International Journal of Engineering Education Vol. 34, No. 2(A), pp. 458–470, 2018
Printed in Great Britain

0949-149X/91 $3.00+0.00
© 2018 TEMPUS Publications.

Examining Novice Programmers' Software Design Strategies through Verbal Protocol Analysis*

MARTIN K.-C. YEH
College of Information Sciences and Technology, The Pennsylvania State University, 25 Yearsley Mill Rd, Media, PA, 19063, USA.
E-mail: martin.yeh@psu.edu

This paper describes the change in software design strategies used by novice programmers over the course of one semester by using verbal protocol analysis. Our participants were nine first-year undergraduate students (novices), and two experts. Overall, we observed that two types of strategy were used by the novice programmers. The most common strategy observed in our participants, at the beginning of the semester, was a UI-based strategy that focused on system components from the user's perspective. This strategy is often overly simplified with little operational and technical details. Another type of strategy used by novices later in the study was a functional-centered strategy in which novices incorporated programming concepts into their design. Novices who used the latter strategy were able to provide more operational detail than when the UI-based strategy was used. We also found that, due to lack of experience, the designs were still very preliminary. In addition, the novices also exhibited opportunistic design behavior more often than systematic behavior (i.e., a top-down or bottom-up strategy) during the semester. We argue that teaching programming knowledge and skills alone will not develop students' software design knowledge effectively.

Keywords: psychology of programming; verbal protocol analysis; software design; human factors

1. Introduction

The first computer programming course (CS1) in undergraduate education is often important yet challenging for many freshmen. As a result, the number of students who graduate with a computer science degree cannot keep up with the job demand. A presidential memorandum signed in 2017 has placed a $200 million grant to allow the Department of Education to fund schools to access high-quality science, technology, engineering, and mathematics programs, specifically in computer science education and coding, to address this crisis. Researchers have been investigating issues related to teaching CS1 from many aspects including educational theories, pedagogical approaches, programming language features, problem-solving skills, design and cognition, and gender differences [1].

Research studies have shown that learning to program is difficult [2] and have identified some of the root causes of the difficulties such as memorizing language syntax and semantics, managing the complexity of a programming project [3], representing and solving a problem using a programming language [4], and continuing in the long process of learning to program [5, 6]. Influenced by Bloom's taxonomy [7], teaching introductory computer programming has been primarily organized by starting with low-level programming skills, then moving toward higher level design knowledge later. This pedagogical model exposes novices to only part of the design task. As indicated by Kinnunen and Malmi's study [3], 30 to 50 of percent students drop out of an introductory computer programming course (CS1) at Helsinki University of Technology. Moreover, they found that one of the difficulties for CS1 students is "*managing extensive unity: for some students, the size of the project work was difficult to handle*", which means up to half of the students felt that programming projects were too complex for them to handle. Learning software design skills is an alternative for managing the complexity of programming projects. Watson and Li [8] reported a pass rate of 67.7% across 51 institutions in their study. After all, the difficulties of developing a software project lie not only in memorizing the syntax or using programming constructs correctly but also in dividing and structuring components in an organized way and managing project complexity [9–12].

The aim of this study is to address how novice programmers acquire design strategies and how these strategies changes through time. Specifically, this study aims to answer the following research questions:

- What are the characteristics of novices' software design strategies?
- Are novices' software design strategies similar to each other?
- Do novices' software design strategies change over time?

2. Related work

Many barriers to novice programmers are rooted in the design of the programming language. Certain

* Accepted 20 November 2017.

programming features and statements are hard to understand and can cause confusion [13]. The choice of words, symbols, and phrases can be less intuitive to non-programmers, therefore to novice programmers as well, than to experienced programmers [14]. For example, apparently using "*repeat*" is more intuitive than using "*for*" when it comes to iteration. Teaching programming with syntactically simple languages (e.g., Python) can improve comprehension and accuracy than using more syntactically complex languages (e.g., Java). Because of these programming features, students tend experience a small number of syntax errors more frequent than others [15].

Other than the language features and constructs, we can attribute novices' failure in CS1 to lack of software design strategies. It is typical that novices learn syntax and semantics knowledge but often cannot put the statements together in a complete and valid program [6]. Novices' lack of design strategies and problem-solving skills can be attributed to the fact that problem-solving skills are in almost every course description and yet the teaching components that are specific to them are vague and abstract [16] and to the way in which programming textbooks are devoted to programming knowledge but few problem-solving skills [17].

Computer programming and software design are seen as two different types of cognitive activities but are both essential to software development [18]. To learn computer programming is to learn the symbolic representations and rules of a programming language; to learn software design is to apply different types of knowledge to various problem-solving situations [19]. Software design skills are regarded as abstract, which include analysis, and synthesis, and are often addressed later in the curriculum of computer science. However, as some studies [16, 20] pointed out, design knowledge and problem-solving skills can be domain independent and do not necessarily come with the acquisition of programming skills. CS1 may experience fewer dropouts and be more successful by introducing design knowledge early [21]. Researchers [22, 23] argue that the longer we defer teaching design knowledge, the more costly it will become, e.g., the cost of fixing design errors and the cost of loss of human capital [3]. In addition, McCracken [24] points out that among the scarce studies on software design, documentation on the progress of learning knowledge of software design is almost nonexistent. Understanding the nature of software design and the development of design knowledge will put us in a better position to retain learners and improve computer science education.

3. Research design

To understand novices' strategies in solving a design problem, we chose to collect and analyze their verbal protocols through the talk aloud method [25] because it can uncover the cognitive process at the moment of performing a task. Verbal Protocol Analysis (VPA) has been used in many studies [18, 19], [26–29]. The validity and reliability issues of using VPA were discussed in [30, 31].

Although novices are the primary target of this study, we collected experts' verbal protocols for comparison. First, experts' verbal protocols can serve as comparison data set for interpreting novices' cognitive ability. Second, experts' verbal protocol can inform coding decisions.

3.1 The Participants

There are two types of participants in this study: two expert designers and eight novice programmers (experts and novices hereafter). Novices—seven males and one female—were college students in the first Java programming course of two sections in the same semester at a large research university in the US. The instructors from both sections were recruited as experts. They were both doctoral students who each had several years of full-time software design experience.

3.2 The design task

It is common that software design studies use one or more design scenarios as problem-solving tasks, superficial or realistic, and then the participants' thinking processes are recorded and analyzed based on different research questions. Some scenarios that could be solved with less than one hundred lines of code were criticized for being artificial and unrealistic because those problems might not require complex thinking [18, 32]. To avoid this pitfall, we designed a scenario that satisfies the following criteria:

- Complexity: To uncover different cognitive processes, the scenario needs to be complex. If the problem only involves a simple solution such as counting a running total, then the only observable cognitive activities are likely to be applying prescribed or memorized solutions.
- Domain knowledge independence: By domain knowledge independence we mean the solution does not depend heavily on domain specific knowledge typically possessed by experts in the domain, but not necessarily by novice programmers, such as a mathematic calculation, understanding some complex physical phenomenon, or using some specialized algorithms. The scenario

is designed so that novices do not feel intimidated by the design task.

The scenario is a modified version of design task in Adelson and Soloway's study [33] that we call an Online Library Management System (OLMS). An OLMS is a library management system that allows patrons to search for library items, look up their library records, change personal information, reserve library items, and renew checked out items. It also allows librarians to check out library items for patrons, send notifications, and modify library items. College students should be familiar with the concept of OLMS.

In addition to the design scenario, a demographic survey was also used to record participants' demographic data, especially the use of computers in daily life, and prior design and development experiences. The survey data can help us understand whether participants transfer knowledge from other experiences and how their prior knowledge affects the assimilation and accommodation processes [34]. Questions were designed to understand what computer applications the participants use, what programming experience they have, and what programming languages they have used before. For example, participants who use spreadsheet formulas in Microsoft Excel often may possess superior skills in problem decomposition that may help to design the OLMS. The survey interview was conducted face-to-face and some additional questions were asked to gain a deeper understanding of their computing background based on participants' responses.

3.3 Procedure

There were three design sessions in one semester (first, seventh, and fourteenth week of a fifteen week semester) for this study. Experts only participated in the first design session. Novices participated in all three sessions separately. All sessions were video recorded. Each design session lasted no more than one hour. In the first session, participants also completed a demographic survey that was audio recorded. They were shown a demonstration video that explained the talk-aloud method.

Participants were given a consent form before they started. Each participant received the same design scenario (OLMS) and was asked to create a design document on paper for such a system so that the design document could be handed to programmers for implementation. The experimenter was in the same room with the participant to record the design sessions throughout the experiment. The experimenter rarely interrupted the subjects except when the talk aloud was unclear to the experimenter. There was no format requirement on the design document, which could include diagrams, text descriptions, etc. Participants were only required to create a proper design document for implementation and talk aloud at the same time. They were not asked to implement the design in any programming language. Participants were also told they were free to discard their design and redo it at any time. The design sessions ended when the participants were satisfied with their design or they could no longer improve on or add to the design.

3.4 Data analysis

Transcripts of video recordings are the primary data. Participants' design artifacts were used to assist the analysis as well. Although the participants were being video recorded, the video camera only focused on their design document, so facial expressions and gestures were not analyzed. Therefore, the analysis is a verbal analysis rather than video interaction analysis. The unit of analysis is the smallest task in a single context the participants were attending to during the design process. As described further below, their cognitive activities were inferred from the verbal protocols and described with respect to changes in cognitive processes.

Adapting from the proposed guidelines by Chi [35], the following functional steps were taken to analyze the data after all video recordings were transcribed:

1. Segmenting protocols by episodes, tasks, and contents.
2. Developing a coding scheme or formalism.
3. Identifying evidence in the coded protocols that constitutes a mapping to some chosen formalism.
4. Depicting the mapped formalism for all sessions across all novices.
5. Seeking pattern(s) in the mapped formalism.
6. Interpreting the pattern(s).

All video recordings were first transcribed into textual protocols, each of which was then segmented into episodes based on the context and tasks to which participants were attending. The reason for segmenting the protocols by context rather than proposition is to preserve information that is related to the single cognitive task. Each episode was then sorted and labeled as one of two separate tasks: design-related or non-design-related. From the design-related episodes, we developed five categories: problem comprehension, decomposition, structuring, mental simulation, and evaluation. Non-design related episodes were not further analyzed. More detailed information for each category is described in the section on coding themes later.

Every episode was labeled by a single rater, according to the coding scheme; this rater read the content and classified the participants' cognitive activity. All adjacent episodes with the same category label were reviewed again to ensure they belonged to different episodes. Long episodes also received another inspection to confirm that they belonged to one episode rather than multiple episodes.

Episodes were labeled as non-design-related when the participants were performing activities that were not directly related to solving the design scenario. These could be verbal data related to general design principles (e.g., button size, interface layouts), capabilities the designers would like to make available that are not in the instructions (e.g., make something similar to Microsoft Surface, include the use of touch screen), and conversations between the participants and the experimenter that were not related to the current design task.

3.5 Coding themes

The following themes emerged from the existing literature [19, 26, 28, 36, 37] and from analyzing experts' verbal protocols. The process of generating the coding themes cycles between the analyses of verbal data and the generation of coding themes until no more themes emerge. Verbal protocol episodes were assigned to one coding theme. In other words, an episode is a segment of verbal protocol that describes a single event or cognitive process.

3.5.1 Problem comprehension

Problem comprehension is a cognitive process in which a set of requirements (problem description) is translated into an organized internal representation through existing schema, prior experience, and intuition. According to schema acquisition theories [34], encoded knowledge stored in long-term memory (LTM) is an individualized mental representation of a design scenario. Problem comprehension is the cognitive process of bridging the external world and the internal representation. A problem can only be understood by accommodating or assimilating to existing schemas. Representing a problem using existing schemas is regarded as the first stage of the software design.

For novices, the internal representation of the OLMS at this stage is probably in a preliminary form. It can be incomplete or incorrect and needs to be modified and restructured during the design process. We expect that changing mental models and re-reading design instructions are both present throughout the design process.

3.5.2 Decomposition

Decomposition is the process of generating intermediate subsystems, subcomponents, and procedures that together can achieve at least a portion of the ultimate design goals. There are two different types of decomposition. First, a designer can divide a problem into smaller problems so that they are manageable. This is referred as *problem decomposition*. It is a common strategy and can be observed in both means-ends or difference-reduction problem-solving processes [5]. For example, a library system can be divided into tasks that fulfill the needs of the librarians and the needs of the patrons. Each need can be broken down further, depending on the designer. Second, the designer can divide a function into smaller steps, which is referred as *procedural decomposition*. Depending on the design, the designer can treat a task as a complete piece or use procedural decomposition to examine its internal mechanism for reuse or for restructuring. For example, the function of checking out a library item can be broken down into checking the availability of the system, verifying patron identity, calculating a due date, modifying the library or the patron's personal records, etc.

3.5.3 Structuring

Structuring is the process of examining and associating subsystems, subcomponents, and procedures that are created from decomposition. Unlike decomposition that concentrates on one component and breaks it down into several subcomponents, structuring concentrates on connecting existing components. It is often represented by arrows pointing from one component to another in a pictorial artifact. Coordination of multiple ideas is the main goal in this cognitive process. As a result, structuring requires the designer to recall information from several components and to manipulate their properties in a way that will achieve one task. A high level of cognitive load is expected in this process because multiple components have to be brought into working memory for processing.

3.5.4 Mental simulation

Mental simulation is a process that uses a hypothetical input and executes it in part of the design to see if the output is as expected. It is often used as a method of verifying the product of structuring. Software designers use two types of mental simulation that require different kinds of knowledge and lead to different design behavior [37]. The first one is *scenario simulation* where designers anticipate the system output based on user input. A designer can imagine herself as being a patron using a library system. She will need a working mental model in her memory to "see" and "operate" the system. Another type of simulation is a *functional simulation* where designers execute a partial or complete system to ensure the sequences of a functional procedure are coherent, ''stepping through'' the execution of programs.

3.5.5 Evaluation

Evaluation is the process of making a judgment about selecting between solutions. Typically, an evaluation process includes phases such as identifying the problem, generating a hypothesis, testing alternatives, and choosing a solution. However, these phases are not always included in an evaluation process. The designer may decide to postpone an issue until she has a better understanding. There can be several benchmarks phrases such as efficiency, consistency, or redundancy in an evaluation process. Evaluation can also be based on intuition because we believe that some designers may have sufficient knowledge that automates the evaluation process. In such cases, the evaluation process can be implicit and seems intuitive to the designers.

The above five coding themes are used to categorize episodes segmented from the verbal protocol. We expect that problem comprehension, decomposition, and structuring activities will constitute the majority of the total cognitive activities because they are common in problem-solving situations. Mental simulation and evaluation require more complex cognitive ability, may appear later in a design process, and designers can finish a draft design without including these activities. Therefor we predict these activities to occur less frequently.

4. Results

Overall, we found that our participants rely on two main strategies: user-interface (UI) based strategy and functional-centered strategy. The UI-based strategy is one that focuses on the elements of the UI as an end-user might. Functional-centered strategy, on the other hand, is one that focuses on the operational implementation of tasks and details. Table 1 shows the overall sum of episodes generated by the experts and novices in each session and suggests that experts are either more thoughtful or better in articulating their strategies in designing the OLMS. It also suggests that overall novices become better at design through the series of tasks over the duration of the semester and are either becoming more thoughtful or better at generating verbal protocols, though not as good as the experts. Fig. 1 shows that the experts engaged in more evaluation and less problem comprehension and structuring than the novices do. The experts seemed to be better at understanding the problem (a smaller ration of problem comprehension episodes), created a more stable design, and tested and evaluated alternatives more often.

Table 1. Comparing average number of episodes by expertise and session (non-design episodes are excluded).

Type of Expertise/Session	Average Number of Episodes
Experts (N = 2)	94.5
Novices/Session 1 (N = 8)	41.4
Novices/Session 2 (N = 8)	46.1
Novices/Session 3 (N = 8)	52.9

Below are the qualitative description and summary for each novice.

4.1 Participant 1

According to the artifacts, participant 1's design strategy evolved from a simple one into a more complex one during the semester. He was able not only to evaluate different options and give rationales but also to execute more sophisticated mental simulation later. Learning Java might have helped him articulate the underlying functionalities of each task, improving his procedural decomposition, and enabling his ability to organize and simulate subcomponents. Although the number of episodes of decomposition (30.2%, 34.4%, and 29.8%) and structuring (30.2%, 21.9%, and 31.6%) were similar across all three design sessions (see Fig. 2), the quality of those cognitive processes did improve. In the decomposition category, he improved from

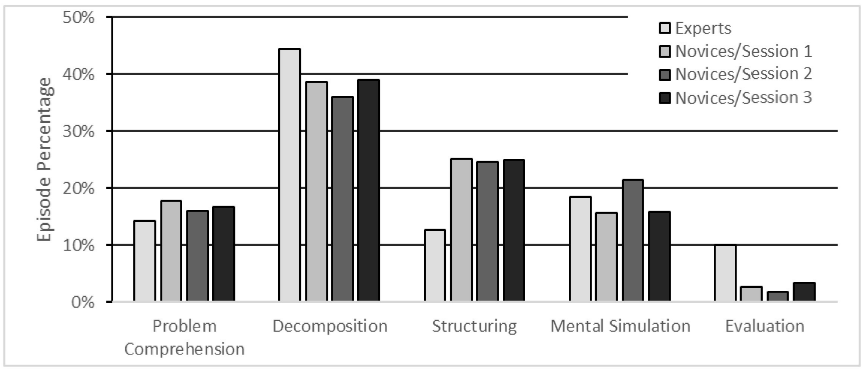

Fig. 1. Comparison between experts (N = 2) and novices (N = 8) in three design sessions among themes.

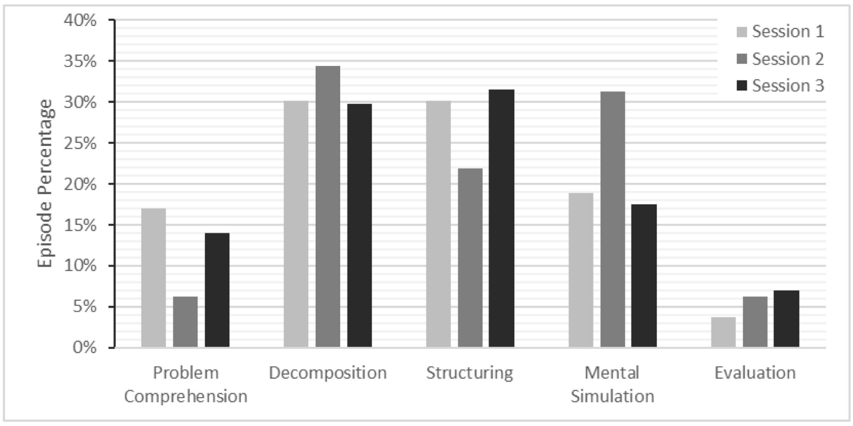

Fig. 2. Design episode percentages for participant 1 in the three design sessions.

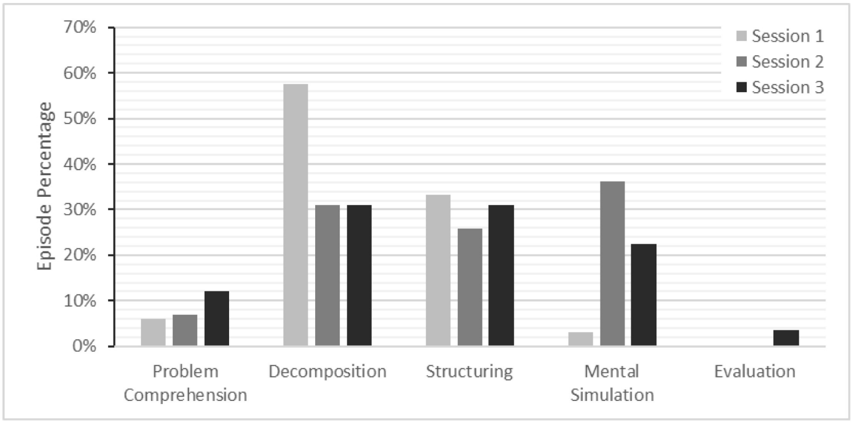

Fig. 3. Design episode percentages for participant 2 in the three design sessions.

centering on the UI to adding more details to the functionality. In the structuring category, participant 1 improved from making connections based on prospective users' behavior to linking components based on the task. To make both of these improvements possible, we argue, he must be able to comprehend how components affect the system as a whole rather than the surface UI features. We see that participant 1's strategy changed from UI-oriented to more functional-centered. It is likely that programming knowledge enabled this transition.

4.2 Participant 2

Figure 3 shows that the biggest difference for participant 2 in learning programming for a semester was his ability to use mental simulation to improve design (3%, 36%, and 22%). Compared to design session one, mental simulation episodes increased dramatically in sessions two and three, which shows his strategies are at the functional level and are no longer isolated components. There were 34 episodes in the first design session, 58 and 60 in the second and third design sessions respectively. In the first design session, participant 2 stopped his design without a reviewing process. Analyzing his verbal protocols reveals that he did not engage in a mental simulation process in the first design session. The mental simulation in the second and third sessions did change his design. In addition, he divided a component into two sub-components and tackled them one by one. He also recognized two similar functions could be combined while he mentally simulated the function in session two, which happened again in session three. These notable improvements suggest that his strategy is moving toward functional-centered strategy.

4.3 Participant 3

Fig. 4 shows that the number of problem comprehension episodes for participant 3 increased in session. Analyzing participant 3's verbal protocols further revealed that the discrepancy may have come from using a task list on which he wrote two categories—patron and librarian—and "check out items", "check in items" in each category. In the first and second design sessions, he created a task list before he started other design activities. Using a task list allowed him to concentrate on design activities other than problem comprehension. On

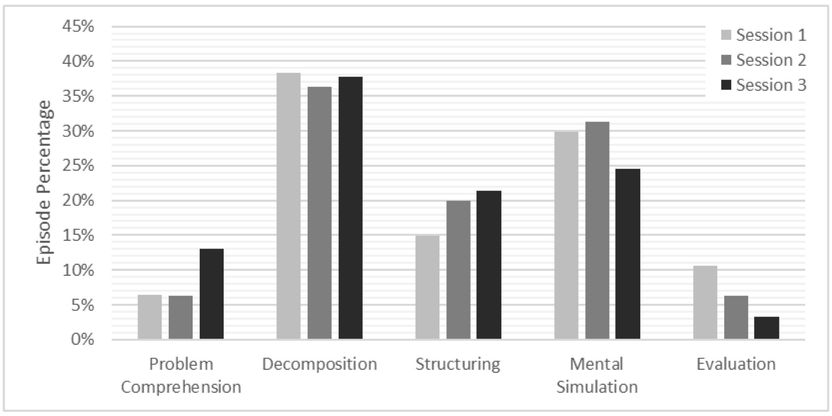

Fig. 4. Design episode percentages for participant 3 in the three design sessions.

the other hand, in design session three, he changed his strategy to walking through UI features, which showed a "depth-first" approach and can be seen by the late problem comprehension episodes during design session three.

Interestingly, Fig. 4 shows a decline of mental simulation in session three and of evaluation from session one to session 3. Further analysis suggests that participant 3 concentrated on non-design a lot in design session three. When participant 3 was using a UI-based strategy in session three, his attention was drawn to usability and consistency. UI appearance, such as the size or location of buttons, became important to him and preoccupied his attention. As a result, other types of cognitive process were reduced.

4.4 Participant 4

Figure 5 illustrates participant 4's lack of mental simulation and evaluation in all three sessions. It is clear from the transcripts that participant 4 could not construct a robust mental model by the end of the study. Although he produced many decomposition episodes, his inability to design came from problems with comprehension, decomposition, or structuring.

> "I kind of get the idea but I just don't know how to put [it] together." [In design session two]
>
> "I just don't know how to draw this... But I get the idea [of] what you said." [Also in design session two]

Even though he claimed that he understood the system, his verbal protocol did not support such a statement. He struggled in understanding the design scenario and was unable to map the problem statements onto his own design document. He also had problems with creating new components to assist him in this design. What he showed in the design documents was largely based on the instruction sheet that was provided. For him, designing a system he had not designed before was challenging, and he had difficulty stepping away from his previous database design experience. There was no observable improvement during the semester in terms of software design. He had no evaluation episodes in any of the three sessions and the number of mental simulation episodes was consistently low (4.8%, 2.3%, and 4.4%).

Participant 4 did try to use object-oriented con-

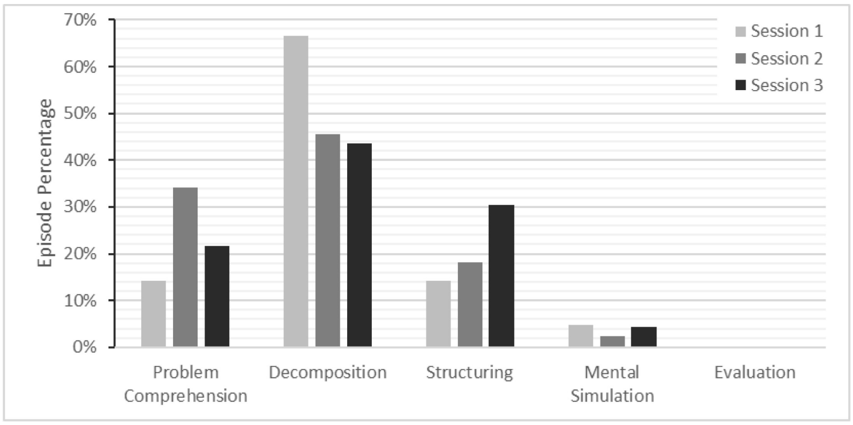

Fig. 5. Design episode percentages for participant 4 in the three design sessions.

cepts in his design in the beginning of session three, something he did not in the session one and two. Programming knowledge enabled him to design software systematically at a lower operational level. Nevertheless, it did not improve his design knowledge. He seemed to want to change his design strategy from UI-based to functional-centered. The outcome of the design, however, did not seem to improve.

4.5 *Participant 5*

Participant 5 demonstrated two different strategies: a UI-based strategy in session one and a functional-centered strategy in sessions two and three. The UI-based strategy allowed him to bypass some of the functional details of the task while he was beginning to learn how to program. As shown in Fig. 6 in design session two, there were very few mental simulations, whereas several mental simulations were identified in session three. He stated that he did not feel comfortable simulating subcomponents when using a functional-centered strategy while he had only studied Java programming for about five weeks at that time. We believe, consequently, he was not capable of reviewing his own design or selecting alternatives; whereas in session three when he used a functional-centered strategy at the end of the semester, he used mental simulation more to review his design. However, there is no evidence to confirm or disapprove this conjecture. It might just be that he chose not to in session two and opted for self-reviewing in session three for other reasons.

4.6 *Participant 6*

Participant 6 used a UI-based mental model in the first session and a functional-centered strategy in the second and third sessions. Figure 7 shows that the increase of decomposition is the most dramatic and obvious change. As discussed in other participants' summaries, a programming-based mental model requires more low-level cognitive processes, such as problem comprehension and decomposition. This phenomenon is depicted in Fig. 7 where the number of decomposition episodes grows steadily (6, 14, and 33).

Having a UI-based mental model (session one) caused participant 6 to use mental simulation more often than having a programming-based mental model (in sessions two and three). Furthermore, there was no mental simulation in session two,

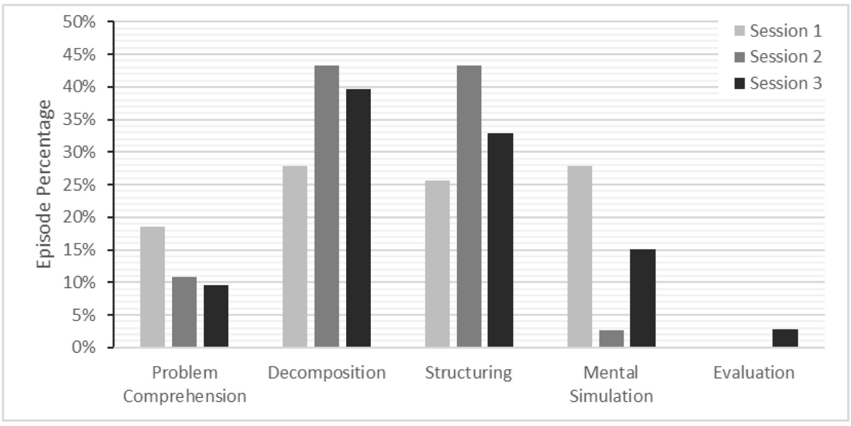

Fig. 6. Design episode percentages for participant 5 in the three design sessions.

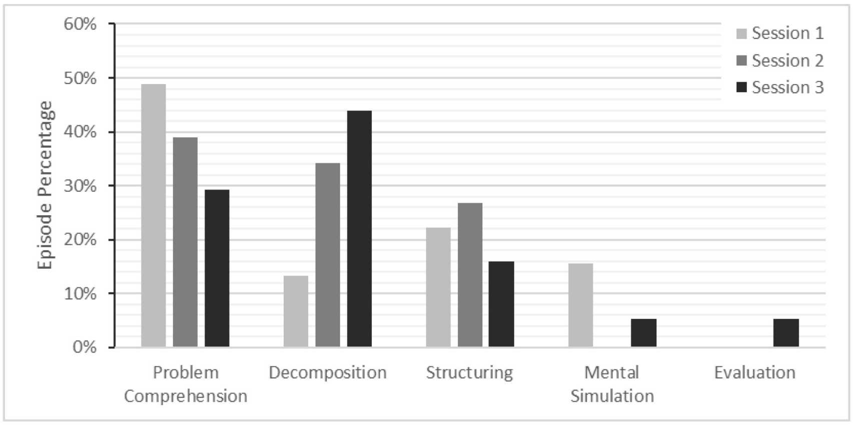

Fig. 7. Design episode percentages for participant 6 in the three design sessions.

which was the first time he had demonstrated using a programming-based mental model. Although he did not mention the reason, it was possible that it was because he had been in Java class for less than two months. To simulate the system using a programming-based mental model would require executing pseudo-code mentally, which might have been too challenging for participant 6 as a novice. In session three, he did demonstrate four mental simulation episodes.

4.7 Participant 7

Analyzing participant 7's verbal protocols from all three sessions indicates that his strategy was rather opportunistic. In the first session, he used diagrams/flow charts to guide his design. In the second session, he used a UI-based strategy for his design. In the third session, he used only text descriptions to describe his design. In all three sessions, he started from a log-in page, and what followed was very different, which suggests his opportunistic strategy. We do not see any indication that learning Java programming for one semester changed his design strategy in any consistent way.

In addition, Fig. 8 shows that the greatest change is the mental simulation in design session two compared with sessions one and three. This is because he used a UI-based strategy in session two. The reason we categorize the strategy as a UI-based is because the simulation is similar to user walk-through and no functional details were observed.

4.8 Participant 8

The result of participant 8's verbal protocols indicates that his strategies were very similar in all three sessions. According to Fig. 9, the biggest difference was that there was no problem comprehension and that the decomposition episodes took up more than 50% of the total episodes in session one. We believe this was due to his opportunistic design behavior. He did not demonstrate a systematic strategy either in following through a function or in breaking up a component to smaller ones. When we examined more carefully the episodes of participant 8's cognitive processes, we found that his design tactics were centered on arranging and categorizing the user tasks, which is another indicator of his UI-based strategy.

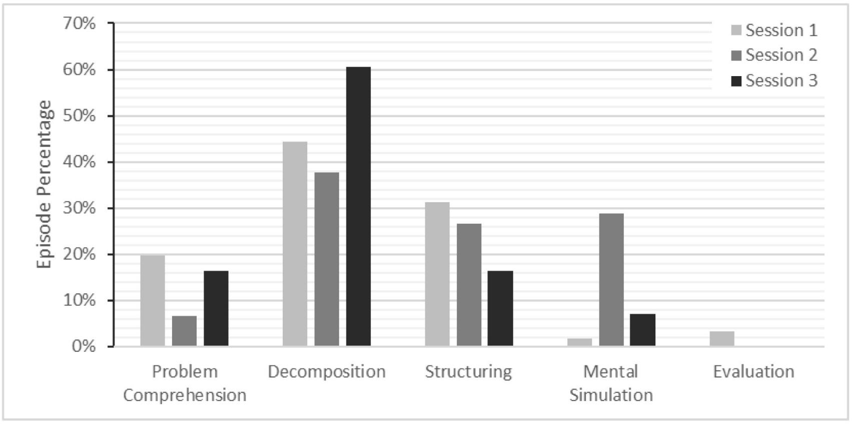

Fig. 8. Design episode percentages for participant 7 in the three design sessions.

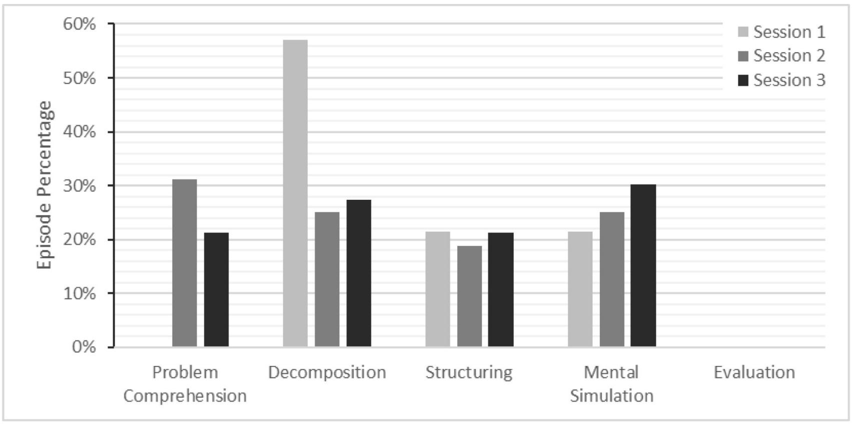

Fig. 9. Design episode percentages for participant 8 in the three design sessions.

Table 2. Design strategy used in each session by participant

Participant	Session 1	Session 2	Session 3
Participant 1	UI-based	Functional-centered	Functional-centered
Participant 2	UI-based	Functional-centered	Functional-centered
Participant 3	UI-based	UI-based	UI-based
Participant 4	UI-based	UI-based	Functional-centered
Participant 5	UI-based	Functional-centered	Functional-centered
Participant 6	UI-based	Functional-centered	Functional-centered
Participant 7	UI-based	UI-based	Functional-centered
Participant 8	UI-based	UI-based	UI-based

"Let's see, look up overdue items, kind of jumping around here, go back to this item here, um, main library page." [In design session three]

The missing evaluation episodes for all three sessions can be seen as an indicator that he is just following the instructions and reorganizing those tasks. Because those tasks are visible for a person who is familiar with the library system, it is not difficult to rearrange those tasks into one design document.

4.9 Summary

Although the participants demonstrated a variety of behaviors, their strategies could be categorized as either UI-based or somewhat functional-centered. While they addressed components by their function, the level of detail was similar in both the UI-based strategy and the functional centered strategy. Based on the data, we list participants' strategy in each session in Table 2.

5. Discussion

Overall, we found novices' design strategy changes slowly from a user's perspective (UI-based) to developer's (functional-centered) perspective during one semester of learning Java programming. The original research questions are answered in the next paragraphs.

- *What are the characteristics of the novices' software design strategies?*
  When novices are facing a novel problem, they rely primarily on prior experience and knowledge. Drawing related knowledge from memory becomes a logical choice to solve the problem [34], [38, p. 195], especially when the problem resembles what is already known. In our study, we observed the same behavior. In the first design session, every participant used the database design concept from their prior database course (programming was not taught in that course) to solve the problem. For example, one of the participants replaced the library system with the online bookstore he designed in the database design course. With the limited resources they have, novices have to find prior knowledge that is compatible with the problem they face. The experience of interacting with similar systems is also activated to assist their design process. So, when facing problems, they cannot use a thoughtful and logical way to come up with a solution. Instead, a template from memory is what a novice can use first. Some novices changed their strategy and adapted their programming knowledge and concepts. For example, we have seen several verbal protocols change from "a patron can check out books. . ." to "this check-out function needs to. . ." which is more focused on what the system should accomplish instead of how the user interact with the system.

  Second, novices' strategies seem unstable and opportunistic. The OLMS is complex and consists of subsystems that are different from each other. Solving these subsystems can be independent to the system as a whole. Novices seem to be restricted to a single strategy. For example, the novices used almost strictly a UI-based or functional-centered strategy in the same design session when the experts were able to focus on one subsystem and use a different approach for different subsystems.
- *Are novices' software design strategies similar to each other?*
  We observed that our novices used two design strategies: UI-based and functional-centered. The UI-based strategy was common to all participants because our participants are all college students who, we argue, have had experience using a library system. In fact, we observed that every participant used the UI-based strategy at some point. Features are treated as black boxes without details when the participant uses the UI-based strategy. Although everyone used a UI-based strategy, each person's design documents were different. This variation comes from their perception of what is important, their experience with using libraries, and the libraries with which they interacted. For example, the payment method for fines, borrowing history, and usability of the interface are some examples that were mentioned by the participants but not in the problem sce-

nario. As a result, the final design documents vary. In design sessions two and three, some participants used a functional-centered strategy with more detailed descriptions related to procedural logic and data structures. Using a functional-centered strategy makes designers focus on lower level cognitive processes, such as decomposition and structuring. Nevertheless, many participants rely on the UI-based strategy while incorporating a functional-centered strategy.

- *Do novices' software design strategies change over time?*
  In terms of design strategy, we observed six out of eight participants moving toward functional-centered strategy throughout the semester. We attributed the change in strategy to the fact that the participants were learning Java programming. For example, starting from session 2, statements such as "creating data members", "something should be a class", and "*is–a* or *has–a* relations" became more frequent. These changes suggest that the participants were applying what they knew from the Java class to the design task.

Although novices did acquire some form of design strategies, the strategies need to be polished and enhanced. From the study, novices have shown the ability to use programming knowledge in design, but in a novel way. Tuner and Hill [39] used robot-based exercises before Java programming task. They found a correlation between robot-based approach and programming success. Koulouri et al. [40] investigated approaches in teaching introductory programming and found that problem-solving activities prior to taking a CS1 course did improve learners programming ability. We argue that based on our results using programming features as a software design strategy should be considered as part of the CS1 course.

This study investigated, from the perspective of design, how novices form and use their mental models and what might cause their mental models to evolve, with the goal of making software engineering education more effective by understanding the learners. We are left with a renewed sense of the importance of helping students understand not only programming concepts but also software design processes, and eventually software design mental models. But, the study raises questions about the assumption that teaching programming skills alone will foster better design processes and therefore better mental models. If novices are to learn complex system design, learning appropriate design skills (i.e., decomposition and diagramming) and design patterns [41, 42] would appear to be important.

6. Conclusion

In this study, we examined eight novices' design strategies over one semester through three design sessions by recording their think-aloud protocols. We first categorized the verbal protocols into five cognitive activities (themes). We further looked at the themes and their design artifacts and found two distinct strategies: UI-based strategy and functional-centered strategy. The former is a design strategy from the user's perspective and the latter is one that is from the designer's perspective.

We also found that novices demonstrated more programming knowledge in design near the end of the semester compare to the beginning of learning to program. Their design strategies also shifted to more functional-centered than in the beginning of the semester. Although the novices were not able to use the new strategy to improve their design drastically, they moved away from the surface feature to focus on the detail of a system. By the end of the study, we observed deeper understanding of the design task and more robust representations of the system started to emerge that could be supported by novices' creating more detailed decomposition, spending longer time on design task, and using more evaluation skills.

However, because the change was inconsistent and not observed globally, there are opportunities to improve design knowledge through teaching. Novices seems to be capable of using their programming knowledge in design. CS1 can benefit from explicitly teaching design and problem-solving skills. These skills can be introduced before teaching the programming content because the skills are not directly related to a programming language. Strategies such as decomposing the problem or evaluating solutions should help learners manage their programming projects and increase learners' success rate in CS1. Different teaching strategies can be employed and further investigated. For example, a problem-based learning environment might increase novices' performance in reasoning, and direction instructions such as verb/noun might help comprehension and decomposition. The work also shows that when novice learners learn to program, they rely on what they already know and what they are taught.

It is felt that the development of cognitive processes and design strategies for novices can be taught and should be addressed early. A model of explicit instruction on transitioning from the UI-based to the functional-centered strategy may be helpful.

Acknowledgements—The author would like to thank Steven Broomell, Stephen Crocker, Fred Fonseca, Dan Gopstein, Joshua B. Gross, Dan Guzek, Doug Hogan, Kyle Peck, Dave Popp, Frank Ritter, and Andrey Soares for assisting in data collection and/or providing feedback to enhance the paper.

References

1. The White House, https://www.whitehouse.gov/the-press-office/2017/09/25/memorandum-secretary-education, accessed 13 October 2017.
2. A. McGettrick, R. Boyle, R. Ibbett, J. Lloyd, G. Lovegrove and K. Mander, Grand challenges in computing: Education—a summary, *The Computer Journal*, **48**(1), 2005, pp. 42–48.
3. P. Kinnunen and L. Malmi, Why students drop out CS1 course?, *Proceedings of the 2006 International Workshop on Computing Education Research*, Canterbury, United Kingdom, 2006, pp. 97–108.
4. B. Du Boulay, Some difficulties of learning to program, *Journal of Educational Computing Research*, **2**(1), 1986, pp. 57–73.
5. J. R. Anderson, *Cognitive Psychology and its Implications*, 5th ed., W. H. Freeman and Company, New York, 1999.
6. L. E. Winslow, Programming pedagogy—A psychological overview, *SIGCSE Bulletin*, **28**(3), 1996, pp. 17–22.
7. B. S. Bloom, M. D. Engelhart, E. J. Furst, W. H. Hill and D. R. Krathwohl, *Taxonomy of Educational Objectives, Handbook I: The Cognitive Domain*. David McKay Co Inc., New York, 1956.
8. C. Watson and F. W. Li, Failure rates in introductory programming revisited, *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, Uppsala, Sweden, 2014, pp. 39–44.
9. E. Soloway, Learning to program = learning to construct mechanisms and explanations, *Communications of the ACM*, **29**(9), 1986, pp. 850–858.
10. J. C. Spohrer and E. Soloway, Novice mistakes: Are the folk wisdoms correct?, *Communications of the ACM*, **29**(7), 1986, pp. 624–632.
11. J. G. Spohrer and E. Soloway, Analyzing the high frequency bugs in novice programs, in E. Soloway, B. Shneiderman, and S. Iyangar (eds), *Empirical studies of programmers: First workshop*, Greenwood Publishing Group Inc., Westport, CT, 1986, pp. 230–251.
12. D. Teague and P. Roe, Collaborative learning: Towards a solution for novice programmers, *Proceedings of the Tenth Conference on Australasian Computing Education*, Wollongong, NSW, Australia, 2008, pp. 147–153.
13. D. Gopstein, J. Iannacone, Y. Yan, L. A. DeLong, Y, Zhuang, M. K.-C. Yeh and J. Cappos, Understanding Misunderstandings in Source Code, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, Paderborn, Germany, 2017, pp. 129–139.
14. A. Stefik and S. Siebert, An empirical investigation into programming language syntax, *ACM Transactions on Computing Education*, **13**(4), 2013, p. 19.
15. P. Denny, A. Luxton-Reilly and E. Tempero, All syntax errors are not equal, *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, Haifa, Israel, 2012, pp. 75–80.
16. S. J. Kimmel, H. S. Kimmel and F. P. Deek, The common skills of problem solving: From program development to engineering design, *International Journal of Engineering Education*, **19**(6), 2003, pp. 810–817.
17. A. Robins, J. Rountree and N. Rountree, Learning and teaching programming: A review and discussion, *Computer Science Education*, **13**(2), 2003, pp. 137–172.
18. R. Guindon, Designing the design process: Exploring opportunistic thoughts, *Human-Computer Interaction*, **5**(2), 1990, pp. 305–344.
19. R. Jeffries, A. A. Turner, P. G. Polson, and M. E. Atwood, The processes involved in design software, in J. R. Anderson (eds), *Cognitive skills and their acquisition*, Lawrence Erlbaum Associates, Ed. Hillsdale, NJ, 1981, pp. 255–283.
20. V. Goel and P. Pirolli, The structure of design problem spaces, *Cognitive Science*, **16**(3), 1992, pp. 395–429.
21. P. Machanick, Teaching Java backwards, *Computers & Education*, **48**(3), 2007, pp. 396–408.
22. B. Curtis, H. Krasner and N. Iscoe, A field study of the software design process for large systems, *Communications of the ACM*, **31**(11), 1988, pp. 1268–1287.
23. R. Guindon, The process of knowledge discovery in system design, in G. Salvendy and M. J. Smith (eds), *Designing and using human-computer interfaces and knowledge based systems*, Elesver, Amsterdam, 1989, pp. 727–734.
24. W. M. McCracken, Research on learning to design software, in S. Fincher and M. Petre (eds), *Computer science education research*, Taylor & Francis, London, 2004, pp. 155–173.
25. D. Gentner, Mental models, Psychology of, in P. Bates and N. Smelser (eds), *International encyclopedia of the social & behavioral sciences*, Elsevier, Amsterdam, 2002, pp. 9683–9687.
26. R. Brooks, Towards a theory of the cognitive processes in computer programming, *International Journal Man-Machine Studies*, **9**(6), 1977, pp. 737–751.
27. M. T. Chi, M. Bassok, M. W. Lewis, P. Reimann and R. Glaser, Self-explanations: How students study and use examples in learning to solve problems, *Cognitive Science*, **13**(2), 1989, pp. 145–182.
28. V. Renumol, D. Janakiram and S. Jayaprakash, Identification of cognitive processes of effective and ineffective students during computer programming, *ACM Transactions on Computing Education*, **10**(3), 2010, p. 10.
29. C. J. Atman and K. M. Bursic, Verbal protocol analysis as a method to document engineering student design processes, *Journal of Engineering Education*, **87**(2), 1998, pp. 121–132.
30. K. A. Ericsson and H. A. Simon, Verbal reports as data, *Psychological Review*, **87**(3), 1989, p. 215.
31. K. A. Ericsson and H. A. Simon, *Protocol Analysis*. MIT Press, Cambridge, MA, 1993.
32. W. Visser and J.-M. Hoc, Expert software design strategies, in J.-M. Hoc, T. R. G. Green, R. Samurcay, and D. J. Gilmore (eds), *Psychology of programming*, Academic Press, London, 1990, pp. 235–250.
33. B. Adelson and E. Soloway, The role of domain experience in software design, *IEEE Transactions on Software Engineering*, **11**(11), 1985, pp. 1351–1360, 1985.
34. D. E. Rumelhart and D. A. Norman, Accretion, tuning, and restructuring three modes of learning, in J. W. Cotton and R. L. Klatzky (eds), *Semantic factors in cognition*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1978.
35. M. T. H. Chi, Quantifying qualitative analyses of verbal data: A practical guide, *The Journal of the Learning Science*, **6**(3), 1997, pp. 271–315.
36. R. Guindon, H. Krasner, and B. Curtis, Breakdowns and processes during the early activities of software design by professionals, in G. M. Olson, S. Sheppard, and E. Soloway (eds), *Empirical studies of programmers: second workshop*, Ablex Publishing Corp., Norwood, NJ, 1987, pp. 65–82.
37. S. Letovsky, J. Pinto, R. Lampert and E. Soloway, A cognitive analysis of a code inspection, in G. M. Olson, S. Sheppard, and E. Soloway (eds), *Empirical studies of programmers: second workshop*, Ablex Publishing Corp., Norwood, NJ, 1987, pp. 231–247.
38. M. P. Driscoll, *Psychology of Learning for Instruction*, 2nd ed. Allyn & Bacon, Boston, MA, 2000.
39. S. Turner and G. Hill, Robots in problem-solving and programming, *8th Annual Conference of the Subject Centre for Information and Computer Sciences*, Thessaloniki, Greece, 2007, pp. 82–85.
40. T. Koulouri, S. Lauria, and R. D. Macredie, Teaching introductory programming: a quantitative evaluation of different approaches, *ACM Transactions on Computing Education*, **14**(4), 2015, p. 26.
41. A. Chatzigeorgiou, N. Tsantalis, and I. Deligiannis, An empirical study on students' ability to comprehend design patterns, *Computers & Education*, **51**(3), 2008, pp. 1007–1016.
42. G. Kolfschoten, S. Lukosch, A. Verbraeck, E. Valentin and G.-J. de Vreede, Cognitive learning efficiency through the use of design patterns in teaching, *Computers & Education*, **54**(3), 2010, pp. 652–660.

Martin K.-C. Yeh is an assistant professor in the College of Information Sciences and Technology at the Pennsylvania State University. His research interests range from software engineering to cognitive modeling of users to study learning and interfaces. He is also interested in developing and evaluating new technology, particularly mobile devices, to help people learn.