SWTbahn: An Embedded Software Demonstrator in Symbiosis with Embedded Software Education

Bernhard M. Luedtke[1], Eugene Yip[1], Gerald Lüttgen[1]

Abstract: Demonstrators, such as model railways, are effective in providing real-world examples that help students to grasp abstract concepts in the engineering of embedded software. Although building such a demonstrator yourself can appear overwhelming, we argue how building your own improves your teaching of embedded software. We report on how we have achieved these improvements with a development process that lets our teaching co-evolve with our own demonstrator, share our main lessons learned, and invite educators to discuss remaining challenges.

Keywords: model railway demonstrator; evolutionary development; embedded software

1 Introduction

Students are exposed to many abstract concepts when learning about software engineering, but thinking abstractly can be a difficult competence for students to acquire. Students often struggle to see relevance in the taught concepts and paradigms, but grounding them with real-world examples can help keep students on track, motivated, and engaged [ALH07]. This is especially true for our Bachelor's module on *Reactive Systems Design (RSD)*, which covers the *model-driven engineering* of *embedded software* that has *real-time* and *safety-critical* concerns. Like other embedded software educators [Hö06; Mc07; Vö18], we have designed and built our own digital model railway, called *SWTbahn*, which serves as a practical demonstrator of embedded systems in our RSD module, projects, and theses, and offers students a realistic case study to put theory into practice. SWTbahn has supported 50% of our RSD practicals and tutorials, about 50% of our thesis topics over the last 6 years, and several semesters of 3 to 4 concurrent projects.

Having built SWTbahn, we take the position that building your own demonstrator improves teaching: (1) by being in control of the development process, you can set the foundation for an adaptable and extendable demonstrator that can keep up with developments in teaching; and (2) the development process generates real-world problems in the area of embedded software that make for great project and thesis topics for students. We elaborate on this in Sect. 2 and describe in Sect. 3 the development process we used to achieve these improvements. We share our main lessons learned in Sect. 4 and conclude this paper in Sect. 5 with remaining challenges that deserve an open discussion among software engineering educators.

[1] University of Bamberg, Software Technologies Research Group, Germany, firstname.lastname@swt-bamberg.de

doi:10.18420/seuh2024_06

2 How Building a Demonstrator Improves Teaching

While demonstrators can be obtained commercially, e. g. Lego Mindstorms[2], we decided to build our own because we could see greater benefits to teaching even though the involved risk, investment, and complexity seemed higher. In the following, we describe two key ways in which building your own demonstrator can improve your teaching.

Adapting to evolving teaching concepts. As educators, we are always striving to improve our teaching, e. g. to explain concepts more intuitively, to adjust to changing student backgrounds, or to use more industry-relevant technologies. Although a demonstrator may initially be a valuable teaching aid, its value can diminish if its capabilities cannot be adapted to support changes in teaching concepts. This is often the case with commercial demonstrators that have been developed with a particular teaching concept and set of applications in mind. In contrast, when building your own demonstrator, you can control the technology stack by developing your own hardware and software solutions, tailor its capabilities to serve your current teaching concepts, and make it adaptable to future concepts. You also gain the know-how needed to make such adaptions. Moreover, an early prototype of the demonstrator can be used in teaching to test the demonstrator's effectiveness and to also refine the teaching itself. If it is not feasible for you to build your own demonstrator, you could try to find an open source alternative that can be adapted to support a wide variety of teaching concepts.

SWTbahn supports the concept-based learning approach of our RSD practical and tutorial sessions. Each session focusses on a selection of key concepts taught in the lectures, and aims to ground the theory on relevant areas of SWTbahn. For instance, students apply their knowledge of model-driven design and synchronous automata on exercises that guide them to model a train's physical behaviour as it reacts to drive and brake commands. To evolve our RSD module more towards research-oriented teaching, we have been able to adapt SWTbahn to showcase research in the areas of modeling, verification, deployment, and integration, and their practical relevance. Moreover, we were able to incorporate industry-relevant tools into our teaching by adapting SWTbahn to interoperate with *KIELER/SCCharts*[3] and *nuXmv*[4].

Offering real-world problems to students. It can be difficult to create project and thesis topics that attract and motivate students, especially those that have outcomes with purpose beyond the work itself. We have found that, by building your own demonstrator, you regularly encounter challenges and topics that deserve further investigation. These you can refine into interesting real-world problem statements for students to analyze, design, and develop solutions for. Such problems appeal to students because they get exposed to challenges unique to embedded systems, become immersed in the engineering of capabilities for a real embedded system, and can see their work in a bigger picture. However, careful problem

[2] `https://www.lego.com/en-de/themes/mindstorms/learntoprogram` (accessed 9 Nov 2023)
[3] `https://github.com/kieler` (accessed 9 Nov 2023)
[4] `https://nuxmv.fbk.eu` (accessed 9 Nov 2023)

selection is needed as not all may be appropriate for student work, e. g. when a timely resolution is needed, or when very specific expertise is required.

As an example, the development of SWTbahn's initial software stack generated numerous topics, where some of the software is available via *GitHub*[5]. A Bachelor project student designed and developed a low-level C library for the model railway communication protocol *BiDiB*[6], who was then hired to prototype a web application for the text-based remote control of SWTbahn[7]. Other projects have extended this with graphical user interfaces and real-time visualisations. Master's and Bachelor's theses have applied model-based testing to user workflows, studied the use of requirements management tools, prototyped a log message analyser to assist in troubleshooting and maintenance work, and developed a *domain-specific language* called *BahnDSL*[8] for the automated consistency checking and generation of configuration files and routing information. SWTbahn still offers many topics to students, e. g. the development of a digital twin, automated safety layer, verifiable models, algorithms for interlocking, train scheduling, and web applications for reactive systems.

3 Why Co-Evolving Your Teaching and Demonstrator is Beneficial

For the two improvements described in Sect. 2, the first relies on developing a demonstrator with adaptability in mind, and the second relies on integrating teaching (e. g. projects and theses) into the demonstrator's development. Hence, the realization of these improvements relies on a development approach in which the teaching and the demonstrator can co-evolve. This section presents our development approach and Sect. 4 shares our lessons learned.

Our development approach can be summarized by a modified *Spiral model* [Bo86], shown in Fig. 1. The original Spiral model was conceived to guide a team through risky software development, which for us also included the risky construction of a model railway with no prior experience. Each time we cycled through the spiral, we aimed to co-evolve the teaching and demonstrator, and to produce more substantial artifacts. This approach helped us to navigate the breadth of possibilities, to engage students with early demonstrator prototypes, and to already improve our teaching before having completely built the demonstrator.

Each cycle through the spiral involves four phases: **(1) Brainstorm and develop teaching concepts** with the use of a demonstrator in mind. They can be refined by mocking up and testing teaching plans. Be creative and postpone feasibility concerns regarding the demonstrator to Phase 3. **(2) Elicit teaching-driven demonstrator requirements and constraints**, which will initially be vague, by analyzing the planned demonstrator usage in the developed teaching concepts. **(3) Design and develop the demonstrator** after carefully selecting an appropriate ecosystem of model railway hardware and software. Be

[5] `https://github.com/uniba-swt?q=swtbahn` (accessed 4 Jan 2024)
[6] `https://bidib.org/`; `https://github.com/uniba-swt/libbidib` (both accessed 4 Jan 2024)
[7] `https://github.com/uniba-swt/swtbahn-cli` (accessed 4 Jan 2024)
[8] `https://github.com/trinnguyen/bahndsl` (accessed 4 Jan 2024)

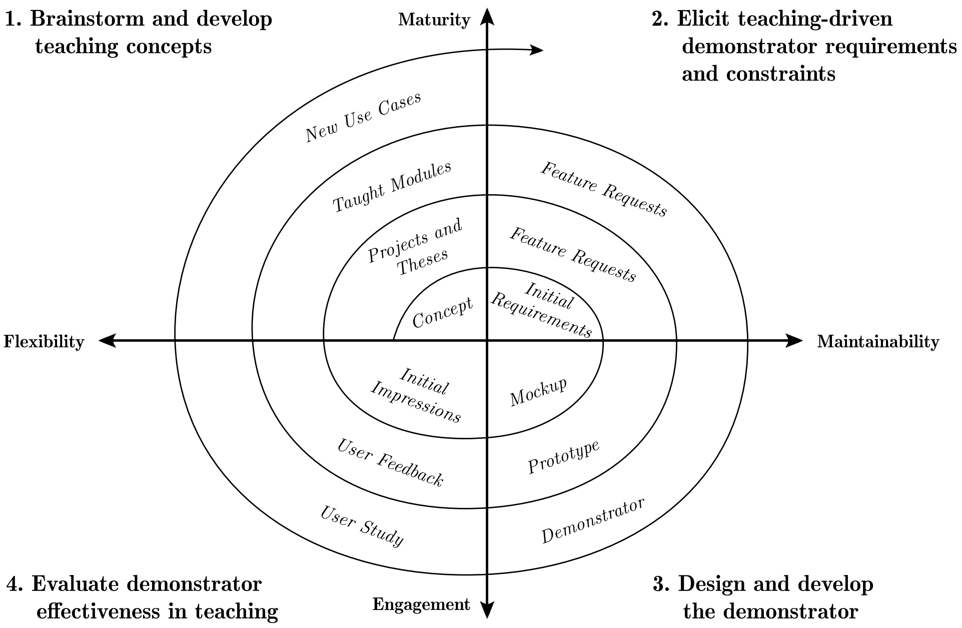

Fig. 1: Spiral model for developing a demonstrator alongside teaching. Example artifacts produced in each of the four phases appear inside the spiral. The axes are labelled with qualities to work towards.

more ambitious with each new prototype, but also aim to improve the construction quality. **(4) Evaluate demonstrator effectiveness in teaching.** Demonstrator prototypes can be phased into teaching to get early feedback on their effectiveness.

During the initial cycles, we established four desirable qualities for our teaching and demonstrator to possess: *maturity* where the demonstrator is an effective teaching aid, and teaching with the demonstrator is streamlined; *flexibility* where the demonstrator caters to a variety of teaching concepts and skill levels; *maintainability* where the demonstrator has documentation and personnel to keep it operational, and the teaching materials reflect the demonstrator's current capabilities; and *engagement* where many students, at different stages of study, work with the demonstrator. We considered our decisions in terms of their effect on the above qualities, as we wanted to strike a balance between the improvements we desired to achieve. For example, when adding capabilities to increase flexibility, we had to consider the increased maintenance effort. Similarly, when involving students in the SWTbahn software development process, we had to consider the quality of their code in relation to the reworking needed to achieve maintainability and maturity.

The Spiral model has been quite flexible in that we have been able to employ various development processes in the approximately 7 cycles since the conception of SWTbahn in 2016, e. g. *rapid prototyping* in earlier cycles, *Waterfall model* in later cycles, *bootstrapping* across the cycles, and *agile development* to manage team momentum. Using these processes, we evolved from a small (1.3×0.65 m) single-loop prototype, that could only support one short train with a few branching tracks and only simple signals, to a larger (3.4×0.9 m) multi-loop track layout, that can support 5 long trains with many large clusters of branching tracks and a variety of railway signals. Students were hired to help construct and configure SWTbahn, from which they gained first-hand experience with embedded systems. The exact processes you employ depends on your team's expertise and experience and the ambitions of your envisioned teaching and demonstrator.

4 Lessons Learned

Our endeavor to co-evolve our teaching and demonstrator using the Spiral model has not always been easy. This section outlines the main lessons we have learned along the way.

A disciplined work attitude is key to good maintenance. The Spiral model, together with staff and students of various skills and expertise, does induce a heritage of artifacts and the resulting patchwork can be difficult to maintain. To manage this, we aim to be disciplined in the recording of decisions, documenting of findings and solutions, updating of employed technologies, and managing of student contributions. To ensure that artifacts and knowledge are retrievable and organized, we work in version-controlled repositories, use the included issue management and wiki, and perform code reviews to improve consistency and quality.

Give students time to immerse themselves in demonstrator-related work. Although students in our RSD practicals have enjoyed working with SWTbahn, they often struggle to complete relatively small exercises. This, however, is because students encounter many of the real challenges that one faces when designing software for embedded systems, e. g. how to communicate with the environment, handle concurrent events, satisfy real-time deadlines, ensure safety, and guarantee correctness. To not overwhelm students, each practical is designed to offer hands-on experience on specific concepts taught in the lectures, and we provide code skeletons to get students started. Our project and thesis students face similar challenges, but at a larger scale. Instead of developing small isolated software components, they are expected to develop substantial software solutions that interface with one or more SWTbahn subsystems. We thus scope the problem statements such that students have the time to familiarize themselves with SWTbahn and its associated technologies and domains.

Expect the development team to also evolve. Being in a university environment, we experience high staff and student turnover. This often disrupts the development momentum and places maintainability at risk. To mitigate the disruption in momentum, we hold weekly meetings with the purpose of assessing our progress and priorities. This has allowed us to adapt to team changes with smoother handovers, onboarding, and redistribution of work when members become busy with other commitments. Furthermore, we always try to assign two long-term staff members to the management of all SWTbahn-related activities to ensure continuity when one of them leaves.

5 Conclusions

Building our model railway demonstrator, SWTbahn, has been a long-term commitment, but as a result we have been able to improve our teaching of embedded systems. We continue to evolve SWTbahn with our teaching concepts, and SWTbahn continues to offer real-world problem statements for students to work on. We have received positive feedback from our students, who have enjoyed using SWTbahn to apply concepts of embedded software design and development. Our modified Spiral model has greatly facilitated our co-evolutionary approach to developing our teaching and demonstrator together, and we would recommend it to others who are interested in building their own demonstrator. However, several challenges remain for which we offer the following questions to the community of software engineering educators to start an open discussion:

(1) How can practicals and projects, which involve a complex demonstrator, be designed to accommodate a large number of participants?
(2) What does the end-of-life of a complex demonstrator look like? When does it make sense to phase out a demonstrator?
(3) With artificial intelligence (AI) powered tools and the embedding of AI into safety-critical systems becoming more widespread, how can a physical demonstrator support the teaching of these topics?

Acknowledgments. We thank the reviewers and our colleagues Jan H. Boockmann and Kerstin Jacob for their suggestions that have helped us to improve the paper.

References

[ALH07] Arnold, R.; Langheinrich, M.; Hartmann, W.: InfoTraffic: Teaching Important Concepts of Computer Science and Math through Real-World Examples. In: SIGCSE. ACM, pp. 105–109, 2007.

[Bo86] Boehm, B. W.: A Spiral Model of Software Development and Enhancement. SIGSOFT Software Engineering Notes 11/4, pp. 14–24, Aug. 1986.

[Hö06] Höhrmann, S.; Fuhrmann, H.; Prochnow, S.; von Hanxleden, R.: A versatile demonstrator for distributed real-time systems: Using a model-railway in education. In: Workshop on Dependable Embedded Systems. ERCIM, 2006.

[Mc07] McCormick, J. W.: Model Railroading and Computer Fundamentals. Computer Science Education 17/2, pp. 129–139, 2007.

[Vö18] Vörös, A.; Búr, M.; Ráth, I.; Horváth, Á.; Micskei, Z.; Balogh, L.; Hegyi, B.; Horváth, B.; Mázló, Z.; Varró, D.: MoDeS3: Model-Based Demonstrator for Smart and Safe Cyber-Physical Systems. In: NASA Formal Methods. Springer, pp. 460–467, 2018.

Session 3

Was heisst "Programmieren" im Zeitalter von LLM-basierten Programmier-Assistenten?

Positionsbeitrag

Thomas R. Gross[1]

Abstract: Auf "Large Language Models" (LLMs) basierende Programmier-Assistenten, wie z.B. GitHub Copilot, AWS Code Whisperer, oder Google Bard stellen die (universitäre) Lehre der Informatik vor neue Herausforderungen. Wenn wir jetzt das "End of Programming" erreicht haben, warum sollen dann Studierende noch Programmieren lernen (und Dozierende dieses Fach unterrichten)? Während verschiedene Beiträge die Reaktionen der Informatik Dozierenden dokumentieren (und die Auswirkungen solcher Werkzeuge auf formative und summative Evaluation untersuchen), so ist auch eine Diskussion über die inhaltlichen Folgen dieser Werkzeuge nötig. In diesem Beitrag argumentiere ich, dass bei Einsatz dieser Werkzeuge (i) Programmieren weiterhin notwendig für die Ausbildung in Software Engineering (und anderen Gebieten der Informatik) ist, und (ii) Programmieren als Grundlage für die Entwicklung von Software Systemen weiterhin relevant ist und als solche unterrichtet werden sollte.

Keywords: LLM-basierte Werkzeuge; Programmieren; Software Technology

1 Einleitung

In den letzten 2 - 3 Jahren sind Programmier-Assistenten, die auf Large Language Models (LLMs) basieren, so weit fortgeschritten, dass sie in gängige Entwicklungsumgebungen (IDEs) integriert werden konnten und damit das Codieren in vielfacher Hinsicht erleichtern. Basierend auf einer grossen Anzahl früher geschriebener Programme können diese Werkzeuge in vielen Situationen Vorschläge machen, und die Benutzer/innen solcher Werkzeuge geben dann Hinweise (oder weitere Anforderungen) ein, um das erwünschte Programm zu erstellen. Beispiele solcher LLM-basierten Werkzeuge sind GitHub Copilot, AWS Code Whisperer, Google Bard oder Meta Code Llama.

Im folgenden beziehe ich mir (nur) auf GitHub Copilot, dass nach den Beobachtungen anderer Dozierender durch verschiedene Ansammlungen von Informatik Programmierproblemen trainiert wurde[Be22]. Insbesondere passen "klassische" Programmierprobleme (die in vielen Büchern zur Einführung in die Programmierung verwendet werden) gut zum Lösungsansatz solcher Werkzeuge, und verschiedene Projekte haben die Fähigkeiten dieser Werkzeuge dokumentiert [DKG23, We23b, BJP23]. Prüfungen in einem solchen Umfeld

[1] ETH Zürich, Departement Informatik, 8092 Zürich, Schweiz

doi:10.18420/seuh2024_07

sind daher eine Herausforderung für universitäre Programmierkurse[CA23, Ou23] und die Dozierenden verschiedener Institutionen haben unterschiedlichste Strategien zur Integration (oder dem Verbot) dieser Werkzeuge vorgeschlagen[LG23].

Allerdings wird durch die Programmier-Assistenz eine Lösung oft nicht sofort (nach Eingabe der Anforderungen, d.h. der zu lösenden Aufgabe) erstellt, sondern die Benutzer/innen müssen die Programmier-Assistenz anleiten, eine vollständige richtige Lösung zu erstellen[2]. Im Raum steht die Behauptung, dass diese Werkzeuge das "End of Programming" zur Folge haben [We23a] und Programmierkenntnisse für zukünftige (und jetztige) Informatiker nicht mehr relevant sind. Niemand wird Software warten müssen – wenn eine neue Herausforderung identifiziert ist (oder ein neuer Bug entdeckt wurde), dann wird das Programm neu mit einem Werkzeug generiert (und nicht durch Modifikation des bestehenden Programms realisiert).

2 Das "End of Programming"?

LLM-basierte Werkzeuge nutzen aus, dass es für ein gegebenes Problem (oder zumindest für ein sehr ähnliches Problem) bereits (Teil)Lösungen gibt, die dann zu einer Lösung für das gegebene Problem kombiniert werden können. Für viele der Probleme, die in Programmierkursen verwendet werden, sind natürlich die Lösungen bekannt, und so haben diese Werkzeuge keine grosse Mühe, eine Lösung zu finden. Aber für viele reale Projekte ist eine grosse Code Basis nicht vorhanden, so dass es viel zu früh ist, vom "End of Programming" zu sprechen[Ye23].

Auch unsere Erfahrung mit GitHub Copilot unterstützt die Einschätzung, dass der Nachruf auf Programmieren verfrüht ist. Tabelle 1 zeigt wieweit Copilot die wöchentlichen Prüfungsaufgaben eines Semesters einer "Einführung in die Programmierung" Vorlesung (die Java als Programmiersprache verwendet) für das 1. Semester lösen konnte. Die 1. Spalte gibt an, in welcher Semesterwoche ein Problem gestellt wurde. Die 2. Spalte gibt das Thema der Aufgabe an. Keine dieser Aufgaben konnte Copilot direkt *ohne weiteren Input* (d.h. ohne Hinweisungen und Anweisungen) lösen. Spalte 3 zeigt an, welchen Prozentsatz der Unit Tests das mit Hilfe von Copilot innerhalb nützlicher Zeit erstellte Programm korrekt ausführen konnte[3].

Es ist nicht überraschend, dass Copilot mit Hinweisen und Anweisungen in der Lage ist, die Aufgaben zu lösen. Aber die Hinweise und Anweisungen setzen profunde Programmierkenntnisse voraus. Insbesondere mussten die Benutzer/innen Hinweise zur Wahl

[2] Diese Einschätzung wird auch von den Entwicklern Werkzeuge geteilt; GitHub Copilot wird als "Your AI *Pair Programmer" beworben.*

[3] Die Programmieraufgaben wurden auf Deutsch gestellt. Frühere Experimente (mit den Sprachen Deutsch und Italienisch) bestätigten, dass die Wahl einer (gängigen) Sprache für die Aufgabenstellung nicht relevant ist. Die automatische Spracherkennung und anschliessende Übersetzung sind gut genug für die LLM-basierte Programmierassistenz.

Woche	Prozentsatz Unit Tests	
W4	Arrays, control flow	100
W5	Data structures, control flow	100
W6	Arrays, recursion	66.67
W7	Graph manipulation, references	100
W8	Paths, references, recursion	100
W9	Objects, simulation, inheritance	100
W10	Collection Framework , exceptions	100
W11	Collection Framework, object design	100
W12	Maps, object design, overriding	32.58

Tab. 1: Bearbeitungserfolg von Copilot für verschiedene Prüfungsaufgaben.

der Datenstrukturen (bzw. Darstellung) geben. Diese Anforderung kann natürlich auch eine Nebenwirkung der Aufgabenstellungen sein, die die Verwendung des Java Collection Frameworks betonen.

Wenn also Programmierer/innen mit diesen Werkzeugen erfolgreich arbeiten wollen, dann müssen sie weiterhin Programmierkenntnisse haben – und da Copilot das volle Spektrum der Programmiersprachen (in unserem Fall: Java) nutzt, müssen Programmier/innen mit allen Konzepten (insbesondere des Objektsystems) vertraut sein. Diese Kenntnisse sind auch erforderlich, wenn man den Output von Copilot überprüfen möchte, z.B., um festzustellen, dass das erstellte Programm nicht unerwünschterweise Informationen weitergibt (wobei es keine Rolle spielt, ob solches Verletzen der Erwartungen absichtlich oder unabsichtlich gemacht wird).

3 Was heisst "Programmieren"?

Der Einsatz von Programmier-Assistenten wie Copilot kann Programmieren interessanter machen, da die Assistenz manche wichtige aber letztlich unkritische Aufgabe übernimmt. Wenn man für eine gegebene Klasse den Konstruktor finden muss, der am zweckmässigsten die Objektexemplare initialisiert, dann macht Copilot oft einen guten Vorschlag (und ist weniger mühsam als aus den Optionen der Code Completion allein (oder gar der API) die beste Auswahl zu treffen). Copilot kann also eine grosse Hilfe für den Schritt des Codierens sein. Aber wenn wir unter "Programmieren" (auch) das Zerlegen eines Problems in Teil-Probleme und das Zusammenfügen der (Teil)-Antworten zur Lösung eines Problems verstehen [Ce16] dann hilft Copilot aber löst nicht das Problem des "Programmierens".

Copilot ist auch nur begrenzt hilfreich im kritischen Lesen der Aufgabenstellungen. Eine Programmier-Assistenz kann helfen, verschiedene Varianten (eines Programms zu erstellen), aber welche der Varianten denn die Anforderungen am besten erfüllt (und welche Aspekte in der Aufgabenstellung noch festgelegt werden müssen, um das Verhalten genügend genau zu spezifizieren) ist eine Frage, für die ein Programmgenerator nur bedingt geeignet ist. Es wäre daher wünschenswert wenn Veranstaltungen, die das Programmieren lehren, verstärkt diese Aspekte der Software Technologie betonen um Studierende auf eine produktive Arbeit mit Programmier-Assistenzen vorzubereiten.

4 Bemerkungen

LLM-basierte Programmmier-Assistenten bieten eine Möglichkeit, das Programmieren interessanter zu machen. Aber die Interaktion mit solchen Werkzeugen setzt noch immer voraus, dass die Benutzer/innen programmieren können. Programmieren ist daher weiterhin notwendig für die Ausbildung in Informatik und insbesondere in Software Engineering. Ein wichtiger Teil des "Programmierens" ist das Erstellen (und Überprüfen) von Anforderungen. Programmieren ist eine Grundlage für den Entwurf und die Entwicklung von Software Systemen und die Ausbildung sollte diesem Umstand Rechnung tragen.

Literaturverzeichnis

[Be22] Berger, E.: , Coping with Copilot. Blog, Aug 2022.

[BJP23] Barke, Shraddha; James, Michael B.; Polikarpova, Nadia: Grounded Copilot: How Programmers Interact with Code-Generating Models. Proc. ACM Program. Lang., 7(OOPSLA1):85–111, 2023.

[CA23] Cipriano, Bruno Pereira; Alves, Pedro: 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. ITiCSE 2023, Association for Computing Machinery, New York, NY, USA, S. 61–67, 2023.

[Ce16] Cerf, Vinton G.: Computer Science in the Curriculum. Commun. ACM, 59(3):7, feb 2016.

[DKG23] Denny, Paul; Kumar, Viraj; Giacaman, Nasser: Conversing with Copilot: Exploring Prompt Engineering for Solving CS1 Problems Using Natural Language. In: Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1. SIGCSE 2023, Association for Computing Machinery, New York, NY, USA, S. 1136–1142, 2023.

[LG23] Lau, Sam; Guo, Philip: From "Ban It Till We Understand It"to "Resistance is Futile": How University Programming Instructors Plan to Adapt as More Students Use AI Code Generation and Explanation Tools such as ChatGPT and GitHub Copilot. In: Proc. ICER'23. ACM, August 2023.

[Ou23] Ouh, Eng Lieh; Gan, Benjamin Kok Siew; Jin Shim, Kyong; Wlodkowski, Swavek: ChatGPT, Can You Generate Solutions for My Coding Exercises? An Evaluation on Its Effectiveness in an Undergraduate Java Programming Course. In: Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1. ITiCSE 2023, Association for Computing Machinery, New York, NY, USA, S. 54–60, 2023.

[We23a] Welsh, Matt: The End of Programming. Commun. ACM, 66(1):34–35, 2023.

[We23b] Wermelinger, Michel: Using GitHub Copilot to Solve Simple Programming Problems. In (Doyle, Maureen; Stephenson, Ben; Dorn, Brian; Soh, Leen-Kiat; Battestilli, Lina, Hrsg.): Proceedings of the 54th ACM Technical Symposium on Computer Science Education, Volume 1, SIGCSE 2023, Toronto, ON, Canada, March 15-18, 2023. ACM, S. 172–178, 2023.

[Ye23] Yellin, Daniel M.: The Premature Obituary of Programming. Commun. ACM, 66(2):41–44, 2023.