Pattern Census: A Characterization of Pattern Usage in Early Programming Courses

Vighnesh Iyer
University of Illinois at Urbana-Champaign
Urbana, Illinois
vniyer2@illinois.edu

Craig Zilles
University of Illinois at Urbana-Champaign
Urbana, Illinois
zilles@illinois.edu

ABSTRACT

Expert programmers rarely think at the syntactic level. Instead, they think at higher levels of abstraction, mentally "chunking" groups of syntactic elements into a single abstraction. Explicitly teaching common "chunks" in early programming courses has been proposed in the research literature using the term "pattern-oriented instruction", but this practice appears not to be emphasized, nor is there a consensus about which patterns to teach or in what order.

In this paper, we explore the set of patterns that students are expected to learn, independent of whether they are taught explicitly or must learn implicitly. Specifically, we studied the instructor solutions to homework and exams from 12 introductory CS courses from nine universities, identifying the presence of 15 patterns throughout the semester. We present results about the relative frequency of the patterns and the order in which the patterns tend to be introduced.

CCS CONCEPTS

• **Social and professional topics → Computer science education**.

KEYWORDS

patterns, schema, plans, CS1, pattern-oriented instruction

ACM Reference Format:
Vighnesh Iyer and Craig Zilles. 2021. Pattern Census: A Characterization of Pattern Usage in Early Programming Courses. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE '21), March 13–20, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3408877.3432442

1 INTRODUCTION

When programming, experts rarely think about syntax during planning and design. Instead they tend to use programming language features in idiomatic ways and tend to think about their programs using higher-level chunks. As discussed in Section 2.1, thinking at a granularity larger than individual syntax elements reduces cognitive load, which makes cognitive resources available for problem solving.

While we could allow our students to discover these common idioms on their own, it has been proposed that instructors actively teach them as "patterns" that students can use to solve specific problems. Such instruction, referred to as *pattern-oriented instruction*, draws inspiration from the notion of "design patterns" developed by the object-oriented programming community [12]. A description of this prior work can be found in Section 2.2, and, in recognition to this previous work, we will use the term *pattern* to describe these common general-purpose programming idioms in this paper.

More recently, a theory of instruction for introductory programming [42] suggests that after learning to read and write syntactic elements, students should be taught to recognize and comprehend common programming patterns and then to use and compose them to solve problems. While it can be assumed that many instructors cover some patterns in at least an ad hoc manner in their courses, we find little documentation in the research literature providing guidance about the patterns that should be introduced and how they should be sequenced.

In an effort to help fill this gap, this paper reports on a study to investigate empirically which patterns students in CS 1 courses are expected to learn. We undertook this study not by looking at the instructional materials to see what the students were taught, but rather by looking at the assessment materials to see what students were expected to know. Specifically, we looked at the instructor solutions for both homework and exams in a collection of 12 early programming courses from nine universities and identified which patterns were present in each assessment in each course. We did this with the intention of answering two research questions:

(1) Is there (empirically) a consensus about which patterns should be taught in early programming courses?
(2) Is there (empirically) a consensus about how these patterns should be sequenced?

In the courses we analyzed, we find that there is significant consensus among which patterns are covered. We found 9 of the 15 patterns that we counted in at least 9 of the 12 classes studied. Furthermore, we found 5 of the remaining patterns in 3 or fewer of the 12 classes. Of these five, we consider three of them to be language dependent, appearing only in specific languages or language families. The remaining pattern was found in 5 of the 12 courses.

In contrast, there seems to be much less consensus about the sequence that the patterns should be introduced. While there are logical, general trends observable from the data, there is such high variance between classes. As such we caution against using the derived sequence information as guidance on what should be done.

SIGCSE '21, March 13–20, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8062-1/21/03…$15.00
https://doi.org/10.1145/3408877.3432442

2 BACKGROUND

2.1 Chunking to reduce cognitive load

An important difference between novice and expert programmers is that novices have trouble identifying important elements of code, because their lack of knowledge forces them to process each element of the code individually [13]. Experts can automatically 'chunk' multiple elements and process them as one unit [7, 13, 21, 41]. These chunks bring multiple, interconnected elements of information into a single cohesive unit with a specific function [20], allowing them to be treated as a single unit and reducing the cognitive load they induce [33]. These chunks, or schemata as they are referred to in Cognitive Load Theory, are developed through repeated experiences that have identifiable features in common [20].

There is significant support for the existence of these pattern-like chunks in the programming education literature, where they have been referred to as plans, templates, schemata, and idioms [7, 30], and they play a significant role in what it is to be an expert programmer. Experienced programmers have been shown to be much better at memorizing code snippets, especially the control flow constructs and the location of separators [15]. Wiedenbeck found that expert programmers carry out low-level programming comprehension activities faster than novices, finding that experts had "automated" these processes, so that little mental attention was required [40]. A leading theory of program understanding for unfamiliar code is bottom-up, where programmers chunk code statements into higher level abstractions which are aggregated further until a high-level understanding of the program is attained [26, 32]. Vessey found that a programmer's ability to chunk effectively was highly predictive of their ability to debug provided programs [38]. Shneiderman and Mayer [32] find that the "encoding process by which programmers convert the program to internal semantics is analogous to the 'chunking' process first described by George Miller in his classic paper, 'The Magical Number Seven Plus or Minus Two.' "

Schemata mean that more complex problems can be handled [25] and working memory resources are available for problem solving [34], but novel problem solving activities (e.g., code writing) are not the most effective ways of developing schemata. The high cognitive load induced by problem solving inhibits schema construction [35]. For novice learners, learning and performing conventional tasks are different and incompatible processes [25].

Instead, lower cognitive load tasks are recommended for schemata construction, including worked examples [3, 25, 35, 37]. Wiedenbeck suggested that the teaching of novice programmers should "stress continuous practice with basic materials" until the novices have automated the practice, including an early emphasis on program comprehension and tracing, with the aim of automating basic skills, so they could later concentrate on problem-solving [40]. Code-reading exercises with automated feedback [10] could be a scalable means of providing low-cognitive load practice. When students had relevant schemata available, Rist found that students could and did reason forward from plan to code [29].

2.2 Pattern-oriented instruction

When facing a novel problem, students often don't know "where to start" and experience difficulties in understanding the essential challenges in a problem and transferring ideas from previous solutions [19, 24]. Previous work has attempted to explicitly facilitate schema construction and plan development in code writing activities. Van Merriënboer proposed completing partially written programs as a lower cognitive load activity than program generation [22, 36]. Several educators have proposed "apprenticeship" and "case study" models, where students learn to program by first reading and modifying programs that have been written by experts [1, 18].

In addition, a number of researchers have proposed teaching design patterns directly to novices, under the moniker *pattern-oriented-instruction* (POI) [4, 8, 9, 14, 23, 24, 28, 39]. Patterns are characterizations of abstract solutions to common problems [11]. Most relevant to this paper are the efforts to identify and teach novice micro-patterns (e.g., selection, process-all-items) [2, 4, 6, 8, 19, 27]. The abstract nature of pattern abstraction, however, can be challenging for novices to understand; novices benefit from constructive rules (do this under this circumstance) more than the descriptive rules of traditional patterns [7].

It has been found that POI enhances problem solving competence [24]. Specifically, it promotes: (1) approaching a problem and formulating an idea for a solution, (2) better recognition of the problem's type, and (3) the ability to recognize subtasks and their corresponding patterns, to identify relationships between subtasks, and to construct an algorithm composed of the subtasks' solutions [24]. However, while designing patterns for CS1 students, it is extremely important to note that if patterns are too narrow or inflexible, students rarely use them [7].

While not specifically patterns, the concept of "Roles of Variables" [5, 17, 31] is closely related. This work recognizes that the usage of many variables, especially in novice code, can be characterized by a relatively small number of patterns, and that explicitly teaching students to recognize these patterns can aide them in learning to read and write code.

Table 1: Properties of courses analyzed in this work

Class	Target Population	Language	Term length
A	(non-CS) STEM majors	Python	semester
B	(non-CS) STEM majors	Python	semester
C	non-technical majors	Python	semester
D	CS majors	Java	semester
E	no prior programming exp.	Java	1st quarter
F	CS majors	Java	1st quarter
G	CS + STEM majors	C++	1st quarter
H	non-majors	Python	semester
I	no prior programming exp.	Python	semester
J	majors + non-majors	Java	semester
K	majors + non-majors	C	semester
L	majors + non-majors	Java	2nd quarter

3 METHOD

We evaluated the expectations on students to learn patterns in introductory programming courses by analyzing the courses assessment materials. We chose to focus on the assessment materials of the courses, rather than the instructional (e.g., lecture, learning objectives) materials, because the assessment materials more directly demonstrate what the students were responsible for learning. In

Table 2: Patterns counted in this work

Name	Description	Previous Usage
booleanOperatorChaining	This pattern involves using compound boolean expressions (e.g., `(x < 7) && (x > -7)` as part of a single conditional statement (e.g., `if`, `while`, `for`).	No previous citations
multiWayBranching	This pattern relates to the appropriate use of `else if` and `else` clauses in complex conditionals	Presented among selection patterns [6]
swapping	This pattern switches the values stored in two variables by using a temporary variable	Used in previous study [42]
digitProcessing	This pattern involves accessing specific digits in a multiple-digit integer by repeatedly using the modulus operator to access the rightmost digit and then using floor division to drop the rightmost digit	Used in previous study [42]
processAllItems	This pattern involves iterating over a collection to perform an operation on each item of a collection	Presented among loop patterns [2]
sum	This pattern involves a numeric variable (often initialized to 0) that has added to it the values in a collection. Summing elements may be done conditionally.	Presented as use of a *gatherer* [31]
average	This pattern is an extension to the *sum* pattern, where the resulting value is then divided by the number of elements in the collection (potentially computed using the *counting* pattern).	No previous citations
counting	This pattern involves an integer variable (typically initialized to 0) that gets incremented under certain circumstances (typically in a loop, perhaps conditionally).	Presented as use of a *stepper* [31]
reverse	This pattern takes an ordered collection and produces the new ordered collection containing the same elements in the reversed ordered.	No previous citations
loopAndHalf	This pattern is used when processing input elements where the end isn't known ahead of time; such loops need to read *and* process each element, and, upon reaching a sentinel value (indicating the end of the input), the loop needs to be executed before processing the sentinel element	Presented among loop patterns [2]
linearSearching	This pattern is a special case of *processAllItems* where each element of a collection is checked to see if it fits a criteria and the first matching element is returned	Presented among loop patterns [2]
findBestInCollection	This pattern is a special case of *processAllItems* where an auxiliary variable is used to track the "best" value encountered so far, and each element of the collection is compared to this "best" variable, replacing it if the new value is "better"	Presented among loop patterns [2]
filterACollection	This pattern is a special case of *processAllItems* that produces a new collection by considering each element of an existing collection for inclusion in the new collection	No previous citations
privateInstanceVars	This pattern involves making class/instance variables private so that they cannot be accessed directly	standard OO encapsulation pattern
publicGetSet	This pattern involves providing public getters and setters to private class/instance variables .	standard OO encapsulation pattern

particular, we focused on collecting *instructor solutions* for each of the code writing assessments in the these courses. Because there are potentially many ways of solving code writing questions, using the instructor solutions clearly indicates the instructor's intended way of solving the problem. We focused on code writing questions and not code tracing, comprehension, or Parson's problems, because they present the clearest picture of the course's expectations for students' code writing abilities. For each course, we reviewed the course's homework assignments, quizzes, and exams. In a small number of cases, practice exams were substituted for exams when exam solutions were not available. In a few courses, lab assignments were also included.

We solicited and received assessment materials from 12 courses from a total of nine different North American universities. These courses varied in the targeted student population and programming language of instruction, as indicated by Table 1. Descriptions of the target population were provided by the instructors that provided the materials. Some of the universities were on quarters, while others were on semesters, and that is also noted in the table. Class L is a CS 2 course, but because it is a quarter length course, it overlaps with the content in some of the semester length CS 1 courses (e.g., class D), so we included it in our analysis. We did not review the instructional materials of these courses, so we do not know the extent to which they practiced pattern-oriented instruction.

The patterns that were used in this census were drawn from three sources: 1) the patterns identified in the pattern-oriented instruction literature discussed in the previous section, 2) reviewing two CS 1 textbooks and attempting to identify patterns from the code examples and homework solutions contained in the books, and 3) identifying additional patterns from the instructor solutions themselves. The authors reviewed the candidate patterns and settled

on the list presented in Table 2 and discussed in Section 4 for this study.

After reviewing a number of code examples to ensure that the authors agreed on which patterns were present, one researcher (Iyer) manually examined each of the instructor solutions to identify the patterns present. The number of instances of each pattern was recorded at the granularity of an assessment (e.g., a homework, quiz, or exam; not individual questions). These records were sorted chronologically for the purpose of the analysis. The analysis of this data is presented in Section 5.

4 CATALOG OF PATTERNS

In this section, we describe the set of patterns that were counted as part of this census. The pattern names, descriptions, and whether the pattern previously appeared in the research literature is indicated in Table 2. There are (in order) two conditional patterns, one data movement pattern, ten repetition patterns, and two object patterns. Longer descriptions and examples of the patterns can be found in Iyer's M.S. thesis [16].

While most of the patterns are straightforward, a few benefit from further explanation. *digitProcessing* is sometimes introduced as an application of loops that doesn't require collections in languages where loops are taught before arrays (i.e., Java/C++). The *reverse* pattern seems to be commonly introduced in the context of strings. While we understand that some might not consider *privateInstanceVars* and *publicGetSet* as patterns per se, but since they meet our criteria of being logical constructs involving collections of syntactic elements, we include them for completeness.

5 RESULTS

Table 3 indicates which of the patterns were present in each of the classes we studied. It can be seen that there is significant variation between the courses, but there are some patterns that students must learn in almost all of the courses. Both branch patterns—*booleanOperatorChaining*, *multiWayBranching*—were found in 11 of the 12 classes, which is not surprising because they are commonly used even in simple programs. Seven of the loop patterns—*processAllItems*, *sum*, *average*, *counting*, *linearSearching*, *findBestInCollection*, and *filterACollection*—were found in at least 9 of the 12 classes. Again, some of these are not surprising since computing totals and averages and counting things are common novice tasks, but we found it surprising that *filterACollection* was so prevalent given that it was not included in previous inventories of elementary patterns.

In the table, we've organized the courses by language of instruction as a way of visualizing whether there is a significant language dependence in which patterns are taught. We don't see evidence of a significant language dependence, with two exceptions. First, as previously noted, the *digitProcessing* pattern is taught in Java/C++ where loops are typically taught before arrays, but not in Python where lists are often introduced before loops. Second, the object patterns were only observed in Java; this result is consistent with Java courses being more likely to emphasize object-orientation concepts than the other languages in this study.

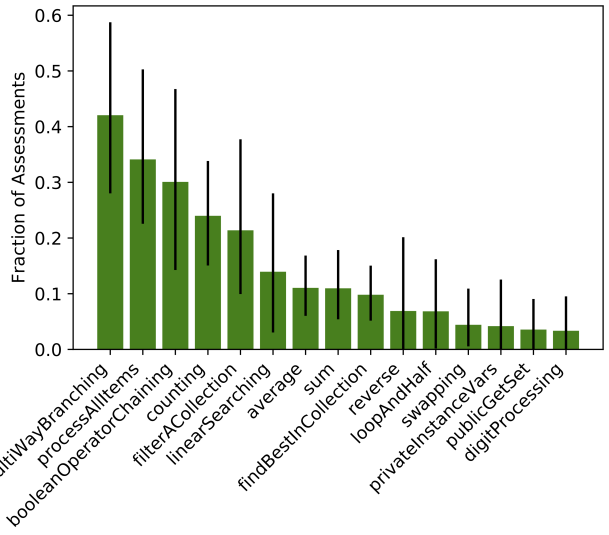

Figure 1: The fraction of assessments in a given course that include each pattern averaged across all courses with 95% confidence interval. The patterns are sorted in descending order.

5.1 Relative Pattern Frequency

Figure 1 shows the average fraction of assessments in which a particular pattern was observed. On the x-axis we list the patterns from most frequently occurring to least frequently occurring. The y-axis denotes the average fraction of assessments in a class that include that pattern; recall that we're treating each homework and exam, which might consist of multiple code writing activities, as a single assessment in our data. We computed for each course the fraction of its assessments that included a given pattern and then averaged these over all of the classes. 95% confidence intervals were computed using a bootstrap with 200 samples.

For the most part, we find these results to make sense. It isn't surprising for the two conditional patterns to be among the top three along with *Process All Items*, as it can be difficult to write non-trivial programs that don't include these patterns. In contrast, patterns like *swapping*, *reverse*, and *digitProcessing* are more gimmicky patterns that are often employed for pedagogical reasons orthogonal to their actual usage frequency in code.

We think the most surprising results are that the *counting* and *filterACollection* patterns are among the most common, even more common than *linearSearching*, *sum*, and *findBestInCollection*. Again, the *filterACollection* pattern was not included in previous elementary pattern inventories. We would have expected *linearSearching*, *sum*, and *findBestInCollection* to be more frequent than they were relative to the other patterns.

5.2 Order of Pattern Appearance

Figure 2 plots the point in the progression of assessments in each course where each pattern first occurs. We present this data to see empirically what it suggests about the order that these patterns should be introduced.

Table 3: The courses typically covered different subsets of the patterns. Columns represent the classes we examined (grouped by language) and rows indicate the patterns that appeared at least once in the course's assessments.

	Python					Java					C++	C
	Class A	Class B	Class C	Class H	Class I	Class D	Class E	Class F	Class J	Class L	Class G	Class K
booleanOperatorChaining	X	X		X	X	X	X	X	X	X	X	X
multiWayBranching	X	X	X	X	X	X	X	X	X	X	X	X
swapping	X					X					X	
digitProcessing						X				X	X	
processAllItems	X	X	X	X	X	X	X	X		X	X	
sum	X	X	X	X	X	X	X	X		X		
average	X	X	X	X	X		X	X	X		X	
counting	X	X	X	X		X	X	X	X	X	X	X
reverse	X										X	
loopAndHalf			X		X	X				X	X	
linearSearching	X	X		X	X	X		X	X	X	X	X
findBestInCollection	X	X		X	X	X	X	X	X	X	X	
filterACollection	X	X	X	X	X	X		X		X	X	
privateInstanceVars						X			X	X		
publicGetSet						X	X		X			

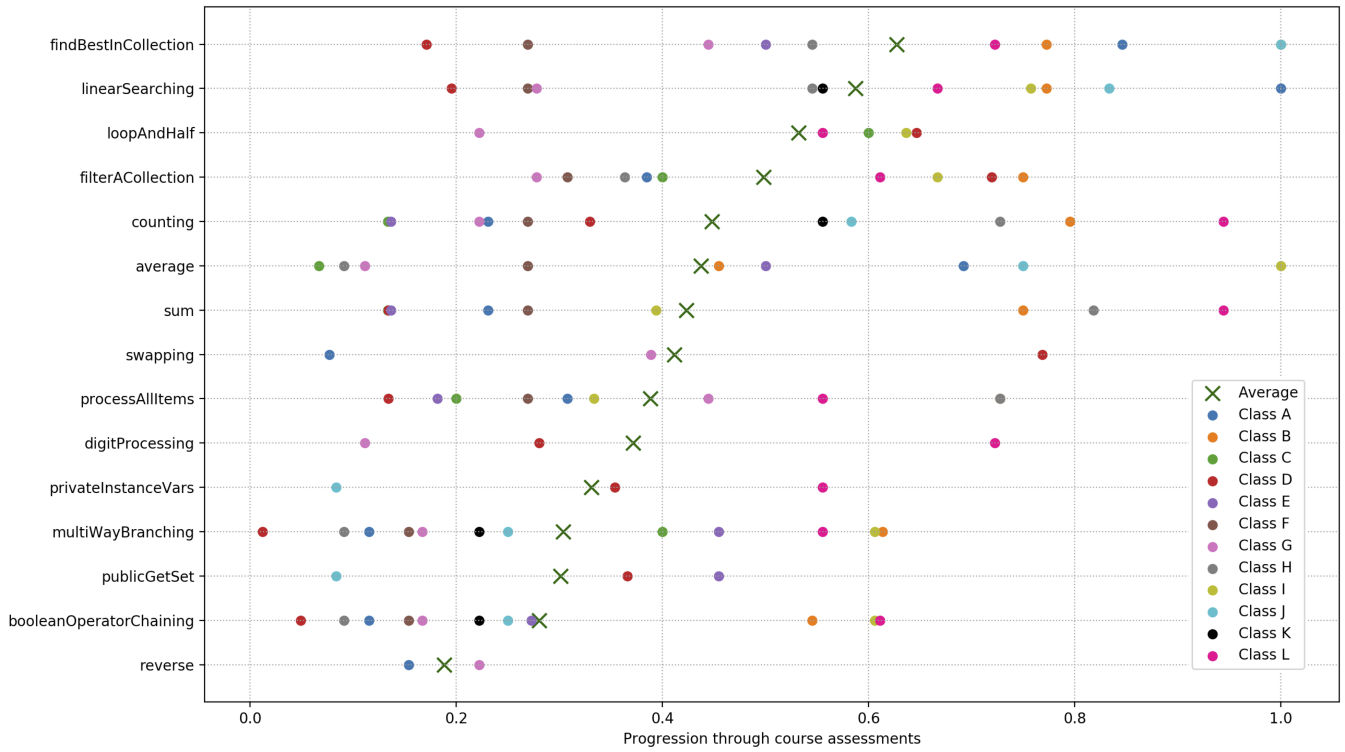

Figure 2: The order in which patterns appear for the first time in each course. The times (x-axis) plotted assuming that the assessments in the course are uniformly distributed throughout the term. The patterns are sorted (from bottom to top, y-axis) by the average point in the term when they were introduced among the classes in which they were observed.

To plot this data, we found the first assessment in each course where a given pattern appeared. We then needed to determine the position in the term at which that assessment occurred. Since we didn't have that information from the instructors that provided their instructor solutions, we made a simplifying assumption. We assumed that the assessments were equally spaced throughout the term. Thus, if there are 36 total assessments in a course and a particular pattern shows up in assessment number 12, then we would indicate that that pattern occurred (12/36) or 33% through the term.

The fact that we had both quarter-length courses (including one CS 2) and semester-length courses further complicates this

analysis. Again, we made a simplifying assumption, which was that 1st quarter classes ran during the first half of semester length courses and that 2nd quarter classes ran during the second half of semester length courses. Obviously this is a gross approximation, but we found these results to be pretty insensitive to the particular assumptions that we made.

Overall, we find most of the results to be not surprising, at least in hindsight. First, the two branch patterns are among the earliest occurring patterns. In addition, the two object patterns occur early in the term for the courses that teach them. The *digitProcessing* pattern generally occurs before the other loop patterns, as it is being used as an application of loops before arrays are taught; recall that Class L is the second quarter course that somewhat artificially pulls this pattern later in the term on average. The *reverse* pattern also occurs early in the classes in which it occurs, which is consistent with it being used shortly after both loops and strings have been introduced.

The later patterns are dominated by the loop patterns, and in general the more sophisticated patterns are later. We see *sum*, *average*, and *counting* occur before *filterACollection*, *loopAndHalf*, *linearSearching*, and *findBestInCollection*. Perhaps most surprising is the apparent lack of agreement of when these later patterns should be introduced, as some of them range from being used in the first 30% of the term in some courses to being first used in the last 30% of the term in other courses.

6 THREATS TO VALIDITY

By far the largest threat to validity of this work is in how the courses studied were selected. Our selection was not particularly well controlled. The courses came from two sources: 1) faculty that the authors had a relationship with that we contacted directly, and 2) respondents from a request on the SIGCSE-members mailing list. While these courses had a significant diversity in a number of dimensions, there is no reason to believe that this small sample is necessarily representative of all of the courses taught even just in North America.

Additionally, since there is not yet a consensus about what the important introductory patterns are, our analysis might have missed some important patterns. While we did our best to remain observant throughout the process to note any recurring idioms, what is an introductory pattern is likely very much in the "eye of the beholder". The bulk of the patterns studied, however, are found in previous literature, which suggests that there would be significant overlap with an independent investigation.

7 CONCLUSION

Learning idiomatic chunks of programming language syntax, what we refer to in this paper as patterns, enables novice programmers to work at a higher level of abstraction and, thus, have lower cognitive load when doing novel problem solving. As such, we believe strongly that these patterns should be explicitly taught to our students.

In this paper, we explore to what degree we can identify what patterns should be taught in early programming courses and in what order. We find that there is significant agreement among the courses we studied about which patterns should be learned, and that most of this consensus is language independent. In contrast, while we find meaning in observed average order that patterns are introduced, there is high variance between courses when students are responsible for demonstrating having learned these patterns. As such, we feel that identifying best practices around how to sequence patterns in pattern-oriented instruction remains an important open research area.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. DUE-1915257.

REFERENCES

[1] Owen Astrachan and David Reed. 1995. AAA and CS 1: The Applied Apprenticeship Approach to CS 1. In *Proceedings of the Twenty-Sixth SIGCSE Technical Symposium on Computer Science Education* (Nashville, Tennessee, USA) *(SIGCSE '95)*. Association for Computing Machinery, New York, NY, USA, 1–5. https://doi.org/10.1145/199688.199694

[2] Owen Astrachan and Eugene Wallingford. 1998. *Loop Patterns*. https://users.cs.duke.edu/~ola/patterns/plopd/loops.html

[3] Maria Bannert. 2002. Managing cognitive load—recent trends in cognitive load theory. *Learning and Instruction* 12, 1 (2002), 139 – 146. https://doi.org/10.1016/S0959-4752(01)00021-4

[4] Walter Beck, S. Rebecca Thomas, Janet Drake, J. Philip East, and Eugene Wallingford. 1996. Pattern Based Programming Instruction. In *1996 Annual Conference*. ASEE Conferences, Washington, District of Columbia. https://peer.asee.org/6228.

[5] Mordechai Ben-Ari and Jorma Sajaniemi. 2004. Roles of Variables as Seen by CS Educators. In *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Leeds, United Kingdom) *(ITiCSE '04)*. Association for Computing Machinery, New York, NY, USA, 52–56. https://doi.org/10.1145/1007996.1008013

[6] Joseph Bergin. 1998. *Patterns for Selection*. http://csis.pace.edu/~bergin/patterns/Patternsv5.html

[7] Michael J. Clancy and Marcia C. Linn. 1999. Patterns and Pedagogy. In *The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education* (New Orleans, Louisiana, USA) *(SIGCSE '99)*. ACM, New York, NY, USA, 37–42. https://doi.org/10.1145/299649.299673

[8] Michael de Raadt. 2008. *Teaching programming strategies explicitly to novice programmers*. Ph.D. Dissertation. University of Southern Queensland.

[9] Michael de Raadt, Richard Watson, and Mark Toleman. 2009. Teaching and Assessing Programming Strategies Explicitly. In *Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95* (Wellington, New Zealand) *(ACE '09)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 45–54. http://dl.acm.org/citation.cfm?id=1862712.1862723

[10] Max Fowler, Binglin Chen, Sush Azad, Matthew West, and Craig Zilles. 2021. Autograding "Explain in Plain English" questions using NLP. In *Proceedings of the SIGCSE Technical Symposium (SIGCSE)*.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1993. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming*. Springer, 406–431.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software* (1 ed.). Addison-Wesley Professional. http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1

[13] Fernand Gobet, Peter CR Lane, Steve Croker, Peter CH Cheng, Gary Jones, Iain Oliver, and Julian M Pine. 2001. Chunking mechanisms in human learning. *Trends in cognitive sciences* 5, 6 (2001), 236–243.

[14] Jacqueline Hundley. 2008. A Review of Using Design Patterns in CS1. In *Proceedings of the 46th Annual Southeast Regional Conference* (Auburn, Alabama) *(ACM-SE 46)*. ACM, New York, NY, USA, 30–33. https://doi.org/10.1145/1593105.1593113

[15] Michelle Ichinco and Caitlin Kelleher. 2017. Towards better code snippets: Exploring how code snippet recall differs with programming experience. In *Visual Languages and Human-Centric Computing (VL/HCC), 2017 IEEE Symposium on*. IEEE, 37–41.

[16] Vighnesh Iyer. 2020. *An analysis of pattern usage in CS1 courses*. M.S. thesis. University of Illinois at Urbana-Champaign, Urbana, Illinois. http://hdl.handle.net/2142/108537

[17] Marja Kuittinen and Jorma Sajaniemi. 2004. Teaching Roles of Variables in Elementary Programming Courses. In *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Leeds, United Kingdom) *(ITiCSE '04)*. ACM, New York, NY, USA, 57–61. https://doi.org/10.1145/1007996.1008014

[18] Marcia C. Linn and Michael J. Clancy. 1992. The Case for Case Studies of Programming Problems. *Commun. ACM* 35, 3 (March 1992), 121–132. https://doi.org/10.1145/131295.131301

[19] Torben Lorenzen, Lee Mondshein, Abdul Sattar, and Seikyung Jung. 2012. A Code Snippet Library for CS1. *ACM Inroads* 3, 1 (March 2012), 41–45. https://doi.org/10.1145/2077808.2077822

[20] Sandra P Marshall. 1995. *Schemas in problem solving*. Cambridge University Press.

[21] Katherine B McKeithen, Judith Spencer Reitman, Henry H Rueter, and Stephen C Hirtle. 1981. Knowledge organization and skill differences in computer programmers. *Cognitive Psychology* 13, 3 (1981), 307–325.

[22] Jeroen J. G. Van Merriënboer. 1990. Strategies for Programming Instruction in High School: Program Completion vs. Program Generation. *Journal of Educational Computing Research* 6, 3 (1990), 265–285. https://doi.org/10.2190/4NK5-17L7-TWQV-1EHL arXiv:https://doi.org/10.2190/4NK5-17L7-TWQV-1EHL

[23] Orna Muller. 2005. Pattern Oriented Instruction and the Enhancement of Analogical Reasoning. In *Proceedings of the First International Workshop on Computing Education Research* (Seattle, WA, USA) *(ICER '05)*. ACM, New York, NY, USA, 57–67. https://doi.org/10.1145/1089786.1089792

[24] Orna Muller, David Ginat, and Bruria Haberman. 2007. Pattern-oriented Instruction and Its Influence on Problem Decomposition and Solution Construction. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Dundee, Scotland) *(ITiCSE '07)*. ACM, New York, NY, USA, 151–155. https://doi.org/10.1145/1268784.1268830

[25] Fred Paas, Alexander Renkl, and John Sweller. 2003. Cognitive Load Theory and Instructional Design: Recent Developments. *Educational Psychologist* 38, 1 (2003), 1–4.

[26] Nancy Pennington. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology* 19, 3 (1987), 295–341.

[27] Viera K Proulx. 2000. Programming patterns and design patterns in the introductory computer science course. *ACM SIGCSE Bulletin* 32, 1 (2000), 80–84.

[28] Michael Raadt, Mark Toleman, and Richard Watson. 2007. Incorporating programming strategies explicitly into curricula. In *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*.

[29] Robert S Rist. 1989. Schema creation in programming. *Cognitive Science* 13, 3 (1989), 389–414.

[30] Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. Learning and Teaching Programming: A Review and Discussion. *Computer Science Education* 13, 2 (2003), 137–172. https://doi.org/10.1076/csed.13.2.137.14200 arXiv:https://doi.org/10.1076/csed.13.2.137.14200

[31] Jorma Sajaniemi. 2008. The Roles of Variables Home Page.

[32] Ben Shneiderman and Richard Mayer. 1979. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences* 8, 3 (01 Jun 1979), 219–238. https://doi.org/10.1007/BF00977789

[33] John Sweller. 2011. Cognitive Load Theory. In *Psychology of learning and motivation*. Vol. 55. Elsevier, 37–76.

[34] John Sweller, Paul Ayres, and Slava Kalyuga. 2011. Cognitive Load Theory, volume 1 of Explorations in the Learning Sciences, Instructional Systems and Performance Technologies.

[35] John Sweller, Jeroen JG Van Merrienboer, and Fred GWC Paas. 1998. Cognitive architecture and instructional design. *Educational psychology review* 10, 3 (1998), 251–296.

[36] Jeroen JG Van Merriënboer and Marcel BM De Croock. 1992. Strategies for computer-based programming instruction: Program completion vs. program generation. *Journal of Educational Computing Research* 8, 3 (1992), 365–394.

[37] Jeroen JG Van Merrienboer and Fred GWC Paas. 1990. Automation and schema acquisition in learning elementary computer programming: Implications for the design of practice. *Computers in Human Behavior* 6, 3 (1990), 273–289.

[38] Iris Vessey. 1985. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies* 23, 5 (1985), 459 – 494. https://doi.org/10.1016/S0020-7373(85)80054-7

[39] Eugene Wallingford. 1996. Toward a First Course Based on Object-oriented Patterns. In *Proceedings of the Twenty-seventh SIGCSE Technical Symposium on Computer Science Education* (Philadelphia, Pennsylvania, USA) *(SIGCSE '96)*. ACM, New York, NY, USA, 27–31. https://doi.org/10.1145/236452.236485

[40] Susan Wiedenbeck. 1985. Novice/expert differences in programming skills. *International Journal of Man-Machine Studies* 23, 4 (1985), 383 – 390. https://doi.org/10.1016/S0020-7373(85)80041-9

[41] Leon E. Winslow. 1996. Programming Pedagogy&Mdash;a Psychological Overview. *SIGCSE Bull.* 28, 3 (Sept. 1996), 17–22. https://doi.org/10.1145/234867.234872

[42] Benjamin Xie, Dastyni Loksa, Greg L. Nelson, Matthew J. Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li, and Andrew J. Ko. 2019. A theory of instruction for introductory programming skills. *Computer Science Education* 29, 2-3 (2019), 205–253. https://doi.org/10.1080/08993408.2019.1565235 arXiv:https://doi.org/10.1080/08993408.2019.1565235