On the Role of Design in K-12 Computing Education

ALANNAH OLESON, BRETT WORTZMAN, and AMY J. KO, University of Washington, USA

Design is a distinct discipline with its own practices, tools, professions, and areas of scholarship. However, practitioners from other fields often leverage aspects of design in their own work, leading to subfields like engineering design and architecture design that are neither wholly design nor wholly the intersecting discipline. Similarly, design and computing are known to intersect in educational contexts. Unfortunately, we do not yet have a clear understanding of how to characterize the kinds of design that may accompany computing topics, resulting in challenges to teaching and learning. This gap is particularly prevalent in K-12 computing education, where design is often used to promote student engagement but rarely studied as its own disciplinary phenomenon. Toward the goal of better understanding the nature and role of design in computing education, this article motivates and describes two qualitative, exploratory analyses of how design skills manifest in popular K-12 computing education curricula and activities. We find evidence to suggest two types of design within existing computing education curricula and standards: nondisciplinary *problem-space design*, which deals with defining software requirements, and disciplinary *program-space design*, which deals with choosing how best to meet those requirements. We find that these two types of computing design may exist independently, but they often overlap, creating an intriguing intersection of discipline-specific computing design educational activity. Finally, we discuss the practical implications of proceeding with research and educational practice in light of these results, highlighting the need for further exploration into the unique overlap of design and computing education.

CCS Concepts: • **Social and professional topics → Computing education**; *K-12 education*;

Additional Key Words and Phrases: Computing education, design, design education, K-12 education

ACM Reference format:
Alannah Oleson, Brett Wortzman, and Amy J. Ko. 2020. On the Role of Design in K-12 Computing Education. *ACM Trans. Comput. Educ.* 21, 1, Article 2 (December 2020), 34 pages.
https://doi.org/10.1145/3427594

1 INTRODUCTION

The field of design is a distinct discipline, with entire academic departments, areas of scholarship, tools, practices, and professions dedicated to it. However, aspects of design often overlap in meaningful ways with other disciplines, blurring the boundaries between it and the fields in which its skills are used. Foundational design literature claims skills such as planning [5, 121], iterative

This material is based upon work supported by the National Science Foundation under Grant No. 1735123, 1539179, 1703304, 1836813, and DGE-1762114, and unrestricted gifts from Microsoft, Adobe, and Google.

Authors' addresses: A. Oleson and A. J. Ko, The Information School, University of Washington, Seattle, WA, 98105; emails: {olesona, ajko}@uw.edu; B. Wortzman, Paul G. Allen School of Computer Science & Engineering, University of Washington, Seattle, WA, 98105; email: brettwo@cs.washington.edu.

1946-6226/2020/12-ART2 $15.00
https://doi.org/10.1145/3427594

problem solving [4, 117, 122], and evaluation of an artifact's effectiveness, utility, costs, and values [75, 79, 105, 145] as core to the field.[1] Designers are clearly not the only ones who practice these skills though. In disciplines like engineering and architecture, practitioners might create, evaluate, and execute project plans, carrying out design work while not necessarily identifying as designers themselves. This entanglement manifests in discipline-specific design subfields (e.g., engineering design, architecture design), which are neither wholly design nor wholly the intersecting discipline.

In professional practice, computing-related areas also experience this disciplinary overlap with design. Work from software engineering shows that developers often make design decisions as they create software. Developers and software engineers may find themselves solely responsible for user interface design in smaller startups or in companies that lack design culture [86], or they may act as gatekeepers for design decisions in open source projects [87]. Even in larger companies with in-house design teams, software engineers commonly collaborate with or even manage designers, often contributing to major design decisions [98].

We see evidence of computing's overlap with design in computing education contexts as well, often encoded into the definitions and standards of what it means to "know computing." The ACM 2005 Computing Curriculum documentation detailing a set of standards for postsecondary computer science education answers "What is Computing?" with the statement [135]:

> "[W]e can define computing to mean any goal-oriented activity requiring, benefiting from, or creating computers. Thus, computing includes *designing* and building hardware and software systems for a wide range of purposes; . . . " (emphasis added)

Similarly, standards and curricula for primary and secondary computing education often integrate design, often as major foci for learning activities. For instance, many of Code.org's *Hour of Code* modules, intended to be students' first exposures to programming, revolve around both envisioning (designing) and creating (engineering) a small animation or game. The first of the AP CS Principles' Big Ideas of Computer Science centers on *creativity* [7], which is often claimed as a central design trait [4, 41, 43, 77]. Efforts to broaden participation in computing per the computing education research (CER) goals established in 2014 [33] often rely on problem solving [37, 40, 114, 134] or creativity [51, 55, 71, 120] to engage students, both of which involve design-related skills and practices. Influential perspectives on computing education may even refer explicitly to design as a means by which learners are motivated to engage with computing in the first place [62]:

> "A student who takes an introductory computer science course wants to *make* something. Even if the student doesn't want to become a professional software developer, they want to *create* software, to *design* something digital." (Guzdial, emphasis preserved)

Design seems to be intrinsically embedded into computing education contexts, especially at the primary and secondary levels.

Unfortunately, the entanglement of design and computing education often results in difficulties teaching, learning, and applying computing knowledge. Prior work around the teaching and learning of design within computing contexts (generally in human-computer interaction classes) repeatedly finds challenges engaging and promoting students' learning [49, 72, 103, 111, 119], assessing students' design-related competencies [16, 136, 148], and enabling instructors to teach design-related material effectively [27, 74, 110]. Further, design problems appear to pose particular challenges to those who create software interfaces [115], architectures [126], and requirements [1]

[1] A more detailed discussion of what design is and what skills it includes can be found in Section 2.

once they enter the workforce, suggesting that whatever design-related education students receive is not yet properly preparing them to practice these aspects of computing professions. And yet, we can't simply ignore design's role in computing practice—to do so would misrepresent the field and, since prior work shows that design proficiencies can contribute to more pronounced computing expertise [101, 116], runs the risk of being detrimental to computing students' career readiness.

We claim that a number of these pedagogical challenges stem at least in part from a lack of understanding on how to characterize and effectively teach design-related skills and topics within computing education contexts. Recently, Wilcox et al. noted this deficit, contributing initial insights into the role of design in human-computer interaction (HCI) courses and calling for more research into the intersection of design and computing education [144]. Additionally, though myriad CER-related work analyzes computing curricula and standards through lenses of diversity (e.g., [17, 20]), career choices (e.g., [24, 147]), and culture (e.g., [10, 53]), little to no work exists that analyzes computing education materials from a design lens, especially in the rapidly growing area of primary and secondary computing education. Postsecondary computing education materials and curricula are often specific to the class instructor or university, lacking the consistency and reach of materials like the CSTA standards or the Code.org or Exploring CS curricula. Further, there is growing administrative and political interest in integrating computing proficiencies into K-12 education. Given the recent and continuing uptake of computing education in K-12 settings, investigating this demographic in particular provides an opportunity to gain insights relevant to a much larger population than those who enroll in computing higher education.

Prior attempts to critically investigate and characterize discipline-specific manifestations of design at all levels have been shown to produce positive results. Design-centered analyses from K-12 and postsecondary engineering education, for instance, surfaced valuable insights around pedagogy and practice [35, 106, 133], and while theories of disciplinary applied design can be difficult to create and generalize, they benefit both practitioners and learners [36, 58, 128]. We can infer that computing education might find similarly informative results through investigation of the field's interactions and overlap with design. Gaining clarity about the nature of design in computing education contexts would likely help inform curricula and pedagogy about what it should mean to "know computing," especially in K-12 contexts where design is used to motivate learners. More clarity might also shed light on how to frame both design and computing to learners in a concise and understandable way.

This article provides an initial investigation into the nature of discipline-specific computing design in educational contexts, toward the goal of motivating and laying groundwork for further research into the overlap of design and computing education. To enable shared understanding and a working definition of design, we first draw upon literature from the design field itself and synthesize five basic cross-cutting skills of design work, which later form the basis of our analysis in Sections 4 and 5. We then review related work from selected design-based education research sub-disciplines, with a particular focus on how characterizing discipline-specific manifestations of design supports improved teaching and learning outcomes. We present two exploratory, qualitative studies of popular K-12 computing education standards and learning activities. The first study asks, *What is the nature of design and computing's overlap in K-12 education?* and finds evidence to suggest two major types of discipline-specific design appearing within computing education: problem-space and program-space design, which may overlap. The second study, informed by the results of the first, asks, *How do problem- and program-space design manifest in K-12 computing education activities?* and finds evidence to suggest that these two types of design can exist separately, but that they often overlap in educational contexts, creating an intriguing intersection of design and computing activity. Finally, we discuss implications for teaching and learning computing topics that arise in light of these results, highlighting the need for further research in this area to support more effective computing education in both K-12 and higher education.

2 BACKGROUND: WHAT IS DESIGN?

2.1 Perspectives on Design

Definitions of design abound, resulting in many perspectives on what design is and what it entails. The goal of this section is to provide a functional understanding of the nature of design as well as to identify core skills of design as suggested by design literature.

2.1.1 Design as Planning. Some early perspectives on design viewed it as a means to plan how an artifact fits into the context of the world. Design from this perspective is a single stage in the process of creating an artifact that "terminates with a commitment to a plan which is meant to be carried out" [121]. Designers analyze the problem space of a given issue, define concrete requirements for a solution's form, and propose a particular solution that fits the given constraints. This view has its roots in the systematic design movement [5, 18, 79]. Extreme versions of this perspective hold that any act of planning an approach, rather than tinkering and learning through trial and error, is an act of design [121].

2.1.2 Design as Iteration. In contrast to design-as-planning's single-phase view, many hold that design is an iterative, reflexive process of finding the most fit solution to a given problem. Fundamentally, this perspective sees designing as "a process of error-reduction" [4]. Designers adjust a solution's form when they discover a mismatch between it and the world's context, iterating through multiple planning, implementation, and testing phases to discover these mismatches. Central to this view are the ideas of productive failure [117] and modularization [4]. Many models of design processes incorporate iteration [45].

2.1.3 Design as Expression. Some perspectives on design hold that the designer embodies aspects of themselves or their values within designed artifacts. For instance, design may be a means of communication [84, 105] or rhetoric [21, 42, 48, 102] (e.g., how Facebook's design expresses Mark Zuckerberg's view of a "post-privacy" world [78]). Each time a designer creates an artifact, they implicitly communicate about its nature in choosing how to represent it [105] and uphold particular values, whether they intend to or not [52, 121]. Designed objects from this perspective are an extension of the designer's self and an outlet to express their creativity and emotions [15, 52, 84, 108].

2.1.4 Design as a Means to Address Complex Problems. Finally, many view design as the only way to address "wicked" problems [19]—ill-structured, often incomplete, complex issues not easily solved through traditional methods [122, 129]. Optimal solutions cannot be found for wicked problems unless unrealistic constraints are imposed, in which case the solutions fail to fully address the original context. Many of the most pressing sociocultural problems of today are wicked. Consider, for instance, finding effective strategies to address climate change or embedded social injustice, which are multifaceted and involve complex tradeoffs. This design perspective manifests itself in DesignX [109] and Transition Design [76], as well as in the popularity of "design thinking" as taught at Stanford's d.school [22].

2.2 Core Design Skills

In order to operationalize these somewhat competing perspectives in a concrete manner for our analysis in Sections 4 and 5, we found it useful to draw out the skills that foundational design literature claims as integral to design. Five skills in particular seemed to cut across the definitions of design discussed above, suggesting they are central to designerly thinking and practice. We note here that though this list of skills is the result of our own synthesis of the literature, it bears similarity to sets of design skills presented in other studies of discipline-specific design education research (e.g., [35]), implying a base level of validity.

2.2.1 Understanding the Context (UtC). Designers use this skill to inform ideation. To fully understand the context, designers must learn about both existing systems that attempt to address a problem and any stakeholders who will interact with the designed artifact. Techniques employed while trying to understand the context involve primary or secondary user research or focus groups [45, 142] and often also involves perspective-taking or empathizing with target users [65, 132, 146]. From these activities, designers identify pain points and opportunities for improvement that exist in current systems. The information designers gain from this work forms the underlying rationale for design requirements. Gaining a rich understanding of the context is critical to ensure that designers adequately address their clients', customers', or users' needs.

2.2.2 Creative Ideation (CI). Creativity is central to design [4, 41, 43, 77]. Designers employ creative ideation as a skill when they begin to generate many possible solutions. The goal of this practice is not to fixate on a single idea, but to explore the solution space and generate many promising approaches. Creative ideation includes activities like prototyping (i.e., ideating through making) and otherwise externalizing one's ideas through sketching or verbal communication. This skill also encompasses practices like "stealing" successful ideas to analogous problems [60, 66] and composing features of those ideas to create novel things [104, 140].

2.2.3 Evaluation & Synthesis (ES). As designers generate ideas, they evaluate them against the constraints of the problem context. The goal of evaluation and synthesis in design is to practice convergent thinking and begin narrowing down the solution space to a handful of feasible options. Formal evaluation processes might include critique sessions [75, 145] to gain constructive feedback from others, evaluation against design heuristics [107], game testing or other empirical evaluations (e.g., [90, 143]), or code reviews. Informal evaluations also occur in the design process: designers make their own expert judgments about which ideas are worth pursuing and feasible to implement. Designers often compare potential solutions against the time and resources they have available or the prioritization of certain user needs over others, performing an analysis of tradeoffs inherent in implementing one solution over another.

2.2.4 Iterative Improvement (II). Design is an inherently iterative discipline. Designers use iterative improvement to optimize their solutions and incrementally update them to fit the problem space. This skill involves the ability to adapt solutions based on feedback, whether that feedback is from a teammate, a user, or even a technological system such as a debugger. Iterative improvement also necessitates an approach to failure as a learning experience, not as a catastrophic ending: breakdowns in the design reveal opportunities for improvement [117]. Each iteration on a solution informs the designer about the problem space's constraints—information that is taken into account in successive iterations.

2.2.5 Communication (Comm). Underlying and supporting all designerly practice is the skill of communication, especially with those in fields other than one's own. Though this skill is by no means exclusive to design, communication is a basic design competency [6]. Without the ability to communicate ideas to teammates, stakeholders, and the general public, designers cannot ensure that their designs are implemented correctly or that stakeholders understand their roles and responsibilities. Examples of this skill in practice include describing concepts and ideas in terms that a particular audience understands, presenting results in fitting manners, and working with teams (e.g., of developers, engineers, or stakeholders) to define realistic common goals [2]. Communication also encompasses the notion of field literacy: designers must be knowledgeable enough in their stakeholders' fields and environments that they can interpret feedback and design a solution appropriate to the context.

3 RELATED WORK: APPLIED DESIGN EDUCATION

Though the core design skills described above remain relatively stable, they manifest in different ways when design overlaps with other disciplines. Many fields have benefited from rigorous investigation of the role design plays within them. Often, the knowledge gained from this work manifests in a subarea of the discipline's existing education research. For instance, there exists engineering education research, which studies the learning and teaching of engineering, but also engineering *design* education research, which studies the learning and teaching of engineering-specific (i.e., disciplinary) design. Here we briefly describe work from architecture design education and engineering design education, two disciplines in which an understanding of how design intersects with and manifests within in the field helped improve curricula and pedagogy. We also present relevant work from STEM education, highlighting how design activities are often used in classrooms to promote understanding of STEM concepts. Finally, we present related work on teaching design skills in computing contexts, framing this article's initial attempts to gain insight into the role of design in computing education as a way to ground future work in disciplinary computing design education research.

3.1 In Architecture

Architecture was one of the earliest fields to recognize and study its overlap with design in educational contexts. For instance, Schön's influential theory of *reflection-in-action* [124] arose in part from observation of architectural design studios. Schön used this theory about the nature of design in educational contexts to inform frameworks for design knowledge and pedagogical strategies to help novice designers become effective professionals [125]. Lawson studied how first-year and final-year architecture students acquired design skills, comparing both groups' cognitive strategies for problem solving to scientists and non-designers. The results of this study not only improved pedagogy by characterizing gaps between first- and final-year design students but also served as a basis for modifying CAD modeling tools to better enable architectural design practice [94]. Goel and Pirolli's recognition of the distinction between design and non-design (engineering) tasks arose from studying architecture students' problem conceptions, which enabled more tailored pedagogy for architecture design problem-solving instruction [58]. Notably, Goel and Pirolli also reported that the skills of design practiced by architecture students seemed to differ from the kinds of design practiced by mechanical engineering or curriculum design students, indicating a basis for discipline-specific design education research.

More recently, work in the area has built on these foundations to explore how well the current state of architecture design education prepares students to deal with the complex problems that will face them when they enter the workforce. Goldschmidt and Sever discovered differences in the role of creative ideation in undergraduate and graduate designers' processes: undergraduate architectural and industrial designers tended toward creating products, while graduate students tended to create services or systems [60]. Based on this analysis, Goldschmidt and Rodgers critiqued the effectiveness of architecture design pedagogy, suggesting that educators focus on preparing students to address ill-structured "wicked" problems rather than teaching design methods [59]. Chiaradia et al. proposed educational strategies for integrating values in urban design education (an offshoot of architecture), instilling ethics and value-sensitive judgment making into student designers [25]. These and other related works contributed to discourse around the nature of design in architecture education, affording both pedagogical improvements and further refinement of the discipline's conceptions on how design manifests these kinds of contexts.

3.2 In Engineering

Engineering design education, while academically younger than architecture design education, has similarly worked to identify the role of design within its discipline. Engineering design is similar to computing design in that the two disciplines involved appear to be distinct on the surface, even though engineers often make design decisions in practice. Sim and Duffy's early attempt to categorize the ways design intersected with engineering education resulted in a generic ontology of engineering design activities [128], which not only shed light on classroom activities but went on to inform product life cycle management [85] and system design [57] as well. The work of Smith et al. work on characterizing "pedagogies of engagement" in the engineering classroom found that while engineering students tended to be more engaged in project-based learning classes where they both designed and implemented their ideas, there were still many open pedagogical and curricular questions about how exactly to teach engineering design practices [131]. Informed by this work, Prince and Felder found that pedagogies that implement engineering design well were often more effective at promoting learning than traditional engineering education [118].

A notable body of work in engineering design education focuses on the development of design expertise in engineering students. For instance, Lemons et al. studied how physically building models of design artifacts interacts with engineering students' design expertise [96]. Though the authors explicitly caution against interpreting their results too broadly due to the limitations of their sample, they found evidence to suggest that model construction can enhance creative thinking and enable awareness of metacognitive strategies—both of which may contribute to design expertise development. The Atman et al. series of studies on the differences between freshman and senior engineering design students (e.g., [8, 9]) helped define concrete learning goals for engineering design pedagogy. To augment this applied work with theoretical understanding, they later analyzed nearly a decade's worth of this work through the lens of Schön's reflective practitioner theory [124], concluding that certain observable classroom design behaviors are reasonably indicative of design expertise [3]. Leveraging the insights afforded by this investigation, they identified problem-setting and engaging in reflective conversations with the design space as two trends of interest in design expertise representation and suggest that future engineering design education research should explore effective ways of imparting these skills to students.

Within the last decade, multiple researchers have called for more emphasis on discipline-specific engineering education research (e.g., [54, 130]), spurred in part by the trend toward discipline-based educational research, but also by the promising results already emerging from the field. Crismond and Adams established a comprehensive framework of pedagogy needed to teach K-12 engineering design effectively, encompassing student misconceptions, teaching strategies, and more [35], helping to define the role design should have in engineering education. Starkey et al. investigated how engineering students' creative ideation processes change throughout the course of a project, finding that the design task itself has an impact on creativity and laying the groundwork for further educational research into how to teach creativity to engineering students [133]. Some of the newest work in engineering design education, Neroni and Crilly's exploration of how engineering students experience design fixation in isolation and in groups, added to the general body of work on design fixation and suggested engineering-specific pedagogy for decreasing fixation [106]. These accomplishments around the learning and teaching of design in engineering contexts would almost certainly not have been possible without the work that came before it that helped clarify the role of design in engineering.

3.3 In STEM Education

In STEM education, design activities are often used not to impart design skills themselves, but to help students learn about STEM-related topics through designing. For instance, Hmelo-Silver's Problem-Based Learning (PBL) approach situates students as active participants in their learning processes [69]. Many of the goals of PBL, such as the development of problem-solving and collaboration skills in students, overlap with the goals of design education and the crosscutting skills of design discussed in Section 2.2. Hmelo et al. contributed a detailed study of how a Learning By Design (LBD) approach [91, 92] could help middle school students gain knowledge of the human respiratory system, finding that students in the LBD classes showed evidence of better learning outcomes than those who received traditional instruction [68]. They gave several recommendations for educational practice when including design activities in instruction but noted the importance of having sufficient time for instruction in this style. Later, Hmelo-Silver and Pfeffer studied the differences in how experts and novices think about complex systems (in this case, the ecosystem of an aquarium) and found that while novices organize knowledge based on perceptually salient structural features (e.g., fish, plants, filters), experts organize knowledge at the behavioral (what the filter does) and functional (why the filter is necessary) levels [70]. This is consistent with prior work in physics education on differences between novices and experts [14]. This line of work later informed the pedagogical strategy of having students describe complex systems in terms in the form of structure-behavior-function models, which promoted deeper understandings of the system as a whole [141].

In a similar fashion, a substantial body of work on engaging students in designing hypermedia and multimedia systems (e.g., [23, 89, 100]) indicates that engaging primary and secondary students in learning through design activities can support higher-order cognitive skills needed to effectively contribute to complex projects. Lehrer offers an example of how secondary American history students took on a more active role in their education when given the opportunity to design systems that reflected their knowledge [95]. Liu studied how hypermedia authoring impacted primary students' higher-order thinking and found that the kinds of skills practiced during design activities contributed to students' holistic growth as learners [99]. In particular, they note that situating skills such as planning, critical reflection, and collaboration within design activities may help students learn to value these skills more highly, which may contribute to success in later project-based education.

Further, design activities in STEM education can promote student engagement and create a more inclusive classroom culture. Kafai's line of work on constructionist learning through designing games [82] aims to help students engage more deeply with STEM concepts. Early on, Kafai et al. explored using game design activities to promote mathematical learning and found that both students and educators who engaged in design practices created higher-quality games, thought in more principled ways about the topic of instruction, and better leveraged their real-world knowledge bases to understand mathematical concepts [80]. They later framed constructionist educational game development as a way of enabling early ownership and participation in digital culture [81]. Others have highlighted the potential for equitable education that arises from design-based instruction. After implementing their LBD approach in middle school science classrooms, Kolodner et al. found that LBD activities helped build a classroom culture in which students of diverse backgrounds and motivations were more comfortable engaging with scientific concepts, often showing evidence of improved learning outcomes compared to control classes [91].

Overall, much of the work concerning design in STEM education uses design as a means to enable learning of other, non-design-related concepts. While this literature presents many valuable insights into how educators can leverage design for use in their classes, using design as a tool to promote understandings of STEM topics is not the same as investigating the teaching and learning of embedded disciplinary design skills. Compared to architecture or engineering design education, relatively little prior work in STEM education positions its main goals as attempts to establish how design manifests in discipline-specific design education (e.g., studying the forms of design that exist in primary and secondary math or science classrooms). In this article, our goal is to understand the nature of design *itself* in computing education, not necessarily how design activities can be used to better impart computing proficiencies.

3.4 In Computing

Computing education research is a young discipline. As a result, we do not yet have an established subarea for disciplinary design education research (analogous to architecture or engineering design education research). Software engineering research provides some insights into the nature of design's role in computing practice. For instance, software professionals often encounter design-related challenges in practices such as requirements elicitation [1] and interface creation [115]. These challenges may be compounded by the often blurry boundaries between the software design and software implementation. Unlike more traditional fields in which designers work with physical media like wood or metal, software designers work with code and are not limited by the constraints of physical material (load-bearing capacities, conductivity, etc.) [115]. These unclear distinctions lead to software development processes in which activities of defining and implementing requirements are tightly, iteratively coupled, such as Boehm's spiral model of software development [13].

A small but rapidly growing focus on education exists in human-computer interaction (HCI) literature, studying in part how computing students learn the design skills needed to create usable, useful software interfaces. This body of work has identified many practical challenges of teaching and learning HCI. For instance, it is often difficult to reliably engage students in HCI classes [72, 103, 119], especially when they view the course content as "inessential" [27], "easy," or "commonsense" [49]. HCI educators also find it difficult to assess students' design work [16, 136, 148], perhaps due to a lack of pedagogical content knowledge for design-related HCI topics [110, 111]. Overall, however, Lewthwaite and Sloan found that there is no agreed-upon pedagogy for teaching HCI principles and that most HCI education work "comprise[s] of teachers' reflections on their own practice and course design" [97], making it difficult to synthesize a generalizable notion of design in HCI from existing literature. Given that HCI education faces a time crunch in already overcrowded computing curricula [27, 63], educators must prioritize some topics and exclude others, but there is no agreement on what the core topics of HCI even are [61].

Though software engineering work describes many challenges developers face that involve disciplinary design, it does not necessarily provide the clarity required for computing educators to teach software design skills effectively or to know how students will respond when learning them. Similarly, HCI education work excels at identifying challenges that exist in design-related computing education, but it also appears to lack unified notions around the role of design in computing education contexts. The goal of this article is to help establish this much-needed clarity so that educators and researchers can address these challenges and create more precise, useful, and effective curriculum and pedagogy. Recent work from Wilcox et al. noted the lack of existing insights into the nature of design in higher education HCI instruction and called for further educational research to untangle the nature of design in supporting computing learning [144]. We hope to build upon this work by providing additional insights into the role of design in K-12 computing education, which has not yet been analyzed through a design lens, and proposing a conceptual taxonomy of design in computing that may be useful at all levels of instruction.

4 STUDY 1: WHAT IS THE NATURE OF DESIGN AND COMPUTING'S OVERLAP IN K-12 EDUCATION?

To begin building an understanding of how design skills manifest in computing education contexts, we performed a deductive qualitative analysis [113] on three sets of popular K-12 computing curricula and standards. We examined the learning objectives from each curriculum for the presence of the five core design skills from the literature described in Section 2.2. We chose to analyze the curricula and standards at the learning objective level because learning objectives and skills represent similar granularities: achieved learning objectives imply proficiency in certain skills. Computing education learning objectives that imply proficiency in design-related skills may reveal connections between design and computing and help us to understand the nature of the two disciplines' intersection.

4.1 Sampling Rationale: Typical Cases

To select the sets of learning objectives, we employed Patton's group characteristics (typical cases) sampling strategy [113] to identify curricula and standards that might be representative of many K-12 students' computing education. This resulted in three sets of learning objectives from the following sources:[2]

- The *CSTA K-12 CS Standards* [32] list what competencies primary and secondary students should have to claim computing proficiency. Most recently updated in 2017, these standards have informed many introductory computing curricula in multiple countries [47, 112].
- Code.org's *CS Discoveries* curriculum takes a self-proclaimed "wide lens on [CS] by covering topics such as programming, physical computing, HTML/CSS, and data" [28]. Freely available online, these materials are designed for early secondary education.
- The *AP CS Principles* course for late secondary education helps students gain familiarity in basic computing concepts, regardless of whether they intend to major in computing in post-secondary education. A record 70,000 students took the AP CS Principles end-of-course exam in May 2018 [30].

Many nations engage in discourse around the boundaries and core concepts of computing education. For instance, the United Kingdom, Australia, and Korea have all participated in the development of primary and secondary computing education standards and curricula, along with numerous other countries. Since it was not feasible to study all the primary and secondary computing curricula that exist, we necessarily had to scope our analysis. For this initial study, we chose three sets of learning objectives developed primarily for U.S.-centric learning contexts, which may limit the generalizability of the findings. Nonetheless, these learning objectives are representative of many students' computing educations. A 2019 report on the status of computer science education policy in the U.S. indicated that 39 of the 50 states had adopted CS standards for education [29]. The standards states adopted were almost always informed by the CSTA standards. As mentioned before, tens of thousands of students take the AP CS Principles exam each year, and more than 100,000 teachers have participated in Code.org's CS Discoveries professional development program to date. Because of this, we feel that the three chosen sets of objectives are sufficient for this exploratory investigation. Future work should apply this mode of analysis to other countries' objectives and standards in order to deepen our understanding around the role of design in computing education in global contexts.

[2]Each of the three sets of learning objectives for Study 1 were collected on January 24, 2019, from the sources found in the references. Our analysis reflects the published materials at that time and does not cover updates between the time of analysis and the time of publication (such as Code.org's recent rework of the CS Discoveries Design Process unit).

4.2 Analysis of Learning Objectives

To ensure that our analysis encompassed as many diverse, relevant perspectives as possible, we had three individuals with varying backgrounds analyze the learning objectives for the presence of design skills. All analysts were either researchers or educators at a large, public, U.S.-based university. They were:

- The first author, a computing education researcher with 5 years of research experience in HCI and design methods at the time of analysis, including 2 years researching the overlap of design and computing education.
- A CS educator with 9 years of experience teaching secondary and post-secondary CS in the U.S. and significant experience designing primary and secondary CS curricula and mapping them to standards. After performing their analysis, the CS educator became interested in the project and joined the research team, becoming the second author.
- A design educator with 4 years of experience teaching design (including interaction design) in the U.S. at the post-secondary level and 7 years of experience as a practicing industrial and architectural designer.

The analysts' diverse backgrounds are essential to understanding the nature of the intersection of design and computing. To achieve robust and reliable understanding, it is necessary to (attempt to) surface all possible connections between design and computing from as many relevant perspectives as possible. This goal would not be possible if the analysis consisted of three similarly trained individuals identifying connections: they would likely miss important relationships between the two fields due to their limited scope of expertise. Correspondingly, our goal for this deductive analysis was not to reach inter-rater agreement (a metric often used to evaluate the consistency of qualitative judgments). Instead, consistent with the view of qualitative work presented by Hammer and Berland [64], we hoped to gain a holistic view of the role of design within computing from many relevant perspectives and surface multiple diverse interpretations.

Each analyst performed their mapping independently. We provided analysts with detailed descriptions of each design skill similar to those presented in Section 2.2. For each computing learning objective, we asked analysts to determine which (if any) design skills they felt were represented in the objective, allowing for multiple skills to be represented in a single objective. We also encouraged analysts to apply their individual expertise when determining which learning objectives mapped to each skill. After each analyst completed their work, the first two authors (who were also two of the three analysts) examined the results for patterns.

4.3 Results

4.3.1 Design Skills in Computing Learning Objectives. In total, the analysts examined 384 learning objectives for the presence of core design skills.

As intended, the analysts surfaced diverse interpretations of which computing learning objectives contained design components. For example, the CS educator interpreted any objective that involved creating a digital artifact or developing a computer program as aligning to the *Creative Ideation*, *Evaluation & Synthesis*, and *Iterative Improvement* skills, as, in their experience, these skills are inherently exercised in the process developing a program or digital artifact. The computing education researcher took a more conservative stance due to their prior knowledge of both design and computing, leading to fewer overall identified instances of design in their results. The design educator identified some learning objectives as falling under a sixth skill they named *Analysis and Synthesis of Information*. After the design educator discussed the rationale behind their choice with the other two, the analysts slightly revised the descriptions of the original five core design skills to clarify language around analysis and synthesis (though they elected to retain only the five existing design skills from the literature) and audited their analyses against the new descriptions. The results presented below reflect the audited analyses.

Some learning objectives did not seem to imply any of the five core design skills, indicated by a lack of design skill codes by any analyst. As might be expected, these were often the most obviously computing-focused objectives, encompassing skills typically ascribed to traditional computer science. Many of these objectives revolved around basic familiarity with computing concepts, such as *"Define an algorithm as the series of commands a computer uses to process information"* (Code.org 1.6) or *"Store, copy, search, retrieve, modify, and delete information using a computing device and define the information stored as data."* (CSTA 1A-DA-05). Other learning objectives that did not seem to contain design skills were more engineering focused, often simply asking students to implement programming constructs, such as *"Create and link to an external style sheet"* (Code.org 2.10), *"Create procedures with parameters to organize code and make it easier to reuse"* (CSTA 2-AP-14), or *"Create and modify an array"* (Code.org 6.10).

The classification of many learning objectives was more ambiguous. Two modes of conflicting agreement surfaced in the objective analyses, delimited by how they manifested in the coding results. (These two types, by their definitions, make up the entirety of analysts' disagreement.) One kind of conflict occurred when some analysts identified design in an objective though others did not. The most common form of this was the design educator and the computing educator agreeing that an objective contained particular design skills while the computing education researcher did not. In particular, the two educators took a broader stance than the researcher on what kinds of learning goals implied *Understanding the Context* and *Communication* skills. For instance, both educators saw UtC and Comm in objectives like *"Explain the beneficial and harmful effects that intellectual property laws can have on innovation"* (CSTA 3A-IC-28) and *"Explain characteristics of the Internet and the systems built on it"* (AP CS P 6.2.1). The computing education researcher did not see design skills in these objectives, conservatively interpreting them as asking students to relay information they had read or heard. A second type of conflict occurred when all analysts indicated there was some kind of design within an objective but disagreed on which particular design skill(s) it contained. For example, each analyst mapped *"Develop a correct program to solve problems"* (AP CS P 5.1.2) to three design skills. The design educator mapped the objective to *UtC*, *CI*, and *ES*; the computing educator to *CI*, *ES*, and *II*; and the computing education researcher to *UtC*, *II*, and *Comm*. This learning objective's description contained examples of what students might do to achieve the objective, though it appears that the analysts all interpreted these examples differently and focused on different aspects. Similar patterns occurred throughout the analysts' results in all three curricula. The two kinds of disagreement surfaced by the analysis highlight the importance of including many diverse perspectives in our initial attempts to understand the nature of discipline-specific computing design. Similarly trained analysts (e.g., three computing education researchers) might have missed some of these connections between the two fields, overlooking important aspects needed to gain a complete understanding of the design's role in computing education.

Despite these conflicts, the three analysts did agree that some computing learning objectives contained specific design skills, indicated by all analysts agreeing on a single skill code. To fulfill the learning objectives that correspond to the following themes, computing students likely must take on design roles.

Situating Choices in the Real World (UtC). All three analysts marked that learning objectives such as *"Identify existing cybersecurity concerns and potential options to address these issues with the Internet and the systems built on it"* (AP CS P 6.3.1) and *"Explain the connections between computing and real-world contexts, including economic, social, and cultural contexts"* (AP CS P 7.4.1) corresponded to the design skill of *Understanding the Context*. Similarly, learning objectives like *"Recommend improvements to the design of computing devices, based on an analysis of how users interact with the devices"* (CSTA 2-CS-01) and *"Systematically design and develop programs for broad audiences by incorporating feedback from users"* (CSTA 3A-AP-19) indicate that students should be able to involve stakeholder perspectives in the process of creating their software. Learning objectives about contextualizing computing tended to occur in units about web development and game development/animation (Code.org), lessons about global impacts of computing and problem analysis (AP CS P), and standards about networks and the Internet or computing systems (CSTA).

Generating Ideas (CI). The analysts often found the design skill of *Creative Ideation* in objectives corresponding to generating many creative ideas or coming up with new concepts. For instance, deceptively simple objectives like *"Design the user interface of an app"* (Code.org 4.7) and more complex objectives such as *"Create a new computational artifact by combining or modifying existing artifacts"* (AP CS P 1.2.2) suggest that students should be able to generate creative, novel ideas for their software. Many objectives falling under this theme explicitly involved brainstorming as well, such as *"Brainstorm ways to improve the accessibility and usability of technology products for the diverse needs and wants of users"* (CSTA 1B-IC-19) and *"Brainstorm potential solutions to a specific problem"* (Code.org 4.6). Learning objectives involving generating ideas tended to occur in units about problem solving and user-centered design (Code.org), in lessons on creating computational artifacts (AP CS P), and in standards about algorithms and programming (CSTA).

Critiquing and Evaluating Tradeoffs (ES). Students learning computing with these objectives are expected to be able to evaluate how well software meets given or discovered constraints. The analysts unanimously identified *Evaluation & Synthesis* learning objectives like *"Test and refine computational artifacts to reduce bias and equity deficits"* (CSTA 3A-IC-25), *"Analyze the correctness, usability, functionality, and suitability of computational artifacts"* (AP CS P 1.2.5), and *"Critique a design through the perspective of a user profile"* (Code.org 4.2). Notably, computing learning objectives in the analyzed sets often expected students to evaluate the tradeoffs implicit in design choices, such as *"Compare tradeoffs associated with computing technologies that affect people's everyday activities and career options"* (CSTA 2-IC-20), *"Evaluate computational artifacts to maximize their beneficial effects and minimize harmful effects on society"* (CSTA 3B-IC-25), and *"Consider the needs of diverse users when designing a product"* (Code.org 6.15). Learning objectives involving critique and evaluation tended to occur in units on data and society or user-centered design, (Code.org), lessons involving analysis of artifacts (AP CS P), and standards about the impacts of computing or algorithms and programming (CSTA).

Incrementally Updating Software (II). Some learning objectives very obviously implied the design skill of *Iterative Improvement*. For example, analysts mapped objectives such as *"Use an iterative process to plan the development of a program by including others' perspectives and considering user preferences"* (CSTA 1B-AP-13) and *"Iteratively improve upon a system for representing information by testing and responding to feedback"* (Code.org 5.2) unanimously to II. The analysts also saw iteration inherent in broader objectives like *"Apply a creative development process when creating computational artifacts"* (AP CS P 1.1.1). Learning objectives involving incremental updates to systems tended to occur in units about problem solving or data and society (Code.org), in lessons involving abstraction (AP CS P), and in standards about algorithms and programming (CSTA).

Working with Others (Comm). The need to communicate with peers surfaced in two main ways throughout each set of learning objectives. First, many objectives expected productive teamwork, evidenced in objectives like *"Communicate and collaborate with classmates in order to solve a problem"* (Code.org 1.1) and *"Collaborate in the creation of computational artifacts"* (AP CS P 1.2.4). Second, some objectives implied the need to become proficient at presenting and documenting design rationales, such as *"Communicate the design and intended use of program"* (Code.org 4.10) and *"Explain the design choices they made on their website to other people"* (Code.org 2.14). This second category of *Communication* is reminiscent of design specifications—documents drafted by designers that contain requirements for developers and engineers to implement (or for managers to approve). Occasionally the two categories overlap: *"Describe choices made during program development using code comments, presentations, and demonstrations"* (CSTA 1B-AP-17) includes both within-group communication (code comments) and external communication (presentations). Learning objectives involving working with others were common throughout each set of learning objectives. They tended to occur in units about human-centered design and web development (Code.org), lessons involving communication and collaboration (AP CS P), and standards about the impacts of computing, data analysis, and algorithms and programming (CSTA).

4.3.2 Types of Design within Computing Education. While examining the results of the design skills analysis, the first and third authors observed that there seemed to be more than one distinct type of design represented in the computing learning objectives. The first author performed preliminary affinity diagramming on the learning objectives that the original analysts identified as containing design skills to identify higher-level categorizations. Further collaborative discussion and refinement of the categories with all authors revealed the following two types of design.

Problem-Space Design: The "What" and the "Why." Problem-space design in computing answers the questions "What will this software do (or enable users to do) in the context of the world?" and "Why does the world need this software?" This form of design involves identifying, defining, and evaluating requirements for what an artifact is and what it should be able to do. The goal of problem-space design is to propose a solution to a particular problem that takes into account the constraints of the context as well as the views and requirements of stakeholders. In industry, requirements that reflect the problem space are generally created by project managers, product managers, or interaction or user experience (UX) designers. On the other hand, learners in computing education contexts typically lead their own projects or work with small teams. As a result, many students practice problem-space design to an extent. The learning objective *"Brainstorm ways to improve the accessibility and usability of technology products for the diverse needs and wants of users"* (CSTA 1B-IC-19) seems to involve only problem-space design, situating software requirements in the real world by expecting students to justify *why* certain software doesn't meet user needs and *what* that software might look like instead. Computing students might also practice problem-space design when they engage in work to define and update high-level software requirements, evaluate the fit of software against real-world constraints, or present their design rationale to peers or instructors.

Program-Space Design: The "How." Program-space design answers the question "How should the available tools and resources be used to effectively implement this software?" This form of design involves deciding how to meet requirements that result from problem-space design activities. Selecting the proper algorithms, data structures, and function calls to generate desired behavior are examples of program-space design. This kind of design is most often seen in software engineering and programming practices. In computing education, students writing code to meet requirements engage deeply in program-space design. Many of the objectives that seemed to involve only program-space design mirrored basic program design and software engineering skills (e.g., *"Develop an algorithm for implementation in a program"* (AP CS P 4.1.1)). Students could also practice program-space design without writing code, such as in "unplugged" activities where they create algorithms to solve problems. In this learning objective, students are not expected to justify *what* they are making or *why* they should address the given problem with an algorithmic approach: they are only expected to choose *how* to implement the algorithm so that it fits given requirements, though the choices they make while doing so still imply design decisions. Computing students might also practice program-space design when they decide which kinds of programming constructs to use in a program (e.g., conditionals vs. loops) or when they define and adhere to programming style guidelines to help communicate with teammates.

Interestingly, some learning objectives seemed to imply both problem- and program-space design practices. For instance, *"Iteratively improve upon a system for representing information by testing and responding to feedback"* (Code.org 5.2) seems to involve the problem-space skill of testing with users and applying feedback to refine higher-level *what* and *why* requirements. It also seems to imply program-space design in the iterative improvement of the program's code-level implementation to refine *how* the system works. The existence of this ambiguous space between problem- and program-space design might be one reason the role of design has traditionally been so unclear and why (as suggested by our analysis in this section) it can be difficult to clearly identify. while program-space design might be considered an inherent, disciplinary form of design that exists within computing, problem-space design shares more characteristics with the discipline of design than it does with computer science or software engineering.

5 STUDY 2: HOW DO PROBLEM- AND PROGRAM-SPACE DESIGN MANIFEST IN K-12 COMPUTING EDUCATION ACTIVITIES?

The results of Study 1 suggest that the nature of design and K-12 computing education's overlap is characterized by two distinct kinds of design: problem-space design, in which students discover and define problem requirements, and program-space design, in which students decide the most fit way to meet the discovered constraints. These dual roles of design arose from analysis of learning objectives that imply certain computing proficiencies. However, what if learning objectives that contain design are simply ignored in practice? What if the curricula and standards we analyzed *do* contain design, but the design work is not instantiated in classroom activities? If the activities computing students participate in don't actually have design components, then the practical need for clarity around design and computing education's overlap becomes less pronounced. To address this possibility, we conducted a second deductive qualitative analysis on three sets of *student activities* from three K-12 computing education curricula.

5.1 Sampling Rationale: Confirming and Disconfirming Cases

To select the curricula for Study 2's analysis, we employed a confirming and disconfirming cases sampling strategy [113], as well as selecting another typical case for adequate coverage.[3]

- Code.org's *CS Discoveries* [28] acts as our confirming case. In Study 1, we analyzed CS Discoveries' learning objectives for design skills; We should expect those design skills to be instantiated in the curriculum's student activities.
- The *AP CS A* course [31] acts as our disconfirming case, insofar as that is possible. In contrast to AP CS Principles' broad goal of increasing participation in computing, AP CS A is designed to mimic a university-level introductory programming course. Upper-level secondary students who take AP CS often intend to major in computer science in postsecondary education and can receive CS1 credit by passing the AP CS A exam. Due to the more technical focus of the course, we can expect its design-related activities to be more like discipline-specific design (program-space) than nondisciplinary (problem-space) design, potentially also occurring at a lower frequency than the other two cases.

[3]The Code.org CSD materials were the same as Study 1 for consistency and were collected on January 24, 2019, from the sources found in the references. The AP CS A and Exploring CS materials were collected on June 26, 2019. As before, our analysis reflects the published materials at that time and does not cover updates between the time of analysis and the time of publication.

- *Exploring Computer Science* [50] was selected as another typical case in order to ensure our analysis would encompass representative material. Exploring CS is a year-long, research-based curriculum intended to teach introductory computing concepts through inquiry-based activities in late secondary education. As of 2018, more than 50,000 students in 25 U.S. states and Puerto Rico had learned computing through the Exploring CS curriculum.

Exploring CS had explicitly labeled student activities within the lesson plans upon which we based our analysis. *CS Discoveries* also had labeled activities for the majority of their lesson plans; for the lessons that did not contain one or more explicitly labeled activities, we considered the entire lesson a single activity. The *AP CS A* course description provided sample instructional activities at the beginning of each unit. Though these were from the point of view of the teacher, they described students' activities, so we were able to use them in our analysis. Of note, the *AP CS A* curriculum guide did not provide sample activities for every lesson, which limits the findings of our analysis to a strict subset of all *AP CS A* student activities.

5.2 Analysis of Student Activities

Two analysts took part in the analysis of design in student activities:

- The first author, the computing education researcher whose expertise is described in Study 1 (Section 4)
- The third author, a researcher and educator at a large, public, U.S.-based university with 10 years of experience in computing education research, 10 years of software engineering research and teaching, and 15 years of HCI + design research and teaching

The first author used a deductive process to qualitatively analyze each student activity for the presence of design based on the two types of design in computing uncovered in Study 1 (Section 4.3) and the skills of design identified in the literature (Section 2.2). Each student activity could be marked with anywhere from 0 (if no design were present) to 10 (5 problem-space design skills + the same 5 program-space design skills) codes. As before, the first author tended toward conservative interpretations of the student activities, relying on each activity's explicit accompanying text and descriptive materials to determine the presence and flavor of design. They also memoized each code with selection rationale.

To establish validity, the third author independently analyzed the student activity sets and marked codes upon which the two analysts disagreed, memoizing their rationale in a short comment. Then, the two analysts met to discuss discrepancies in their results and decide upon final categorizations. As mentioned in Section 4, we adhere to the perspective on qualitative coding presented by Hammer and Berland [64], treating the results of deductive coding as organizations of claims about data rather than quantitative data in themselves. As a result, collaborative discussion about disagreements led to refinement of the codeset. During their discussion, the two analysts noted the following major patterns in their disagreement:

- Applications of the *Understanding the Context* skill code for both problem- and program-space design differed based on whether one considered information gathering outside of the context of any particular problem to be a design activity (e.g., "Groups of students create lists of their ideas of what a computer is," ECS 1.1). Analysts decided that an activity should only be coded as UtC if there was a specific, defined problem the student was trying to solve, as opposed to discussing problems in general.
- Applications of various program-space codes differed based on whether one considered "unplugged" activities (which teach computing concepts without programming or computers) to be program-space or problem-space activities. After reviewing and discussing the

definitions of problem- and program-space design, the analysts decided that program-space design work did not necessarily imply writing code and that unplugged activities could include program-space design.

- Applications of program-space *Evaluation & Synthesis* codes differed based on whether one considered critiquing tradeoffs of software created by an external person or entity to be a design activity in itself. Based on design literature, analysts decided that any evaluation of code for fitness to a certain constraint should be considered program-space ES.
- Applications of the *Evaluation & Synthesis* codes during presentation-based activities (e.g., ECS's gallery walks) differed based on how conservative the one was with their inference of activities. Analysts decided to limit their inference of activities and apply codes based only on the text of activity descriptions and supporting materials.
- Applications of program-space *Iterative Improvement* codes in debugging-focused activities differed based on whether one considered "improvement" to require implementing changes or simply proposing them. After discussing the core design skill definitions, analysts decided that debugging activities should have students actually implement changes to count as program-space II codes.

The two analysts collaboratively updated their final results to address these conceptual discrepancies. These clarifications also served to refine the definitions of the design skills and types of design in computing presented in this article. The first author then analyzed the results for patterns and themes.

5.3 Results

5.3.1 Characterization of Cases. Table 1 summarizes the types of design skills that appeared in student activities by each unit of the three curricula.

As expected, the majority of Code.org CS Discoveries (CSD) activities included design: the analysts marked 100 of 150 activities as containing at least one skill of design. Of those 100, 52 only contained evidence of problem-space design skills, 37 contained only program-space design skills, and 11 contained both problem- and program-space design skills. In the Code.org CSD activities, problem- and program-space design practices were often tightly interleaved. As students created computational artifacts, they typically decided what they wanted to create and defined requirements for their artifact (problem-space design), then decided the most fit way to meet those constraints with programming or algorithmic constructs (program-space design), and finally implemented their plan. Frequently, the design work in both spaces was scaffolded by provided project guides that led students through activities like brainstorming, storyboarding, algorithm creation, and selection of programming constructs. Students often also justified their design rationale in both design spaces during peer review or formal presentations.

Design did surface in the AP CS A activities, much to the analysts' surprise. Notably, design practices in AP CS A were limited to the program-space: out of the 36 activities, 29 contained program-space design, and none contained problem-space design. In software engineering terminology, students were often asked to come up with strategies to meet requirements, but they were never asked to define those requirements themselves. Generally, the program-space design activities centered around generating algorithms that fit certain given constraints (*Creative Ideation*) or analyzing programs for correctness or equivalence (*Evaluation & Synthesis*). Students occasionally worked in pairs and communicated their program design rationale as well.

Of the 288 Exploring CS activities, the analysts identified 96 that contained design: 46 problem-space only, 43 program-space only, and 7 containing both problem- and program-space design skills. Similar to the Code.org curriculum, Exploring CS activities often interleaved problem-space

Table 1. The Kinds of Design Skills That Appeared in Study 2's Analyzed Computing Education Student Activities

	Problem-Space Design					Program-Space Design				
	UtC	*CI*	*ES*	*II*	*Comm*	*UtC*	*CI*	*ES*	*II*	*Comm*
Code.org CSD										
Problem Solving	✓	✓	✓	✓	✓	✓	✓			
Web Development	✓	✓				✓	✓	✓	✓	✓
Interactive Animation and Games	✓	✓	✓		✓	✓	✓	✓	✓	✓
The Design Process	✓	✓	✓	✓	✓	✓	✓		✓	✓
Data and Society	✓	✓	✓	✓	✓		✓		✓	
Physical Computing	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
AP CS A										
Primitive Types								✓	✓	
Using Objects							✓	✓		
Boolean Expressions and "if" Statements							✓	✓	✓	✓
Iteration							✓	✓		
Writing Classes								✓		
Array							✓		✓	✓
ArrayList						✓	✓	✓		
2D Arrays							✓	✓		
Inheritance							✓	✓		
Recursion								✓		✓
Exploring Computer Science										
Human Computer Interaction	✓	✓	✓		✓					
Problem Solving	✓						✓	✓		✓
Web Design	✓	✓	✓	✓	✓		✓	✓	✓	
Introduction to Programming	✓	✓	✓		✓	✓	✓	✓	✓	✓
Computing and Data Analysis	✓		✓	✓	✓					
Robotics	✓	✓	✓		✓		✓	✓	✓	✓

The leftmost column contains the names of units within each curricula. A checkmark (✓) indicates both analysts agreed that particular design skill appeared at least once in the unit's activities. Notably, the AP CS A curriculum activities did not seem to contain problem-space (nondisciplinary) design, though all kinds of program-space (disciplinary) design skills were represented.

design and program-space design. Peer review, critique, and feedback (in both design spaces) were especially common in Exploring CS activities. Students often justified their design rationale and implementation choices to at least their teammates, if not the larger class. Often, they were asked to synthesize problem-space-level feedback from users or peers into program-space design choices, then implement the results to incrementally improve their computational artifacts. Exploring CS activities also supported problem-space *Creative Ideation* very well by providing brainstorming guides where students wrote down multiple approaches to solving a problem with computational artifacts, selected one, and justified their selection before they implemented it.

5.3.2 Non-design Activities. In activities where students were not actively solving a problem, they did not practice design. Often, these took the form of simple identification exercises such as *"For each [provided code] segment, have students trace through the execution of a loop with smaller bounds to see what boundary cases are considered, and then use that information to determine the number of times each loop executes with the original bounds"* (AP CS A 4.5). Non-design activities might also involve students simply manipulating data without necessarily having a higher-level goal in mind, such as in *"Groups create bar and mosaic plots with the data they have collected and additional contextual data sets"* (ECS 5.8). In the case of this last activity, the goal of the lesson was to have students become familiar with different types of plots, not to have them use those plots for any analysis. Similarly, some activities simply asked students to learn about general programming constructs and did not require application of that knowledge to solve a particular problem. For instance, the Code.org CSD activity *"Web Lab: Intro to CSS"* (Code.org 2.10) led students through a guided tutorial on how CSS can be used to change the appearance of HTML websites. While students would later use their knowledge of CSS to style their own web pages, this specific activity did not ask students to make design decisions—only to implement CSS styles as directed.

5.3.3 Exclusively Problem-Space Design Activities. The two analysts found instances of student activities containing only problem-space (nondisciplinary) design in both the Code.org CSD and Exploring CS curricula. When students perform these activities, they are working to define the "what" and the "why" of their artifact: motivating its existence and defining high-level requirements for its implementation. Problem-space design activities appeared in every unit of the Code.org CSD and ECS curricula, with most instances occurring in user-centered design (Code.org), human-computer interaction, and data analysis (ECS) units. Below, we present some themes that arose from final analysis results and indicate which of the five core design skills (Section 2.2) the group of activities tended to correspond to.

Research to Motivate Artifact Creation (UtC and ES). One way in which problem-space design manifested in the activities was through information gathering used to situate whatever computational artifact students planned to create in real-world contexts. Sometimes this took the form of user research, as in activities from Code.org's user-centered design unit: *"Looking through a user's eyes"* and *"Responding to products"* activities (both Code.org 4.2) asked students to take the perspectives of different users to evaluate an existing product. After doing so, students identified opportunities for improvement in the existing products' designs. Another form this information gathering could take was market research and data analysis. For instance, in the activity *"Brainstorming app ideas"* (Code.org 4.9), students analyzed applications already on the market to see how they addressed certain problems, then came up with an app idea that would differ from already-existing solutions. At other times, students sifted through data to characterize problems, as in the activity *"Groups do statistical analysis with mean, median, maximum, and minimum using the data they have collected and additional contextual data sets"* (ECS 5.10). ECS groups later used this data analysis to inform their final unit projects.

Requirement Generation and Refinement (UtC, ES, and II). Problem-space design activities sometimes had students come up with and iterate on requirements for artifacts they planned to create, rather than specifying all requirements in the student materials. For instance, the *"Define"* activity in Code.org's lesson 4.3 had students perform a version of stakeholder analysis to identify target user populations' needs. Similarly, *"Identify evaluation criteria and work in groups to evaluate websites using the rubric"* (ECS 1.3) asked students to identify requirements for determining the credibility of websites, then evaluate various sites against those requirements. Students later used these initial sets of constraints to inform project requirements.

Prototyping Concepts and Interfaces (CI and ES). Both curricula in which problem-space design appeared favored a design-before-implementation mentality, often instantiated in storyboarding and prototyping activities. Problem-space prototypes were generally low fidelity, but all represented a concept for a final system. These systems did not necessarily have to be technological: for instance, in the *"Create a representation"* (Code.org 5.8) activity, students designed a paper punch card with categories of information they would use to represent their "perfect days." However, when the system was technological, storyboarding and prototyping activities often helped students envision user interfaces, such as in the *"Paper prototypes"* activity (Code.org 4.7), where students created and tested user flows of their apps by sketching screens on notecards. Sometimes, students were led to consider user perspectives other than their own when prototyping: the notes for the *"Create a storyboard for a multipage web site"* (ECS 3.8) activity suggest that teachers prompt their students to consider the many kinds of diverse users that might use their website, though students are not evaluated upon this requirement.

Managing Group Expectations and Decisions (Comm, CI, and ES). When doing group work, teams often practiced communication around problem-space design decisions and how their group would meet particular design requirements. Both the Code.org CSD and Exploring CS curricula contained some form of team management activity such as *"Groups discuss roles and responsibilities"* (ECS 5.3) and *"Team contract"* (Code.org 4.8). These kinds of activities focused on how the team would effectively work together to create a final computational artifact. Sometimes groups worked together to generate a single set of design requirements, such as in *"Building an Aluminum Boat"* (Code.org 1.1), where teams came up with different ways to create boats from aluminum foil that would float in water when weighed down with coins. Other times, teams had to come to a consensus on which requirements to prioritize and implement. For instance, teams came into the activity *"Groups work to merge individual data sets together"* (ECS 5.1) with data they had individually collected (with no particular constraints on how to categorize it) and were asked to merge the data sets together. In doing so, groups had to agree upon which dimensions of their disparate data sets to change so that the data in the final set ended up at the same granularity, which inherently involved making decisions about what data was important. Other times, team consensus building happened in response to feedback: using the results of *"User Testing"* (Code.org 4.11), teams discussed and prioritized feature implementations and bug fixes. Problem-space communication underlies all these activities and helps students to successfully create artifacts while working with teams.

Justifying Problem-Space Design Choices (Comm). Finally, when practicing problem-space design, students were often asked to justify why their solution fit a particular problem. Note that this is different than discussing *how* they implemented their solution (which would represent problem-space design justification). For example, in the *"Student teams present projects"* (ECS 1.2) activity, pairs of students worked together to explain their rationale for giving a particular user a specific computer hardware configuration (based on the fictitious person's typical usage patterns). During the *"Presentation prep"* and *"Presentations"* (both Code.org 4.16) activities, students wrote up and presented thorough design specification documents, including their problem motivation and framing, existing solutions and their shortcomings, examples of how their artifact addressed these shortcomings, and their user testing refinement process.

5.3.4 Exclusively Program-Space Design Activities. Instances of activities containing only program-space (disciplinary) design appeared in all three curricula. When students perform these activities, they determine the "how" of their artifact: given some problem-space constraints, how at the algorithmic level those constraints should be implemented. Program-space design activities occurred in every unit of the Code.org CSD and AP CS A curricula, though it was absent from two units of the Exploring CS curriculum (human-computer interaction, and computing and data analysis).

Understanding Programming Environment Affordances (UtC). Students often worked to understand the kinds of functionality they had available to them before they began to implement their artifacts. This is different than simply learning about programming language constructs (e.g., *for* loops or arrays), which are general concepts that underlie all programming practice. Instead, students exercising this kind of program-space design skill learn about unique features that are not necessarily transferable to other programming environments. For instance, in activities like *"Intro to Sprites"* (Code.org 3.6) and *"The Draw Loop"* (Code.org 3.7), students explored the functionality of Code.org's Game Lab platform. Later, they used their knowledge of Game Lab's affordances to create interactive animations and games. Similarly, in the *"AppLab exploration"* (Code.org 4.10) activity, students explored implemented apps in the AppLab environment in order to understand the functionality available to them. In the *"Pairs investigate features of Scratch and start name assignment"* (ECS 4.1) activity, students experimented with Scratch blocks to figure out what they could accomplish within the environment. Each of these activities is a form of the program-space skill of *Understanding the Context*, teaching students the affordances of their specific environments or IDEs.

Designing Algorithms and Program Behavior (CI and II). A common form of program-space design was devising algorithms or structuring programs to fit given constraints. Sometimes, the constraints were defined by the teaching materials and simply required iterative modification of a base program, requiring students to figure out the best way to meet those constraints. This occurred in activities like *"Provide students with a method [with some functionality]... ask students... to write a similar method that, given a student number as input, returns the name of a student from a String containing the first name of all students in the class, each separated by a space"* (AP CS A 4.1-4.4). At other times, requirements were defined for students, but they had to figure out how to meet them without a base example, as in *"Develop an Age program"* (ECS 4.9), where students created a program that responded to different numerical (age) inputs in various specified ways, and *"Making music"* (Code.org 6.11), where students used a physical circuit board to create audible buzz patterns. Notably, these kinds of activities did not necessarily include writing code: *"In groups, participate in the candy bar activity"* (ECS 2.2) had students come up with an algorithm to split a candy bar in the least amount of cuts. Students also practiced problem-space design when they created code architectures that fit a particular set of design requirements, as in the activity *"Given a class design problem that requires the use of multiple classes in an inheritance hierarchy, students identify the common attributes and behaviors among these classes and write these into a superclass..."* (AP CS A 9.2-9.4).

Program Analysis and Rnement (ES and II). Students often analyzed programs for correctness or equivalence. This kind of program-space evaluation activity was particularly prevalent in the AP CS A curriculum, making up the majority of the curriculum's design activities. In the AP CS A curriculum, students often had to compare actual output against expected output, then propose and implement fixes to align program behavior with requirements. For example, in *"Provide students with code that contains syntax errors. Ask students to identify and correct the errors... [and] have them verify their conclusion by using a compiler and an IDE that does not autocorrect errors"* (AP CS A 1.1), students identified and fixed syntax-level errors. Students also evaluated code against requirements at the semantic level, as in *"Provide students with several error-ridden code segments containing array traversals along with the expected output of each segment. Ask them to identify any errors that they see on paper and to suggest fixes to provide the expected output..."* (AP CS A 9.4). Debugging and refinement activities appeared throughout all three curricula, in activities like *"Web Lab: Smash those Bugs"* (Code.org 2.8), *"Enhance the variable example"* (ECS 4.9), and *"Test the robot frequently and refine program and hardware"* (ECS 6.10). Program analysis activities often also asked students to determine equivalence of statements (*"Provide students with a code segment that utilizes conditional statements and a compound Boolean expression, and ask them to choose an equivalent code segment that uses a nested conditional statement..."* (AP CS A 3.6)). Finally, students participated in program-space design evaluation when comparing different algorithms: in *"Model the tower building algorithm"* (ECS 2.6) and *"Groups participate in the various parts of the*

CS Unplugged: Lightest and Heaviest activity" (ECS 2.7), students evaluated the tradeoffs of using particular algorithms over others, then selected the best fit for their situation.

Justifying Implementation and Code-Level Teamwork (Comm and ES). When working in groups, students had to communicate with others about program design decisions in order to build functional programs. The Exploring CS curriculum almost always had students working in pairs or groups when creating software. In the curriculum's Intro to Programming unit, activities like *"Pairs complete Map Route Activity"* (ECS 4.4), *"Develop Social Media Quiz program"* (ECS 4.10), and *"Create a timer block with a parameter"* (ECS 4.14) all required students to work with at least one partner. In doing so, students often discussed and justified their program-space design decisions with peers. These kinds of discussions were explicitly scaffolded by project guides in the CSD curriculum (Code.org 3.22 and 6.16), where students collaboratively decided upon program behavior and generated pseudocode before they began implementing their solutions. Pair programming activities were prevalent in all three curricula (e.g., *"Have students use pair programming to solve an array-based free-response question..."* (AP CS A 6.4); *"Pair programming"* (Code.org 2.4)). Sometimes pair programming activities involved evaluation and critique, as in *"Provide students with the pseudocode to multiple recursive algorithms, and have students write the base case of the recursive methods and share it with their partner. The partner should then provide feedback, including any corrections or additions that may be needed"* (AP CS A 10.1). Notably, Code.org's CSD activities often embedded instruction about code style best practices into students' introductions to programming concepts. For instance, the activity *"Programming with variables"* (Code.org 3.5), students' first exposure to variables, instructed students on and had students adhere to style guidelines for variable naming and commenting. Subsequent activities encouraged students to keep up these practices in order to facilitate better within-team communication about program implementation choices. Finally, similar to the previously mentioned problem-space practice, students often had to justify their implementation choices, describing how their solution met (or enabled them to meet) requirements. During whole-class discussions or demo days (e.g., *"Participate in discussion of solutions"* (ECS 2.2), *"Groups present final projects"* (ECS 2.9)), students often described and defended their program-level design choices to peers.

5.3.5 Overlapping Problem- and Program-Space Design Activities. The two analysts identified a handful of student activities in which both problem- and program-space design overlapped in the Exploring CS and Code.org CSD curricula. Students tended to practice both kinds of design in tight iterative cycles in activities fitting this categorization, drilling down through requirement definition to algorithm design and finally implementation, then popping back up into a design space to either further refine or gain new perspectives on their computational artifacts. Activities that contained both problem- and program-space design appeared in five of the six Code.org CSD units, with the majority in *Interactive Animations and Games* and *Physical Computing*, and three of the six Exploring CS units, with the majority in *Intro to Programming*.

Understanding Existing Solutions at Multiple Levels (UtC and CI). Sometimes, students were asked to understand both the problem and program-level design choices inherent in existing solutions to a particular problem, both defining and ideating on requirements. For instance, at the beginning of the Physical Computing unit, Code.org students participated in *"Innovation research"* (Code.org 6.1). During this lesson, students researched existing physical computing devices and answered questions like "What problem does it solve?" (a problem-space design question) and "How do you interact with it?" and "How could you improve it?" (both program-space design questions). Later, they used these findings to inform their own robot creations. Similarly, ECS students practiced both kinds of design in *"Participate in Rock Paper Scissors discussion"* (ECS 4.11). In this activity, students first worked to define the program-space requirements for a game of rock paper scissors, then (with help from a partner) defined a program-space pseudocode algorithm to determine the winner of a given round of the game. Throughout both these activities, students practiced each kind of design.

Simultaneous Evaluation in Both Design Spaces (ES and Comm). Occasionally, when students evaluated each other's work, they critiqued both how well the artifact fit its requirements (program-space) and how well those requirements represented the problem at hand (problem-space). One example of this is found in a lesson where students created interactive greeting cards with animations: during the *"Peer review"* portion of the assignment (Code.org 3.14), students critiqued others' cards. In the program space, students evaluated the implementation of the card—ensuring that multiple properties were updated in the animation-rendering loop, that the code was properly modularized, and so on. During the same critique, they also evaluated how well the card met problem-space requirements, such as ensuring that it responded to multiple kinds of user input and suggesting ideas for future problem-space requirements. Similar dual critiques occurred in the ECS activity *"A member of each group will demonstrate and explain a program modification to their assigned gallery walk group"* (ECS 6.6).

Improving Artifacts by Implementing Feedback (II and ES). When implementing feedback from critiques and user testing, students practiced problem-space design when they synthesized the feedback into new requirements, and program-space design when they extended the functionality of their artifact to meet the new requirements. The activity *"Fixing bugs and adding features"* (Code.org 4.15) asked students to incrementally improve their prototypes after a round of user testing. Bug fixes almost certainly involved program-space design to patch the code, while new feature implementation likely included both defining requirements and writing code to meet them. Code.org students responded to user feedback in both design spaces in multiple activities throughout the curriculum (e.g., *"Prepare [to finalize an artifact]"* (Code.org 5.15), *"Iterate - revise prototypes"* (Code.org 6.16)).

Project Planning and Implementation (All Skills). Finally, the overlap of problem- and program-space design showed up most often in final project activities. Often, these projects occurred at the end of a unit and were intended to showcase students' learning as they worked in teams to define and motivate a problem, plan the design of a computational artifact to address the problem, then implement, test, and refine their solutions. For example, *"Develop and Create Your Own Adventure animation"* (ECS 4.5) asked students to practice a wide swath of design skills. In this activity, students worked in pairs to create a multilevel interactive adventure game. In the problem space, pairs storyboarded their adventure before implementing it, creatively generating requirements and discussing design rationale for their levels. In the program space, students attempted to meet these requirements by implementing event handlers for user input (exactly how they did this was left up to students to decide), evaluating and updating their code until it met the problem-space constraints, and communicating while pair programming. Most project planning activities also contained both forms of design, even if the actual implementation of the design was left to a later activity. For instance, the *"Unplugged: program planning"* activity (Code.org 6.9) asked students to create some sort of interactive game with physical computing hardware. The problem-space requirement definition process was left entirely up to student groups, along with any program-space ideations around how to respond to user input. When students implemented their plans, they iterated on their solutions until they felt like they were ready to test it with peers, at which point they facilitated user testing and improved their prototypes based on the results. Another project activity *"Use the planning document to plan the robot"* (ECS 6.11) had students go through a similar process to create an autonomous "search and rescue" explorer robot. Each of these project activities had students acting as both designers and developers as they moved through the full artifact creation process.

Table 2. Examples of Problem-Space (Nondisciplinary) and Program-Space (Disciplinary) Design Activities in Computing Education Contexts, Organized by the Core Design Skill They Correspond To

	Problem-Space Design	**Program-Space Design**
	Why does the world need this software? *What* should this software do (or enable users to do) in the world?	*How* should this program be structured? *How* should I implement this software's requirements most effectively?
Understanding the Context	• Requirement generation • User research • Market research • Exploratory data analysis	• Learning environment affordances • Understanding existing code • Decomposing problem into modules
Creative Ideation	• Storyboarding • Brainstorming possible solutions • Sketching and prototyping	• Creating algorithms • Writing pseudocode • Defining code architecture
Evaluation & Synthesis	• Using feedback to inform requirements • Critique • Evaluating solution against constraints	• Verifying correctness and equivalence • Analyzing implementation tradeoffs • Program behavior analysis • Code reviewing
Iterative Improvement	• Refining problem conception • Updating requirements	• Debugging • Refactoring code-level architecture • Extending existing code
Communication	• Presenting design rationale • Collaboratively defining requirements • Managing group roles/responsibilities	• Presenting implementation rationale • Adhering to style guidelines • Pair programming

Each of these activities appeared in at least one of the analyzed computing curricula. While we present these activities in two clean columns, we note that some activities may exist in the intersection of problem- and program-space design depending how they are implemented in a particular classroom or course (e.g. "Creating algorithms" might require students to draw both on implementation knowledge and knowledge about the world).

6 DISCUSSION AND CONCLUDING REMARKS

Our analysis in the previous sections suggests that there are design skills in many of our most widely disseminated computing education standards and the activities that stem from them. Table 2 summarizes some of the kinds of design-related activities in computing curricula, broken down by their alignment with the five core skills of design from Section 2.2 that served as our basis for analysis. The skills identified include both *disciplinary* forms of computing-specific design, such as devising and refining algorithms (referred to as program-space design), and skills from the broader discipline of design, which involve understanding and situating software in the world (referred to as problem-space design). While problem-space design activities tend to align more with most people's conception of general (i.e., not discipline-specific) design, program-space design activities seem closely linked to computing in a way that suggests they are more aligned with discipline-specific computing design. As evidenced by our Study 2 results, there were also a number of computing education activities that contain both kinds of design, either simultaneously or interleaved with each other.

As it stands, many of our K-12 computing education materials mask the presence of design in computing education by either simply ignoring it or by claiming many design practices and skills as computing proficiencies. However, continuing to do this might be detrimental to teaching and learning in a number of ways. Students learning computing through design-laden curricula may mistakenly interpret design practices as computing-specific skills. Though this is appropriate and authentic for program-space design practices, these students would likely fail to recognize the existence of problem-space design and distinguish it from computing. They might then be misled into pursuing careers in software engineering when their interests are more suited to user experience design or product management, for instance. Educators may also suffer from a lack of clear boundaries between design and computing, which complicates teaching. Prior work has shown that the pedagogical content knowledge (PCK) required to teach design is meaningfully different than the PCK required to teach computing [73, 110, 111], since PCK is domain specific [67, 127]. The PCK needed to teach problem- and program-space design might also differ in meaningful ways. Any effort teachers spend figuring out how to teach computing skills may not transfer if the skills they teach are, in fact, design.

If the status quo is problematic in these ways, how might this overlap be addressed? One possibility would be to try and remove nondisciplinary problem-space design (the "what" and the "why") from computing education, while retaining disciplinary program-space design (the "how"). The result of this would be a set of competencies that are solely concerned with computing concepts and the design skills required to architect, plan, construct, and refine programs. Such work, however, would be difficult, time-consuming, and likely even counterproductive to the goal of training well-rounded computing students. As Section 5 demonstrated, many nominally computing-related learning objectives *only* concerned problem-space design, while others contained both problem- and program-space skills. Pragmatically, we probably should not spend time and resources to undo the work that has already gone into creating and defining computing curricula.

Even if we decided to move ahead with the work of fully separating design and computing in educational contexts, prior work suggests that aspects of problem-space design are important for understanding how computing connects to the world. For instance, those who create software *should* understand that there are differing software usage styles, and that the software they create will almost certainly be used by stakeholders who are not like them [56, 137]. Computing literacy *should* also extend to understanding the ethics and justice issues inherent to computing and the impact that computational approaches can have in society [27, 123, 138]. In the same vein as the Chilana et al. *conversational programmers* [26], who learn to code not to become developers, but so that they can communicate better with developers, computing education might aim to educate computing students who are *conversational designers*, able to communicate effectively with designers in their teams and organizations.

Problem-space design skills are also often used in computing education to motivate students to engage with computing material (see Section 1). Removing the *what* and *why* from computing education and leaving only the *how* might harm interest and engagement in computing education. While demand for all occupations is projected to grow 7% by 2026, the demand for computing-related occupations is projected to grow by 19% [46]. If we are to have a chance at meeting the demand for a computing-literate workforce, we cannot afford to dissuade learners early on by focusing solely on the design of programs and not on their use in the broader world. Eliminating problem-space design might even lower the overall diversity of the computing field: prior work suggests that situating computing within creative or design-based topics can encourage women [51, 83, 114] and underrepresented ethnic groups [38] to engage with computing where they otherwise might not.

If we can't ignore design's presence in computing, and since it seems counterproductive to try and excise non-disciplinary problem-space design from computing materials, another option is to embrace the entanglement of design and computing and attempt to better understand the two fields' unique intersection through further research. Given the evidence presented in this article and the arguments presented in prior work [144], there is a meaningful interaction between design and computing education that would be well served by further scrutiny. As mentioned in

Section 3, other fields that overlap with design have found success in this approach. For example, by recognizing design as a distinct yet necessary part of K-12 engineering education, Crismond and Adams created a framework of "misconceptions, learning trajectories, instructional goals and teaching strategies that instructors need to know to teach engineering design effectively" [35]. In architecture education, one of the earliest disciplines to recognize and study its overlap with design education, Lawson studied how students acquire design skills in order to suggest improvements to computer-aided design (CAD) systems [94]. A branch of research specifically studying the intersection of computing education with design could produce similarly fruitful and interesting results.

Toward this end, future work should build upon the distinctions, intersections, and opportunities identified in this article to further investigate how to effectively integrate design and computing education. Such work might reveal how design and computing interact not only in the program space but also in the largely unexplored problem-space design of computing artifacts. For example, studies might begin to reveal how difficulties that students face with debugging (part of program-space design) might lead students to change their requirements (part of problem-space design), or how constraints discovered while designing an algorithm (program-space design) reveal insights about a program's value in the world (problem-space). Investigating design and computing integration would also enable us to draw upon both fields' theoretical and pedagogical bases. For instance, *Iterative Improvement*, a design skill described in Section 2.2, sits squarely in this intersection. Researchers might ask questions like *What are the most effective pedagogical strategies to teach iteration and the notion of productive failure in computing contexts?* Design education pedagogy could inform particular approaches to the question, while computing education work might suggest adaptations and pitfalls specific to computing contexts, leading to unique results that inform about the overlap of design and computing.

Since the intersection of computing education and design is not yet well explored, understanding starts with the CER community and future exploratory work. Embracing a research area that recognizes the nuanced role of design in computing opens up many new exciting avenues of investigation. For example, what are concrete strategies to address design-related learning barriers of students in computing education concepts (such as those recently described in our previous work [111])? For students in computing classes who decide they are more interested in design than computing, how should we effectively route them to design disciplines? How can learning be structured to isolate program-space design from problem-space design? What are effective ways to teach design-related skills like perspective taking, communication, and interpreting user feedback? And what kinds of pedagogical content knowledge do educators need to effectively teach design in computing contexts? There also fascinating questions about learner identity. How can explicitly naming design's presence in computing and teaching design skills change students' perceptions of the field and their roles within it? Can embracing design in computing draw more students to the field who do not fit into traditional computer science stereotypes, as has been shown in some prior work [38, 51, 114]? By answering these questions, we may be able to more effectively leverage design to effectively and equitably engage diverse learners.

One of the first places the above kinds of research might appear is in curriculum and instructional design. Recognizing and identifying the presence of problem- and program-space design in computing might require the reframing of existing materials as well as creation of new materials. For example, existing materials that already contain design skills (such as those studied in Sections 4 and 5) may require a framing update. Resources might clearly distinguish and name both design and computing skills within lesson plans, so that the teachers using them know how to prepare. A potential example of this would be something similar to the tags used in Code.org's curricula (which inform the teacher at a glance if the lesson's material requires a physical computer, uses the Game Lab platform, etc.). In K-12 education, materials might even provide explicit instruction on the role of design within computing and the two disciplines' overlapping natures. For instance, content creators could include more information about fields in the intersection of design and computing and explicitly name them (e.g., "If you liked this lesson, consider careers in interaction design"). Some of this work has already begun. Since the time of our analysis in Sections 4 and 5 (early to mid-2019) and the time of publication, Code.org's CS Discoveries curriculum has already modified its unit on the design process to explicitly integrate more information about user-centered design best practices and empathy with diverse kinds of users, among other updates.

These shifts in curriculum come with their own changes to teacher preparation. Pre-service and professional development on K-12 computing education might need to explicitly distinguish between computing and design, sharing the concepts in this article and revealing how they play out in classrooms. Teachers may need to learn to discriminate between computing skills and design skills (some initial work on this topic is described in Section 3.4). They may also need to recognize and respect the fact that students can have differing motivations for learning computing, some of which may lead to design professions rather than computing ones. Teachers may also need to understand that a student being proficient at computing concepts does not mean that they will also be proficient at design concepts (and vice versa). Higher education faculty may also need to embrace these concepts, suggesting a need for structural change. For example, embracing the role of design might involve making the field of human-computer interaction more central in computer science curricula (as opposed to its current role as a popular elective). It might also involve changes in how the rest of computing is taught, leveraging design's studio approaches [25, 41, 119], ideas of productive failure and iteration (e.g. [11, 117]), and approaches to teaching [39, 77, 93] and assessing [41, 44] creativity and prototyping processes. In these ways, teachers at all levels can use design to broaden the definition of computing while simultaneously respecting its role as a distinct discipline.

In addition to teachers, there are many opportunities for school administrators to embrace design more systematically. Some secondary schools in the United States, for example, already teach graphic design courses, but these courses are often siloed to arts departments and disconnected from computing education. K-12 administrators could support leveraging both design and computing in these respective learning contexts. Such administrative changes might also require higher-level policy change that provides more comprehensive definitions of computing that recognize the distinct role of design.

As with any work that suggests directions for the future of computing education, there are some intrinsic limitations to what can be realized in practice. First, primary and secondary curricula coverage for any area is necessarily constrained by limitations of time, and computing education is no exception to this. Deciding which computing topics to prioritize over others in order to better integrate an understanding of computing design becomes a question of curriculum design. Our results uncovered and characterized the presence of design *within* established computing curricula, suggesting that implementing the changes discussed above might be less of a full excision of particular topics and more of a change to the framings and approaches we use to better represent the design that is already there. The goal of this article is not to provide a definitive answer as to which computing topics should be cut in future curriculum, but rather to discuss and provide insights into what computing education might gain if it were to prioritize making the presence of design more explicit. In this way, our contribution is similar to recent work that suggests improving computing education by approaching it from perspectives like sociopolitical identity [139] or culture [38]. Another limitation is that K-12 educators and administrators will have questions about what the results presented in this article imply for their particular contexts and courses, some of which we cannot yet answer definitively. As discussed in Section 3.4, teaching and learning design within computing contexts is difficult for a number of reasons, and while recent work has begun to explore this space, much of it focuses on postsecondary education. We still need to develop pre-service preparation programs, useful materials, and concrete pedagogical strategies for teaching computing design topics. These areas provide directions for future research on design and computing's overlap in primary and secondary educational contexts.

While there are challenging choices to make about computing curriculum, acting on these implications in some way is critical to help students gain a holistic understanding of computing in the broader world. Understanding the ways that design manifests in computing contexts—in decisions about how software fits into the world and how to implement real-world requirements—enables computing students to better recognize and respond to pervasive embedded biases in software [12, 34, 88], whether they are the software's creator or simply a user who experiences its effects. Because even basic changes in deploying computing education across the globe are already challenging, many of these implications may seem intractably hard. On the other hand, changing computing education infrastructure, such as curricula, programs, and policy, will never be easier than it is now, where there is little inertia, much momentum, and immense opportunities for literacy.

ACKNOWLEDGMENTS

We thank our third analyst, Aubree Ball, whose perspectives on design helped us better understand the discipline of design and how it interacts with computing.

REFERENCES

[1] Lucas F. Abreu, Glivia A. R. Barbosa, Ismael S. Silva, and Natalia S. Santos. 2016. Characterizing software requirements elicitation processes: A systematic literature review. In *Proceedings of the XII Brazilian Symposium on Information Systems: Information Systems in the Cloud Computing Era - Volume 1 (SBSI'16)*. Brazilian Computer Society, Porto Alegre, Brazil, 26:192–26:199. http://dl.acm.org/citation.cfm?id=3021955.3021988.

[2] Piotr D. Adamczyk and Michael B. Twidale. 2007. Supporting multidisciplinary collaboration: Requirements from novel HCI education. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'07)*. ACM Press, San Jose, California, 1073. DOI: https://doi.org/10.1145/1240624.1240787

[3] Robin S. Adams, Jennifer Turns, and Cynthia J. Atman. 2003. Educating effective engineering designers: The role of reflective practice. *Design Studies* 24, 3 (May 2003), 275–294. DOI: https://doi.org/10.1016/S0142-694X(02)00056-X

[4] Christopher Alexander. 1964. Notes on the synthesis of form. *Harvard Graduate School of Design* (Jan. 1964).

[5] L. Bruce Archer. 1965. *Systematic Method for Designers.* Council of Industrial Design.

[6] Henrik Artman and Mattias Arvola. 2008. Studio life: The construction of digital design competence. *Nordic Journal of Digital Literacy* 3, 2 (2008), 78–96.

[7] Owen Astrachan and Amy Briggs. 2012. The CS principles project. *ACM Inroads* 3, 2 (2012), 38–42.

[8] Cynthia J. Atman, Robin S. Adams, Monica E. Cardella, Jennifer Turns, Susan Mosborg, and Jason Saleem. 2007. Engineering design processes: A comparison of students and expert practitioners. *Journal of Engineering Education* 96, 4 (2007), 359–379. DOI: https://doi.org/10.1002/j.2168-9830.2007.tb00945.x

[9] Cynthia J. Atman, Justin R. Chimka, Karen M. Bursic, and Heather L. Nachtmann. 1999. A comparison of freshman and senior engineering design processes. *Design Studies* 20, 2 (March 1999), 131–152. DOI: https://doi.org/10.1016/S0142-694X(98)00031-3

[10] Lecia Jane Barker, Kathy Garvin-Doxas, and Michele Jackson. 2002. Defensive climate in the computer science classroom. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education (SIGCSE'02)*. ACM, New York, NY, 43–47. DOI: https://doi.org/10.1145/563340.563354

[11] R. J. Barnes, D. C. Gause, and E. C. Way. 2008. Teaching the unknown and the unknowable in requirements engineering education. In *2008 Requirements Engineering Education and Training*. 30–37. DOI: https://doi.org/10.1109/REET.2008.6

[12] Ruha Benjamin. 2019. Race after technology: Abolitionist tools for the new Jim code. *Social Forces* 98, 4 (2019), 1–3.

[13] Barry W. Boehm. 1988. A spiral model of software development and enhancement. *Computer* 5 (1988), 61–72.

[14] John D. Bransford, Ann L. Brown, Rodney R. Cocking, Bransford, Brown, and Cocking. 2000. *How People Learn.* Vol. 1. Washington, DC: National Academy Press.

[15] Andrea Branzi. 1986. We are the primitives. *Design Issues* 3, 1 (1986), 23–27. DOI: https://doi.org/10.2307/1571638

[16] Robin Braun, Wayne Brookes, Roger Hadgraft, and Zenon Chaczko. 2019. Assessment design for studio-based learning. In *Proceedings of the 21st Australasian Computing Education Conference (ACE'19)*. ACM, New York, NY, 106–111. DOI: https://doi.org/10.1145/3286960.3286973

[17] Amy Bruckman, Maureen Biggers, Barbara Ericson, Tom McKlin, Jill Dimond, Betsy DiSalvo, Mike Hewner, Lijun Ni, and Sarita Yardi. 2009. "Georgia computes!": Improving the computing education pipeline. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE'09)*. ACM, New York, NY, 86–90. DOI: https://doi.org/10.1145/1508865.1508899

[18] Louis L. Bucciarelli. 1994. *Designing Engineers*. MIT Press.

[19] Richard Buchanan. 1992. Wicked problems in design thinking. *Design Issues* 8, 2 (1992), 5–21. DOI: https://doi.org/10.2307/1511637

[20] Sheryl Burgstahler. 2011. Universal design: Implications for computing education. *Transactions on Computing Education (TOCE)* 11, 3 (Oct. 2011), 19:1–19:17. DOI: https://doi.org/10.1145/2037276.2037283

[21] Kenneth Burke. 1941. Four master tropes. *Kenyon Review* 3, 4 (1941), 421–438. https://www.jstor.org/stable/4332286

[22] Maria Camacho. 2016. David Kelley: From design to design thinking at stanford and IDEO. *She Ji: The Journal of Design, Economics, and Innovation* 2, 1 (2016), 88–101. DOI: https://doi.org/10.1016/j.sheji.2016.01.009

[23] Sharon M. Carver, Richard Lehrer, Tim Connell, and Julie Erickson. 1992. Learning by hypermedia design: Issues of assessment and implementation. *Educational Psychologist* 27, 3 (1992), 385–404.

[24] LaVar J. Charleston. 2012. A qualitative investigation of African Americans' decision to pursue computing science degrees: Implications for cultivating career choice and aspiration. *Journal of Diversity in Higher Education* 5, 4 (2012), 222–243. DOI: https://doi.org/10.1037/a0028918

[25] Alain J. F. Chiaradia, Louie Sieh, and Frances Plimmer. 2017. Values in urban design: A design studio teaching approach. *Design Studies* 49 (March 2017), 66–100. DOI: https://doi.org/10.1016/j.destud.2016.10.002

[26] Parmit K. Chilana, Rishabh Singh, and Philip J. Guo. 2016. Understanding conversational programmers: A perspective from the software industry. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI'16)*. ACM Press, Santa Clara, CA, 1462–1472. DOI: https://doi.org/10.1145/2858036.2858323

[27] Elizabeth F. Churchill, Anne Bowser, and Jennifer Preece. 2013. Teaching and learning human-computer interaction: Past, present, and future. *Interactions* 20, 2 (March 2013), 44–53. DOI: https://doi.org/10.1145/2427076.2427086

[28] Code.org. 2018. CS Discoveries Curriculum Guide 2018-2019. Retrieved January 14, 2019, from https://curriculum.code.org/csd-18/.

[29] Code.org. 2019. *The State of K-12 Computer Science*. Technical Report. Code.org. 24 pages. https://code.org/files/Code.org-Annual-Report-2019.pdf.

[30] The College Board. 2016. AP Computer Science Principles: The Course. Retrieved January 14, 2019, from https://apcentral.collegeboard.org/courses/ap-computer-science-principles/course?course=ap-computer-science-principles.

[31] The College Board. 2019. AP Computer Science A: The Course. Retrieved June 26, 2019, from https://apcentral.collegeboard.org/courses/ap-computer-science-a/course?course=ap-computer-science-a.

[32] Computer Science Teachers Association. 2017. CSTA K-12 Computer Science Standards, Revised 2017. Retrieved January 14, 2019, from, http://www.csteachers.org/standards.

[33] Steve Cooper, Shuchi Grover, Mark Guzdial, and Beth Simon. 2014. A future for computing education research. *Communications of the ACM* 57, 11 (Oct. 2014), 34–36. DOI: https://doi.org/10.1145/2668899

[34] Sasha Costanza-Chock. 2020. *Design Justice: Community-led Practices to Build the Worlds We Need*. MIT Press.

[35] David P. Crismond and Robin S. Adams. 2012. The informed design teaching and learning matrix. *Journal of Engineering Education* 101, 4 (2012), 738–797. DOI: https://doi.org/10.1002/j.2168-9830.2012.tb01127.x

[36] Nigel Cross. 1982. Designerly ways of knowing. *Design Studies* 3, 4 (Oct. 1982), 221–227. DOI: https://doi.org/10.1016/0142-694X(82)90040-0

[37] Ethan Danahy, Eric Wang, Jay Brockman, Adam Carberry, Ben Shapiro, and Chris B. Rogers. 2014. LEGO-based robotics in higher education: 15 years of student creativity. *International Journal of Advanced Robotic Systems* 11, 2 (Feb. 2014), 27. DOI: https://doi.org/10.5772/58249

[38] James Davis, Michael Lachney, Zoe Zatz, William Babbitt, and Ron Eglash. 2019. A cultural computing curriculum. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE'19)*. ACM, New York, NY, 1171–1175. DOI: https://doi.org/10.1145/3287324.3287439

[39] Elise Deitrick, Brian T. O'Connell, and R. Benjamin Shapiro. 2014. The discourse of creative problem solving in childhood engineering education. In *Learning and Becoming in Practice: The International Conference of the Learning Sciences (ICLS'14), Vol. 1.* International Society of the Learning Sciences, 591–598.

[40] Elise Deitrick, R. Benjamin Shapiro, Matthew P. Ahrens, Rebecca Fiebrink, Paul D. Lehrman, and Saad Farooq. 2015. Using distributed cognition theory to analyze collaborative computer science learning. In *Proceedings of the 11th*

Annual International Conference on International Computing Education Research (ICER'15). ACM, New York, NY, 51–60. DOI: https://doi.org/10.1145/2787622.2787715

[41] Halime Demirkan and Yasemin Afacan. 2012. Assessing creativity in design education: Analysis of creativity factors in the first-year design studio. *Design Studies* 33, 3 (May 2012), 262–278. DOI: https://doi.org/10.1016/j.destud.2011.11.005

[42] Carl DiSalvo. 2012. *Adversarial Design.* MIT Press.

[43] Kees Dorst and Nigel Cross. 2001. Creativity in the design process: Co-evolution of problem-solution. *Design Studies* 22, 5 (Sept. 2001), 425–437. DOI: https://doi.org/10.1016/S0142-694X(01)00009-6

[44] Steven P. Dow, Alana Glassco, Jonathan Kass, Melissa Schwarz, Daniel L. Schwartz, and Scott R. Klemmer. 2010. Parallel prototyping leads to better design results, more divergence, and increased self-efficacy. *ACM Transactions on Computer-Human Interaction* 17, 4 (Dec. 2010), 18:1–18:24. DOI: https://doi.org/10.1145/1879831.1879836

[45] Hugh Dubberly. 2008. *How Do You Design? A Compendium of Models.* Dubberly Design Office.

[46] Wendy DuBow and Allison-Scott Pruitt. 2019. NCWIT Scorecard: The Status of Women in Computing [2019 Update]. www.ncwit.org/scorecard.

[47] Caitlin Duncan and Tim Bell. 2015. A pilot computer science and programming course for primary school students. In *Proceedings of the Workshop in Primary and Secondary Computing Education (WiPSCE'15).* ACM, New York, NY, 39–48. DOI: https://doi.org/10.1145/2818314.2818328

[48] Anthony Dunne. 2006. *Hertzian Tales: Electronic Products, Aesthetic Experience, and Critical Design.* MIT Press.

[49] Alistair D. N. Edwards, Peter Wright, and Helen Petrie. 2006. HCI education: We are failing - why? In *Proceedings of HCI Educators Workshop 2006.* 23–24.

[50] Exploring Computer Science. 2015. ECS Curriculum: Requests & Downloads. Retrieved June 26, 2019, from http://www.exploringcs.org/curriculum.

[51] Allan Fisher and Jane Margolis. 2002. Unlocking the clubhouse: The Carnegie Mellon experience. *SIGCSE Bulletin* 34, 2 (June 2002), 79–83. DOI: https://doi.org/10.1145/543812.543836

[52] Batya Friedman and David Hendry. 2019. *Value Sensitive Design: Shaping Technology with Moral Imagination.* MIT Press.

[53] Carol Frieze, Jeria L. Quesenberry, Elizabeth Kemp, and Anthony Velázquez. 2012. Diversity or difference? New research supports the case for a cultural perspective on women in computing. *Journal of Science Education and Technology* 21, 4 (Aug. 2012), 423–439. DOI: https://doi.org/10.1007/s10956-011-9335-y

[54] Jeffrey E. Froyd, Phillip C. Wankat, and Karl A. Smith. 2012. Five major shifts in 100 years of engineering education. *Proceedings of the IEEE* 100, Special Centennial Issue (2012), 1344–1360.

[55] Daniel D. Garcia, Valerie Barr, Mark Guzdial, and David J. Malan. 2013. Rediscovering the passion, beauty, joy, and awe: Making computing fun again, Part 6. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE'13).* ACM, New York, NY, 379–380. DOI: https://doi.org/10.1145/2445196.2445308

[56] Andrea Alessandro Gasparini. 2015. Perspective and use of empathy in design thinking. In *ACHI, the Eighth International Conference on Advances in Computer-Human Interactions.* 49–54.

[57] Ashok K. Goel, Swaroop Vattam, Bryan Wiltgen, and Michael Helms. 2012. Cognitive, collaborative, conceptual and creative - Four characteristics of the next generation of knowledge-based CAD systems: A study in biologically inspired design. *Computer-Aided Design* 44, 10 (Oct. 2012), 879–900. DOI: https://doi.org/10.1016/j.cad.2011.03.010

[58] Vinod Goel and Peter Pirolli. 1992. The structure of design problem spaces. *Cognitive Science* 16, 3 (July 1992), 395–429. DOI: https://doi.org/10.1016/0364-0213(92)90038-V

[59] Gabriela Goldschmidt and Paul A. Rodgers. 2013. The design thinking approaches of three different groups of designers based on self-reports. *Design Studies* 34, 4 (July 2013), 454–471. DOI: https://doi.org/10.1016/j.destud.2013.01.004

[60] Gabriela Goldschmidt and Anat Litan Sever. 2011. Inspiring design ideas with texts. *Design Studies* 32, 2 (March 2011), 139–155. DOI: https://doi.org/10.1016/j.destud.2010.09.006

[61] Sukeshini Grandhi. 2015. Educating ourselves on HCI education. *Interactions* 22, 6 (Oct. 2015), 69–71. DOI: https://doi.org/10.1145/2834811

[62] Mark Guzdial. 2017. Balancing teaching CS efficiently with motivating students. *Communications of the ACM* 60, 6 (May 2017), 10–11. DOI: https://doi.org/10.1145/3077227

[63] Rich Halstead-Nussloch and Han Reichgelt. 2013. Teaching HCI in a "crowded" computing curriculum. *Journal of Computing Sciences in Colleges* 29, 2 (Dec. 2013), 184–190. http://dl.acm.org/citation.cfm?id=2535418.2535447.

[64] David Hammer and Leema K. Berland. 2014. Confusing claims for data: A critique of common practices for presenting qualitative research on learning. *Journal of the Learning Sciences* 23, 1 (2014), 37–46.

[65] Martyn Hammersley and Paul Atkinson. 2007. *Ethnography: Principles in Practice.* Routledge.

[66] Steve Harrison and Deborah Tatar. 2011. On methods. *Interactions* 18, 2 (March 2011), 10–11. DOI: https://doi.org/10.1145/1925820.1925823

[67] Heather C. Hill, Deborah Loewenberg Ball, and Stephen Schilling. 2008. Unpacking pedagogical content knowledge: Conceptualizing and measuring teachers' topic-specific knowledge of students. *Journal for Research in Mathematics Education* 39 (2008), 372–400. DOI: https://doi.org/10.1145/3025453.3025609

[68] Cindy E. Hmelo, Douglas L. Holton, and Janet L. Kolodner. 2000. Designing to learn about complex systems. *Journal of the Learning Sciences* 9, 3 (July 2000), 247–298. DOI: https://doi.org/10.1207/S15327809JLS0903_2

[69] Cindy E. Hmelo-Silver. 2004. Problem-based learning: What and how do students learn? *Educational Psychology Review* 16, 3 (Sept. 2004), 235–266. DOI: https://doi.org/10.1023/B:EDPR.0000034022.16470.f3

[70] Cindy E. Hmelo-Silver and Merav Green Pfeffer. 2004. Comparing expert and novice understanding of a complex system from the perspective of structures, behaviors, and functions. *Cognitive Science* 28, 1 (2004), 127–138.

[71] Beryl Hoffman, Ralph Morelli, and Jennifer Rosato. 2019. Student engagement is key to broadening participation in CS. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE'19)*. ACM, New York, NY, 1123–1129. DOI: https://doi.org/10.1145/3287324.3287438

[72] Chenglie Hu. 2016. Can students design software?: The answer is more complex than you think. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE'16)*. ACM, New York, NY, 199–204. DOI: https://doi.org/10.1145/2839509.2844563

[73] Peter Hubwieser, Johannes Magenheim, Andreas Mühling, and Alexander Ruf. 2013. Towards a conceptualization of pedagogical content knowledge for computer science. In *Proceedings of the 9th Annual International ACM Conference on International Computing Education Research (ICER'13)*. ACM, New York, NY, 1–8. DOI: https://doi.org/10.1145/2493394.2493395

[74] C. D. Hundhausen, D. Fairbrother, and M. Petre. 2012. An empirical study of the "prototype walkthrough": A studio-based activity for HCI education. *ACM Transactions on Computer-Human Interaction (TOCHI)* 19, 4 (Dec. 2012), 26:1–26:36. DOI: https://doi.org/10.1145/2395131.2395133

[75] Hengameh Irandoust. 2006. The logic of critique. *Argumentation* 20, 2 (Oct. 2006), 133–148. DOI: https://doi.org/10.1007/s10503-006-9012-0

[76] Terry Irwin. 2015. Transition design: A proposal for a new area of design practice, study, and research. *Design and Culture* 7, 2 (April 2015), 229–246. DOI: https://doi.org/10.1080/17547075.2015.1051829

[77] Karl K. Jeffries. 2007. Diagnosing the creativity of designers: Individual feedback within mass higher education. *Design Studies* 28, 5 (Sept. 2007), 485–497. DOI: https://doi.org/10.1016/j.destud.2007.04.002

[78] Bobbie Johnson. 2010. Privacy no longer a social norm, says Facebook founder. *The Guardian* (Jan. 2010). https://www.theguardian.com/technology/2010/jan/11/facebook-privacy.

[79] John Christopher Jones. 1970. *Design Methods: Seeds of Human Futures*. Wiley-Interscience.

[80] Y. B. Kafai, M. L. Franke, C. C. Ching, and J. C. Shih. 1998. Game design as an interactive learning environment for fostering students' and teachers' mathematical inquiry. *International Journal of Computers for Mathematical Learning* 3, 2 (May 1998), 149–184. DOI: https://doi.org/10.1023/A:1009777905226

[81] Yasmin B. Kafai. 2006. Playing and making games for learning: Instructionist and constructionist perspectives for game studies. *Games and Culture* 1, 1 (Jan. 2006), 36–40. DOI: https://doi.org/10.1177/1555412005281767

[82] Yasmin B. Kafai. 2012. *Minds in Play: Computer Game Design as a Context for Children's Learning*. Routledge. DOI: https://doi.org/10.4324/9780203052914

[83] Yasmin B. Kafai, Eunkyoung Lee, Kristin Searle, Deborah Fields, Eliot Kaplan, and Debora Lui. 2014. A crafts-oriented approach to computing in high school: Introducing computational concepts, practices, and perspectives with electronic textiles. *Transactions on Computing Education (TOCE)* 14, 1 (March 2014), 1:1–1:20. DOI: https://doi.org/10.1145/2576874

[84] Gyorgy Kepes. 1965. Education of vision. (1965).

[85] Dimitris Kiritsis. 2011. Closed-loop PLM for intelligent products in the era of the Internet of Things. *Computer-Aided Design* 43, 5 (May 2011), 479–501. DOI: https://doi.org/10.1016/j.cad.2010.03.002

[86] Amy J. Ko. 2017. A three-year participant observation of software startup software evolution. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. IEEE Press, 3–12. DOI: https://doi.org/10.1109/ICSE-SEIP.2017.29

[87] Amy J. Ko and Parmit K. Chilana. 2011. Design, discussion, and dissent in open bug reports. In *Proceedings of the 2011 iConference*. ACM, 106–113. DOI: https://doi.org/10.1145/1940761.1940776

[88] Amy J. Ko, Alannah Oleson, Neil Ryan, Yim Register, Benjamin Xie, Mina Tari, Matt Davidson, Stefania Druga, Dastyni Loksa, and Greg Nelson. 2020. It's time for more critical CS education. *Communications of the ACM (CACM)* 63, 11 (Nov. 2020), 31–33. DOI: 10.1145/3424000

[89] Matthew J. Koehler and Richard Lehrer. 1998. Designing a hypermedia tool for learning about children's mathematical cognition. *Journal of Educational Computing Research* 18, 2 (1998), 123–145.

[90] Ron Kohavi, Alex Deng, Brian Frasca, Toby Walker, Ya Xu, and Nils Pohlmann. 2013. Online controlled experiments at large scale. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'13)*. ACM, New York, NY, 1168–1176. DOI: https://doi.org/10.1145/2487575.2488217

[91] Janet L. Kolodner, Paul J. Camp, David Crismond, Barbara Fasse, Jackie Gray, Jennifer Holbrook, Sadhana Puntambekar, and Mike Ryan. 2003. Problem-based learning meets case-based reasoning in the middle-school science classroom: Putting learning by design(tm) into practice. *Journal of the Learning Sciences* 12, 4 (Oct. 2003), 495–547. DOI: https://doi.org/10.1207/S15327809JLS1204_2

[92] Janet L. Kolodner, David Crismond, Jackie Gray, Jennifer Holbrook, and Sadhana Puntambekar. 1998. Learning by design from theory to practice. In *Proceedings of the International Conference of the Learning Sciences*, Vol. 98. 16–22.

[93] Chinmay E. Kulkarni, Richard Socher, Michael S. Bernstein, and Scott R. Klemmer. 2014. Scaling short-answer grading by combining peer assessment with algorithmic scoring. In *Proceedings of the 1st ACM Conference on Learning @ Scale Conference (L@S'14)*. ACM, New York, NY, 99–108. DOI: https://doi.org/10.1145/2556325.2566238

[94] Bryan Lawson. 1979. Cognitive strategies in architectural design. *ERGONOMICS* 22 (Jan. 1979), 59–68. DOI: https://doi.org/10.1080/00140137908924589

[95] Richard Lehrer. 2013. Authors of knowledge: Patterns of hypermedia design. In *Computers as Cognitive Tools*. Routledge, 205–236.

[96] Gay Lemons, Adam Carberry, Chris Swan, Linda Jarvin, and Chris Rogers. 2010. The benefits of model building in teaching engineering design. *Design Studies* 31, 3 (May 2010), 288–309. DOI: https://doi.org/10.1016/j.destud.2010.02.001

[97] Sarah Lewthwaite and David Sloan. 2016. Exploring pedagogical culture for accessibility education in computing science. In *Proceedings of the 13th Web for All Conference (W4A'16)*. ACM, New York, NY, 3:1–3:4. DOI: https://doi.org/10.1145/2899475.2899490

[98] Paul Luo Li, Amy J. Ko, and Andrew Begel. 2017. Cross-disciplinary perspectives on collaborations with software engineers. In *IEEE/ACM 10th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE'17)*. IEEE, 2–8. DOI: https://doi.org/10.1109/CHASE.2017.3

[99] Min Liu. 1998. The effect of hypermedia authoring on elementary school students' creative thinking. *Journal of Educational Computing Research* 19, 1 (1998), 27–51.

[100] Min Liu. 2003. Enhancing learners' cognitive skills through multimedia design. *Interactive Learning Environments* 11, 1 (2003), 23–39.

[101] N. Mangano, T. D. LaToza, M. Petre, and A. van der Hoek. 2015. How software designers interact with sketches at the whiteboard. *IEEE Transactions on Software Engineering* 41, 2 (Feb. 2015), 135–156. DOI: https://doi.org/10.1109/TSE.2014.2362924

[102] Katherine McCoy. 2000. Information and persuasion: Rivals or partners? *Design Issues* 16, 3 (Sept. 2000), 80–83. DOI: https://doi.org/10.1162/07479360052053342

[103] D. Scott McCrickard, C. M. Chewar, and Jacob Somervell. 2004. Design, science, and engineering topics?: Teaching HCI with a unified method. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'04)*. ACM, New York, NY, 31–35. DOI: https://doi.org/10.1145/971300.971314

[104] Eduardo Navas, Owen Gallagher, and Borrough, xtine. 2014. *The Routledge Companion to Remix Studies*. Routledge.

[105] George Nelson. 1965. *Problems of Design*. Whitney Library of Design.

[106] Maria Adriana Neroni and Nathan Crilly. 2019. Whose ideas are most fixating, your own or other people's? The effect of idea agency on subsequent design behaviour. *Design Studies* 60 (Jan. 2019), 180–212. DOI: https://doi.org/10.1016/j.destud.2018.05.004

[107] Jakob Nielsen and Rolf Molich. 1990. Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'90)*. ACM, New York, NY, 249–256. DOI: https://doi.org/10.1145/97243.97281

[108] Donald A. Norman. 2004. *Emotional Design: Why We Love (or Hate) Everyday Things*. Basic Books.

[109] Donald A. Norman and Pieter Jan Stappers. 2015. DesignX: Complex sociotechnical systems. *She Ji: The Journal of Design, Economics, and Innovation* 1, 2 (Dec. 2015), 83–106. DOI: https://doi.org/10.1016/j.sheji.2016.01.002

[110] Alannah Oleson, Christopher Mendez, Zoe Steine-Hanson, Claudia Hilderbrand, Christopher Perdriau, Margaret Burnett, and Amy J. Ko. 2018. Pedagogical content knowledge for teaching inclusive design. In *Proceedings of the 2018 ACM Conference on International Computing Education Research (ICER'18)*. ACM, New York, NY, 69–77. DOI: https://doi.org/10.1145/3230977.3230998

[111] Alannah Oleson, Meron Solomon, and Amy J. Ko. 2020. Computing students' learning difficulties in HCI education. In *ACM CHI 2020 Conference on Human Factors in Computing Systems*. 14. DOI: https://doi.org/10.1145/3313831.3376149

[112] Arno Pasternak. 2016. Contextualized teaching in the lower secondary education: Long-term evaluation of a CS course from grade 6 to 10. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE'16)*. ACM, New York, NY, 657–662. DOI: https://doi.org/10.1145/2839509.2844592

[113] Michael Quinn Patton. 2014. *Qualitative Research & Evaluation Methods: Integrating Theory and Practice*. SAGE Publications.

[114] Anne-Kathrin Peters. 2018. Students' experience of participation in a discipline - A longitudinal study of computer science and IT engineering students. *ACM Transactions on Computing Education (TOCE)* 19, 1 (Sept. 2018), 5:1–5:28. DOI: https://doi.org/10.1145/3230011
[115] Marian Petre and André van der Hoek. 2013. *Software Designers in Action: A Human-Centric Look at Design Work* (1st ed.). Chapman & Hall/CRC.
[116] Marian Petre and André van der Hoek. 2018. Beyond coding: Toward software development expertise. *XRDS* 25, 1 (Oct. 2018), 22–26. DOI: https://doi.org/10.1145/3265889
[117] Henry Petroski. 2006. *Success through Failure: The Paradox of Design*. Princeton University Press.
[118] Michael J. Prince and Richard M. Felder. 2006. Inductive teaching and learning methods: Definitions, comparisons, and research bases. *Journal of Engineering Education* 95, 2 (2006), 123–138. DOI: https://doi.org/10.1002/j.2168-9830.2006.tb00884.x
[119] Yolanda Jacobs Reimer and Sarah A. Douglas. 2003. Teaching HCI design with the studio approach. *Computer Science Education* 13, 3 (2003), 191–205.
[120] Lauren Rich, Heather Perry, and Mark Guzdial. 2004. A CS1 course designed to address interests of women. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'04)*. ACM, New York, NY, 190–194. DOI: https://doi.org/10.1145/971300.971370
[121] Horst W. J. Rittel. 1987. The reasoning of designers. In *International Congress on Planning and Design Theory in Boston*. 1–9.
[122] Horst W. J. Rittel and Melvin M. Webber. 1973. Dilemmas in a general theory of planning. *Policy Sciences* 4, 2 (June 1973), 155–169. DOI: https://doi.org/10.1007/BF01405730
[123] Luciana Salgado, Roberto Pereira, and Isabela Gasparini. 2015. Cultural issues in HCI: Challenges and opportunities. In *Human-Computer Interaction: Design and Evaluation (Lecture Notes in Computer Science)*, Masaaki Kurosu (Ed.). Springer International Publishing, 60–70.
[124] Donald A. Schön. 1984. *The Reflective Practitioner: How Professionals Think in Action*. Basic Books.
[125] Donald A. Schön. 1987. Educating the reflective practitioner. (1987). Jossey-Bass San Francisco.
[126] Mary Shaw and David Garlan. 1996. *Software Architecture*. Vol. 101. Prentice Hall Englewood Cliffs.
[127] Lee Shulman. 1987. Knowledge and teaching: Foundations of the new reform. *Harvard Educational Review* 57, 1 (1987), 1–23. DOI: https://doi.org/10.17763/haer.57.1.j463w79r56455411
[128] Siang Kok Sim and Alex H. B. Duffy. 2003. Towards an ontology of generic engineering design activities. *Research in Engineering Design* 14, 4 (Nov. 2003), 200–223. DOI: https://doi.org/10.1007/s00163-003-0037-1
[129] Herbert A. Simon. 1973. The structure of ill structured problems. *Artificial Intelligence* 4, 3 (Dec. 1973), 181–201. DOI: https://doi.org/10.1016/0004-3702(73)90011-8
[130] Susan Singer and Karl A. Smith. 2013. Discipline-based education research: Understanding and improving learning in undergraduate science and engineering. *Journal of Engineering Education* 102, 4 (Oct. 2013), 468–471. DOI: https://doi.org/10.1002/jee.20030
[131] Karl A. Smith, Sheri D. Sheppard, David W. Johnson, and Roger T. Johnson. 2005. Pedagogies of engagement: Classroom-based practices. *Journal of Engineering Education* 94, 1 (Jan. 2005), 87–101. DOI: https://doi.org/10.1002/j.2168-9830.2005.tb00831.x
[132] Penny Sparke and Fiona Fisher. 2016. *The Routledge Companion to Design Studies*. Routledge.
[133] Elizabeth Starkey, Christine A. Toh, and Scarlett R. Miller. 2016. Abandoning creativity: The evolution of creative ideas in engineering design course projects. *Design Studies* 47 (Nov. 2016), 47–72. DOI: https://doi.org/10.1016/j.destud.2016.08.003
[134] Matti Tedre, Simon, and Lauri Malmi. 2018. Changing aims of computing education: A historical survey. *Computer Science Education* 28, 2 (June 2018), 1–29. DOI: https://doi.org/10.1080/08993408.2018.1486624
[135] The Association for Computing Machinery (ACM). 2019. Computing Curricula 2005: The Overview Report. https://www.acm.org/binaries/content/assets/education/curricula-recommendations/cc2005-march06final.pdf.
[136] Charles Thevathayan and Margaret Hamilton. 2017. Imparting software engineering design skills. In *Proceedings of the 19th Australasian Computing Education Conference (ACE'17)*. ACM, New York, NY, 95–102. DOI: https://doi.org/10.1145/3013499.3013511
[137] Harold Thimbleby. 2009. Teaching and learning HCI. In *Universal Access in Human-Computer Interaction. Addressing Diversity (Lecture Notes in Computer Science)*, Constantine Stephanidis (Ed.). Springer, Berlin, 625–635.
[138] Nicholas True, Jeroen Peeters, and Daniel Fallman. 2013. Confabulation in the time of transdisciplinarity: Reflection on HCI education and a call for conversation. In *Human-Computer Interaction. Human-Centred Design Approaches, Methods, Tools, and Environments (Lecture Notes in Computer Science)*, Masaaki Kurosu (Ed.). Springer, Berlin, 128–136.
[139] Sepehr Vakil. 2018. Ethics, identity, and political vision: Toward a justice-centered approach to equity in computer science education. *Harvard Educational Review* 88, 1 (2018), 26–52.

[140] Anna Vallgarda and Ylva Fernaeus. 2015. Interaction design as a bricolage practice. In *Proceedings of the 9th International Conference on Tangible, Embedded, and Embodied Interaction (TEI'15)*. ACM, New York, NY, 173–180. DOI:https://doi.org/10.1145/2677199.2680594

[141] Swaroop S. Vattam, Ashok K. Goel, Spencer Rugaber, Cindy E. Hmelo-Silver, Rebecca Jordan, Steven Gray, and Suparna Sinha. 2011. Understanding complex natural systems by articulating structure-behavior-function models. *Journal of Educational Technology & Society* 14, 1 (2011), 66–81. https://www.jstor.org/stable/jeductechsoci.14.1.66.

[142] Jane Webster and Richard T. Watson. 2020. Analyzing the past to prepare for the future: Writing a literature review. *MIS Quarterly* 26, 2 (2020), xiii–xxiii.

[143] Cathleen Wharton, John Rieman, Clayton Lewis, and Peter Polson. 1994. Usability inspection methods. John Wiley & Sons, Inc., New York, NY, 105–140. http://dl.acm.org/citation.cfm?id=189200.189214.

[144] Lauren Wilcox, Betsy DiSalvo, Dick Henneman, and Qiaosi Wang. 2019. Design in the HCI classroom: Setting a research agenda. In *Proceedings of the 2019 on Designing Interactive Systems Conference (DIS'19)*. ACM, New York, NY, 871–883. DOI:https://doi.org/10.1145/3322276.3322381

[145] Tracee Vetting Wolf, Jennifer A. Rode, Jeremy Sussman, and Wendy A. Kellogg. 2006. Dispelling "design" as the black art of CHI. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'06)*. ACM, New York, NY, 521–530. DOI:https://doi.org/10.1145/1124772.1124853

[146] Peter Wright and John McCarthy. 2008. Empathy and experience in HCI. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'08)*. ACM, New York, NY, 637–646. DOI:https://doi.org/10.1145/1357054.1357156

[147] Sarita Yardi and Amy Bruckman. 2007. What is computing?: Bridging the gap between teenagers' perceptions and graduate students' experiences. In *Proceedings of the 3rd International Workshop on Computing Education Research (ICER'07)*. ACM, New York, NY, 39–50. DOI:https://doi.org/10.1145/1288580.1288586

[148] Helen Z. Zhang, Charles Xie, and Saeid Nourian. 2018. Are their designs iterative or fixated? Investigating design patterns from student digital footprints in computer-aided design software. *International Journal of Technology and Design Education* 28, 3 (Sept. 2018), 819–841. DOI:https://doi.org/10.1007/s10798-017-9408-1

Received June 2019; revised September 2020; accepted September 2020