Do Away with the Frankensteinian Programs! A Proposal for a Genuine SE Education

Simon Dierl, Falk Howar, Malte Mues, Stefan Naujokat, and Till Schallau
Department of Computer Science, TU Dortmund University, 44227 Dortmund, Germany
Email: {simon.dierl, falk.howar, malte.mues, stefan.naujokat, till.schallau}@tu-dortmund.de

***Abstract*—It is widely accepted by now that the discipline of *Software Engineering* is distinct from both Computer Science and Electrical Engineering, and that it requires bespoke higher education programs. In this paper, we argue that previous attempts at designing such programs have often failed to fully account for three essential characteristics of the discipline. We propose a design philosophy for undergraduate Software Engineering programs addressing these particularities and outline a corresponding program. Incorporating this philosophy would make Generation Alpha the first generation to receive a genuine Software Engineering education.**

I. Introduction

When the term *Software Engineering* was coined during the NATO conferences of 1968 [1] and 1969 [2], the participants were in general agreement that some form of higher education for software engineers was merited. They did not, however, agree on the nature and contents of such an education program. Discussed approaches included purely abstract programs, programs including hardware design and programs focusing on teaching programming.

Today, with the ever-increasing digitization of all aspects of society during the past decades, there is an urgent and unprecedented demand for qualified software engineers: Engineers traditionally build the infrastructure of a society and the infrastructure of the 21st century is built as software. Communication, science, health care, transport, utilities, government, entertainment – today, all of these societal systems rely on a digital infrastructure. Software will enable more sophisticated automation and autonomous systems in the future. It is high time that the engineers of these systems receive a genuine engineering education and embody the values traditionally associated with an engineering discipline: economic viability, ethical behavior, positive societal impact, and plannable results.

Universities have implemented *Software Engineering* and *Computer Science* programs (which we collectively refer to as *Informatics* programs) with different scopes, philosophies, and contents during the last half-century. Since the field of Software Engineering has matured substantially during this time[1], we can positively define many aspects of Software Engineering today – without the need for analogies. E.g., instead of working towards the start of production during design, the start of delivery during production, and the end of service obligations during operation with completely different challenges and disciplines, software is developed and released continuously in many instances – leading to genuine *Software* product life-cycles, process models, and organizational structures.

This progress allows us to move away from designing Software Engineering education as a specialization in other disciplines and to recognize three *misconceptions* in current programs stemming from design by analogy: Software Engineering is neither analogous to Computer Science plus programming, nor a classic Engineering discipline for computers, and it does not require prior domain knowledge. We reckon that the time to act on this knowledge is now. By starting today with designing and implementing genuine Software Engineering programs that address these issues, Generation Alpha, i.e., the generation entering universities between 2030 and 2045, will benefit from this education.

We will present our proposal by outlining the underlying philosophy, then sketching a corresponding program, and finally projecting on further evolution over the next 20 years. Other program proposals and related research in Software Engineering education are discussed throughout the paper.

II. Program Design Philosophy

This section presents three misconceptions about the nature of Software Engineering and its communication often found in current educational program designs. Subsequently, we will present three theses as counterpoints that characterize our design philosophy for Software Engineering programs.

A. Misconceptions in Design-by-Analogy

1) Software Engineering is Computer Science Plus Programming: In this model, Software Engineering is taught using a modified version of a Computer Science program. A set of modules is replaced by more programming-focused courses and business administration modules; alternatively, these are taught in a subsequent master's program. The remaining courses are shared as-is with the Computer Science program. This pattern can be seen in the recommendations of the GI[2]

[1]Consider, e.g., Shaw's 1990 [3] ("not yet a true engineering discipline, but it has the potential to become one") and 2016 [4] ("[i]t's emerging, but still very spotty") progress estimations.

[2]The GI is Germany's counterpart to the ACM, https://gi.de.

for Informatics [5] and Technical Informatics programs [6]. The AICTE[3] model curriculum for Computer Science & Engineering [7] does not include any Software Engineering and therefore industrial trainings have to fill this gap [8], [9].

2) Software Engineering is Engineering for Computers: In this model, Software Engineering programs are derived from other Engineering programs. Substantial parts of these programs are devoted to multidisciplinary science while mathematics is taught in the form of scientific computing. This model underlies AICTE's [7] and the Russian Ministry of Science and Education's [10] model curricula and was advocated by Parnas in [11].

3) Software Engineering Requires Prior Domain Knowledge: This model argues that to learn Software Engineering, one needs to understand (and specialize in) an application domain. Students must select a technical or engineering program and pass several foundational courses. Such skills are recommended by the GI [5] and the ACM/IEEE Joint Task Force on Computing Curricula [12].

B. Design Principles Beyond Analogies

We argue that all three designs are ill-conceived and derive three essential characteristics of Software Engineering from the discussion on them.

1) Software Engineers Build Digital Infrastructure: Computer Science and Software Engineering are two different professions, which needs to be reflected in their education. Vincenti [13] describes the difference between practitioners as the purpose of their work: Engineers create practical utility while scientists want to attain understanding. Their work requires a fundamentally different perspective, even though they apply the same knowledge. When learning about, e.g., finite automata, the computer scientist may care about creating similar models and studying their properties, while the software engineer cares about the applications in system design.

As a result, a genuine Software Engineering program will share parts of its subject matter with a Computer Science program, but needs to instill a different understanding of this matter and a distinct set of skills (e.g., design-space exploration instead of theory building). Sharing similar modules with a Computer Science program necessarily hampers the understanding of one group of students. Ideally, both programs only use bespoke modules tailored to the work the students train for. We discuss this issue in detail in Section III-A2.

2) Software Engineering is Founded in Structural Science: Software Engineering is an Engineering discipline because its focus is the design of technical systems. However, due to the abstractions provided by logic circuits, its pursuit does not require a general education in natural sciences, but in Mathematics. Weizsäcker [14] described Mathematics and Informatics as *structural sciences*, i.e., the study of a well-defined, axiomatized structure, to distinguish it from *natural sciences* such as Physics that study natural phenomena. Software Engineering uses axiomatized structures to automate processes located in the physical world. Instead of the foundational knowledge of natural sciences taught to other engineers, software engineers thus require mathematical foundational skills to perform their work. This is discussed further in Section III-A1.

Therefore, a Software Engineering program must contain mathematical fundamentals (e.g., Proof Theory), discrete mathematics (e.g., Algebra and Type Theory), and continuous mathematics (e.g., Analysis and Statistics). It should neither include the natural science courses (e.g., Chemistry) nor the scientific computing courses found in other engineering programs.

3) Software Engineering is Domain-Agnostic: The essential skills and methods of a software engineer (e.g., elicitation of requirements, design, validation, cost estimation, etc.) are independent of concrete application domains, long-lived, and stable. Practical knowledge (e.g., about concrete designs, solutions, metrics, etc.) can be domain-specific and short-lived.

Modern software systems have become sufficiently complex that undergraduate programs must focus on imparting a T-shaped skill set [15]. A T-shaped software engineer has foundational knowledge of general Computer Science topics and is deeply specialized in the engineering of IT systems. As requirements elicitation becomes part of their day-to-day business, they do not need to learn the specifics of a certain domain (e.g., online shops or automotive software). Instead, they can employ their engineering skill set to adapt smoothly to the domain when tackling a new engineering task.

Therefore, a Software Engineering program should not include specific application domains. Specialization should instead focus on classes of systems (e.g., web applications or real-time systems). See Sections III-A5 and III-A8 for discussions of this issue.

III. Sample Program Outline

While the main point of our proposal is the novel design philosophy described in the previous Section, we now apply it to derive a (fictitious) Software Engineering program. We do not provide a topic-precise allocation of credits or a list of modules, but assign a weight to certain *knowledge areas* and *special activities*. Weight should correspond to the time spent on each area and the credits allocated to its modules. In the following sections, we will detail the rationale for inclusion or exclusion, contents and proposed classroom activities for each knowledge area and special activity.

We summarize our proposal in Table I and compare it to model curricula from Germany, India, Russia and the USA available online. In addition, we compare it to two Computer Science programs with Software Engineering minors from Carnegie Mellon University [16], [17] and the University of Illinois [18], [19]. While our proposed program substantially deviates from the model curricula, it is closer in weight distribution to the programs taught at these schools.

A. Knowledge Areas

1) Mathematical Foundations: As Mathematics has been used for modeling in technical subjects for centuries, it is

[3]AICTE is the Indian advisory board for technical education.

Knowledge Area	Ours	GI 1[1]	GI 2[1]	A/I E-1[2]	Weight AICTE[3]	POOP[4]	Parnas[5]	CMU[6]	UoI[7]
Mathematical Foundations	7 %	3 %	12 %	8 %	9 %	13 %	(✓)	13 %	16 %
Computer Science Foundations	7 %	17 %	12 %	8 %	11 %	5 %	(✓)	15 %	16 %
Programming Foundations	12 %	9 %	7 %	8 %	8 %	11 %	✓	4 %	5 %
Engineering Foundations	—	—	—	—	24 %	8 %	✓	10 %	9 %
Software Engineering Core	22 %	12 %	19 %	25 %	—	9 %	(✓)	20 %	18 %
Advanced Topics	17 %	23 %	26 %	8 %	9 %	13 %	✓	14 %	22 %
Economics, Ethics, and Management	13 %	12 %	—	13 %	5 %	5 %	(✓)	15 %	14 %
Technical Writing and Presentation	4 %	2 %	2 %	—	2 %	5 %	—	3 %	—
Informatics Elective	—	10 %	—	21 %	11 %	12 %	—	5 %	—
Secondary Subjects and General Education	—	—	14 %		11 %	10 %	✓	<1 %	—
Standalone Project	—	5 %	—	—	—	3 %	—	—	—
Industrial Internship	8 %	—	—	—	—	2 %	—	—	—
Senior Thesis or Project	10 %	8 %	8 %	8 %	9 %	4 %	✓	—	—

[1] GI recommendations [5], sample programs 1 and 2. [2] ACM/IEEE Joint Task Force on Computing Curricula's guidelines [12], pattern E-1 for three-year programs. [3] AICTE Model Curriculum for Undergraduate Degree Courses in Computer Science & Engineering [7]. [4] Russian Ministry of Science and Education's Sample Core Curriculum [10]. [5] Example from [11]. ✓ signifies that the area is present, and (✓) that it significantly deviates from our approach. [6] Carnegie Mellon University's B.S. in Computer Science [16] with Software Engineering Minor [17]. [7] University of Illinois' B.S. in Computer Science [18] with Software Engineering Certificate [19].

TABLE I
COMPARISON OF THREE-YEAR BACHELOR'S PROGRAM LAYOUTS FOR INFORMATICS

not surprising that software engineers require it as well. In addition, mathematical notation is omnipresent in the Informatics world. The curriculum should cover some of the mathematical foundations listed in the SWEBOK [20], focusing on logic, proof techniques, discrete probability, and algebraic structures. Our weight allocation is similar to most model curricula. We do not advocate the use of Scientific Computing courses focusing on the application of formulas often taught in other engineering disciplines, since a "deeper" background in math is an essential part of the skill set.

2) Computer Science Foundations: For Computer Science foundations, Software Engineering students should learn the basics of data structures and algorithms as well as theoretical foundations, covering formal languages, automata, and complexity theory. The focus should be on a basic understanding of the relevant theorems and proofs, and the application of the contents in Software Engineering, so dedicated courses for Software Engineering students are advisable.

As this block only forms the foundation of the software engineer's T-shape, we reduce the weight in comparison with most Computer Science-derived programs. Since this area already focuses on practical applications, the didactic format should follow. Frontal instructions should be reduced, while the students increasingly train their skills in modeling and programming projects.

3) Programming Foundations: As programming techniques, programming paradigms, and programming languages are a fundamental part of a software engineer's "toolbox", the program aims for in-depth knowledge in these areas. The use of software tools (e.g., build tools and debuggers) is equally important, so a module similar to the "The Missing Semester of your CS Education" [21] should also be included. This block's modules would require students to write programs in different (not necessarily mainstream) languages following various programming paradigms, such as imperative, declarative, and functional. This block is substantially larger than in most models. We refer to the survey of Marques et al. [22] for an overview of effective didactic approaches for this and other Software Engineering subjects but believe that a focus on practical application in projects is essential.

4) Software Engineering Core: The Software Engineering topics in the curriculum should provide in-depth knowledge about software construction and operations, spanning the complete software life-cycle, i.e., requirements elicitation and software development, production, and maintenance. As engineers have to construct systems using principled methods, this block is the "heart" of the Software Engineering program. Since software engineers must be fluent in modeling languages, these should also be taught and applied in this block.

Students will learn to understand the knowledge codified by design patterns and how to write architecture descriptions for their own software systems. A Software Architecture course should showcase different state-of-the-art architecture patterns (e.g., microservices or lambda architectures) and compare them to more traditional patterns like client-server systems. The focus should be on the trade-offs made when choosing an architecture. This helps students to develop a better understanding of important metrics for monitoring architectures during operation.

Students have to learn how to create realistic test scenarios and extrapolate a system's real performance from measured values. The focus should be on performance indicators that might help to identify early bottlenecks in operation. E.g., performing a load performance test against a server should become part of a hands-on project. Finally, security has become such a central concern for any type of software development that it should be included in this core area. Students will learn to identify and avoid vulnerabilities, but also perform analyses such as penetration testing. For these modules, hands-on experience is invaluable, so frontal instruction should be reduced to a minimum. Techniques such as problem-based

learning [23] can make the creation of solutions a core part of the learning experience and are already applied in case-studies with promising results [22], [24]–[26]. The learning outcome should be a profound understanding of the design, construction, and validation of Software, combining an engineering skill set and knowledge of existing methods and approaches.

5) Advanced Topics: While the previous area focused on Software Engineering fundamentals, this area covers more specific fields such as computer organization, database systems, human-computer interactions, networking, operating systems, and web technologies. A software engineer should possess basic knowledge in all of these areas [27], although a program could permit some degree of choice here by offering introductory and in-depth modules – potentially grouped by specializations – for each. Our weight allocation lies between the high amount recommended by the GI and the low emphasis placed by the ACM/IEEE, reflecting our emphasis on T-shaped people: they should be well-versed in their core Software Engineering skills and reasonably proficient in a Software Engineering-related topic. Didactic approaches to this field should be identical to those recommended for the core Software Engineering modules, i.e., a heavy emphasis on practical application.

6) Economics, Ethics, and Management: Software engineers have to understand business models, cost controlling, legal restrictions, and licensing concerns. In addition, they have to lead development teams during their career or work together with other software engineers, necessitating team organization and project management skills [8], [24]. With the increasing effect of software-based automation on our daily lives, ethics becomes an important aspect of the formation of a software engineer. An education in Ethics and an intense discussion about a professional code of conduct should be included in the education. Notably, many model curricula allocate far less space to this, with the GI omitting both legal and ethical aspects entirely. Classroom techniques will need to vary between more instructive formats for Economics and Law and discussion-centric formats for Ethics.

7) Technical Writing and Presentation: The program must prepare students for writing scientific theses as well as documenting and presenting their work in a professional or academic setting. In this area, students should learn the fundamentals of scientific workmanship and writing as well as presentation skills. Formats for this might include workshop-like formats and seminar papers.

8) Informatics Elective Courses and Secondary Subjects: Often, programs reserve space for elective courses or secondary subjects (e.g, Physics or Electrical Engineering) [5], [11]. We do not include secondary subjects in our program, since we do not recommend them for undergraduate programs. Elective courses should focus on the Computer Science foundations and advanced Software Engineering areas.

B. Special Activities

1) Standalone Projects: Programs frequently include a "project" module in which students are supposed to create a software artifact, often in a group (e.g., [9], [28], [29]). While we consider this to be an essential skill for software engineers, we consider the notion of a dedicated module to be misguided. Instead, project work must be a constant companion throughout the program. If possible, each module should be accompanied by a project, fitted to the learning outcomes.

2) Industrial Internships: Whenever we talk to company representatives, they ask us how they might cooperate better with academia. We are convinced that one way of cooperation between industry and academia are industrial internships. When interning at a company, students will experience real-world Software Engineering. We expect this to stimulate new ideas and areas of interest and trigger discussion among the students. This way, they get a first glimpse of industry, while industry has a chance to seed new discussions in universities. The internship model works quite well in many countries, but is virtually absent from German Informatics in Academia.

3) Senior Thesis or Project: We conclude the program with a thesis or project. While in Computer Science programs, a senior thesis concentrates on scientific work (e.g., studying a hypothesis or proving a theorem), a Software Engineering program's conclusion should be based on engineering work. This might either take the form of a senior project (i.e., the creation of a complete software-based solution), which is primarily judged on its technical merits, or a more theoretical work in the field of Software Engineering in the form of a thesis. To accommodate the time-intensive creation of a complete software project, we allocate more weight to this than existing model curricula.

IV. Outlook and Conclusion

In this paper, we focused on designing a genuine program for Software Engineering education that is both distinct from current practice and earlier propositions. Of course – being just the first step – the program will have to evolve along major changes in the field. We state three example areas that we expect to have a considerable impact in the near future.

1) Artificial Intelligence in Software Engineering: Apart from being an immensely fast-growing field of research, widespread applications of AI are becoming more and more popular. We expect AI to take a modular role in software systems (e.g., for processing sensor data), similar to data structures and algorithms today. Thus, software engineers need to be taught AI basics to effectively evaluate and integrate an AI-based solution into their system, without requiring a deep understanding of the methods.

2) Standardization and Professional Organizations: The call for professional certification of software engineers – similar to most other engineering disciplines – as, e.g., discussed by McConnel [30] in 2004 and by Kruchten [31] in 2008, has long accompanied Software Engineering. If these demands are realized, education programs will be forced to change to match. E.g., if a standardized verification technique or modeling language becomes a certification requirement, universities must include it in their curriculum to either prepare for a subsequent certification exam or to have degrees recognized as equivalent.

3) Low Code and DSLs: As the ubiquitous need for automation likely will continue to grow exponentially [32], [33], we will reach the point where automation tasks can't be performed by professional software engineers alone anymore. A way to counter this is to provide better abstractions enabling a profession of "automation craftsmen" to develop solutions based on architectures with built-in design decisions, like, e.g., the currently trending "Low Code" platforms [34]. Another way is to enable workers of all application domains to perform automation tasks during their day-to-day work, for which they will require simple and specialized configuration options. Here, domain-specific (modeling) languages [35], [36] with an automatically evaluating environment seems promising. For both ways, software engineers need to develop the required concepts, frameworks, and tools. Thus, in the future, metamodeling and language engineering skills need to be emphasized in Software Engineering education.

By implementing the proposed program today and continuously adapting it to the field's ongoing evolution, a mature higher education for software engineers will be available when Generation Alpha begins entering universities in ten years.

Authors' Profile

All authors research and teach (as professor, post-doc, and PhD students) at the chair for Software Engineering of the Computer Science department at TU Dortmund University, Germany. Our teaching activities range from basics in Software Engineering (incl. programming lab) over (domain-specific) modeling techniques to formal aspects like automata theory and software verification. The analyses, projections and positions expressed in this paper stem from extensive discussions among all authors on how to change lectures, individual teaching techniques, and even curricula, so that Informatics students can become more proficient in software development.

References

[1] P. Naur and B. Randell, Eds., *Software Engineering*, Garmisch, Germany, Jan. 1969.

[2] J. N. Buxton and B. Randell, Eds., *Software Engineering*, Rome, Italy, Apr. 1970.

[3] M. Shaw, "Prospects for an engineering discipline of software," *IEEE Software*, vol. 7, no. 6, pp. 15–24, 1990, doi: 10.1109/52.60586.

[4] M. Shaw, "Progress toward an engineering discipline of software," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, Austin, TX, USA. IEEE, 2016, pp. 3–4.

[5] O. Zukunft, *Empfehlungen für Bachelor- und Masterprogramme im Studienfach Informatik an Hochschulen*, ser. GI-Empfehlungen. GI, Jul. 2016. [Online]. Available: https://dl.gi.de/handle/20.500.12116/2351

[6] E. Maehle, *Curriculum für Bachelor- und Masterstudiengänge Technische Informatik*, ser. GI/ITG-Empfehlungen. GI, Mar. 2018. [Online]. Available: https://dl.gi.de/handle/20.500.12116/16384

[7] All India Council for Technical Education, *Model Curriculum for Undergraduate Degree Courses in Engineering & Technology*. New Delhi: All India Council for Technical Education, Jan. 2018, vol. I. [Online]. Available: https://www.aicte-india.org/education/model-syllabus

[8] K. Garg and V. Varma, "Software engineering education in india: Issues and challenges," in *2008 21st Conference on Software Engineering Education and Training*. IEEE, 2008, pp. 110–117.

[9] R. Mahanti and P. Mahanti, "Software Engineering Education From Indian Perspective," in *18th Conference on Software Engineering Education & Training (CSEET'05)*. IEEE, 2005, pp. 111–117.

[10] I. V. Rudakov, A. V. Proletarskij, and T. I. Buldakova, *Sample Core Curriculum*. Federal Educational-Methodical Association in the System of Higher Education in "Informatics and Computer Engineering", Sep. 2017, in Russian. [Online]. Available: http://xn--n1aabc.xn--p1ai/poop/1cc44c15cc8843e8abc46e225ea930fa

[11] D. L. Parnas, "Software Engineering programs are not Computer Science programs," *IEEE Software*, vol. 16, no. 6, pp. 19–30, 1999, doi: 10.1109/52.805469.

[12] The Joint Task Force on Computing Curricula, *Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*, ser. Computing Curricula. New York, NY, USA: Association for Computing Machinery, Feb. 2015, doi: 10.1145/2965631.

[13] W. G. Vincenti, *What Engineers Know and How They Know It*. Baltimore and London: John Hopkins University Press, 1990.

[14] C. F. von Weizsäcker, *Die Einheit der Natur: Studien*. München: Hanser, 1971.

[15] D. L. Johnston, "Scientists become managers-The 'T'-shaped man," *IEEE Engineering Management Review*, vol. 6, no. 3, pp. 67–68, 1978, doi: 10.1109/EMR.1978.4306682.

[16] Carnegie Mellon University. Computer Science program: Curriculum - B.S. in Computer Science. [Online]. Available: http://coursecatalog.web.cmu.edu/schools-colleges/schoolofcomputerscience/undergraduatecomputerscience/#bscurriculumtext

[17] Carnegie Mellon University. Software Engineering minor. [Online]. Available: https://www.isri.cmu.edu/education/undergrad/se-minor/index.html

[18] University of Illinois. B.S. in Computer Science. [Online]. Available: https://cs.illinois.edu/academics/undergraduate/degree-program-options/bs-computer-science

[19] University of Illinois. Software Engineering certificate. [Online]. Available: https://cs.illinois.edu/academics/undergraduate/degree-program-options/software-engineering-certificate

[20] P. Bourque and R. E. Fairley, Eds., *Guide to the Software Engineering Body of Knowledge*, version 3.0 ed. Los Alamitos, CA: IEEE Computer Society, 2014. [Online]. Available: https://www.computer.org/education/bodies-of-knowledge/software-engineering

[21] A. Athalye, J. Gjengset, and J. J. Gonzalez Ortiz. Why we are teaching this class. [Online]. Available: https://missing.csail.mit.edu/about/

[22] M. R. Marques, A. Quispe, and S. F. Ochoa, "A systematic mapping study on practical approaches to teaching Software Engineering," in *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*, 2014, pp. 1–8, doi: 10.1109/FIE.2014.7044277.

[23] H. S. Barrows, "Problem-based learning in Medicine and beyond: A brief overview," *New Directions for Teaching and Learning*, vol. 1996, no. 68, pp. 3–12, 1996, doi: 10.1002/tl.37219966804.

[24] M. Gnatz, L. Kof, F. Prilmeier, and T. Seifert, "A practical approach of teaching Software Engineering," in *Proceedings 16th Conference on Software Engineering Education and Training*, 2003, pp. 120–128, doi: 10.1109/CSEE.2003.1191369.

[25] D. Dahiya, "Teaching Software Engineering: A practical approach," *SIGSOFT Softw. Eng. Notes*, vol. 35, no. 2, p. 1–5, Mar. 2010, doi: 10.1145/1734103.1734113.

[26] N. M. Paez, "A flipped classroom experience teaching Software Engineering," in *2017 IEEE/ACM 1st International Workshop on Software Engineering Curricula for Millennials (SECM)*, 2017, pp. 16–20, doi: 10.1109/SECM.2017.6.

[27] K. Claypool and M. Claypool, "Teaching Software Engineering through game design," *ACM SIGCSE Bulletin*, vol. 37, no. 3, pp. 123–127, Sep. 2005, doi: 10.1145/1151954.1067482.

[28] A. Baker, E. O. Navarro, and A. Van Der Hoek, "An experimental card game for teaching software engineering processes," *Journal of Systems and Software*, vol. 75, no. 1–2, pp. 3–16, Feb. 2005, doi: 10.1016/j.jss.2004.02.033.

[29] P. N. Robillard, "Teaching Software Engineering through a project-oriented course," in *Software Engineering Education, Conference on*. Los Alamitos, CA, USA: IEEE Computer Society, Apr. 1996, p. 85, doi: 10.1109/CSEE.1996.10004.

[30] S. McConnel, *Professional Software Development: Shorter Schedules, Higher Quality Products, More Successful Projects, Enhanced Careers*. Addison-Wesley, 2004.

[31] P. Kruchten, "Licensing software engineers?" *IEEE Software*, vol. 25, pp. 35–37, 2008, doi: 10.1109/MS.2008.149.

[32] R. C. Martin. (2014, Jun.) My lawn. [Online]. Available: https://blog.cleancoder.com/uncle-bob/2014/06/20/MyLawn.html

[33] SlashData, "The global developer population 2019," Tech. Rep., Jul. 2019. [Online]. Available: https://sdata.me/GlobalDevPop19
[34] B. Atkins. (2020, Nov.) The most disruptive trend of 2021: No code / low code. [Online]. Available: https://www.forbes.com/sites/betsyatkins/2020/11/24/the-most-disruptive-trend-of-2021-no-code--low-code/
[35] S. Kelly and J.-P. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation*. Hoboken, NJ, USA: Wiley-IEEE Computer Society Press, 2008, doi: 10.1002/9780470249260.
[36] M. Fowler and R. Parsons, *Domain-specific languages*. Addison-Wesley / ACM Press, 2011.