TAs in programming courses are computer scientists with graduate-level training. We expected that some fraction of these users could benefit from the ability to change the prompt and leave this option open. This assumption, however, has to be explored. We hope that a study that engages real instructors or TAs in producing worked examples for their course might provide interesting data on end-user work with a prompt. The ultimate way to address these limitations and collect valuable information is to run a multi-semester-long study engaging instructors to use the tool to produce explanations. Such a study will also enable us to assess the quality of explanations produced through human-AI collaboration and their value for students in introductory programming classes.

Bibliography

Brusilovsky, Peter, Michael V. Yudelson, and I-Han Hsiao (2009). "Problem Solving Examples as First Class Objects in Educational Digital Libraries: Three Obstacles to Overcome". In: *Journal of Educational Multimedia and Hypermedia* 18, pp. 267–288.

Chen, Eason et al. (2023). "GPTutor: A ChatGPT-Powered Programming Tool for Code Explanation". In: *Artificial Intelligence in Education. Posters and Late Breaking Results, Workshops and Tutorials, Industry and Innovation Tracks, Practitioners, Doctoral Consortium and Blue Sky*. Ed. by Ning Wang et al. Cham: Springer Nature Switzerland, pp. 321–327. ISBN: 978-3-031-36336-8.

Chi, Michelene T. H. et al. (2018). "Translating the ICAP Theory of Cognitive Engagement Into Practice". In: *Cognitive Science* 42.6, pp. 1777–1832.

Choi, Yunseok et al. (2023). "READSUM: Retrieval-Augmented Adaptive Transformer for Source Code Summarization". In: *IEEE Access* 11, pp. 51155–51165.

Davidovic, Alex, James R. Warren, and Elena Trichina (2003). "Learning benefits of structural example-based adaptive tutoring systems". In: *IEEE Trans. Educ.* 46, pp. 241–251.

Deitel, Harvey M. and Paul J. Deitel (1994). *C How to Program, 2nd Edition*. New York: Prentice Hall.

Ericson, Barbara, Mark Guzdial, and Briana B. Morrison (2015). "Analysis of Interactive Features Designed to Enhance Learning in an Ebook". In: *Proceedings of the eleventh annual International Conference on International Computing Education Research*.

Hosseini, Roya et al. (2018). "PCEX: Interactive Program Construction Examples for Learning Programming". In: *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*. Koli Calling '18. Koli, Finland: Association for Computing Machinery. ISBN: 9781450365352.

Hosseini, Roya et al. (2020). "Improving Engagement in Program Construction Examples for Learning Python Programming". In: *International Journal of Artificial Intelligence in Education* 30.2, pp. 299–336. ISSN: 1560-4306.

Hsiao, I-Han and Peter Brusilovsky (2011). "The Role of Community Feedback in the Student Example Authoring Process: an Evaluation of AnnotEx". In: *British Journal of Educational Technology* 42.3, pp. 482–499.

Kelley, Al and Ira Pohl (1995). *C by Dissection : The Essentials of C Programming*. New York: Addison-Wesley.

Khandwala, Kandarp and Philip J. Guo (2018). "Codemotion: expanding the design space of learner interactions with computer programming tutorial videos". In: *Proceedings of the Fifth Annual ACM Conference on Learning at Scale*.

Leinonen, Juho et al. (2023). *Comparing Code Explanations Created by Students and Large Language Models*.

Li, Jierui et al. (2023). *Explaining Competitive-Level Programming Solutions using LLMs*.

Linn, Marcia C. and Michael J. Clancy (1992). "The case for case studies of programming problems". In: *Commun. ACM* 35, pp. 121–132.

MacNeil, Stephen et al. (2023). "Experiences from Using Code Explanations Generated by Large Language Models in a Web Software Development E-Book". In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. SIGCSE 2023. Toronto ON, Canada: Association for Computing Machinery, 931–937. ISBN: 9781450394314.

Morrison, Briana B. et al. (2016). "Subgoals Help Students Solve Parsons Problems". In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*.

Park, Jungkook et al. (2018). "Elicast: embedding interactive exercises in instructional programming screencasts". In: *Proceedings of the Fifth Annual ACM Conference on Learning at Scale*.

Peng, Han et al. (Dec. 2022). "Rethinking Positional Encoding in Tree Transformer for Code Representation". In: *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. Abu Dhabi, United Arab Emirates: Association for Computational Linguistics, pp. 3204–3214.

Phillips, Jesse et al. (2022). "Improved Evaluation of Automatic Source Code Summarisation". In: *Proceedings of the 2nd Workshop on Natural Language Generation, Evaluation, and Metrics (GEM)*.

Sarsa, Sami et al. (2022). "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*. ICER '22. Lugano and Virtual Event, Switzerland: Association for Computing Machinery, 27–43. ISBN: 9781450391948.

Sharrock, Rémi et al. (2017). "CODECAST: An Innovative Technology to Facilitate Teaching and Learning Computer Programming in a C Language Online Course". In: *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*.

Shi, Yang et al. (2022). "Code-DKT: A Code-based Knowledge Tracing Model for Programming Tasks". In: *ArXiv* abs/2206.03545.

Sim, Julius and Chris C Wright (Mar. 2005). "The Kappa Statistic in Reliability Studies: Use, Interpretation, and Sample Size Requirements". In: *Physical Therapy* 85.3, pp. 257–268. ISSN: 0031-9023.

Sorva, Juha, Ville Karavirta, and Lauri Malmi (2013). "A Review of Generic Program Visualization Systems for Introductory Programming Education". In: *ACM Trans. Comput. Educ.* 13, 15:1–15:64.

Tian, Haoye et al. (2023). *Is ChatGPT the Ultimate Programming Assistant – How far is it?*

White, Jules et al. (2023). *A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT*.

Williams, Joseph Jay et al. (2016). "AXIS: Generating Explanations at Scale with Learnersourcing and Machine Learning". In: *Proceedings of the Third (2016) ACM Conference on Learning @ Scale*. ACM, pp. 379–388.

Zamfirescu-Pereira, J.D. et al. (2023). "Why Johnny Can't Prompt: How Non-AI Experts Try (and Fail) to Design LLM Prompts". In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. CHI '23. Hamburg, Germany: Association for Computing Machinery. ISBN: 9781450394215.

Zhou, Denny et al. (2022). "Least-to-Most Prompting Enables Complex Reasoning in Large Language Models". In: *ArXiv*.