Pragmatic Use of Generative AI in an Introductory Programming Course (extended abstract)

Thomas Overklift
Delft University of Technology
The Netherlands
t.a.r.overkliftvaupelklein@tudelft.nl

Andy Zaidman
Delft University of Technology
The Netherlands
a.e.zaidman@tudelft.nl

Abstract

This paper introduces and discusses an early intervention of Generative AI (GenAI) in a first year university-level introductory programming course of a Computer Science programme. In our quest to convince our critical student audience that deep knowledge and understanding of programming is still relevant, and not everything can be vibe coded, we designed 3 interventions that we deployed at the start, around the middle, and towards the end of the 10-week course. From interactions with students, we observed that they were surprised at some sublte mistakes that GenAI solutions contained. For us teachers, it solidified the idea that we should keep teaching students in-depth programming knowledge and understanding.

CCS Concepts

• **Software and its engineering** → **Object oriented development**; • **Social and professional topics** → **Computer science education**.

Keywords

programming education, generative AI, LLM

ACM Reference Format:
Thomas Overklift and Andy Zaidman. 2026. Pragmatic Use of Generative AI in an Introductory Programming Course (extended abstract). In *34th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE Companion '26), July 5–9, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3803437.3805778

1 Introduction

How to convince a critical audience of young 1st year Computer Science undergraduate students to study and understand how to program in a world of vibe coding? That was an important question that we had in our mind when preparing for the 2025 edition of the "Introduction to Programming" course at Delft University of Technology. Over the past year, this question has become a frequent concern for students and parents alike during lectures and open days at the university, but also curriculum designers have pondered this question [3]. And who would blame them? With industry leaders like Satya Nadella, Sundar Pichai, and Mark Zuckerberg claiming that between 20 and 50% of the code companies like Microsoft, Google, and Meta produce in 2025 will be created by Generative AI[1,2], there is an obvious concern that computer science degrees are no longer relevant in a world of AI automation, and that jobs in the sector will become more scarce.

While an obvious career recommendation for students could be to focus on areas that AI can not (yet) handle, e.g., higher-level system design or complex problem-solving areas, our intention was to convince these young students that in-depth knowledge about programming can go hand-in-hand with the use of generative AI during programming. In fact, we wanted to convince them that in-depth knowledge of, and experience with programming is essential when pair programming with a GenAI system [22], or *vibe coding* a software system.

In this vision paper, **we explain how we designed three GenAI interventions in our introductory programming course to make it clear to 1st year students why deep knowledge and understanding of programming is still relevant in the era of generative AI**. We designed our interventions taking into account the varying backgrounds of our student population. These 3 interventions coincide with major milestones in our course: (1) the very first lecture, (2) the lecture before the mid-term exams, and (3) one of the final lectures before the end-term exams.

2 Structure of the Introduction to Programming Course

The *Introducton to Programming* course at Delft University of Technology is a 10-week course that is offered as one of the first 3 courses that our 1st year Computer Science students follow [12]. We teach our introductory programming course to approximately 500 students, and we teach our students the basic principles of programming through Java. While this course is only attended by students who major in computer science, a recurring theme is that some students already consider themselves experienced programmers, while other students have little to no programming experience. Therefore, we organize the course such that both a novice and more advanced programmer can follow it. We do so by organizing additional sessions for novice programmers, and offering additional assignments for programmers with more experience.

Table 1 gives an overview of the course activities. Each lecturing week (1—4 + 5—9), we have 3 2-hour lectures in which we combine explaining the theory and living coding. Throughout the week, students can attend practical sessions where teaching assistants can help them with their weekly non-graded non-mandatory programming assignments. Each Friday afternoon, we hold special sessions

This work is licensed under a Creative Commons Attribution 4.0 International License.
FSE Companion '26, Montreal, QC, Canada
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2636-1/26/07
https://doi.org/10.1145/3803437.3805778

[1] https://mashable.com/article/llamacon-mark-zuckerberg-ai-writes-meta-code?test_uuid=04wb5avZVbBe1OWK6996faM&test_variant=a, last visited January 21st, 2026.
[2] https://www.theverge.com/2024/10/29/24282757/google-new-code-generated-ai-q3-2024, last visited January 21st, 2026.

Week(s)	Content/activities
1—4	Lectures and assignments on primitive types, methods, classes, arrays, container types, inheritance, polymorphism, etc. Programming assignments are done in a web-based environment.
5	Mid-term exam (mostly theory).
6–9	Lectures and assignments on I/O, code quality, software design, multi-threading, functional programming, etc. Programming assignments are done in an actual IDE (IntelliJ).
8	Mock programming end-term exam to let students get to know exam environment and gauge their skill level two weeks before actual exam.
10	End-term exams (theory & programming).

Table 1: Course Overview

with additional programming exercises for students that consider themselves novice programmers.

With the mid-term exam in week 5 and the mock programming exam in week 8, we try to provide formative feedback to the students and they can earn a small bonus for a successful exam. The end-term exams in week 10 are split into a theoretical part, and a programming part. During the programming exam, the students need to solve a small assignment using an IDE; they have access to local resources, e.g., Javadoc and course slides, but do not have access to any other internet resources. For completeness sake, we mention that both end-term exam components can be resit 7 weeks after the original exam.

3 Intervention 1: Making it Clear What GenAI Can(not) Do, Even If You Cannot Program Yet

> **Question**: *How do you convince a critical audience of young students — often without programming knowledge — that they simply cannot vibe code everything, and that additionally you need to be very critical towards the results that you obtain from GenAI?*

We use the very first lecture of our course to introduce the learning goals of the course, introduce the lecturers involved, explain the general setup of the course, including the practical assignments and how they can get help if they are stuck with them, and we explain how the examination of the course works. We also introduce optional books that students can use to read a slightly different explanation of the concepts that we try to teach them. Obviously, they can also use these books to sometimes deepen themselves in subjects that we cover only partially. For the 2025 edition of the course, we for the first time also explicitly mention GenAI as a valuable source of information: if students cannot get their heads around a concept or an assignment, we stimulate them to ask GenAI to explain the concept or assignment to them. Obviously, we also advocate that the teaching assistants and lecturers remain available to them in case GenAI does not offer them a suitable answer or insight.

This is the moment where we question the use of GenAI, and where we openly ask the class whether we should still teach a full-blown introductory programming course to them. Keeping in mind the variety of backgrounds, we start by asking them whether they are familiar with the concept of *prime numbers* from mathematics. The general reaction of the students is that they are familiar with the concept.

Step 1. Ask for a definition of prime numbers. To start our discussion, we showed our students a video in which we asked ChatGPT-5[3] to generate a definition of prime numbers. As teachers, we gave students a minute or two to critically reflect upon the definition that ChatGPT provided. The general consensus from the audience was that this was a good and correct definition, an assessment that we as teachers agreed with.

Step 2. Ask GenAI to help us identify prime numbers. For Step 2, we created a short video recording in which we ask ChatGPT to identify prime numbers. We created this recording up front, because ChatGPT is (1) non-determinstic, and (2) does learn from its mistakes. We provided ChatGPT with the following question: *Is X a prime number*. We repeated this question a number of times, and ChatGPT mostly gave the right answer. However, for the ninth number that we entered, ChatGPT falsely indicated that the number was *not* prime, while we were convinced it was prime. Interestingly enough, when we re-prompted ChatGPT to ask whether it was sure that the number was non-prime, it admitted its mistake.

This raised two important points to consider: (1) the fact that GenAI can make mistakes, and (2) how can we recognize that GenAI solutions can contain mistakes.

Step 3. Relection. In relation to mistake that ChatGPT made, we explained that deep knowledge and understanding of the area (here: prime numbers) helped us to understand that the initial definition about prime numbers that ChatGPT had provided was a good one, but equally that ChatGPT made a mistake at classifying a number as non-prime, while it was prime. We concluded the reflection with the observations that (1) deep knowledge and understanding is required to understand the output of GenAI, and (2) that we should always remain critical towards the results of GenAI.

4 Intervention 2: Making it Clear that Deep Understanding (and Details) Matter

> **Question**: *How do you convince students that are learning to program that also in trivial coding tasks, we need to stay critical towards the results of ChatGPT?*

In week 4 of our 10-week course we discuss the concept of *equality* in Java [20]. In short, Java has a different notion for equality of primitive types and reference types (`==` versus `.equals()`. For reference types, e.g., classes, the Java language prescribes that the equals method should have the following method signature `public boolean equals(Object obj)` and it should adhere to the following behavioural properties[4]:

(1) Reflexivity: x.equals(x) always returns true.
(2) Symmetry: x.equals(y) returns true if and only if y.equals(x) returns true.
(3) Transitivity: for any non-null reference z, if x.equals(y) and y.equals(z) return true, then x.equals(z) should return true.

[3] We used the then newly released ChatGPT-5 in August 2025.
[4] Java 21 specification of the equals() method of Object: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/Object.html

(4) Consistency: multiple invocations of x.equals(y) consistently return true or false, provided that no information used in comparing the objects is modified.
(5) non-nullity[5]: x.equals(null) must return false.

Step 1. In the lecture subsequent to the introduction of the `equals()` method and its properties, we asked ChatGPT to generate an entire class, including its equals method. For a `class Person` with two attributes `name` and `age`, ChatGPT delivered us the following:

```
1  @Override
2  public boolean equals(Object obj) {
3      if (obj == null) return false;
4      if (this == obj) return true;
5      if (!(obj instanceof Person))
6          return false;
7
8      Person other = (Person) obj;
9      if ((age != other.age) ||
           (!this.name.equals(other.name))
10         return false;
11     else
12         return true;
13 }
```

However, on line 5 we see the `obj instanceOf Person` predicate, which returns true if `obj` is of type `Person` or any of its subtypes [21]. In case `Person` has a sub-type, the equals method implementation of Person loses the *symmetry* property [20].

Example: To illustrate the loss of symmetry, let us assume that `Person` has a sub-type `Student`. `Student` has an additional attribute `studentnumber`. Assuming that the `equals()` method implementation of `Student` is similar to Person's, we would expect a predicate like `obj instanceOf Student` (similar to line 5 of the equals implementation of Person). This would lead to the following situation:

```
1  Person a = new Person("A", 31);
2  Student b = new Student("A", 31, 121212);
3
4  assertTrue(a.equals(b)); // true
5  assertTrue(b.equals(a)); // false
```

Step 2. When we introduced the concept of equality during the lecture preceding the ChatGPT intervention, we actually made a note of how older Java code uses the `instanceOf` operator, while more modern Java code uses the `getClass()` construct. Using `getClass()` ensures symmetry, as comparisons across types (e.g., comparing a `Student` and a `Person` will always yield `false`). Indeed, during the lectures we put the following code example central:

```
1  @Override
2  public boolean equals(Object obj) {
3      if (obj == null) return false;
4      if (this == obj) return true;
5      if (this.getClass() !=
           obj.getClass())
6          return false;
7
8      Person other = (Person) obj;
9      if ((age != other.age) ||
           (!this.name.equals(other.name))
10         return false;
11     else
12         return true;
13 }
```

Again, we reflected with the students that without prior and deep knowledge of the Java language, an error could easily have slipped in here: the Java compiler will compile this and will not give any warnings. In the discussion we also highlighted the importance of (unit) tests [11] and remain critical towards results from GenAI in terms of correctness, an observation that is confirmed by Ran et al. [18]. Additionally, we discussed that the sub-optimal solution (`instanceOf` versus `getClass()`) that ChatGPT delivered is likely due to the training corpus, which might contain a number of older Java projects.

5 Intervention 3: Putting it All Together: How Well Can GenAI Solve the Exam

> **Question**: *How do you convince students that are towards the end of their introductory course on programming, that the course is still relevant, and by extension that they are still relevant in a world of GenAI?*

In Section 2 we explained that the end-term exam for our course consists of two parts: a theoretical part and a programming assignment that students need to finish within 3 hours without any outside help, but access to the Javadoc and course syllabus.

Step 1. We took one of the old programming assignment's exams. This assignment consists of roughly 4 pages of text describing the problem domain, and some simple requirements that the command-line application should be able to fulfill. The assignment was to read in data from a file, design appropriate classes that could contain the data in the file, and perform certain actions on the data (e.g., filtering). The assignment also contains hints towards class design, has requirements like writing `equals()` methods, explicitly asks to write Javadoc documentation, and asks to ensure the quality of the solution through unit tests. During class, we asked ChatGPT to solve the programming assignment. ChatGPT returned us a compiling version of the solution.

Step 2. Alongside each programming exam, we create an assessment rubric [6] that provides a detailed overview of the elements that we will grade students on. Following the example of Section 4, where we discussed equality, a number of potential rubric items could be:

(1) All classes that contain data have an `equals()` method.
(2) All `equals()` methods are correctly implemented (e.g., `getClass()` versus `instanceOf`).
(3) All classes that have an implementation for `equals()`, also have an implementation for `hashCode()` (see [4, p.50] for a detailed explanation.

[5]The term non-nullity is due to Bloch [4].

During class, we took a detailed look at the rubrics and scored the ChatGPT solution. While both the lecturer and the students agreed that the solution seemed good at first sight, a closer scrutiny of the solution with the help of the rubrics meant that we would have given the ChatGPT solution a ~6.7 out of 10 score. As a side note, critically looking at an exam solution using the lecturers' rubrics, was in itself an insightful experience given the number and type of questions that the students asked.

Step 3. While together with the students we agreed that the ChatGPT solution lost some points on "details", e.g., not adhering the coding conventions that were not explicitly stated on the exam, but where known to the students, ChatGPT's solution also lost points on actual mistakes, e.g., not generating a `hashCode` method, or making a trivial mistake in translating a formula from the assignment into code. Other major elements that the ChatGPT solution lost points on are: incomplete unit tests, sub-optimal class design (e.g., no use of inheritance) resulting in duplicated code, and lack of documentation. The consensus was that ChatGPT offers a great start to the solution, which can be further refined and should be critically scrutinized before it can be considered either complete or correct.

Additionally, triggered by the missing `hashCode()` implementation in Intervention 3 and the wrong implementation of `equals()` in Intervention 2, we also pointed to the auto-generate functionality of several methods in the IDE that we are using. More specifically, we pointed our students to the *determinstic* generation of getters, setters, equals, and hashCode methods, as compared to the non-deterministic GenAI generation.

6 Related Work

Eldh states that universities are surprisingly conservative in their curricula, and pushes the community to innovate their teaching with GenAI [8]. While Joyce et al. agree with the slow-moving and conservative nature of teaching innovation [16], they classify GenAI interventions in programming education at 3 levels (A) GenAI avoidance, (B) hands-off GenAI interaction, and (C) guided interaction with GenAI. The interventions presented in this paper put us (mostly) at level C.

Puryear and Sprint experimented with GitHub Copilot and observed that programming assignments from an introductory class obtain human-graded scores ranging from 68% to 95% [19]. These results are inline, or slightly more positive, with the result that we have seen with our 3rd intervention.

Areej et al. acknowledges that GenAI is increasingly being used in higher education computing classrooms across the USA [1]. They studied policies and practices that instructors are actually adopting with regard to GenAI. In addition, they also look at how the use of GenAI is communicated to computer science students through course syllabi. They collected 98 computing course syllabi from 54 R1 institutions in the USA. Their analysis shows that (1) most instructions related to GenAI use were as part of the academic integrity policy for the course and 2) most syllabi prohibited or restricted GenAI use, often warning students about the broader implications of using GenAI.

Zihan et al. focus on how ChatGPT could assist students during testing education [9]. Their results highlight how students acknowledged ChatGPT's effectiveness in accelerating task completion and addressing programming language challenges. However, they also note that a reliance on ChatGPT may hinder students' deeper engagement with new concepts, which is crucial for comprehensive learning, as they often interact superficially with AI-generated responses without employing the critical thinking necessary to evaluate the information provided. Similar observations stem from the study done by Mezzaro et al. [17]. They found that students wrote fewer tests and less useful tests when they had unrestricted access to large language models, and they call to raise awareness about the implications of using Generative AI in computer science education.

Bull and Kharrufa advocate for integrating GenAI into programming education [5]. While they highlight productivity gains that GenAI can offer, they warn against "*convincing but incorrect code*", i.e., code that compiles and is seemingly functionally correct, but may not have the desired or requested outcome. This is similar to what we observed through our second intervention.

De Souza et al. highlight how students use LLMs in software engineering education, and how they perceive their use as either legitimate or cheating, depending on context [7]. Their findings indicate that LLM cheating is largely shaped by workload pressure, assessment structure, and unclear or inconsistent guidance, rather than by a lack of awareness of academic integrity norms.

7 Lessons Learned & Discussion

From the interactions with students during the lecture, we feel that students did either already know, or through our interventions, grasp that GenAI can be a useful tool to "get them started", or "get them unstuck". However, from our discussions with students, they also expressed surprise at some of the mistakes that we together uncovered during the lectures that contained the 3 aforementioned interventions. Additionally, the consensus was that deep knowledge of programming is indeed required to uncover and understand sub-optimal or incorrect programming situations.

This leads to a follow-up question: **Should we replace an introductory course on programming with a vibe coding course, early on in the curriculum?** Based on our impressions, we would argue for *no*. In particular, the mistakes that we encountered during the live coding sessions with GenAI are important enough, and we need to train our students to (1) recognize the wrong or sub-optimal constructs (and understand why they are wrong or sub-optimal), and (2) make our students critical towards the — at first sight — good solutions that GenAI provides.

Based on our findings, we are now looking to introduce follow-up software engineering courses that integrate GenAI far more deeply, e.g, similar to Fein et al. [10]. In particular, we want to teach our students how to quickly prototype applications, and to then cope with the potential fall-out of either incomplete or incorrect solutions that GenAI might provide; extensive quality assurance practices can play an important role there.

From a more high-level education perspective, we also want to reflect upon Bloom's taxonomy [14], a hierarchical framework for classifying educational learning objectives. Education in our

CS curriculum follows Bloom's taxonomy, starting with courses that help students *understand* basic principles and slowly working towards synthesis as they *create* their thesis. GenAI enables students (with limited experience) to create (seemingly) good solutions with relative ease, which can easily lead students to believe that they are further along than they actually are. The introduction of GenAI may well result in the revision of models such as Bloom's taxonomy [15]. We argue that for students to truly work in a symbiotic way with GenAI they need the ability to *evaluate* generated solutions. The students in our course were quick to trust AI generated code, and we see it as our task as educators to make students aware of both the strengths and limitations of GenAI, and continuously stress the need for critical reflection [13]. The issue of trust has previously also been touched upon by Amoozadeh et al.; they found that students have different levels of trust in GenAI in relation to confidence and motivation [2].

Data availability

The video material that we used for the first intervention can be found here: https://youtu.be/1BCOMnB_Ezs.

Acknowledgments

This research was partially funded by the Dutch science foundation NWO through the Vici "TestShift" grant (No. VI.C.182.032).

References

[1] Areej Ali, Aayushi Hingle Collier, Umama Dewan, Nora McDonald, and Aditya Johri. 2025. Analysis of Generative AI Policies in Computing Course Syllabi. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education (SIGCSE)*. ACM, 18–24.

[2] Matin Amoozadeh, David Daniels, Daye Nam, Aayush Kumar, Stella Chen, Michael Hilton, Sruti Srinivasa Ragavan, and Mohammad Amin Alipour. 2024. Trust in Generative AI among Students: An exploratory study. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE)*. ACM, 67–73.

[3] Brett A. Becker, Michelle Craig, Paul Denny, Hieke Keuning, Natalie Kiesler, Juho Leinonen, Andrew Luxton-Reilly, Lauri Malmi, James Prather, and Keith Quille. 2023. Generative AI in Introductory Programming. In *Computer Science Curricula 2023*, Amruth N. Kumar, Rajendra K. Raj, Sherif G. Aly, Monica D. Anderson, Brett A. Becker, Richard L. Blumenthal, Eric Eaton, Susan L. Epstein, Michael Goldweber, Pankaj Jalote, Douglas Lea, Michael Oudshoorn, Marcelo Pias, Susan Reiser, Christian Servin, Rahul Simha, Titus Winters, and Qiao Xiang (Eds.). ACM Press, IEEE Computer Society Press and AAAI Press, 438–440.

[4] Joshua Bloch. 2018. *Effective Java: Third Edition*. Addison-Wesley.

[5] Christopher Bull and Ahmed Kharrufa. 2024. Generative Artificial Intelligence Assistants in Software Development Education: A Vision for Integrating Generative Artificial Intelligence Into Educational Practice, Not Instinctively Defending Against It. *IEEE Software* 41, 2 (2024), 52–59.

[6] Andrea Cockett and Carole Jackson. 2018. The use of assessment rubrics to enhance feedback in higher education: An integrative literature review. *Nurse Education Today* 69 (2018), 8–13.

[7] Ronnie de Souza Santos, Italo Santos, Maria Bento, Giuseppe Destefanis, Cleyton Magalhães, and Mairieli Wessel. 2026. LLM Use, Cheating, and Academic Integrity in Software Engineering Education. In *Companion Proceedings of the 34th ACM Symposium on the Foundations of Software Engineering (FSE)*.

[8] Sigrid Eldh. 2024. Generative AI Is Changing How and What We Learn. *IEEE Software* 41, 2 (2024), 4–5.

[9] Zihan Fang, Jiliang Li, Anda Liang, Gina R. Bai, and Yu Huang. 2025. A Comparative Study on ChatGPT and Checklist as Support Tools for Unit Testing Education. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion)*. ACM, 871–882.

[10] Benedikt Fein, Gordon Fraser, and Steffen Herbold. 2026. AI-Driven Software Development: A New Course Concept and Assessment Model for the Era of Large Language Models. In *International Conference on Software Engineering – Software Engineering Education Track (ICSE-SEET)*.

[11] Khalid El Haji, Carolin E. Brandt, and Andy Zaidman. 2024. Using GitHub Copilot for Test Generation in Python: An Empirical Study. In *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST)*. ACM, 45–55.

[12] Stefan Hugtenburg and Andy Zaidman. 2022. First Impressions of Using Stack Overflow for Education in a Computer Science Bachelor Programme. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 2 (SIGCSE)*. ACM, 1146.

[13] Vassilka D. Kirova, Cyril S. Ku, Joseph R. Laracy, and Thomas J. Marlowe. 2024. Software Engineering Education Must Adapt and Evolve for an LLM Environment. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE)*. ACM, 666–672.

[14] David R Krathwohl. 2002. A revision of Bloom's taxonomy: An overview. *Theory into practice* 41, 4 (2002), 212–218.

[15] Anitia Lubbe, Elma Marais, and Donnavan Kruger. 2025. Cultivating independent thinkers: The triad of artificial intelligence, Bloom's taxonomy and critical thinking in assessment pedagogy. *Education and Information Technologies* (2025), 1–34.

[16] Joyce Mahon, Brian Mac Namee, and Brett A. Becker. 2024. Guidelines for the Evolving Role of Generative AI in Introductory Programming Based on Emerging Practice. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1 (ITiCSE)*. ACM, 10––16.

[17] Simone Mezzaro, Alessio Gambi, and Gordon Fraser. 2024. An Empirical Study on How Large Language Models Impact Software Testing Learning. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (EASE)*. ACM, New York, NY, USA, 555–564.

[18] Ran Mo, Dongyu Wang, Wenjing Zhan, Yingjie Jiang, Yepeng Wang, Yuqi Zhao, Zengyang Li, and Yutao Ma. 2025. Assessing and Analyzing the Correctness of GitHub Copilot's Code Suggestions. *ACM Trans. Softw. Eng. Methodol.* 34, 7, Article 194 (2025), 32 pages.

[19] Ben Puryear and Gina Sprint. 2022. Github copilot in the classroom: learning to code with AI assistance. *J. Comput. Sci. Coll.* 38, 1 (Nov. 2022), 37–47.

[20] Chandan R. Rupakheti and Daqing Hou. 2008. An empirical study of the design and implementation of object equality in Java. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds (CASCON)*. ACM, 111–125.

[21] Chandan R. Rupakheti and Daqing Hou. 2010. An Abstraction-Oriented, Path-Based Approach for Analyzing Object Equality in Java. In *2010 17th Working Conference on Reverse Engineering*. IEEE, 205–214.

[22] Diomidis Spinellis. 2024. Pair Programming With Generative AI. *IEEE Software* 41, 3 (2024), 16–18.