Decades of Striving for Pedagogical and Technological Alignment

Lassi Haaranen
lassi.haaranen@aalto.fi
Aalto University
Helsinki, Finland

Lukas Ahrenberg
lukas.ahrenberg@aalto.fi
Aalto University
Helsinki, Finland

Arto Hellas
arto.hellas@aalto.fi
Aalto University
Helsinki, Finland

ABSTRACT

Computing educators have a tradition of building systems for supporting students and teachers alike. Systems and their evaluations are also an important topic in computing education research. Learning systems range from very specific such as algorithm visualizations to broad learning management systems, including supporting systems such as Q&A services. Over the years, calls for interoperability have also emerged, fueled by the desire for sharing best practices and content.

In this discussion paper, we consider the challenges with evolving technology and aligning technology with pedagogy. We see that building and using systems is a learning effort that can help the community grow, while we also note that there are no perfect solutions and that there always will be trade-offs in developing and using technology in computing education.

CCS CONCEPTS

• **Applied computing** → **Education**; *Learning management systems*; **E-learning**; **Computer-managed instruction**.

KEYWORDS

computing education, pedagogical alignment, learning technology, interoperability, learning management system, automated assessment

ACM Reference Format:
Lassi Haaranen, Lukas Ahrenberg, and Arto Hellas. 2023. Decades of Striving for Pedagogical and Technological Alignment. In *23rd Koli Calling International Conference on Computing Education Research (Koli Calling '23), November 13–18, 2023, Koli, Finland.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3631802.3631809

1 INTRODUCTION

Our behavior with systems is shaped by the design of the systems [17, 44]. Shaping of behavior can be explicit as in gamification where positive behaviors are linked with rewards [25], implicit as in providing a user interface for exploration [13] and hoping that the user learns to use it, or unintentional as in users learning to exploit a system (e.g. speedrunning games by exploiting bugs [53]). The way how the systems are connected and connect individuals can also shape our behaviors and actions – in the words of Bucher [10]:

> *"Facebook and other software systems support and shape sociality in ways that are specific to the architecture and material substrate of the medium in question."*

Although small things in user interfaces – such as Facebook's like button [20] – can influence behavior significantly, shaping of behavior is not limited to user interfaces but relates also to the underlying technology and infrastructure. As an example, the performance of an underlying server can influence the usability of a user interface that is in interaction with the server.

The same phenomena exists also in systems built for teaching and learning. Design choices and underlying technology and infrastructure have an impact on users, namely learners and instructors. By extension, they also have an impact on learning and pedagogy. While some technical contributions where learning is at the core do discuss pedagogical underpinnings and context (e.g. [15, 63, 64]), they are omitted from other articles (e.g. [24, 57]). Computing educators are known for their tendency for building systems [19].

In this work, we discuss the interplay of technology and pedagogy in computing education research (CER), including recent calls for system interoperability. Our discussion is framed through analysis of our prior experiences in both building and using a multitude of educational systems. We frame the discussion through a loose collection of case studies outlining issues in possible approaches to CS learning systems that we are aware of.

We start our discussion by presenting two perspectives of the evolution of the computing education research field in Section 2; one of them focuses on evolution of systems while the other focuses on the evolution of pedagogies for programming, followed by a discussion on the recent call for interoperability. In Section 3, we outline loose collection of case studies with the issues relating to misalignment between technology and pedagogy. In Section 4 we consider and discuss the implications of our work. We conclude in Section 5 by calling the community to action.

2 BACKGROUND

2.1 Evolution of Supporting Systems and Learning Content

The development and study of systems for supporting computing education and learning programming has a history spanning over half a century. Automated assessment tools for programming have been developed at least since the late 1950's [26], and there has been an ongoing evolution since [4, 12, 28, 45, 61]. The developments have included supporting and adopting specific programming approaches such as Test-Driven Development [15], making it easier to submit exercises for assessment and to facilitate feedback from tutors [30], making technological advances in how exercises are submitted [34], and emphasizing the need for professional tools and being supported within them while learning programming [5, 63].

Koli Calling '23, November 13–18, 2023, Koli, Finland
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1653-9/23/11.
https://doi.org/10.1145/3631802.3631809

During this evolution, there has been a call to help students understand how programs work [52] to avoid forming a faulty picture of how programs are executed [6]. One stream of research that has sought to address this is the work into algorithm and program visualization tools [22, 35, 47, 58] that are used to create and play animations of program execution, possibly line by line or even at a finer detail [56]. Due to the observation that engagement with such tools is one of the key contributors to learning [43], more recent program visualization tools have included the possibility to interact with the visualizations and use them as exercises [60].

While the above highlights examples of the evolution of supporting systems for learning programming, there has been also a broader evolution in the delivery of learning content. Fueled by the emergence of the web as a platform, computing education researchers have explored the possibility of integrating program visualizations into online learning materials [51] and considered the broader needs for interactive online materials [33], highlighting also the need for interoperability of existing systems [9]. This work has in part also included open online ebook projects such as the OpenDSA [54] and Runestone interactive [41] platforms.

As with any software and systems, there is bound to be decay and abandonment as the chosen technologies and approaches age and newer options emerge. A recent ITiCSE working group [7] charted software used by computing educators and researchers. Through literature review and surveys, they identified software to facilitate the building and maintenance of it by the community – in essence, calling for increased community adoption and maintenance and reduced duplicate effort. However, within computing education and CER, we like to create systems [19].

2.2 Evolution of Pedagogy and Pedagogical Innovation

Similar to systems used to support teaching programming, pedagogy of programming has also evolved over time [16, 38, 50]. Introductory programming education aims to help the learner to form an understanding of how computer programs are written and how they function. The interpretation of the functionality of programs might differ from the reality, as students might be presented with an idealized form of how the programs work [14, 18] with the objective of helping with the oft-mentioned problem of students struggling to write and trace programs [36, 37, 40]. The pedagogical approaches are many-fold, including using tools to provide students with something to reason about the execution of the programs [18, 46] to working with and manipulating specific types of objects such as media [21].The way how classroom instruction is organized has also been explored, where teaching approaches like peer instruction [55] have gained popularity along with media computation and pair programming [48].

The research evidence seems to point out that effort into improving programming courses yield benefits, as articles on the topic tend to have positive outcomes [62]. There is, however, a possibility of reporting bias, where researchers tend to report successes whilst hiding failures [11], and the lack of incentives for conducting replication studies [3] does not certainly improve the situation.

Many of the pedagogical innovations have been backed by technology. For example, the original LOGO project had a physical "robot" that followed instructions [46], newer notional machines tend to be interactive and hosted online [18], media computation to some extent has relied on a tailored solution for working with graphics [21], peer instruction has required clickers or some other way to facilitate answering of questions [55], including tools for writing programs in peer instruction classes [64], and so on. This explicit interplay of technology and pedagogy is rarely discussed, however, even though researchers and practitioners have called for increased adoption of educational systems [7] and increased interoperability of the systems [9].

2.3 Increasing Interoperability and Technology Adoption

There have been various efforts over the years to imbue learning tools with interoperability protocols such as SCORM [8], LTI [29], and xAPI [2] and other systems and approaches for incorporating learning materials and online learning activities from multiple sources (e.g. [31, 57]). But managing content, especially if it is interactive and technically complex, in multiple places is not trivial [24].

Clearly, a call and a desire to increase adoption of different learning technologies exists within the community. In 2014, an ITiCSE working group [9] sought to classify smart learning content (SLC) and increase its adoption. Almost a decade later their proposed *"ideal vision for embedding SLCs and storing data collected by them using new standards and protocols [...]"* has not yet arrived, which is also the case for the *"[...] pragmatic vision for achieving these goals based on existing, albeit imperfect, standards and protocols"*. There have been some advances on that front, however. As an example, the community has developed formats for sharing programming snapshot data [27, 49], and there have been attempts to increase interoperability and reuse of programming exercises [42]. While the future remains to be seen, so far the not-invented-here syndrome seems to still remain somewhat strong, and we foresee that many practitioners continue to stick to their own existing approaches – what incentives or reasons would they have to change this?

We see that one of the key problems in adopting existing interoperable content is the pedagogical alignment of objectives and technologies. We draw on an analogy of a book written so well that it is hard to put down – the likelihood of such a book being written by multiple authors who have not discussed the overall narrative with each other is rather rare. A course that draws upon multiple technologies, each written by specific teams with their own pedagogical viewpoints and contexts, would potentially end up reflecting even conflicting pedagogical objectives. Following Norman, we see that like with all design, course design *"is really an act of communication, which means having a deep understanding of the person with whom the designer is communicating"* [44].

3 A LOOSE COLLECTION OF CASE STUDIES

Combined, the three authors have over half a century of higher education teaching experience. During the years, the authors have built and developed both general purpose and tailored learning systems, used numerous learning systems – also those dubbed as interoperable – witnessed multiple university-wide learning management system (and other system) transitions, and participated in university-level committee work on future directions of learning management systems. In addition, the three authors also have years of industry experience ranging from founding their own companies and building software for and through those companies to consulting some of the largest companies in the world.

In the following, we describe a loose collection of case studies, starting from building systems to support learning and the eventual software decay, proceeding to more specific cases with issues in the use and adoption of software.

3.1 Platforms and Software Decay

The first case study draws from experiences building platforms for delivering interactive online learning materials. Over the years, the authors have been involved with developing multiple platforms and tailored services, including adopting existing services and components. The technologies used to deliver content to students have evolved over time, as have the expectations from the technologies. The authors have explored Java-based desktop applications, Java Applets, IDE integrations, HTML with JavaScript and CSS, and more modern approaches to delivering content on the web, including the use of somewhat more modern web libraries such as React[1], not to mention a range of server-side technologies used to create interactive content and allow remote experiences that have also evolved over the years. Already these evolutions have led to multiple software rewrites, refactorings, and the development of new platforms, in part due to the decay of the technolgies and software.

Slightly less than four years ago, there was again a need to improve the delivery of learning materials to students, to allow easier development of interactive functionality, and to improve the user experience. In particular, due to a new effort to provide easy-to-start learning experiences to life-long learners where the learners would not have to register but still could practice programming in an interactive online environment, a call for a system to support this emerged. Thus *Platform A* was developed. At the time, Gatsby[2] was one of the leading technologies for combining content and data and for improving the performance of websites. Gatsby allowed easy integration of interactive React components into the materials with MDX[3], where materials could be authored in Markdown format and interactivity could be easily added through React components interspersed in the markdown. Similarly, providing a consistent and contemporary user experience – and design – was a priority.

However, as is the case with modern web software, the technologies are built on top of existing libraries, some of which cease to be maintained. As the MDX support for Gatsby was (seemingly) not a top priority, perhaps in part due to underlying acquisition discussions[4], there was a period spanning over a year where the official MDX integration was not maintained at the same pace with Gatsby. This led to a situation, where using MDX required an older version of Gatsby, where both (outdated) versions had security issues. In addition, during the development of the platform, we had opted for a user interface style library that has not been continuously updated with changes to React, which has further blocked some of the refactoring efforts (and consequently also required refactorings and partial rewrites).

[1]https://react.dev/
[2]https://www.gatsbyjs.com/
[3]https://mdxjs.com/
[4]https://www.gatsbyjs.com/blog/gatsby-is-joining-netlify/

At the present state, the platform hosts a handful of courses, and there is already a need for a major refactoring (or even a rewrite), in part due to the initial decision of the technology used to combined data and content (i.e., Gatsby). This need stems from both newer versions of Gatsby lacking compatibility with older versions, relatively slow production build times of the platform, and the emergence of newer technologies such as Astro[5] that allow a more versatile approach in choosing the user interface libraries and improve the developer experience. This movement highlights the current pace of contemporary web technologies – we're slightly less than four years out from starting the effort, and there is already an effort underway to move away from the core technology powering the user interface of the platform.

> The technologies used to deliver and display content have significantly evolved over the years. As the expectations of users from learning platforms evolve with the expectations of other software, there is a need to keep up with the technology. Sticking to old is not always possible, as was evidenced with Java Applets, while keeping up with the new can also lead to issues. The evolution of technologies is also intertwined with improvements to developer experience, which has an effect on the *fun* in developing software – not keeping up with the field also eventually influences whether there are volunteering students who help in building and maintaining the platform(s).

3.2 Using an Online Platform does not Replace Local Tooling

The second case study draws from a recent experience from building an online learning platform and materials for learning web development. To facilitate a fast start on the topic, the platform provides an easy-to-use online interface for creating web applications, including writing server-side and client-side logic and database functionality within the browser. In addition, the platform provides feedback on the progress through instructor-authored tests that are executed by running a suite of unit tests and end-to-end tests on the web applications that students work on. The materials of the course and the easy-to-use online interface are hosted on a platform that intertwines theory and practice, providing a rapid possibility for practice whenever new topics are introduced.

When the course moves towards more complex topics and the applications continue to grow, students are provided a Docker Compose-based walking skeleton that includes Docker images for a web server, a database, end-to-end tests, and database migrations. The walking skeleton comes with a worked example that outlines how a similar project would be created, including demonstrating how a database running in a container is accessed and how the Dockerfiles and the Docker Compose file are created and organized. A video that outlines the use of the walking skeleton and how one would develop web applications using Visual Studio Code[6] and the walking skeleton is also provided.

[5]https://astro.build/
[6]https://code.visualstudio.com/

At this point, the way how course assignments are returned is also adjusted. Instead of working using the online interface, students are encouraged to use a zip-based submission mechanism, where they return a copy of their local project for grading.

In this case, the problem lies partially in encouragement and partially in the easy-to-use online interface. In the platform, even after course assignments become more complex, students still have the option of reverting to working with the interface instead of working on the course assignments locally and returning the assignments using the zip-based submission mechanism. Due to this, we have witnessed many cases of the metaphorical frog in the pot, where students continue working on assignments in the course platform at a point where they would significantly benefit from taking the next step that is encouraged in the materials. Ultimately, this has led to significant amounts of wasted time, both from the instructors who are supporting the students and from the students who struggle due to the increased complexity of the projects where a local environment would trump the online interface.

Course technology, opportunities, and experience of the technology shape students' behavior. Even when the technology of a course is aligned with the course pedagogy and the course provides a progression mechanism (i.e., a "scaffolding"), students may seek to maintain their existing behaviors and avoid the adoption of new technology. In essence, opportunities provided by online platforms may lead to local tooling being replaced, even when the use of local tooling would have the potential for improved learning outcomes. It seems that, at times, encouragement is not enough.

3.3 Explicit and Implicit Lock-ins in Platforms

Sooner or later the course materials need to be updated. Reasons arise from pedagogy when materials are out of date or in need of significant improvements, or from technology as specifications are updated or requirements fail. Plenty of CS specific learning technology was written with Java in the past decades which fell out of use with web-based approaches (e.g. [32, 39]).

As described in the first case study in 3.1, Web-based learning technology is also rapidly changing and evolving from a technology perspective as is the web itself. Many CS courses (e.g. [54, 59]) have interactive online materials, often taking the shape of ebooks with text, images, and examples interspersed with automatically assessed exercises. These materials can be produced with different markup languages such as HTML, reStructuredText (RST), Markdown, and MDX. No matter the technology used, there is always a lock-in in terms of opportunity cost. Changing from a format, technology, or platform to another is always costly in terms of time and effort, not to mention the mental burden that this places on teachers (and students). When it comes to commercial platforms the lock-in might be more explicit and tied to the vendor.

This case study concerns such an online course that consists of approximately 70,000 words of text, dozens of code examples and illustrations, and automatically assessed exercises using multiple technologies. The material was originally created with RST – using directives built by a teacher for another course – with automated exercises using a combination of exercises graded on the server and within the browser. Due to the limitation of a prevalent platform used by many other CS courses in the context of the case study, the decision was made to implement a new customized platform to create technology that was more aligned with the course pedagogy. An additional motivation to move away from the previous platform was the horrendous developer experience in creating the RST-based materials. As an added benefit, this migration to the new platform enabled also conducting research in new ways.

The new platform supported creation of material with MDX which provided a nicer experience in converting the earlier material and writing new components. However, the platform was written to support the same exercises of the old online course with interoperability protocols, which meant that most of the old exercises stayed as they were and required no additional significant efforts. However, maintaining a platform for just a single course is expensive in terms of effort, so there were plans to migrate the course to *Platform A* – described in the first case study. Even though the platform used very similar technologies (e.g. MDX), migrating content and especially the exercises proved to be very challenging and time-consuming, as the core underlying APIs and data formats of the platforms were different.

Beyond the observed technical challenges, *Platform A* was written with a particular pedagogical approach in mind that was *built-in* to the platform. Despite existing interoperability protocols, there were technical challenges in migrations exasperated by a difference in pedagogical approaches that were built into the two platforms.

As with the previous case study, course technology, opportunities, and experience of the technology shape *teachers'* creation of material. The platforms we choose influence the way we write materials, how they are organized, and what types of exercises we create. Succinctly, the perfect platform does not exist and even when the instructor creates the platform there are still issues in material development and pedagogical alignment. Interoperability does not and can not address pedagogical alignment.

3.4 Online Support and Chat Systems do not Replace Dialogue

Chat and collaboration software have become a common ingredient in the set of tools and teaching 'channels' in many courses. In many cases they have replaced forums and email for remote communication. Examples of such software suits and services are Discord, Microsoft Teams, Slack, Telegram, and Zulip. These tools are not completely interchangeable, and have different strengths, weaknesses and specific use-cases. However, they support collaboration in the form of chat channels around named topics, private messages, and in some cases group messages.

Chats are often employed on courses in the hope that group discussions will support learning in the form of student participation, and reduce load on other forms of communication, such as email. Administratively, it allows students to get in touch with teaching staff to ask questions and get help. This is potentially beneficial to students and teachers alike, as general issues ("when will the exam be ready?", "the course web pages are down", "there's a mistake with the handout", etc.) can be shared. Pedagogically, such tools can foster building a community where discussion can help learning, and questions lead to dialogue.

However, chat systems are not always designed for teaching, nor with pedagogy in mind. Most are intended to increase and support communication and collaboration during remote work in company settings and to allow free-form dialogue between acquaintances. Some design choices natural for these areas translate poorly to course settings. In large classes with hundreds – or even thousands – of students, channels can fill up quickly with repeated questions. That is, often the same or similar question is asked multiple times by students. It is often easier for a student to ask the question again than to search the history of already answered questions.

At the same time, it is hard to keep track of what has been answered/discussed, and by whom. Some chat systems support "pinned" posts, which can be used as a FAQ for channels, and thus alleviate the problem somewhat. But this will not completely solve the issue, as everyone does not check pinned messages before posting. Overall, this is a self-reinforcing problem – the more posts there are the harder it will be to check, and thus it is easier to just ask again.

Another part of this problem is simply that course staff are individual users on the system - perhaps with elevated moderation rights, but still separate users - and by default lack the tools necessary to sort through and coordinate, divide work, and support each other. It is therefore easy that discussions are left in the middle (as teaching assistants take shifts) or simply missed altogether, which is frustrating for the student and detrimental to learning.

Additionally, students make use of the private messages to reach out to course staff directly. While private communication is sometimes necessary, for example when discussing details of individual tasks or exercises, or private course matters, it removes the intended benefits of the system, and could in many cases be better solved by other means. In fact, one of the most common chat questions in programming courses, "what is wrong with my code?" could be the start of a dialogue in an exercise session, but hard to follow up in a chat due to its asynchronous design.

One might argue that a solution for the issues described above is a certain amount of discipline for teaching staff and students, so that repeated questions are not answered, and private messages ignored. While this is true to an extent, and we have employed such tactics, we have also observed that it decreases student engagement in the system overall. Students do not necessarily ask repeated questions because they are too lazy to search, or send private messages because they cannot wait. Instead, the usage follow from opportunities created by technical features and limitations in the system. We note that the best way to discourage such use is not by rules, but by providing students with ample opportunity to ask these questions in other forums, such as exercise sessions, which provide an opportunity for dialogue and learning.

Chat systems – and other supporting systems – can be beneficial in teaching computing. However, as they are not designed with pedagogy in mind, students' and teachers' needs need to be taken into account when using them. As with other learning tools and systems, students will find their ways of using available features, potentially causing serious issues, duplicated effort, or leaked answers.

4 MISERY IS THE ONLY CONSTANT

4.1 Eventual mismatch between pedagogy and technology

Even if the pedagogy and technology is decided by a single instructor, mismatches will eventually emerge. Approaches that students take might not be what was intended by the teachers, as seen in Section 3.2. This has also been observed in prior literature, which highlights that students can actively avoid issues that require effort, like solving merge conflicts [23].

To align pedagogy and technology more closely, we see that there are different *types* of systems used in computing education. Figure 1 arranges the tools we use in our courses along two axes: whether they are intended for CS education and to what extent the system has a specific purpose. We encourage instructors, researchers, and developers to consider where the tools they are using and developing fall into.

"CS Learning Technology" – Particular and Specific: Top right quadrant is systems intended for CS education that serve a particular purpose. Examples of these are algorithm and program visualization systems as well as automated graders.

"Supporting Systems" – Particular and Agnostic: These systems are discipline agnostic – they might be used in many other fields as well – but they are created for particular pedagogical purpose. Q&A systems would be one example of these, chat systems – discussed in Section 3.4 – are another example of supporting systems.

"Learning Management Systems" – General and Agnostic: We consider systems aimed at generally facilitating (online) learning to belong to the lower left quadrant. These are typically university-wide learning management systems (LMS) used to host course material from many disciplines and their use is shaped on a per-course basis. E.g. some teachers might use them actively for many purposes (discussions, static assign submissions, hosting lecture materials) and some might opt out of using them almost completely. It is worth noting that Blanchard et al. [7] found learning management systems being the least favorable class of tools. Potentially due to them being university-wide tools with little consideration for pedagogical approaches.

"CE(R) Specific Platforms" – General and Specific: Top left in the axis are platforms intended for computing education (research). These are platforms that offer generic functionality that one would find in a typical LMS but at the same time they offer features that are tailored specifically for CS education – which might also include features built for experimentation and CER in mind. *Platform A* belongs in this category.

When considering the case outlined in 3.3, one of the issues was that the *Platform A* had been built as a CS-specific learning

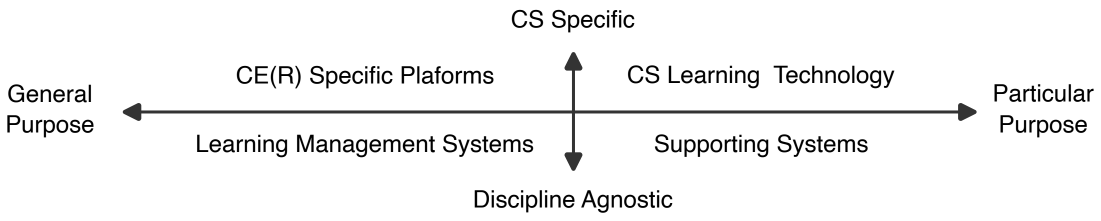

Figure 1: Different types of systems used in computing education.

platform with a particular pedagogical approach in mind. Seeking to transfer content from one system to another, even when the approach used to write the concrete materials (i.e. MDX) was the same, mismatches in the intended course pedagogy and the pedagogy that the platform had been built in mind with made the effort rather painful. Similarly, the original creation of the platform described in 3.3 (i.e., the move away from the prevalent platform in the context of the case study) emerged from the need to have a better alignment with pedagogy and technology, which the more broadly used and prevalent platform failed to support.

4.2 Eventual issues with interoperability

As Andrew Tanenbaum quipped *"The nice thing about standards is that you have so many to choose from"*. There have been various attempts at increasing interoperability, both within the CE(R) community and also in the larger space of pedagogical technology. We argue, that even in the ideal world where accessible, functional, and developer and teacher friendly interoperability protocols would exist, their adoption would not solve the fundamental issue of technical and pedagogical alignment. Ultimately, instructors would still end up implementing their own exercises, and researchers in computing would still want to "scratch their own itch".

All this being said, we see interoperability efforts and sharing of exercises, materials, and tools as highly valuable for both teachers and researchers. Duplicating the effort of creating similar technology or exercises types does not make sense, and adopting existing technologies should be prioritized. At the same time, we encourage efforts to keep learning through implementing new tools for CS education – even if they are not interoperable. That is, coming up with technological solutions to solve problems is in the veins of CS educators and researchers, and solving problems by itself is a learning effort. Would we adopt solutions from others, how familiar would we be with the design decisions – both pedagogical and technical – that underlie the adopted solutions, and could we even end up in a situation where we unintentionally create problems due to pedagogical misalignment of the solutions that we have adopted?

4.3 Eventual software decay and the need to reinvent the wheel

As discussed in 3.1, technologies have evolved over time, and there are times when it is effectively mandatory to switch technologies (e.g. the abandonment of Java Applets and moving from desktop to the web). Creating production-ready software in university settings is challenging at the best of circumstances, and often the setting is less than ideal for software development. At the same time, creating software for university settings is software development, and it should be treated as such [24] – a part of software development is maintenance, and due to the evolution of technology, there is an eventual need for larger refactorings and rewrites.

In addition, evolving technology means that our expectations of technology, as users and developers, evolve as well. While the students' learning is the main goal, learning is shaped by technology and materials, those who create them, and their experiences while creating them. Learning is easier when one does not have to learn to explicitly use a new system; at times, catching up is needed, as "users prefer your [platform] to work the same way as all the other [platforms] they already know" [1][7]. Not keeping up, on the contrary, can lead to unnecessary effort, and especially in the case of lifelong learning where decisions on what content to use can be made rather rapidly, can also lead to potential students dropping out.

Contrary to the position outlined in [7], we see that there is a need to reinvent the wheel every now and then.

5 CONCLUSION

Ultimately, there are no solutions only trade-offs. Technology changes, standards are replaced, and pedagogical theory improves. In our experience, the only constant in development, adoption, and use of educational technology is eventual misery that equates with a need for change. Sisyphus, in Greek mythology, was cursed to forever push the same boulder up a hill – as the rock was about to crest, it rolled back instead, and the toil started anew. This is also the fate of the computing educator striving to eternally align technology and pedagogy. However, the inevitable misery is not a reason to give up; we should keep on building systems and innovating pedagogy and technology alike. Interoperability is a great asset but it will not solve all our technology woes, and even platforms with similar technologies may have been built with a very different pedagogy in mind.

REFERENCES

[1] Jakob Nielsen. 2000. End of Web Design. https://www.nngroup.com/articles/end-of-web-design/.
[2] Rustici Software. 2023. What is the Experience API? https://xapi.com/overview/.
[3] Alireza Ahadi, Arto Hellas, Petri Ihantola, Ari Korhonen, and Andrew Petersen. 2016. Replication in computing education research: researcher attitudes and experiences. In *Proceedings of the 16th Koli calling international conference on computing education research*. 2–11.
[4] Kirsti M Ala-Mutka. 2005. A survey of automated assessment approaches for programming assignments. *Computer science education* 15, 2 (2005).
[5] Anthony Allowatt and Stephen H Edwards. 2005. IDE support for test-driven development and automated grading in both Java and C++. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*. 100–104.

[7]Author note, replaced "site" with "platform".

[6] Mordechai Ben-Ari. 2001. Constructivism in computer science education. *Journal of computers in Mathematics and Science Teaching* 20, 1 (2001).

[7] Jeremiah Blanchard, John R Hott, Vincent Berry, Rebecca Carroll, Bob Edmison, Richard Glassey, Oscar Karnalim, Brian Plancher, and Seán Russell. 2022. Stop Reinventing the Wheel! Promoting Community Software in Computing Education. In *Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education.* 261–292.

[8] Oliver Bohl, Jörg Scheuhase, Ruth Sengler, and Udo Winand. 2002. The sharable content object reference model (SCORM)-a critical review. In *International Conference on Computers in Education, 2002. Proceedings.* IEEE, 950–951.

[9] Peter Brusilovsky, Stephen Edwards, Amruth Kumar, Lauri Malmi, Luciana Benotti, Duane Buck, Petri Ihantola, Rikki Prince, Teemu Sirkiä, Sergey Sosnovsky, et al. 2014. Increasing adoption of smart learning content for computer science education. In *Proceedings of the Working Group Reports of the 2014 on Innovation & Technology in Computer Science Education Conference.* 31–57.

[10] Taina Bucher. 2018. *If... then: Algorithmic power and politics.* Oxford University Press.

[11] Phillip Dawson and Samantha L Dawson. 2018. Sharing successes and hiding failures:'reporting bias' in learning and teaching research. *Studies in Higher Education* 43, 8 (2018), 1405–1416.

[12] Christopher Douce, David Livingstone, and James Orwell. 2005. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing (JERIC)* 5, 3 (2005), 4–es.

[13] Stephen W Draper and Stephen B Barton. 1993. Learning by exploration and affordance bugs. In *INTERACT'93 and CHI'93 Conference Companion on Human Factors in Computing Systems.* 75–76.

[14] Benedict Du Boulay, Tim O'Shea, and John Monk. 1981. The black box inside the glass box: presenting computing concepts to novices. *International Journal of man-machine studies* 14, 3 (1981), 237–249.

[15] Stephen H Edwards. 2003. Using test-driven development in the classroom: Providing students with automatic, concrete feedback on performance. In *Proceedings of the international conference on education and information systems: technologies and applications EISTA*, Vol. 3. Citeseer.

[16] Gerald E Evans and Mark G Simkin. 1989. What best predicts computer proficiency? *Commun. ACM* 32, 11 (1989), 1322–1327.

[17] Nir Eyal. 2014. *Hooked: How to build habit-forming products.* Penguin.

[18] Sally Fincher, Johan Jeuring, Craig S Miller, Peter Donaldson, Benedict Du Boulay, Matthias Hauswirth, Arto Hellas, Felienne Hermans, Colleen Lewis, Andreas Mühling, et al. 2020. Capturing and characterising notional machines. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education.* 502–503.

[19] Sally Fincher and Marian Petre. 2004. *Computer science education research.* CRC Press.

[20] Jan Fox. 2018. An unlikeable truth: Social media like buttons are designed to be addictive. They're impacting our ability to think rationally. *Index on Censorship* 47, 3 (2018), 11–13.

[21] Mark Guzdial. 2003. A media computation course for non-majors. In *Proceedings of the 8th annual conference on Innovation and technology in computer science education.* 104–108.

[22] Jyrki Haajanen, Mikael Pesonius, Erkki Sutinen, Jorma Tarhio, Tommi Terasvirta, and Pekka Vanninen. 1997. Animation of user algorithms on the Web. In *Proceedings. 1997 IEEE Symposium on Visual Languages (Cat. No. 97TB100180).* IEEE, 356–363.

[23] Lassi Haaranen and Teemu Lehtinen. 2015. Teaching git on the side: Version control system as a course platform. In *Proceedings of the 2015 ACM conference on innovation and technology in computer science education.* 87–92.

[24] Lassi Haaranen, Giacomo Mariani, Peter Sormunen, and Teemu Lehtinen. 2020. Complex online material development in CS Courses. In *Proceedings of the 20th Koli Calling International Conference on Computing Education Research.* 1–5.

[25] Juho Hamari, Jonna Koivisto, and Harri Sarsa. 2014. Does gamification work?–a literature review of empirical studies on gamification. In *2014 47th Hawaii international conference on system sciences.* Ieee, 3025–3034.

[26] Jack Hollingsworth. 1960. Automatic Graders for Programming Classes. *Commun. ACM* 3, 10 (oct 1960), 2 pages.

[27] David Hovemeyer, Arto Hellas, Andrew Petersen, and Jaime Spacco. 2017. Progsnap: Sharing Programming Snapshots for Research. In *SIGCSE.* 709.

[28] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli calling international conference on computing education research.* 86–93.

[29] IMS Global Learning Consortium. 2010. Learning Tools Interoperability. http://www.imsglobal.org/toolsinteroperability2.cfm.

[30] David Jackson and Michelle Usher. 1997. Grading student programs using ASSYST. In *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education.* 335–339.

[31] Ville Karavirta, Petri Ihantola, and Teemu Koskinen. 2013. Service-oriented approach to improve interoperability of e-learning systems. In *2013 IEEE 13th International Conference on Advanced Learning Technologies.* IEEE, 341–345.

[32] Ville Karavirta and Clifford A Shaffer. 2013. JSAV: the JavaScript algorithm visualization library. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education.* 159–164.

[33] Ari Korhonen, Thomas Naps, Charles Boisvert, Pilu Crescenzi, Ville Karavirta, Linda Mannila, Bradley Miller, Briana Morrison, Susan H Rodger, Rocky Ross, et al. 2013. Requirements and design strategies for open source interactive computer science ebooks. In *Proceedings of the ITiCSE working group reports conference on Innovation and technology in computer science education-working group reports.* 53–72.

[34] Andy Kurnia, Andrew Lim, and Brenda Cheang. 2001. Online judge. *Computers & Education* 36, 4 (2001), 299–315.

[35] SP Lahtinen, T Lamminjoki, E Sutinen, J Tarhio, and AP Tuovinen. 1996. Towards automated animation of algorithms. In *Proceedings of Fourth International Conference in Central Europe on Computer Graphics and Visualization*, Vol. 96. 150–161.

[36] Raymond Lister, Tony Clear, Dennis J Bouvier, Paul Carter, Anna Eckerdal, Jana Jacková, Mike Lopez, Robert McCartney, Phil Robbins, Otto Seppälä, et al. 2010. Naturally occurring data as research instrument: analyzing examination responses to study the novice programmer. *ACM SIGCSE Bulletin* 41, 4 (2010), 156–173.

[37] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the fourth international workshop on computing education research.* 101–112.

[38] Andrew Luxton-Reilly, Ibrahim Albluwi, Brett A Becker, Michail Giannakos, Amruth N Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory programming: a systematic literature review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education.* 55–106.

[39] Lauri Malmi, Ville Karavirta, Ari Korhonen, Jussi Nikander, Otto Seppälä, and Panu Silvasti. 2004. Visual algorithm simulation exercise system with automatic assessment: TRAKLA2. *Informatics in education* 3, 2 (2004), 267–288.

[40] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In *Working group reports from ITiCSE on Innovation and technology in computer science education.* 125–180.

[41] Brad Miller and David Ranum. 2014. Runestone interactive: tools for creating interactive course materials. In *Proceedings of the first ACM conference on Learning@ scale conference.* 213–214.

[42] Divyansh S Mishra and Stephen H Edwards. 2023. The Programming Exercise Markup Language: Towards Reducing the Effort Needed to Use Automated Grading Tools. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1.* 395–401.

[43] Thomas L Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, et al. 2002. Exploring the role of visualization and engagement in computer science education. In *Working group reports from ITiCSE on Innovation and technology in computer science education.* 131–152.

[44] Don Norman. 2013. *The design of everyday things: Revised and expanded edition.* Basic books.

[45] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. 2022. Automated assessment in computer science education: A state-of-the-art review. *ACM Transactions on Computing Education (TOCE)* 22, 3 (2022), 1–40.

[46] Seymour Papert. 1980. Mindstorms: Children, Computers and Powerful Ideas. (1980).

[47] Willard C Pierson and Susan H Rodger. 1998. Web-based animation of data structures using JAWAA. *ACM SIGCSE Bulletin* 30, 1 (1998), 267–271.

[48] Leo Porter, Mark Guzdial, Charlie McDowell, and Beth Simon. 2013. Success in introductory programming: What works? *Commun. ACM* 56, 8 (2013).

[49] Thomas W Price, David Hovemeyer, Kelly Rivers, Ge Gao, Austin Cory Bart, Ayaan M Kazerouni, Brett A Becker, Andrew Petersen, Luke Gusukuma, Stephen H Edwards, et al. 2020. Progsnap2: A flexible format for programming process data. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education.* 356–362.

[50] Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. Learning and teaching programming: A review and discussion. *Computer science education* 13, 2 (2003), 137–172.

[51] Guido Rößling, Thomas Naps, Mark S Hall, Ville Karavirta, Andreas Kerren, Charles Leska, Andrés Moreno, Rainer Oechsle, Susan H Rodger, Jaime Urquiza-Fuentes, et al. 2006. Merging interactive visualizations with hypertextbooks and course management. In *Working group reports on ITiCSE on Innovation and technology in computer science education.* 166–181.

[52] Jorma Sajaniemi and Marja Kuittinen. 2008. From procedures to objects: A research agenda for the psychology of object-oriented programming education. *Human Technology: An Interdisciplinary Journal on Humans in ICT Environments* (2008).

[53] Rainforest Scully-Blaker. 2014. A practiced practice: Speedrunning through space with de certeau and virilio. *Game Studies* 14, 1 (2014).

[54] Clifford A Shaffer, Ville Karavirta, Ari Korhonen, and Thomas L Naps. 2011. Opendsa: beginning a community active-ebook project. In *Proceedings of the 11th Koli Calling International Conference on computing education research*. 112–117.

[55] Beth Simon, Michael Kohanfars, Jeff Lee, Karen Tamayo, and Quintin Cutts. 2010. Experience report: peer instruction in introductory computing. In *Proceedings of the 41st ACM technical symposium on Computer science education*. 341–345.

[56] Teemu Sirkiä. 2014. Exploring expression-level program visualization in CS1. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*. 153–157.

[57] Teemu Sirkiä and Lassi Haaranen. 2017. Improving online learning activity interoperability with acos server. *Software: Practice and Experience* 47, 11 (2017), 1657–1676.

[58] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A review of generic program visualization systems for introductory programming education. *ACM Transactions on Computing Education (TOCE)* 13, 4 (2013), 1–64.

[59] Juha Sorva and Otto Seppälä. 2014. Research-based design of the first weeks of CS1. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*. 71–80.

[60] Juha Sorva and Teemu Sirkiä. 2010. UUhistle: a software tool for visual program simulation. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. 49–54.

[61] Draylson M Souza, Katia R Felizardo, and Ellen F Barbosa. 2016. A systematic literature review of assessment tools for programming assignments. In *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*. IEEE, 147–156.

[62] Arto Vihavainen, Jonne Airaksinen, and Christopher Watson. 2014. A systematic review of approaches for teaching introductory programming and their influence on success. In *Proceedings of the tenth annual conference on International computing education research*. 19–26.

[63] Arto Vihavainen, Thomas Vikberg, Matti Luukkainen, and Martin Pärtel. 2013. Scaffolding students' learning using test my code. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*. 117–122.

[64] Daniel Zingaro, Yuliya Cherenkova, Olessia Karpova, and Andrew Petersen. 2013. Facilitating code-writing in PI classes. In *Proceeding of the 44th ACM technical symposium on Computer science education*. 585–590.