What Do We Do When We Teach Software Engineering?

Joseph Maguire
School of Computing Science
University of Glasgow
Glasgow, Scotland
joseph.maguire@glasgow.ac.uk

Steve Draper
School of Psychology
University of Glasgow
Glasgow, United Kingdom
steve.draper@glasgow.ac.uk

Quintin Cutts
School of Computing Science
University of Glasgow
Glasgow, United Kingdom
quintin.cutts@glasgow.ac.uk

ABSTRACT

Many UK higher education institutions offer software engineering programmes, but the purpose and relevance of these programmes within computing science departments is not always obvious. The reality is that while advanced economies require many more skilled software engineers, universities are not delivering them. This is at least true in the context of the United Kingdom, where there are high numbers of software engineering vacancies and unemployed software engineering graduates. A possible explanation could be that curriculum content of software engineering programmes in universities needs to be reconsidered to meet the needs of industry. However, reconsidering curriculum content alone is unlikely to be transformative as there is little to be gained from changing to an emerging methodology, language or framework. Instead, an alternative direction could be to reconsider curriculum delivery and the identity of software engineering within computing science itself. In this paper, we contextualise the challenge by considering the history of software engineering education and some of its key developments. We then consider some of the alternative delivery approaches, before arguing cooperative programmes provide a opportunity for institutions to reconsider software engineering education.

CCS CONCEPTS

• **Social and professional topics** → *Computing education.*

KEYWORDS

work based learning, software engineering, computing science education

ACM Reference Format:
Joseph Maguire, Steve Draper, and Quintin Cutts. 2019. What Do We Do When We Teach Software Engineering?. In *UK Ireland Computing Education Research Conference (UKICER), September 5–6, 2019, Canterbury, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3351287.3351295

1 INTRODUCTION

Friedrich Ludwig Bauer employed the term 'software engineering' over 50 years ago as both the problem and solution to the perceived software crisis of the era [34, 51]. The specifics of the profession emerged from the 1968 NATO Conference and its use was deliberately provoking. The aim was to recognise the specific skills required to deliver software that was both significant in complexity and scale [10].

The provocation to traditional programmers and theorists was arguably effective as in the ensuing years, large-scale systems have came to represent significant assets not only for developed economies, but emerging markets as well [7]. The continued significance of software engineering is expected only to increase as the governors of advanced economies identify software systems and services as strategically important to economic growth [24]. Universities across the world are serving the perceived priority, for example 95 of the 130 publicly funded universities in the United Kingdom deliver computing science and/or software engineering degrees [53].

However, while the significance and importance of software to modern society has only increased, many of the same problems persist. In 1955, industry perceived universities as not delivering computing science graduates that satisfied their requirements [36]. In the current climate, at least in the United Kingdom, industry still perceives universities as not delivering computing science graduates that satisfy their requirements [53]. The solution to the problem is not entirely obvious. This is partly due to the fact the problem is not clear. The problem could be curriculum content, delivery or it could be both. Parnas argues that software engineering should be more engineering-focused and not delivered as a formal science [44]. However, Gibbs argues that fragmenting computing science and software engineering is detrimental to computing as a whole, as theorists should always be mindful of the applied environment [22].

In this paper, we attempt to propose a solution to the challenge of software engineering education, by considering some of its past. The contributions of this paper are:

- to contextualise the issue by reminding readers that the identity of software engineering education has been debated just as long as computing science, but there is still no sign of a resolution.
- to review some of the developments in the education of software engineers.
- to identify of some of the different delivery models used for software engineering education and suggest a way forward.

2 TRACING A HISTORY FOR SOFTWARE ENGINEERING EDUCATION

In order to chart a future for software engineering education it requires appreciation of the events that shaped its past. The following section attempts to understand the current settlement by tracing some of the developments in the history of software engineering education.

2.1 Magic Tricks and Moon Shots

A system is only as powerful as the mind that wields it. The challenge for early computer manufacturers was to demonstrate the potential of their expensive systems, a situation that required manufacturers to train programmers [18]. In the 1950s while a software industry was emerging, programmers had their own personal practices and systems had their own incantations. The immediate concern was to get the most out of manufactured systems.

However, the Apollo Guidance and Navigation programme in the 1960s serves as a classic example of the difficulty in delivering a complex software solution at scale to solve a specific challenge: spacecraft control. David Hoag, programme technical director, stated that 1,400 person hours were applied to land on the moon alone and that the "effort needed for the software turned out to be grossly underestimated" [28]. Hoag stated at the peak of activity, in 1968, 350 programmers were working on software. While this figure that may seem trivial by modern standards, it was significant for the era: recall that computer manufacturers typically trained programmers in the era. For context, in 1954, a manager from the computer manufacturer UNIVAC, stated that the annual training capacity for all computer manufacturers was approximately 260 programmers [18]. Consequently, the Apollo Guidance and Navigation programme was a hugely significant software project.

Margaret Hamilton, software lead on the programme, regularly used the term "software engineering" to emphasise that devising software was just as much an engineering challenge as others on the programme [40]. Hamilton was aware that minds had to shift to appreciate that software delivery was less a hobby and more a profession. This important shift was increasingly occurring in industry and academia. In general, industry was becoming tired of the lack of programmers with transferable skills and systems that required bespoke solutions [69]. In academia there was increasing recognition that programming had to move from a craft to a discipline [16, 68]. Consequently, there was increasing consensus for the need for a discipline for computing generally.

2.2 Forging a Discipline

In the summer of 1960, IBM organised a conference for computing centre directors [38]. There was, unsurprisingly, agreement among directors that universities should fund computing centres, much like libraries and other central resources. The directors also agreed that computing centres should deliver more advanced computing courses that were credit bearing and contributed to undergraduate programmes. The conversation of advancing the discipline continued throughout the 1960s with much of the focus on defining a suitable foundation, possibly in mathematics, systems or information processing.

There was much debate, but European institutions were generally not convinced that information processing was a suitable foundation as it was arguably an established discipline in itself [26]. There was also a general consensus that mathematics would be a suitable foundation, leading to more abstract and theoretical course content. However, in 1964, Varga argued that "the systems programming expert is neither a numerical analyst, a pure logician or at the other extreme a computer coder" [63]. Varga proposed a curriculum for computing that was more closely related to the educating of engineers, rather than mathematicians. Neverthless, mathematics remained the favoured foundation for the discipline and this was reflected in "Curriculum 68" published by the ACM [4].

The early attempts to outline curricula were formative in nature, weaponised to guide the formation of the discipline [17]. The argument could be made that a mathematical foundation was favoured as it may ensure the theoretical progression of the discipline. However, Atchison argued that the focus on mathemaitcs in Curriculum 68 is not particularly surprising given that many of the committee members were mathematicians [3].

2.3 Theory versus Reality

In the 1970s, the path for the discipline was beginning to form, but not necessarily in a direction that favoured software engineering. Coates argued that computing science departments were concerned more with theory and less with reality [14, 39]. Coates argued that graduates were not immediately productive and required additional training as they no exposure to real world systems. The situation led to some institutions introducing more industry considerate courses. Parnas reported one such course that delivered a project-oriented course in software engineering methods at Carnegie-Mellon University [43]. Parnas delivered the course twice, one cohort had industrial programming experience, while the other had completed only programming courses. Parnas argued that when devising courses with different cohorts, tailored versions should be favoured.

Further consideration of the optimal curriculum continued in the 1970s. The Model Curricula Subcommittee of the IEEE Computer Society Education Committee was formed to bridge the gap between emerging computing science curricula and computing engineering curricula [39]. While not specifically focused on software engineering, Wasserman and Freeman argued the recommendations from the Model Curricula Subcommittee could act as the strong foundation for curricula optimal for software engineering [65]. Consequently, even in the 1970s consideration was been given to the significance and importance of software engineering.

2.4 Economic importance

The significance of software engineering increased into the 1980s with many advanced economies beginning to assess the importance of software to defence and economic growth. The Software Engineering Institute (SEI) was established, a key outcome of the US Department of Defense Strategic Computing Initiative in the early 1980s. The aim of the SEI was not only to increase the production of software, but also to increase the dependability and quality of it [6, 19]. In the UK, there was concern the country was not leading in software engineering and computing initiatives. Grindley reported that the UK software industry contributed to only 2% of the world's software value [25]. In 1983, the Alvey Programme was launched, primarily to improve collaboration between academia and industry [59]. The programme considered many technological areas and challenges, but a key concern was improving software engineering within the United Kingdom [47].

The importance of software engineering and relevancy to employment was also beginning to be recognised by UK institutions. In 1984, University College London (UCL) adopted an engineering approach to their undergraduate programme, delivering graduates that were valuable to industry [67]. Winder et al. argued this was not about equipping students with a "bag of skills" but rather, developing intellectual skills that permit them to progress rapidly within their chosen career. Similarly, Garratt and Edmunds reported on the introduction of a compulsory software engineering course into the computing science curriculum at the University of Southampton [21]. The project-led course spanned 12 weeks and was focused more on group interaction and less on formal lecturers and exercises. There was growing recognition that software engineering needed to be considered within the computing science UK curriculum.

In 1989, the British Computing Science (BCS) and the then Institution of Electrical Engineers (IEE) formed a joint initiative to consider a curriculum for undergraduate software engineering [57]. Hoyle argued this was a direct response to the outcomes of the Alvey programme [29]. The focus of the initiative was to ensure software engineering was perceived as more than just programming. There was an increasing sense that a curriculum for software engineering was emerging.

2.5 Threatening Split from Computing Science

In the 1990s, Ford argued that software engineering had been evolving over the past 20 years, but that despite best efforts the necessary skills and knowledge were not present in the majority of computing science curricula [19]. Gibbs argued that computing science and software engineering were facing an impending split, much the same way that computing science split from mathematics [22]. Gibbs argued that software engineering in the early 1990s was in a state of evolution similar to computing science in the 1960s and that by the end of the century it would splinter. In 1997, Garlan et al. reported on their postgraduate software engineering programme that they had spent almost nine years refining and was jointly delivered by the Software Engineering Institute and Department of Computing Science at Carnegie Mellon University [20].

The postgraduate programme required students to have earned an undergraduate in computing science and have at least two years of relevant industrial experience. An interesting aspect was that approximately 50% of the initial cohort comprised of students employed by large corporations who paid their tuition fees. The expectation was that such leaders would return to workplace and influence existing practice and essentially represent agents of change. However, while the programme was promising the cohort was comparably small, initially only educating 20 students at some expense. In 1999, as the century closed there was still much concern that computing science departments were not embracing software engineering education. The Software Engineering Institute released a body of knowledge for software engineering as to support development of bespoke courses and programmes [5].

2.6 Prioritising Skills Development

In 2000, Shaw outlined a road map for software engineering education [55]. Shaw argued while progress had been made, focus had to move away from content and consider how the curriculum could reflect different software engineering roles. However, concern still persisted that computing science graduates were not aligned with industry expectations. Begel and Simon investigated graduate software developers experience in the workplace [8]. They reported that while students demonstrated generally good design skills, they lacked other important skills relevant to software engineering, such as communication.

The trend continued in the 2010s with more research and practitioners discussing the importance of skills. In 2013, Radermacher and Walia argued that students not only lacked strong communication and collaboration skills, but were also weak in design and development [48]. They argued that curricula changes were still necessary to address the technical and professional skill deficiencies in many graduates [48]. Similarly, Radermacher et al. argued that graduating students do not possess skills to deliver on large-scale software engineering projects [49]. They interviewed project managers and hiring personnel in the US and Europe and reported that graduates lacked project experience, had poor communication skills and had little knowledge of testing [49]. Almi et al. argued that despite the best efforts of educators and industry, there still existed a gap between academic curricula and the requirements of industry [1].

The importance of skills and the lack of focus on them is arguably reflected in the present day settlement, at least in the United Kingdom. A reality confirmed by the high unemployment of computing science graduates versus the high demand for such skills [53]. Numerous reviews of such a paradox have not identified any single clear, consistent factor [64].

3 SOFTWARE ENGINEERING AS A BRANCH OF ENGINEERING

If graduate employment is an acceptable metric then computing science departments still need to refine their efforts to ensure they are producing graduates that are aligned with the requirements of industry. The concern is that computing science departments are not equipping graduates with the necessary skills. An alternative perspective could be that industry are incorrectly expecting computing science departments to deliver software engineers, rather than computing scientists. Consequently, a solution to the current challenge could be to reconsider the foundation of software engineering education.

Hoyle argues one of the strongest outcomes of the BCS/IEE curriculum report in 1989 was the definition of software engineering: "Software Engineering is not simply a more organised approach to programming than that which was prevalent in the early days of computer science and remains widespread among amateurs or through lack of education and training" [29, 57]. Similarly, Parnas argues a software engineer is not simply a good programmer, but a professional who is responsible for the solutions they deliver [44].

However, this thinking is not clearly reflected in many software engineering programmes and courses. Tomayko argues that some computing scientists display a general lack of respect towards engineering, essentially not favouring or even believing in "messy" solutions [61]. The assumption could be that high-quality software solutions can be delivered by thinking about them hard. However, perfect solutions are not always possible, as any potential solution will represent any number of hidden assumptions.

The failure of the original Tacoma bridge demonstrates the challenge of hidden assumptions [46]. Celebrated bridge engineer Leon Moisseiff was the leading engineer on the bridge that collapsed shortly after opening to the public. The design of the bridge utilised narrow and shallow girders, unusual at the time for such suspension bridges. The narrow and shallow girders were not sufficiently rigid and were easily swayed by the winds until the eventual collapse of the bridge.

Othmar Ammann, leading bridge engineer investigating the collapse, stated: "The Tacoma Narrows bridge failure has given us invaluable information... It has shown [that] every new structure [that] projects into new fields of magnitude involves new problems for the solution of which neither theory nor practical experience furnish an adequate guide. It is then that we must rely largely on judgment and if, as a result, errors or failures occur, we must accept them as a price for human progress" [2].

Moisseiff had never thoroughly considered the winds as they had not been a concern in prior bridge projects. Consequently, it does not matter how hard Moisseiff and other thought about the solution, as Ammann suggests no prior theory or experience shed light on the problem. Petroski argues that many projects demand engineering judgement and that such judgement does not come from deeper understanding of theory or strong command of computational tools, but by learning from experience and failure [45]. In the context of software engineering, exposure to existing systems and building messy solutions may be a stronger starting point. The use of robust or defensive programming is one such technique that may be valuable for software engineers to learn to deal with hidden assumptions [9].

Therefore, adopting a more engineering perspective may be valuable in the delivery of software engineering education. Roy and Veraart argues such a curriculum does not need to be widely different from existing computing science curricula, but would include more engineering science courses and relevant skills [52]. Parnas argues software engineering does not need to be a sub-field of computing science, but needs to consider not just content, but delivery as well [44].

4 THE DIFFERENT APPROACHES TO SOFTWARE ENGINEERING EDUCATION

There have been attempts to perform mappings of software engineering education [12, 35]. However, while such attempts provide insight into methods and trends, they arguably do not consider the wider delivery strategy or approach to software engineering education within computing science programmes.

In the UK context, computing science departments use many different approaches to deliver software engineering education and skills. The "approaches" discussed here are our groupings of examples considered from literature, experience and other sources. The listed approaches and examples are not exhaustive, and others may favour alternative groupings and different examples.

4.1 The "icing" approach

The icing approach delivers software engineering within existing courses or as one area within computing science. It affords institutions the option to embed content within existing courses, such as operating systems, and/or deliver bespoke courses that focus solely on software engineering concerns, such as processes and practices.

There are many examples of the aforementioned approach discussed in §2, but there are many inventive methods to consider. Dawson suggests "*twenty dirty tricks to train software engineers*" that can introduce some of the richness of the real world back into the sheltered and sterile environment of the computing laboratory [15]. Dawson argues that software upgrades of workstations, downtime of important resources, moving deadlines and fluctuating team members are part of professional software engineering. However, the university environment strives to perform software upgrades and maintain resources outside semesters and have numerous policies to manage missing team members. Dawson reasons that such disruptions should be introduced back into courses as to provide students with sufficient experience and skills in negotiating such challenges. The approach from Dawson seems valuable, but also relatively inexpensive to implement. Nevertheless, care would be required in employing such tricks, considering the current climate within universities and concerns around student mental health. An alternative approach would be to create an artificial real-world environment.

Tvedt et al. proposes the Student Software Factory an organisation that is staffed and effectively managed by students [62]. The software factory comprises of eight semester courses that students complete sequentially. Each course represents a role within the software engineering profession, from the junior to senior. A student would complete an initial course on software tools and processes in a first semester course, before starting the role of system tester in the second semester. The conclusion in the seventh and eighth semester is for students to act as project managers, effectively coordinating and managing junior students as well as considering risk and deadlines. A potential limitation of the proposed software factory approach is that students are not able to reflect fully on some roles, such as project management. The student effectively completes the course and graduates.

Another approach is studio teaching, students typically have a dedicated space with mentors that support reflection and critique of the ongoing production of software [27]. Lee et al. reports on the use of studio teaching for three software engineering courses on a software engineering program [32]. Lee et al. states the studio comprises of a dedicated lab space with teaching staff in the space for specific contact hours. Tomayko argues teaching software engineering in a studio environment affords greater opportunity for students to engage and reflect on the production of software [60]. The concept of studio education is long since established, but definitions in the context of software engineering and computing science are often vague. Bull et al. argues that the lack of such a precise definition makes such a method difficult to evaluate and to determine its overall effectiveness [11].

Nevertheless, the icing approach has advantages in that it allows students to focus on theory will still be exposed to material relevant to industry. The concern is that students have limited to no exposure to industry itself.

4.2 The “sandwich” approach

Two fairly common complaints from employers are that: graduates are not aligned with their requirements; and/or that they lack exposure to the challenges of the workplace [14]. The sandwich approach exposes students to industry early, by expecting students to complete a summer or even a year in the workplace in the middle of their degree programme. The Shadbolt Review reports that integrated work placements are crucial in addressing the unemployment of computing science graduates [53]. Shadbolt states that students that completed sandwich degree programmes reported lower levels of general unemployment (6% vs 15% non-sandwich) and the lowest level of non-graduate unemployment (6% vs 25% non-sandwich). Consequently, Shadbolt recommends increased use of placements and internships within programmes. However, while the Shadbolt review may recommend the increased use of such sandwich programmes and some countries make them compulsory [31], such an approach focuses on exposure to industry as the solution to the problem rather than the students themselves.

The typical approach for many sandwich programmes is that students are encouraged to engage with specific resources, such as the programme lead, placement coordinator or placement office. The student is largely responsible for identifying and securing the placement with scaffolding from the institution. The approach itself requires the student to engage in their preparation and exposure to the workplace. In a compulsory approach students are thrust upon employers and potentially view the placement as just another course.

Clark and Zukas report on the experience of two different students exploring placements in software engineering [13]. They argue that students being required to engage in the process of securing a placement is an important component of the experience. Clark and Zukas suggest that not all students would benefit from a placement experience as they may not have the general qualities for the workplace, that are effectively demonstrated in securing a placement. That is demonstrating independence by research, identifying a suitable placement and determining the value to be extracted from it. Clark and Zukas also argue that compulsory placements not only remove such a process, but also have the potential to undermine the value of them.

Similarly, O’Briain et al. report that a significant advantage of a placement, other than the placement itself, is the process of identifying, engaging and securing it [42]. However, O’Briain et al. do suggest that some students struggle to adjust to academic processes and structures upon returning from the workplace. Silva et al. argues that while sandwich placements and internships can enhance graduate employment, the most effective structure or approach is still unclear [56]. Silva et al. suggests that institutions need to consider more the value in the entry and exit processes to placements. It is not clear that placement students experience a different path or curriculum upon return from their employment, they likely experience the same courses as non-sandwich students. Parnas discussed this concern in 1972 as a primary consideration on his software engineering course, while students with industrial experience did not perform any better than inexperienced students, a course approach that was optimal for both could not be found [43]. Similarly, Tvedt et al. argues one of the difficulties of sandwich placements is that they are slightly “big-bang”, in that students encounter many new things all at once, and the curriculum has not necessarily prepared them for it [62].

In many ways the sandwich approach can be compared to that of a joint academic degree programme between two different disciplines. A student essentially studies all the hardest parts of both, with no one managing their situation.

4.3 The “cooperative” approach

The cooperative approach to software engineering education is for academia and industry to partner to deliver graduate professionals. The cooperation could be relatively small scale and only involve a few courses and internships or it could involve a shared curriculum. The approach effectively expects students not only to learn on-campus at university, but also in industry at the workplace. The approach is commonly referred to as cooperative programmes in North America, Duales Studium in Germany and is effectively being introduced in the United Kingdom through Degree Apprenticeships.

The cooperative approach is a delivery model that attempts to address the isolation of theory and practice by integrating them in a single programme where students are exposed to theory at university and practice it in the workplace [50]. The approach has the benefits of both the aforementioned approaches while avoiding some of the concerns. In particular the academic and industrial partner work together to form a programme that is an optimal balance of theory and practice.

The concern is that many universities do not have sufficient experience to rapidly deliver traditional academic degrees, such as software engineering, as cooperative programmes. This is not to say that universities are ineffective at working with industry [30]. However, devising cooperative curriculum is a significant challenge as universities need to ensure quality and effective attainment of higher education objectives.

A successful work-based learning programme requires close collaboration with industrial partners [66]. The aspiration is that knowledge transfer can occur between both industry and academia. Research is transferred from university and makes an impact on industry and companies can inform universities of the knowledge and skills they expect of software engineering graduates. There is potential for cooperative education programmes to engage more research-led universities with software engineering as more researchers are able to connect with practitioners in industry. Shaw argues that “Good science depends on strong interactions between researchers and practitioners" [54]. Similarly, Meyer argues that institutions should take on the challenge of delivering a programme of teaching and research that is engaging and scientifically rigorous [37].

Nevertheless, the concern is that rather than being a mutually assured partnership, universities will be ‘reverse-colonised’ by industry [23]. The concern is that universities will seek to satisfy industry requirements when under pressure rather than ensure attainment of higher education objectives. Nevertheless, universities could adopt different solutions to avoid such challenges, a novel approach could be for academic institutions to initially act as the academic environment and workplace by training research software engineer apprenticeships [33].

Staehr et al. report on the delivery and use of cooperative learning for computing science students [58]. They argue that the primary advantage of the approach is reflection, that students are able to consider their practice away from the workplace.

Similarly, Nerland reports on the experiences of graduates participating in work-based learning in the roles as computing engineers [41]. Nerland argues that the role of computer engineer, not dissimilar to that of a software engineer, requires interaction with multiple objects that vary and are tailored to a given context. Moreover, the relevant technology and implementation is in constant flux. Nerland suggests professional development requires an individual to continually update and question knowledge. Consequently, work-based learning has the potential not only to prepare students for industry, but academia has the potential to shape students for professional development.

4.4 The "do nothing" approach

There is more than one powerful case for *doing nothing*, i.e. not delivering finished computing professionals.

The less discussed approach for computing science departments is to not engage regards software engineering education. These departments may favour simply focusing on core computing science. There is nothing to say that such graduates would not be eminently employable. In such universities, employers may simply look to other disciplines to fill roles that are perceived as being delivered by computing science graduates. The reality is that Physics could deliver programmers, Statistics could deliver data scientists and Business Schools could deliver project managers.

There is also the case that the role of computing science departments is not to produce perfect professionals, but rather strong entry candidates for subsequent professional programmes. These programmes could be delivered by higher education institutions, by companies themselves or a combination of the two. An approach that is broadly similar to the traditional USA model of professional education for medicine which traditionally only tackled applied concerns in postgraduate programmes, not in undergraduate programmes.

5 REDISCOVERING SOFTWARE ENGINEERING EDUCATION

The primary concerns of software engineering education seem to persist, despite many academics and practitioners rediscovering them over the past 70 years. The future challenges for software engineering education for academia and industry seem largely the same now as we near 2020 as they did in the 1950s.

However, a significant aspect has changed in the ensuing decades for software engineering education. In the 1950s and 1960s, students at universities would have scheduled access to a computing system and wait days for results. In 2020, students can wander into any university library and find countless desktop systems just waiting for a key press. That is to say that computing systems have progressed from being incredibly costly to inexpensive, whereas the same can not be said for the individuals that operate them.

Consequently, individuals have to be productive for companies as soon as possible. If graduate employment is an important metric, then (some) computing science departments (at least in the UK) are not delivering graduates aligned with the expectations of industry. The challenge becomes how to solve the misalignment problem and deliver graduates that meet expectations.

From the non-exhaustive list of considered approaches, see §4, computing science departments could consider progress in two different routes. The first option could be to adopt the "do nothing" approach, where computing science departments focus more on a computing science curriculum for undergraduates and tackle professional concerns at the postgraduate level. Alternatively, industry could hire graduates from other departments, such as programmers from Physics and data scientists from Statistics.

The second route could be to emphasise the responsibilities of industry and embrace the "cooperative" approach. Industry can engage more in defining the knowledge and skills they want from software engineering students. Academia can begin to devise and develop programmes that deliver such graduates. There is great potential in cooperative programmes to deliver valuable graduates, but only if industry appreciate their responsibility in delivering professional software engineers. Cooperative programmes also have the potential to engage traditional computing scientists in the concerns and requirements of industry. This could potentially not only produce valuable research partnerships but could also result in industry utilising existing research outputs, leading to greater research impact for traditional academics.

6 CONCLUSION

The history of software engineering education reveals a debate which has persisted for many decades with no clear progress. Contrasting that to Moore's law brings home how shocking that stasis is. Cooperative programmes could be a fruitful starting point for departments interested in moving forward, but are no silver bullet. A sounder basis for progress may be for us each not only to acknowledge, but to take to heart, that: (a) software engineering is not all about programming, equally (b) it is not a receptacle for all applied and professional concerns and (c) there are numerous different roles that are filled by computing graduates, each of which implies different course content.

REFERENCES

[1] Nurul Ezza Asyikin Mohamed Almi, Najwa Abdul Rahman, Durkadavi Purusothaman, and Shahida Sulaiman. 2011. Software engineering education: The gap between industry's requirements and graduates' readiness. In *Computers & Informatics (ISCI), 2011 IEEE Symposium on*. IEEE, 542–547.

[2] Othmar H Amman, Theodore von Kármán, and Glenn B Woodruff. 1941. The failure of the Tacoma Narrows bridge. *Bulletin of the Agricultural and Mechanical College of Texas* (1941).

[3] William F Atchison. 1981. Computer education, past, present, and future. *ACM SIGCSE Bulletin* 13, 4 (1981), 2–6.

[4] William F Atchison, Samuel D Conte, John W Hamblen, Thomas E Hull, Thomas A Keenan, William B Kehl, Edward J McCluskey, Silvio O Navarro, Werner C Rheinboldt, Earl J Schweppe, et al. 1968. Curriculum 68: Recommendations for academic programs in computer science: a report of the ACM curriculum committee on computer science. *Commun. ACM* 11, 3 (1968), 151–197.

[5] Donald J Bagert, Thomas B Hilburn, Greg Hislop, Michael Lutz, and Michael McCracken. 1999. *Guidelines for software engineering education version 1.0.* Technical Report. Carnegie Mellon University.

[6] Mario R Barbacci and A Nico Habermann. 1985. The Engineering Institute: Bridging Practice and Potential. *IEEE software* 2, 6 (1985), 4.

[7] Momodu Ibrahim Bayo, Nnebe Samuel Ekene, Sadiq Fatai Idowu, et al. 2007. Software development: An attainable goal for sustainable economic growth in developing nations: The Nigeria experience. *International Journal of Physical Sciences* 2, 12 (2007), 318–323.

[8] Andrew Begel and Beth Simon. 2008. Struggles of new college graduates in their first software development job. In *ACM SIGCSE Bulletin*, Vol. 40. ACM, 226–230.

[9] Matt Bishop and Deborah Frincke. 2004. Teaching robust programming. *IEEE Security & Privacy* 2, 2 (2004), 54–57.

[10] Frederick P Brooks Jr. 1975. The mythical man-month: Essays on Software Engineering. (1975).

[11] Christopher N Bull, Jon Whittle, and Leon Cruickshank. 2013. Studios in software engineering education: towards an evaluable model. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 1063–1072.

[12] Orges Cico and Letizia Jaccheri. 2019. Industry trends in software engineering education: a systematic mapping study. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*. IEEE Press, 292–293.

[13] Martyn Clark and Miriam Zukas. 2016. Understanding successful sandwich placements: A Bourdieusian approach. *Studies in Higher Education* 41, 7 (2016), 1281–1295.

[14] DC Commission on Engineering Education, Washington. 1968. *Proceedings of the Meeting on Computer Science in Electrical Engineering of the Commission on Engineering Education, Stanford University, October 24-25, 1968*. http://www.eric.ed.gov/contentdelivery/servlet/ERICServlet?accno=ED027219

[15] Ray Dawson. 2000. Twenty dirty tricks to train software engineers. In *Proceedings of the 22nd international conference on Software engineering*. ACM, 209–218.

[16] Edsger W Dijkstra. 1970. Notes on structured programming. (1970), 1–82.

[17] Sebastian Dziallas and Sally Fincher. 2015. ACM Curriculum Reports: A Pedagogic Perspective. In *Proceedings of the Eleventh Annual International Computing Education Research*. ACM, 81–89.

[18] Nathan L Ensmenger. 2012. *The computer boys take over: Computers, programmers, and the politics of technical expertise*. Mit Press.

[19] Gary Ford. 1990. *1990 SEI Report on Undergraduate Software Engineering Education*. Technical Report. Carnegie Mellon University.

[20] David Garlan, David P Gluch, and James E Tomayko. 1997. Agents of change: Educating software engineering leaders. *Computer* 30, 11 (1997), 59–65.

[21] PW Garratt and G Edmunds. 1988. Teaching software engineering at university. *Information and software technology* 30, 1 (1988), 5–11.

[22] Norman E Gibbs. 1991. Software engineering and computer science: the impending split? *Education and Computing* 7, 1-2 (1991), 111–117.

[23] Paul Gibbs. 2013. Work-based quality: a collusion waiting to happen?

[24] Energy Great Britain. Department for Business and Industrial Strategy. 2017. Industrial Strategy: building a Britain fit for the future. (2017).

[25] Peter C Grindley. 1988. *The UK software industry: a survey of the industry and evaluation of policy*. Centre for Business Strategy, London Business School.

[26] Gopal K Gupta. 2007. Computer science curriculum developments in the 1960s. *IEEE Annals of the History of Computing* 29, 2 (2007), 40–54.

[27] Orit Hazzan. 2002. The reflective practitioner perspective in software engineering education. *Journal of Systems and Software* 63, 3 (2002), 161–171.

[28] David G Hoag. 1983. The history of Apollo onboard guidance, navigation, and control. *Journal of Guidance, Control, and Dynamics* 6, 1 (1983), 4–13.

[29] BS Hoyle. 1991. The software engineering initiative: the past and the future. In *IEE Colloquium on Teaching of Software Engineering-Progress Reports*. IET, 12–1.

[30] James Kewin, Iain Nixon, Abigail Diamond, Martin Haywood, Helen Connor, and Alexandra Michael. 2011. Evaluation of the higher education transforming workforce development programme. (2011).

[31] Markus Klein and Felix Weiss. 2011. Is forcing them worth the effort? Benefits of mandatory internships for graduates from diverse family backgrounds at labour market entry. *Studies in Higher Education* 36, 8 (2011), 969–987.

[32] Jaejoon Lee, Gerald Kotonya, Jon Whittle, and Christopher Bull. 2015. Software Design Studio: A Practical Example. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 389–397. http://dl.acm.org/citation.cfm?id=2819009.2819071

[33] Joseph Maguire, Quintin Cutts, Jack Parkinson, Matthew Barr, and Derek Somerville. 2019. Devising Work-based Learning Curricula with Apprentice Research Software Engineers. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '19)*. ACM, New York, NY, USA, 313–313. https://doi.org/10.1145/3304221.3325576

[34] Michael S Mahoney. 2004. Finding a history for software engineering. *IEEE Annals of the History of Computing* 26, 1 (2004), 8–19.

[35] Maíra R Marques, Alcides Quispe, and Sergio F Ochoa. 2014. A systematic mapping study on practical approaches to teaching software engineering. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*. IEEE, 1–8.

[36] M. E. Mengel. 1995. Present and projected computer manpower needs in business and industry. In *Proceedings of the Conference on Training Personnel for the Computing Machine Field*, Vol. 1. Wayne University Press, 4–9.

[37] Bertrand Meyer. 2001. Software engineering in the academy. *Computer* 34, 5 (2001), 28–35.

[38] Philip M Morse. 1960. Report on a conference of university computing center directors (June 2-4, 1960). *Commun. ACM* 3, 10 (1960), 519–521.

[39] Michael C Mulder. 1975. Model Curicula for Four-Year Computer Science and Engineering Programs: Bridging the Tar Pit. *Computer* 8, 12 (1975), 28–33.

[40] NASA. 2016. Margaret Hamilton, Apollo Software Engineer, Awarded Presidential Medal of Freedom. https://www.nasa.gov/feature/margaret-hamilton-apollo-software-engineer-awarded-presidential-medal-of-freedom

[41] Monika Nerland. 2008. Knowledge cultures and the shaping of work-based learning: The case of computer engineering. *Vocations and learning* 1, 1 (2008), 49–69.

[42] Sian O'Briain, Susan Bergin, Martina Bourgoin, Aidan Mooney, Paula Murray, and Qingyang Zhao. 2013. Student Work Placement: Friend or Foe? A study of the perceptions of university students on industrial work placement. (2013).

[43] DL Parnas. 1972. A course on software engineering techniques. *ACM SIGCSE Bulletin* 4, 1 (1972), 154–159.

[44] David Lorge Parnas. 1999. Software engineering programs are not computer science programs. *IEEE software* 16, 6 (1999), 19–30.

[45] Henry Petroski. 1993. Failure as source of engineering judgment: Case of John Roebling. *Journal of performance of constructed facilities* 7, 1 (1993), 46–58.

[46] Henry Petroski. 2018. *Success through failure: The paradox of design*. Vol. 59. Princeton University Press.

[47] Paul Quintas and Ken Guy. 1995. Collaborative, pre-competitive R&D and the firm. *Research Policy* 24, 3 (1995), 325–348.

[48] Alex Radermacher and Gursimran Walia. 2013. Gaps between industry expectations and the abilities of graduates. In *Proceeding of the 44th ACM technical symposium on Computer science education*. ACM, 525–530.

[49] Alex Radermacher, Gursimran Walia, and Dean Knudson. 2014. Investigating the skill gap between graduating students and industry expectations. In *Companion Proceedings of the 36th international conference on software engineering*. ACM, 291–300.

[50] Joseph A Raelin. 1997. A model of work-based learning. *Organization science* 8, 6 (1997), 563–578.

[51] Brian Randell. 1996. The 1968/69 nato software engineering reports. *History of Software Engineering* (1996), 37.

[52] Geoffrey G Roy and Valerie E Veraart. 1996. Software engineering education: from an engineering perspective. In *Proceedings 1996 International Conference Software Engineering: Education and Practice*. IEEE, 256–262.

[53] Nigel Shadbolt. 2016. Shadbolt review of computer sciences degree accreditation and graduate employability: April 2016. (2016).

[54] Mary Shaw. 1990. Prospects for an engineering discipline of software. *IEEE Software* 7, 6 (1990), 15–24.

[55] Mary Shaw. 2000. Software engineering education: a roadmap. In *ICSE-Future of SE Track*. 371–380.

[56] Patrícia Silva, Betina Lopes, Marco Costa, Ana I Melo, Gonçalo Paiva Dias, Elisabeth Brito, and Dina Seabra. 2018. The million-dollar question: can internships boost employment? *Studies in Higher Education* 43, 1 (2018), 2–21.

[57] British Computer Society and Institute of Electrical Engineers. 1989. A Report of Undergraduate Curricula for Software Engineering.

[58] Lorraine Staehr, Mary Martin, and Ka Chan. 2014. A multi-pronged approach to work integrated learning for IT students. *Journal of information technology education: innovations in practice* 13 (2014), 1–11.

[59] D Talbot and RW Witty. 1983. *Alvey Programme: Software Engineering: Strategy*. Alvey Directorate.

[60] James E Tomayko. 1991. Teaching software development in a studio environment. In *ACM SIGCSE Bulletin*, Vol. 23. ACM, 300–303.

[61] James E Tomayko. 1998. Forging a discipline: An outline history of software engineering education. *Annals of Software Engineering* 6, 1-4 (1998), 3–18.

[62] John D Tvedt, Roseanne Tesoriero, and Kevin A Gary. 2001. The software factory: combining undergraduate computer science and software engineering education. In *Proceedings of the 23rd international conference on Software engineering*. IEEE Computer Society, 633–642.

[63] Richard S Varga. 1964. Computer technology at Case. In *Proceedings of the 1964 19th ACM national conference*. ACM, 121–301.

[64] William Wakeham. 2016. Wakeham Review of STEM degree provision and graduate employability. (2016).

[65] Anthony I Wasserman and Peter Freeman. 1977. Special Feature Software Engineering Concepts and Computer Science Curricula. *Computer* 10, 6 (1977), 85–91.

[66] Tim Wilson. 2012. A review of business–university collaboration. Department for Business Innovation and Skills.

[67] Russel Winder, Charles Easteal, and Robert Cole. 1987. Software engineering in a first degree. *Software Engineering Journal* 2, 4 (1987), 133–139.

[68] Niklaus Wirth. 2008. A brief history of software engineering. *IEEE Annals of the History of Computing* 30, 3 (2008), 32–39.

[69] Jo Anne Yates. 1995. Application Software for Insurance in the 1960s and Early 1970s. *Business and Economic History* (1995), 123–134.