Introductory Programming: A Systematic Literature Review

Andrew Luxton-Reilly
University of Auckland
New Zealand
andrew@cs.auckland.ac.nz

Simon
University of Newcastle
Australia
simon@newcastle.edu.au

Ibrahim Albluwi
Princeton University
United States of America
isma@cs.princeton.edu

Brett A. Becker
University College Dublin
Ireland
brett.becker@ucd.ie

Michail Giannakos
Norwegian University of Science and Technology
Norway
michailg@ntnu.no

Amruth N. Kumar
Ramapo College of New Jersey
United States of America
amruth@ramapo.edu

Linda Ott
Michigan Technological University
United States of America
linda@mtu.edu

James Paterson
Glasgow Caledonian University
United Kingdom
james.paterson@gcu.ac.uk

Michael James Scott
Falmouth University
United Kingdom
michael.scott@falmouth.ac.uk

Judy Sheard
Monash University
Australia
judy.sheard@monash.edu

Claudia Szabo
University of Adelaide
Australia
claudia.szabo@adelaide.edu.au

ABSTRACT

As computing becomes a mainstream discipline embedded in the school curriculum and acts as an enabler for an increasing range of academic disciplines in higher education, the literature on introductory programming is growing. Although there have been several reviews that focus on specific aspects of introductory programming, there has been no broad overview of the literature exploring recent trends across the breadth of introductory programming.

This paper is the report of an ITiCSE working group that conducted a systematic review in order to gain an overview of the introductory programming literature. Partitioning the literature into papers addressing the student, teaching, the curriculum, and assessment, we explore trends, highlight advances in knowledge over the past 15 years, and indicate possible directions for future research.

CCS CONCEPTS

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

KEYWORDS

ITiCSE working group; CS1; introductory programming; novice programming; systematic literature review; systematic review; literature review; review; SLR; overview

ACM Reference Format:
Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Giannakos, Amruth N. Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory Programming: A Systematic Literature Review. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE Companion '18), July 2–4, 2018, Larnaca, Cyprus.* ACM, New York, NY, USA, 52 pages. https://doi.org/10.1145/3293881.3295779

ITiCSE Companion '18, July 2–4, 2018, Larnaca, Cyprus

ACM ISBN 978-1-4503-6223-8/18/07…$15.00
https://doi.org/10.1145/3293881.3295779

1 INTRODUCTION

Teaching students to program is a complex process. A 2003 review by Robins et al. [554] provided a comprehensive overview of novice programming research prior to that year. The first paragraph of the review sets the general tone:

> *Learning to program is hard […] Novice programmers suffer from a wide range of difficulties and deficits. Programming courses are generally regarded as difficult, and often have the highest dropout rates. [554, p137]*

However, more recent studies have suggested that the situation is not as dire as previously suggested. Studies indicate that dropout rates among computing students are not alarmingly high [64, 692], and it has been suggested that the difficulties faced by novices may be a consequence of unrealistic expectations rather than intrinsic subject complexity [393]. Better outcomes are likely to arise from focusing less on student deficits and more on actions that the computing community can take to improve teaching practice. In this paper we investigate the literature related to introductory

programming and summarise the main findings and challenges for the computing community.

Although there have been several reviews of published work involving novice programmers since 2003, those reviews have generally focused on highly specific aspects, such as student misconceptions [530], teaching approaches [679], program comprehension [578], potentially seminal papers [502], research methods applied [598], automated feedback for exercises [321], competency-enhancing games [675], student anxiety [478], and program visualisation [631].

A review conducted contemporaneously with our own, by Medeiros et al. [436], is somewhat broader in scope than those mentioned above, but not as broad as our own. It investigates the skills and background that best prepare a student for programming, the difficulties encountered by novice programmers, and the challenges faced by their instructors.

2 SCOPE

We review papers published between 2003 and 2017 inclusive. Publications outside this range are not included in the formal analysis, but may be included in discussion where appropriate.

In selecting papers for review, we make a clear distinction between those involving introductory programming — the focus of our review — and those about other aspects of introductory computing. For example, the literature of computing includes many papers on aspects of computational thinking. This review addresses such papers only where they where they have a clear focus on introductory programming.

We have limited our scope to units of teaching corresponding to introductory programming courses, thus ruling out shorter and less formal units such as boot camps and other outreach activities. As it became apparent that we needed to reduce the scope still further, we also excluded work on introductory programming courses at school level (also known as K–12) and work explicitly concerning introductory computing courses for non-computing students (also known as non-majors). Some papers in these areas are still included in our discussion, but only if they contribute to our principal focus on introductory programming courses for students in computing degrees.

As recommended by an ITiCSE working group on worldwide terminology in computing education [609], this report tends in general to avoid the term 'computer science', preferring instead the less restrictive term 'computing'.

3 METHOD

The working group conducted a systematic literature review by adapting the guidelines proposed by Kitchenham [335]. In this review, we followed a highly structured process that involved:

(1) Specifying research questions
(2) Conducting searches of databases
(3) Selecting studies
(4) Filtering the studies by evaluating their pertinence
(5) Extracting data
(6) Synthesising the results
(7) Writing the review report

3.1 Research Questions

This review aims to explore the literature of introductory programming by identifying publications that are of interest to the computing community, the contributions of these publications, and the evidence for any research findings that they report. The specific research questions are:

RQ1 What aspects of introductory programming have been the focus of the literature?
RQ2 What developments have been reported in introductory programming education between 2003 and 2017?
RQ3 What evidence has been reported when addressing different aspects of introductory programming?

3.2 Conducting Searches

Selecting search terms for a broad and inclusive review of introductory literature proved challenging. Terms that are too general result in an unwieldy set of papers, while terms that are too specific are likely to miss relevant papers. After some trial and error with a range of databases, we selected a combined search phrase that seemed to capture the area of interest:

> "introductory programming" OR "introduction to programming" OR "novice programming" OR "novice programmers" OR "CS1" OR "CS 1" OR "learn programming" OR "learning to program" OR "teach programming"

To check whether this search phrase was appropriate, we applied it to a trial set of papers and compared the outcome with our own thoughts as to which papers from that set would fall within the scope of our review. We chose the papers from the proceedings of ICER 2017 and ITiCSE 2017, 94 papers in all. Ten members of the working group individually decided whether each paper was relevant to the review. The members then formed five pairs, discussed any differences, and resolved them.

The inter-rater reliability of this process was measured with the Fleiss-Davies kappa [148], which measures the agreement when a fixed set of raters classify a number of items into a fixed set of categories. In principle we were classifying the papers into just two categories, yes and no, but some members were unable to make this decision for some papers, introducing a third category of undecided. The Fleiss-Davies kappa for individual classification was 61%. It has been observed [607] that classification in pairs is more reliable than individual classification, and this was borne out with our paired classification, which resulted in a Fleiss-Davies kappa of 73% — and the disappearance of the undecided category.

When measuring inter-rater reliability, an agreement of less than 40% is generally considered to be poor, between 40% and 75% is considered fair to good, and more than 75% is rated excellent [45]. By this criterion, our individual agreement was good and our paired agreement was substantially better. This process resulted in the selection of 29 papers, those that a majority of pairs agreed were pertinent to our review.

We then automatically applied the search terms to the same set of 94 papers, resulting in a selection of 32 papers: 25 of the 29 that we had selected and seven false positives, papers that were indicated by the search terms but not selected by us. This proportion of false positives was not a major concern, because every selected

paper was going to be examined by at least one member of the team and could be eliminated at that point. There were also four false negatives, papers that we deemed relevant but that were not identified by the search terms. False negatives are of greater concern because they represent relevant papers that will not be identified by the search; but, unable to find a better combination of search terms, we accepted that our search might fail to identify some 15% of pertinent papers.

The search terms were then applied to the title, abstract, and keyword fields of the ACM Full Text Collection, IEEE Explore, ScienceDirect, SpringerLink and Scopus databases. The search was conducted on 27 May 2018, and identified the following numbers of papers:

- ACM Full text collection: 2199
- IEEE Explore: 710
- ScienceDirect (Elsevier): 469
- SpringerLink (most relevant 1000): 1000
- Scopus (most relevant 2000): 2000; 678 after removal of duplicates
- Total: 5056

3.3 Selecting Studies

The next stage of a systematic review is to select the papers that will form the basis for the review. The search results were divided among the authors, who examined each title and abstract, and the corresponding full paper if required, to determine its relevance to the review. We eliminated papers that were irrelevant, papers that were less than four pages long (such as posters), and papers that were clearly identified as work in progress. The biggest reductions were seen in the more general ScienceDirect and SpringerLink databases, where, for example, 'CS1' can refer to conditioned stimulus 1 in a behavioural studies paper, cesium 1 in a paper on molecular structures, and connecting segment 1 in a paper on pathology. This process reduced the pool of papers by more than half, as shown below.

- ACM Full text collection: 1126 (51%)
- IEEE Explore: 448 (63%)
- ScienceDirect (Elsevier): 62 (13%)
- SpringerLink (most relevant 1000): 204 (20%)
- Scopus: 349 (51%)
- Total: 2189 (43%)

3.4 Filtering and Data Analysis

Following the selection of papers, the team collectively devised a set of topics that might cover the papers we had been seeing. This began with a brainstormed list of topics, which was then refined and rationalised. The topics were then gathered into four high-level groups: the student, teaching, curriculum, and assessment. The first three of these groups have together been called the 'didactic triangle' of teaching [70, 314]. While it is not one of these three core elements, assessment is a clear link among them, being set by the teacher and used to assess the student's grasp of the curriculum.

The 2189 papers were divided among the authors, each of whom classified approximately 200 papers using the abstract and, where necessary, the full text of the paper. During this phase some 500 further papers were excluded upon perusal of their full text and some 25 papers because the research team was unable to access them. The remaining 1666 papers were classified into at least one and often several of the categories. This classifying, or tagging, constituted the first phase of the analysis: the data extracted from each paper were the groups and subgroups into which the paper appeared to fit.

Small groups then focused on particular topics to undertake the remaining steps of the systematic process: evaluating the pertinence of the papers, extracting the relevant data, synthesising the results, and writing the report. The data extracted at this stage were brief summaries of the points of interest of each paper as pertaining to the topics under consideration. As the groups examined each candidate paper in more depth, a few papers were reclassified by consensus, and other papers were eliminated from the review. At the completion of this phase, some of the initial topics were removed because we had found few or no papers on them (for example, competencies in the curriculum group), and one or two new topics had emerged from examination of the papers (for example, course orientation in the teaching group).

In a systematic literature review conducted according to Kitchenham's guidelines [335], candidate papers would at this point have been filtered according to quality. This process was followed in a limited manner: as indicated in the two foregoing paragraphs, some papers were eliminated upon initial perusal and others upon closer examination. However, our focus was more on the pertinence of papers to our subject area than on their inherent quality, so at this stage we could be said to have deviated somewhat from Kitchenham's guidelines.

A further deviation from Kitchenham's guidelines arises from the number of papers identified by our search. It would be impractical to list every paper that has addressed every topic, and even more impractical to discuss any papers in depth. Therefore our intent is to give a thorough view of the topics that have been discussed in the literature, referring to a sample of the papers that have covered each topic. Except where the context suggests otherwise, every reference in the following sections should be understood as preceded by an implicit 'for example'.

4 OVERVIEW OF INTRODUCTORY PROGRAMMING RESEARCH

Table 1 shows the number of papers in each group and the subgroups into which some or all of these papers were classified. The majority of papers fall into the teaching category, most of them describing either teaching tools or the various forms of delivery that have been explored in the classroom. A substantial number of papers focus on the students themselves. We see fewer papers that discuss course content or the competencies that students acquire through the study of programming. The smallest of the broad topic areas is assessment, which is interesting since assessment is such a critical component of courses and typically drives both teaching and learning.

Given the comprehensive nature of this review, it is inevitable that some papers will be discussed in more than one section. For example, a paper exploring how students use a Facebook group to supplement their interactions in the university's learning management system [408] is discussed under both student behaviour

Table 1: Initial classification of 1666 papers, some classified into two or more groups or subgroups

Group	Papers	Optional subgroups
The student	489	student learning, underrepresented groups, student attitudes, student behaviour, student engagement, student ability, the student experience, code reading, tracing, writing, and debugging
Teaching	905	teaching tools, pedagogical approaches, theories of learning, infrastructure
Curriculum	258	competencies, programming languages, paradigms
Assessment	192	assessment tools, approaches to assessment, feedback on assessment, academic integrity

(section 5.1.3) and teaching infrastructure (section 6.5). While further rationalisation might have been possible, we consider that readers are best served by a structure that considers broad categories and then surveys the papers relevant to each, even if that entails some duplication.

Figure 1 shows the number of papers that we identified in the data set, arranged by year. It is clear that the number of publications about introductory programming courses is increasing over time.

To check whether introductory programming is a growing focus specifically in the ACM SIGCSE conferences, we counted the papers in our data set that were published each year in ICER (which began in 2005), ITiCSE, or the SIGCSE Technical Symposium, and compared this with the total number of papers published each year in those three conferences. Figure 2 shows that publication numbers in the three main ACM SIGCSE conferences remain fairly stable between 2005 and 2017. Publications from these venues focusing on introductory programming, although somewhat variable, also remain relatively stable across the same period. We conclude that the main growth in publications is occurring outside SIGCSE venues, which might indicate that programming education is of growing interest to the broader research community. Alternatively, it might indicate that authors are seeking more venues because there is no growth in the numbers of papers accepted by the SIGCSE venues.

5 THE STUDENT

This section explores publications that focus primarily on the student. This includes work on learning disciplinary content knowledge, student perceptions and experiences of introductory programming, and identifiable subgroups of students studying programming. Table 2 gives an overview of the categories and corresponding numbers of papers. The sum of the numbers in the table does not match the number in Table 1 because some papers were classified into more than one category.

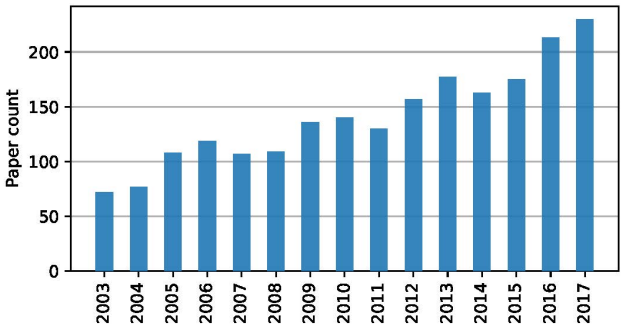

Figure 1: Introductory programming publications identified by our search

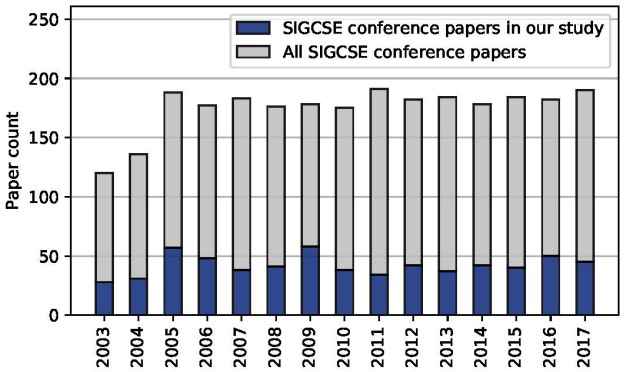

Figure 2: Introductory programming publications identified by our search and published in ICER, ITiCSE or SIGCSE, compared with total publications in ICER, ITiCSE and SIGCSE

Table 2: Classification of papers focused on students

Category	N	Description
Content		
– Theory	17	Models of student understanding
– Literacy	58	Code reading, writing, debugging
– Behaviour	69	Measurements of student activity
– Ability	169	Measuring student ability
Sentiment		
– Attitudes	105	Student attitudes
– Engagement	61	Measuring/improving engagement
– Experience	18	Experiences of programming
Subgroups		
– At risk	17	Students at risk of failing
– Underrep.	25	Women and minorities

5.1 Content

The categories in this section relate to measuring what students learn and how they learn it. We begin by considering work that applies a cognitive lens to student understanding. We then move to publications on what we term code literacy (i.e., reading, writing, and debugging of code), before moving to measurable student behaviour. The section concludes by considering broad ways that student ability is addressed in research. Figure 3 illustrates the growth of papers focusing on the interaction between students and the content taught in introductory programming courses.

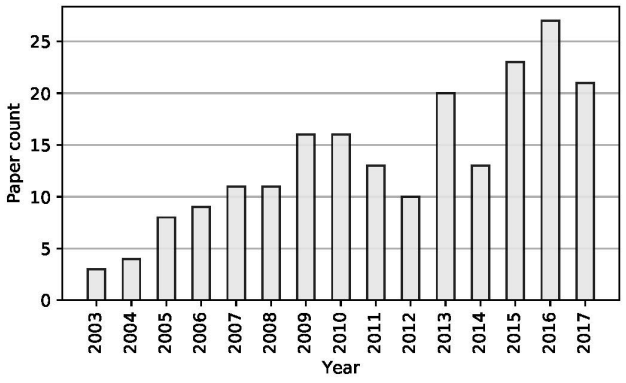

Figure 3: Number of publications focusing on interaction between students and the content — theory, literacy, behaviour and ability — by year

5.1.1 *Theory.*

Several papers grounded in various theoretical perspectives study the thinking processes of novice programmers. The number of papers focusing on the theoretical perspectives is relatively small (no more than 3 papers in any given year), with no discernible trend over the period of our study.

The constructivist point of view suggests that learners construct their own mental models of the phenomena they interact with [59]. Several papers have investigated novice programmers' viable and non-viable mental models of concepts such as variables [628], parameter passing [401], value and reference assignment [399], and how objects are stored in memory [627]. Students were found to hold misconceptions and non-viable mental models of these fundamental concepts even after completing their introductory programming courses. To address this issue, Sorva [627, 628] recommends the use of visualisation tools with techniques from variation theory, while Ma et al. [398, 400] recommend the use of visualisation tools with techniques from cognitive conflict theory. Interestingly, both Madison and Gifford [401] and Ma et al. [399] report that students holding non-viable mental models sometimes still manage to do well on related programming tasks, suggesting that assessment techniques beyond conventional code-writing tasks might be needed to reveal certain misconceptions.

Proposing a conceptual framework and a graphical representation that can be used to help students construct a viable mental model of program-memory interaction, Vagianou [674] argues that program-memory interaction exhibits the characteristics of a *threshold concept*, being troublesome, transformative, and potentially irreversible. Sorva [629] distinguishes between threshold concepts and *fundamental ideas*, proposing that threshold concepts act as 'portals' that transform the students' understanding, while fundamental ideas "run threadlike across a discipline and beyond". Sorva, who has conducted a comprehensive review of research on mental models, misconceptions, and threshold concepts [630], suggests that abstraction and state might be fundamental ideas while program dynamics, information hiding, and object interaction might be threshold concepts.

Lister [381] and Teague et al. [655–658] apply a neo-Piagetian perspective to explore how students reason about code. They discuss the different cognitive developmental stages of novice programmers and use these stages to explain and predict the ability or otherwise of students to perform tasks in code reading and writing. The most important pedagogical implication of this work is that instructional practice should first identify the neo-Piagetian level that students are at and then explicitly train them to reason at higher levels. They contrast this with conventional practices, where they argue that teaching often happens at a cognitive level that is higher than that of many students [381, 656–658].

Due to the qualitative nature of research done both on mental models and on neo-Piagetian cognitive stages, more work is needed to quantitatively study what has been observed. For example, it is still not clear how widespread the observed mental models are or which neo-Piagetian cognitive stages are more prevalent among novice programmers in different courses or at different times in the same course. The small numbers of participants in these qualitative studies suggest the need for more studies that replicate, validate, and expand them.

5.1.2 *Code 'Literacy'.*

Literacy, a term traditionally applied to the reading and writing of natural language, is concerned with making sense of the world and communicating effectively. In modern usage this has broadened to refer to knowledge and competency in a specific area, for example, 'computer literacy'. Here, however, we apply the term in the traditional sense to coding, using it to mean how students make sense of code and how they communicate solutions to problems by writing executable programs. We distinguish between reading and writing, and consider research that seeks insights into the students' processes in each.

Code reading and tracing. The process of reading programs is essential both in learning to program and in the practice of programming by experts. We found 28 papers reporting on issues related to students' code reading. These included papers where the reading process involved tracing the way a computer would execute the program, which adds a significant dimension to 'making sense' that is not present in the largely linear process of reading natural-language text.

A number of papers study the reading process in order to gain insight into students' program comprehension, for example by relating reading behaviour to well-known program comprehension models [13] or the education-focused block model [702]. There has been recent interest in the application of eye-tracking techniques to novice programmers, for example to study changes in reading

process as students progress through a course and to identify clusters of students with similar learning paths [499, 721]. Although useful work has been done, the process of reading code, and the way this process changes as a novice programmer gains expertise, are not well understood.

There have been major studies of student code-reading and tracing skills, for example by Lister et al. [382], which have raised concerns about weaknesses in many students' abilities. The relationship between code reading and other skills, notably code writing, has also been widely studied [383, 388, 562, 611], leading to the conclusion that code-reading skills are prerequisite for problem-solving activities including code writing. Given this, there is a strong incentive to help students to develop reading skills, and a number of papers focus on the effects on writing skills of specific activities that are designed to provide practice in reading [142, 644]. Tools have been developed to support students in code reading by helping them to identify 'beacons' in code [369] and to help teachers to target support by visualising students' code-tracing processes [126]. Given the evidence that has been found for the value of code-reading skills in novice programmers, there is an ongoing need to explore further ways of encouraging the development of these skills.

Code writing and debugging. Papers in this category focus on students' ability to write programs, how they approach the creation of programs, and ways of supporting the process. Creating programs involves writing code and debugging that code in order to reach a working solution. We found 48 papers focusing on aspects of writing and debugging code.

The McCracken report from an ITiCSE working group in 2001 [426], which raised concerns about students' skills at the end of the introductory programming course, has been influential in the study of student code-writing skills during the years covered by this review. Utting et al. revisited the topic in a subsequent ITiCSE working group [673], and discovered a closer match between teachers' expectations and student skills than the original study. They noted as a possible reason that teachers' expectations have lowered, possibly as a consequence of previous research. A number of papers focus not just on students' performance on writing tasks, but also on what this reveals about the difficulty and suitability of those tasks, such as the canonical 'rainfall problem' [357, 586, 606, 703]. The types of task that we give to students, and the way we present and scaffold these tasks, are important to students' performance and to their learning through undertaking the tasks, and insights in this area will continue to be of value.

Other papers focus on supporting students in writing tasks, generally addressing the way students deal with the errors they encounter as they write code rather than the process of planning or designing their programs. Zehetmeier et al. [729] describe teaching interventions designed to help with common errors. Compilation is an important part of the process of code writing and there has been a significant body of work related to compiler error messages, including studying the comprehensibility of standard compiler messages [667] and providing enhanced messages to help novices [56]. A number of approaches and tools have been proposed to encourage students to take a systematic approach to debugging, including a serious game [442], error messages that are pedagogically designed [411] or based on social recommendations from other students' activities [251], and programming environments that allow students to interactively ask questions about program output [339]. An ITiCSE working group [615] developed a repository of videos to help students with debugging. Bennedsen and Schulte [66] studied the impact of using the BlueJ debugger to visualise code execution, but there are actually very few papers that study how students use the debuggers built into most IDEs. These environments provide rich tools to provide insight into the execution of code, and there are open questions around their accessibility and usefulness in the early stages of learning to program.

Summary. Research in this area has advanced understanding of the ways in which students read and write code and the expectations that are realistic for teachers to hold for their ability to do so. A key finding is the importance of code-reading skills to underpin other aspects code literacy. Linguists hypothesise that we acquire writing style by reading, and there is evidence that this applies to programming as well as to natural language. Guidance has also been derived for designing appropriate code reading and writing tasks and supporting the students in these processes. For example, emerging techniques such as eye tracking promise new insights into the process of code reading and may also guide teachers in presenting code examples in a 'readable' form to enhance learning.

5.1.3 *Student Behaviour.*

Students generate a great deal of data as they go about their activities, for example in the way they interact with coding environments, learning tools, and the classroom. There is increasing interest in gathering and analysing this data in order to learn about the ways in which students behave. This section reports on the 69 papers we identified that report data and findings on student behaviour. More than half of these (36 papers) were published in the last three years (2015-2017) of the fifteen-year period covered in this review (see Figure 4). This may reflect an increasing awareness of the value of data and the emergence and wide use of machine learning and data mining, although by no means all of the papers use these techniques.

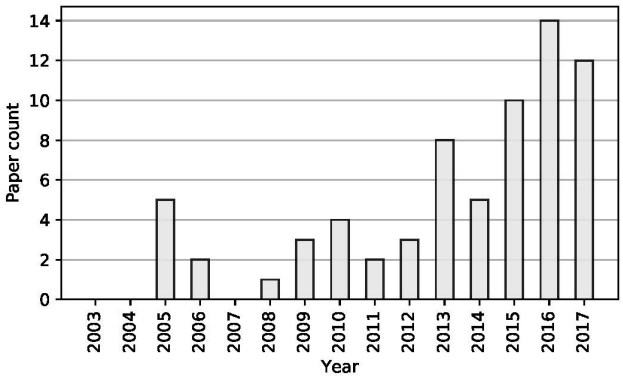

Figure 4: Papers classified as being about student behaviour have increased rapidly over the past few years

Many of these papers can be considered to describe learning analytics. Hui and Farvolden [280] recently proposed a framework to classify ways in which learning analytics can be used to shape a course, considering how data can address the instructors' needs to understand student knowledge, errors, engagement, and expectations, and how personalisation of feedback based on data can address students' needs to plan and monitor their progress. Many aspects of their framework are reflected in the themes discovered in the current review. An understanding of student behaviour is of value to educators in at least the following ways:

- Predicting success in examinations and other assessments
- Identifying student difficulties and interventions designed to mitigate them
- Designing tools that respond to specific behaviours in a way that is helpful to the students
- Encouraging students to alter their behaviour to improve their likelihood of success
- Detecting undesirable behaviour such as cheating

A previous ITiCSE working group [294] has reviewed the literature on educational data mining and learning analytics in programming (not restricted to introductory programming), describing the state of the art in collecting and sharing programming data and discussing future directions. The report from that working group complements this work by providing in-depth coverage of this specific topic.

Coding behaviour data. Students in introductory programming courses spend a lot of time creating code, but what are they actually doing when they code? Some papers describe the use of data to provide insights into how students go about and experience the process of coding, reporting analyses of observations of the students as they code or of the products of the coding.

The largest number of papers report analysis of compilation behaviour. Brown et al. [93] describe a repository of compilation data that is populated by users worldwide of the BlueJ IDE. This follows earlier work by the BlueJ team on recording compilation behaviour [299], and allows other researchers to use the tool to gather data for their own research [555]. Other researchers have embedded the capability to record similar data in the tools that their students use to write code [69, 205, 460]. In order to extract insights from compilation data, Jadud [299] defined an error quotient (EQ) metric, which was further explored by Petersen et al. [507], while Becker [57] has proposed an alternative metric. Compilation data has been used to provide feedback to students in the form of enhanced error messages, although the results have been mixed in terms of the value of these messages to students [58, 509]. Compilation can provide insights into aspects of behaviour other than the errors the student has to deal with; for example, Rodrigo and Baker [555] used compilation data to study student frustration.

Compilation is an important aspect of the coding process, but other aspects have been studied, including keystrokes and timing within a session [367] and evolution of code over an extended period [514].

In addition to the process, the code artefacts created by students have been studied. The correctness of code, typically measured by whether it passes automated tests on submission, has been used to identify common student difficulties [137] and competencies [69], and to predict course outcomes and target at-risk students [6, 37]. Code quality has been analysed to allow instructors to target improvements in the guidance given to students. Measurements of code quality have included traditional [508] and novel [112] software metrics and a continuous inspection tool [41]. While coding quality is a notion that comes from software engineering, Bumbacher et al. [99] propose a set of metrics, which they refer to as 'coding style', that can be used to monitor progress in learning and to predict help-seeking. Hovemeyer et al. [273] and Pero [503] have used code structures, in the form of abstract syntax trees derived from students' code, to draw conclusions about student skill levels. Loksa and Ko [387] combine student code with think-aloud protocols to investigate self-regulation of learning in programming.

Other behaviour data. While coding tools play an important part in studying programming, students interact with a range of other more generic learning resources such as learning management systems (LMS). Although such resources are widely used and studied across disciplines, there is a growing body of work specifically focused on their use in introductory programming.

Some researchers have applied techniques for machine learning [323, 370] and data mining [36] to study interaction by introductory programming students with an LMS in order to identify behaviour patterns and investigate whether these can predict academic success. Other papers focus on introductory programming students' use of specific types of resource provided within a learning environment, such as video lectures [149, 428] and interactive reading and homework materials [183] designed to enable a flipped classroom approach. The latter paper identifies some key implementation factors that encourage engagement with interactive materials. There has also been research on the use of social media, beyond the LMS, for online learning [408].

Although there is an emphasis in the literature on online behaviours, Hsu and Plunkett [274] study the correlation between class attendance and academic grades, while Chinn et al. [125] investigate several study habits, including online and other activities.

Summary. The value of data in providing insights and predictive power is becoming increasingly recognised in many aspects of human activity, and techniques and tools for data science are becoming increasingly accessible. Computing education is beginning to make use of these, but we have as yet only scratched the surface of what can be done. In particular, it seems that there is significant potential to learn about student learning through analysis of coding behaviour. There has been progress in mechanisms for gathering data, but there is a great deal of scope for exploring the correlation between behaviour and learning outcomes [637], and perhaps for understanding the differences in behaviour between novice and expert programmers.

5.1.4 *Student Ability.*

Students' ability in introductory programming courses can be thought of as what they can achieve in terms of learning, understanding, and applying programming. In this category we identified 169 entries, most of which infer students' ability from measurements related to their performance and success in programming [21, 48, 709, 734]. The use of the word 'ability' in this context should not be taken to infer that this is a fixed quantity. In particular,

it does not contradict the notion of 'growth mindset', which recognises that students' ability can change as their studies progress.

Students' ability has been measured by various means such as code assessment [318], self-reporting [223], knowledge acquisition tests [106], exams [101, 357], and various learning analytics [323]. Some studies combine different types of measurement [264, 318]. Although students' grades and traditional assessment are still the dominant measurement of their ability, there are moves towards more sophisticated measurements coming from high-frequency data collections.

A number of papers focus on the capacity of student-generated data to provide accurate estimations of students' ability. In recent years we have seen papers using student-generated data such as source code and log files to compare variables that can predict students' performance [694]. In addition, we have seen large-scale student-generated data that can be analysed to guide (or question) teachers' decisions and judgement about students' ability [92]. Despite the difficulties in adopting and using student-generated data (e.g., from large-scale projects and sophisticated pedagogical practices) to infer students' ability, student-generated data is a growing area of research in introductory programming [92, 323] that promises more accurate and reliable measurements of students' ability, for example by tracking learning progression during programming.

Many papers propose specific pedagogical practices such as community-of-inquiry learning, pair programming, and team-based learning [101, 425], suggesting that these practices can potentially improve students' ability to learn programming. Such papers generally report on the introduction of a practice to cope with a specific difficulty or misconception in programming, and often conclude with a possibly rigorous empirical evaluation and a discussion of lessons learned.

Some recent publications focus on the relationship between students' programming ability and other abilities such as problem solving [377] and engagement [313, 323], traits such as conscientiousness [570] and neuroticism [569], and attitudes such as self-efficacy [313].

In summary, publications on the ability of introductory programming students typically address one or more of these questions:

- How can students' ability, that is, their success or performance in programming, be measured more accurately and reliably?
- How can the various types of student-generated data inform students/teachers about ability and predict future success?
- What is the relationship between students' programming ability and their other abilities and personality traits, attitudes, and engagement?
- What pedagogical practices have the capacity to enhance students' ability?

5.2 Sentiment

In this section we report on research that investigates the student perspective of learning to program — the attitudes of students, the extent of their engagement, and the experience of learning to program. Figure 5 illustrates the growth of papers focusing on student sentiment.

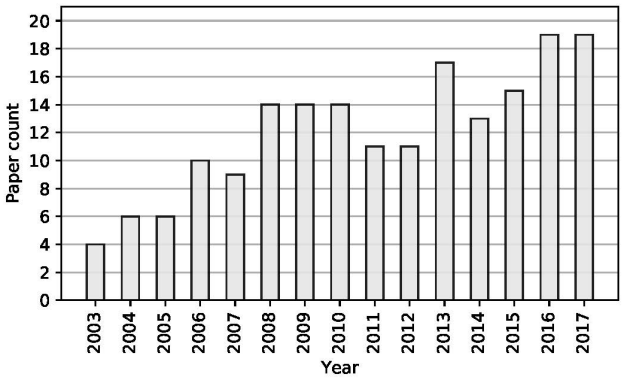

Figure 5: Number of publications per year focusing on student attitudes, engagement, or experience

5.2.1 *Student Attitudes.*

For the purpose of classifying papers, 'attitudes' has a broad definition. Eagly and Chaiken [180] define attitude as "a psychological tendency that is expressed by evaluating a particular entity with some degree of favour or disfavour". We take that to include self-perceptions, where the 'entities' are the students themselves or certain of their characteristics, such as current level of programming skill. We also take it to include perceptions of particular aspects of introductory programming, whether more abstract, such as programming as a discipline, or more concrete, such as a tool.

Given the challenges that educators face in the introductory programming course, particularly with respect to retention [325, 372], ways of supporting students to develop positive attitudes have received considerable attention in the literature (105 papers). This is perhaps underscored by McKinney and Denton's 2004 paper [431], which reports a concerning decline in affective factors during the first programming course, a challenge that persists more than ten years after its publication [587].

Although a considerable number of attitudinal influences have been linked with measures of course success, no construct has received more attention than self-efficacy and self-perception. This body of work includes key papers by Kinnunen and Simon which examine self-efficacy through a theoretic lens [327] and which identify sometimes counter-intuitive processes through which students' self-efficacy can change [328]. Though the importance of the construct is consistent with other areas [350], recent work throws into question whether conventional designs for learning activities that are intended to lead to enacted mastery experiences are appropriate and readily transferable to the domain of introductory programming. Further complicating this challenge is the notion that practices could disproportionately affect different groups. Some papers on this topic are covered in sections 5.3.1 and 5.3.2. However, it is worth noting that recent work continues to highlight differences in self-efficacy between genders [378, 531].

Within this domain is a considerable body of work on the perceived difficulty of programming courses [355] and its relation to persistence in computing degrees [189]. Only the most engaged students are good at predicting their performance in a course [159],

so when students tend to pick tasks that they perceive to be easier [665], it is not clear whether the difficulties they perceive are in the task or in themselves as individuals. The literature seems to acknowledge that when addressing such perceptions, the first impression is important [338], as is the consideration of perceived difficulty in the design of pedagogic approaches [223].

Several studies have explored ways to enrich student attitudes, often with the intention of improving some aspect of course 'health' or the student experience. These interventions include collaborative scaffolding [194], problem-based learning [208], peer tutoring [220], pedagogical code review [282], facilitated e-learning [365], and peer instruction [731].

A related area where successful interventions are also reported focuses on Dweck's notion of 'mindset', the level of belief that one can grow and develop. Murphy and Thomas [464] warn educators of the dangers of students developing a 'fixed' mindset in the introductory programming context. Further literature has since explored the design of mindset interventions [245, 584, 616]. Cutts et al. [143] report some success by teaching students about the concept of mindset and using messages embedded in coursework feedback to reinforce a 'growth' mindset.

There remains a need for further work comparing the effect sizes of interventions in order to guide educators towards the most effective practices. Student self-beliefs therefore remain a fertile field of research with continuing challenges. Calls for greater methodological rigour [190] might pave the way for such comparisons. For example, a number of measurement instruments have been developed and formally validated for use in the introductory programming context [171, 583] and their use might facilitate new developments in this field.

Another core area of attitudinal research considers attitudes towards particular tools, techniques, platforms, and pedagogical approaches. Much of this research is combined into studies exploring the tools themselves, and is covered in section 6.4. Examples of such attitudinal research include impressions of pair programming [242, 425, 427], attitudes towards the use of tools such as Scratch [418], satisfaction with self-assessment [474], impressions of prior experience [651], and the use of consistent instructors [424].

This area has considerable breadth and leverages a rich set of methodologies. Although validity is a spectrum rather than an absolute [472], there are several ways that methodological rigour could be improved in future work. Many of the recommendations published in the 2008 methodological review by Randolph et al. [537] are yet to be widely adopted, and there is scope to address similar concerns surrounding measurement [190]. Furthermore, papers in this area during the review period tend to focus on WEIRD populations (western, educated, industrialised, rich, and democratic) [258], exposing a lack of cross-cultural studies and undermining the generalisability of the research to an international audience. There is also a shortage of research replicating experiments that use validated measurement instruments; such research would offer reassurance of the consistency with which particular attitudes are observed in different contexts.

5.2.2 *Student Engagement.*

Student engagement in introductory programming has received considerable attention over the period of this review, encompassing papers focused on time on task, encouragement of self-regulated learning, and the issues surrounding disengagement. It is no surprise that a major factor in students' success is their self-motivation and ability to engage with the learning opportunities available to them [552]. However, students disengage and subsequently drop out for many reasons, and these are quite multifaceted [325]. On the other hand, there are students who complete their studies successfully after adapting to the difficulties by devising new techniques and increasing their efforts [505].

While much research on student engagement in computing takes a broader approach, the introductory programming context has been the focus of a number of studies. Such investigations often overlap with attitudes as antecedents to motivation (see section 5.2.1), tools and teaching approaches as interventions to improve motivation (see sections 6.4 and 6.3), and behaviour, in the sense of students doing things that improve their level of achievement and their probability of persisting with their study (see section 5.1.3). However, given the specific interest of the research community in engagement, it is worth highlighting key contributions to student engagement in a separate section here.

Many of the papers published during the period report on empirical studies focusing on the internal characteristics of students [111] and their role in self-regulated learning [378]. Some of these studies highlight particular constructs deserving of further attention, for example, achievement goals [734] and perceived instrumentality [504], as well as how these may differ between different subpopulations in some introductory classes [600]. Researchers also examine a range of motivational sub-factors, for example finding a weak correlation between introductory programming performance and social motivation, the desire to please somebody [576].

There are many explorations of the effectiveness of interventions on student engagement. Examples include full course redesigns [134, 406, 476]; implementing a holistic and better integrated curriculum [134]; teaching activity design, such as the use of in-class response systems [119]; strategies to enrich student behaviour, such as online journaling [604] and reading interventions [183]; changes to infrastructure, allowing for smaller classes [87] and closer monitoring of individuals [32]; the application of e-learning [80, 365, 652]; flipped classrooms [363]; peer communication tools such as anonymous chat rooms [666], discussion forums [590], and Facebook [488]; collaboration, including pair programming [115], team-based learning [192], and strategies such as think-pair-share [346]; interactive learning experiences [337, 415, 669]; the use of robots [397, 435]; and physical computing approaches [417, 565].

Other papers report attempts to improve student engagement through changes in the assessment process, introducing assessment practices and cycles such as feedback-revision-resubmission [262]; contexts that are meaningful to students [292]; contexts that correspond to the real world [344]; the use of tools and equipment, such as 3D printers [316] and robot olympics [582], that can grab attention while facilitating creativity; emphasising the utility of computing through simulated research [493]; tapping into social

computing paradigms earlier in introductory programming [592]; and game development [130].

Of course, games also feature prominently in the literature on engagement for their motivational qualities [122, 568]. The use of games and game-like tools in introductory programming is not unique to the most recent literature. However, over the period of our review there has been a clear increase in papers on the topic of 'gamification' (or 'gameful design' as some authors prefer [162]), describing how these ideas and techniques can be applied in the introductory programming context. Though 'gamification' was coined as a term in 2002, it has received considerably more attention in educational spheres since 2010 when Lee Sheldon published *The Multiplayer Classroom* [599] and Jesse Schell endorsed the approach at the 2010 DICE (Design, Innovate, Communicate, Entertain) Summit. This has manifested in the introductory programming literature through schemes such as *Game2Learn* [49] and *JustPressPlay* [155], as well as the gamified practical activities in *TRAcademic* [249]. Yet few papers refer explicitly to any underlying motivational theory or design frameworks [38, 127], and further research may be necessary to determine their applicability to improving the design of interventions that engage students to do more programming.

Approaches that promote engagement tend to emphasise active learning over passive learning, eliminate barriers to practice, play to students' intrinsic motivations, explicitly introduce extrinsic motivations, and strive to resolve 'hygiene' problems. Also, some holistic interventions that approach course design as a system of interrelated factors tend to report at least some success. However, the question of how best to facilitate engagement for most students remains an open one. Several high-quality papers describe successful interventions and explain their efficacy using motivational theories such as three-motivator theory [476], self-determination theory [336], and control-value theory [583]. Although such theories are discussed elsewhere in the literature, there is little work that addresses them within the scope of our focus on introductory programming. Therefore, future work focusing on introductory programming should strive to apply these theories in explaining their interventions. It is also somewhat surprising that learning analytics have not played a more prominent role in this section of the introductory programming literature. The use of such tools to measure engagement would help to identify particular 'hygiene' factors or problematic topics that tend to disengage students. Such data could be collected through classroom response systems, virtual learning environments, version control systems, integrated development environments, and more.

5.2.3 *The Student Experience.*

When considering the student experience in introductory programming courses, we limit this to mean those aspects of study that are beyond the scope of tools and approaches related to learning. Thus, papers in this category focus on the experience itself, covering practices and systems for support, social integration, affect, and pastoral issues surrounding the transition into higher education. Many of these issues overlap with retention risk (section 5.3.1) but have been reported in this category because they focus on experiences or because they describe social or pastoral support structures rather than focusing on individual characteristics.

With only 18 papers identified by the search, as a topic in and of itself the experience of learning introductory programming does not seem to have received focused attention in the period of this review. A small number of articles explore the influences of learning programming in a foreign language [147], but most focus on interventions to support learners who experience anxiety and frustration in the face of difficulty. These include peer-assisted study schemes [163], mentoring systems [175], and programmes to support students from underrepresented groups [471]. Most such studies apply an action-research approach incorporating analysis of data collected through survey and interview. However, a clear nomological network has yet to emerge across such intervention studies, suggesting that further work is needed to identify and converge upon key factors associated with the study of the student experience.

Even if not as an explicit goal, work towards such a model does exist. For example, Barker et al. [46] found that factors such as peer interaction predicted intention to continue studying computing, and Schoeffel et al. [576] found correlations between social motivation and performance. Other researchers such as Haatainen et al. [234] considered social barriers such as whether students felt comfortable requesting help. Indeed, many such social integration challenges have been identified in the literature. However, rather than a sole focus on predictors of achievement or retention, the identification of a holistic set of factors would support work into interventions.

An aspect of the student experience which has received much attention is affect, in the sense of emotions. Notable qualitative studies [279, 326, 376] analyse the emotional experiences and 'tolls' that programming students tend to encounter as they complete introductory programming tasks. There are interventions to explore how the modality of learning programming can influence the emotions experienced by students [417]. Nolan and Bergin [478] systematically reviewed many such papers in 2016, focusing on programming anxiety. They concluded that although much work promotes an awareness of student anxieties, there is a need for greater focus and granularity by leveraging tools that can measure a student's anxiety during particular programming tasks. A number of tools have been proposed and developed to aid this research [84, 235].

Our search found only a small amount of work on the student experience in the explicit context of introductory programming. We are aware that more general work has been carried out on this topic, and we feel that such work could have considerable impact on concerns specific to introductory programming, so the area remains a fertile field for further research. In addition to receiving more attention, there is also the need to identify key facets and drivers of the student experience. The methods of phenomenography and grounded theory are well suited to the identification of new factors [329]. Researchers should also consider identifying and drawing together appropriate theories, models, and frameworks from other fields to elicit potential factors [409]. The development of new tools to measure experiential effects would also support work into the identification of particular tasks that heavily influence the student experience, and would help to determine the effectiveness of particular interventions.

5.3 Student Subgroups

In this section we report on two commonly studied subgroups: students who are at risk of failure, and students in underrepresented groups. Figure 6 illustrates the growth of papers focusing on these subgroups.

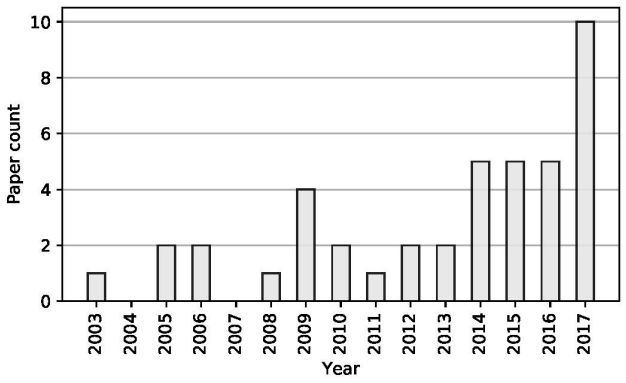

Figure 6: Number of publications per year focusing on underrepresented groups or students at risk

5.3.1 Students at Risk.

The early 2000s was a time of low enrolments in computing degrees in many parts of the world. The low enrolments, along with a perception that an atypically high percentage of students were not successfully completing introductory programming courses, resulted in a developing interest in identifying students who are at risk of failing the introductory programming course. However, in the scope of our own study the majority of papers in this area have appeared in the past four years. Many of the recent papers focus on automated methods for identifying students at risk as the enrolment in introductory computing courses surges.

In one of the earliest papers in our review period [456], students with little or no programming background were encouraged to take a preliminary Alice-based course in which they would first meet the fundamental programming concepts that they would later encounter in the 'normal' introductory programming course. Students in this treatment group ultimately performed substantially better in introductory programming, and showed a better retention rate, than the students with a similar background who went directly into the later course. Although not statistically significant, the students who began in the Alice course also displayed higher self-confidence. Other researchers also found low confidence or self-efficacy to be a factor in students who were at risk in introductory programming [220, 694, 716].

Several groups of researchers developed automated methods for identifying at-risk students early in the semester. Many of these techniques are based on automatically monitoring student activity as the students develop code [6, 37, 135, 196, 556, 649, 693]. Measurements are made of various attributes of the interactions with the development environment, such as numbers of errors and times between compilations. Some researchers make use of data-mining techniques [135] or predictive modelling [37]. A consistent rationale for automated investigations of programmer behaviour is their ease of use compared to test-based methods that attempt to identify relevant demographic, personality, cognitive, or academic attributes of the students. A comparative study of behaviour-based and test-based attributes showed that the former are better predictors of programmer success [694]. The only test-based attribute that was a good predictor was self-efficacy.

Estey et al. [196] took a somewhat different approach by examining student behaviour using BitFit, a tool for practising programming that checks solutions and provides hints. They found that students who frequently use hints without developing and submitting code tend not to succeed. Another approach for identifying at-risk students relies on the use of responses to clicker-based questions in a classroom environment [374, 520, 521].

Several researchers have identified behaviours that can be observed physically in a lab environment to be indicators of future poor performance. Examples include being bored or confused in lab [556], stopping working, using code samples verbatim that may not be relevant, and making frantic changes [716]. Badri et al. [40] developed a web-based IDE with a variety of dashboards and visualisations that allow an instructor to closely monitor and to some extent control student progress. Haden et al. [235] measured students' affective state after lab and found that students who found a problem both difficult and boring, or who recognised material as being familiar but did not see a plan to solve the problem, were at risk.

Nearly all of these approaches have shown some success at identifying at-risk students early in the semester. Some researchers have identified concepts that are covered in the first few weeks of the semester and continue to trouble students on exams and in assignments throughout the semester. This underscores the need for students to learn these early fundamental concepts in order to subsequently succeed. A few papers discuss interventions that have had a positive impact in this regard. Wood et al. [716] found that when students were matched in pair-programming teams based on their confidence levels, the matched pairs were more likely to successfully complete assignments. The effect was strongest for the least confident students. Whittinghill et al. [706] found that when the course was directed at the inexperienced student, those with little interest in programming at the beginning of the semester had a similar level of interest to the rest of the class at the end of the semester.

On the other hand, after encouraging students who performed poorly on an early exam to attend voluntary tutorial sessions conducted by undergraduates, Punch et al. [528] note that few students attended and the intervention had little effect on the final course outcomes. Similarly, Hare [247] reports that the use of peer mentors did not have the anticipated effect of increasing student success. Further investigation revealed that many of the students had competing time constraints, such as jobs, which prevented them from taking advantage of the peer mentoring opportunities.

The research in this area indicates that early success in an introductory programming course is important for a student's ultimate success. Many researchers have shown that students having difficulty can be identified early in the semester. A critical piece of information that was not discussed in many of these papers is what specific concepts were covered in the early weeks of the courses being investigated. Future work could focus on identifying if there

are specific concepts that confuse students early, and on methods or techniques for presenting the material to enhance student understanding.

5.3.2 *Underrepresented Groups.*

While many STEM fields have seen increasing enrolments by women and members of underrepresented minorities over the past few decades, female enrolments in computing degrees in the US have stayed low since declining in the mid-1980s. At the same time, non-white, non-Asian, male populations have traditionally been poorly represented in computing degree programs. Much of the research examining issues surrounding the lack of diversity in computing has occurred during the current decade, with the initial focus primarily on gender diversity. As the gateway course, introductory programming can play a critical role in attracting students to a computing major or degree and encouraging them to persist with that study; alternatively, it can discourage students from pursuing study in computing. A key factor that has emerged through various studies is the importance of self-efficacy. Multiple studies have found that females have lower self-efficacy when enrolled in an introductory programming course [16, 55, 172, 378, 587] and that this can affect performance and persistence.

Studies have also shown that students' self-efficacy can be influenced by the pedagogy employed in introductory programming. An early paper by Dorn and Sanders [172] describes an environment for learning introductory programming concepts such as control structures, methods, and objects. When the environment was used for four weeks at the beginning of an introductory programming course, students indicated that it increased their comfort and confidence in learning to program, with the effect more pronounced for females and for males from underrepresented groups. Similarly, Beck et al. [55] found that students given cooperative learning exercises performed significantly better than a control group, with females and males from underrepresented groups appearing to benefit more. Newhall et al. [471] report on a modified introductory programming course with a peer mentoring program that after several years resulted in the demographics of the course more closely matching the overall college demographics.

A study of students in an introductory programming course at a large Midwestern US university [378] used measures of self-regulated learning. As in other studies, self-efficacy was found to have the strongest direct effect on performance. Interestingly, male and female students corrected the accuracy of their self-evaluation at different rates. Measures of self-efficacy were taken at three points over the semester. For females the correlation between self-efficacy and end-of-term performance increased significantly between the first and second measures of self-efficacy, whereas males showed a statistically significant increase in correlation between the second and third measures.

The issue of lower self-efficacy in females is not limited to the United States. A study of students in introductory programming across multiple institutions in Ireland and an institution in Denmark showed significant gender differences [531]. Although women indicated a higher previous achievement in mathematics, female students rated their programming self-efficacy negatively while males rated theirs positively and predicted that they would perform better. Males did perform better at programming early in the course, but the final pass rate for females was significantly higher than that for males.

Another factor that influences continuation from introductory programming to the subsequent course is a student's intended major (in institutions where students choose majors after beginning their degrees). Data from nine US institutions on the demographics of majors and non-majors in CS1 [575] indicated that women make up a greater percentage of non-majors and tend to have less programming experience than men. A multi-year study at a single university [39] found that of the students enrolled in CS1, a lower percentage of the women intended to major in computer science. Male students showed a significant correlation between grades and continuing to CS2, but there was no such correlation for female students. Among self-taught students and students with no prior background, males were more likely than females to persist to CS2.

The research to date shows convincingly that developing self-efficacy is a key factor in the persistence and success of students from underrepresented groups. While some approaches show promise, additional research on pedagogical techniques to foster the development of self-efficacy would provide better guidance to those involved in teaching introductory programming.

6 TEACHING

Table 3 shows that papers on teaching fall into five broad categories: theories, orientation, delivery, tools, and infrastructure. We found a small number of papers focusing on theories of teaching and learning, including those related to Bloom's taxonomy [25] and cognitive load theory [648]. The orientation category includes papers that describe the overall approach to the structure of an entire course, such as using a flipped-learning approach. The category of delivery includes papers describing techniques or activities that could enhance learning by improving the way the content is delivered by teachers and experienced by students. The majority of papers in the tools section focus on tools that support teaching and learning. Finally, papers classified as infrastructure focus on aspects such as the physical layout of laboratories, networks and other technology, teaching assistants, and other support mechanisms.

Table 3: Classification of papers focused on teaching

Category	N	Description
Theories	19	Models, taxonomies and theories
Orientation	53	Overall course structure
Delivery	156	Techniques/activities used to teach
Tools	254	Tools that support teaching/learning
Infrastructure	15	Institutional/environmental support

6.1 Theories of Learning

As with theoretical perspectives of students (section 5.1.1), there are relatively few papers on theories of learning, with no obvious trends across the period of our review.

Pears et al. [501] observed that papers in computing education are generally system-oriented and rarely build on a theoretical foundation. Nine years later Malmi et al. [409] found that 51% of the

papers that they examined explicitly applied at least one theory in their work. However, only 19 of the papers that we examined were coded as theory-related papers, and the majority of these papers deal with the notion of learning styles [404, 726, 735], which have been largely discredited as "pseudoscience, myths, and outright lies" [332].

The papers identified in our literature review apply theories and frameworks related to students' progress from exploration, through tinkering and constructive learning theories, to knowledge acquisition [71, 103]. Bloom's taxonomy, which has been widely referenced by researchers as a benchmark for assessment of students' learning, has seen some use in introductory programming research [174, 661]. Self-regulated learning theory is another body of knowledge that has been used to identify guidelines for when and how interventions should occur [170].

Soloway, a pioneer in computing education research, viewed student learning through the lens of cognitive theory, which sees basic knowledge as chunks of information that a student or programmer can recall and use in a constructive way. This is directly connected with another theory used in computing education research, cognitive load theory, which posits that learning degrades as the learner is required to remember more items than the capacity of their working memory [389]. In introductory programming, which is known to be difficult to learn, it is critical to assist students to develop the necessary basic knowledge, but also to consider their cognitive load and their capacity to absorb that knowledge.

For introductory programming in the object-oriented paradigm, recent work has focused on the development and empirical validation of competency models [458], which highlight the structural knowledge of programming novices and the potential dimensions of such knowledge [458].

Some research initiatives focus on building theories of how learners acquire programming knowledge [181]. However, the dominant approach to knowledge construction in computing education research is to design innovative tools and teaching approaches and evaluate them empirically through possibly rigorous studies. This approach constructs knowledge related to particular examples and contexts, but contributes little to a more generalised understanding of computing.

Nevertheless, studies of local interventions do create opportunities for constructing more abstract knowledge. Thus, future research in introductory programming should consider theoretical implications by going beyond case studies and specific tools. The role of theory should be not to replace the case studies and empirical analyses but to annotate them and construct knowledge that is more general. This could be achieved by grounding or connecting future research to learning theories and documenting potential theoretical implications; by extending learning theories to address the particularities of introductory programming research; or by performing meta-analyses to construct bodies of knowledge that are more abstracted.

6.2 Course Orientation

In this section we look at broader approaches related to the overall course structure, the 'orientation' of the course, that have been adapted for use in introductory programming courses. We have identified several broad categories in the literature: self-paced learning, exploratory learning, inverted classrooms, and online courses. The growing interest in student-directed learning and the rise in popularity of online learning have resulted in relatively steady growth in this area, as indicated in Figure 7, with over half of the papers being published in the past four years.

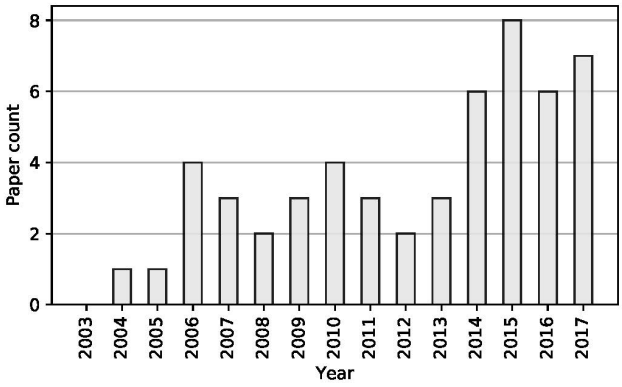

Figure 7: Papers classified as being about course orientation have increased over the period of study

6.2.1 *Self-Paced.*

Mastery learning is an approach in which students are expected to demonstrate that they have reached an appropriate level of mastery in a given topic before they continue to more advanced material in that topic. It is widely believed that learning to program requires students to understand basic elements of syntax before learning more complex elements [553], and researchers have expressed the need for assessment practices to allow students to demonstrate knowledge of individual components before they solve problems involving multiple concepts [396, 506]. Despite the apparent alignment between the process of learning to program and the mastery learning approach [394], there have been few reported examples of mastery learning in the introductory programming classroom. This is perhaps explained by the difficulty of resolving the tension between the constraints of traditional semester-based delivery and the freedom of self-paced learning.

There is some evidence that mastery learning is of more benefit to weaker students than to stronger students because it helps to 'level the playing field', and that those benefits continue into subsequent courses [423]. In an attempt to ensure a minimum standard of expertise in team projects, Jazayeri [303] required students to demonstrate mastery of basic programming skills before they were permitted to participate in a group project in the second half of an introductory programming course. This approach meant that fewer teams were burdened by students with inadequate skills.

Purao et al. [529] report on the iterative refinement over five years of a self-paced learning approach based on Keller's Personalized System of Instruction. Although several issues were encountered, their most significant concern was student procrastination. They advocate that teaching using self-paced learning should attend to student motivation and ensure that students continue to study at

the pace required to complete the course material. They also identify a trade-off between allowing students to work at their own pace and the benefits of working with peers. In similar implementations, students preferred an environment that accommodated a variety of ability levels by having exercises that they could work through by themselves, but supported by scheduled laboratory sessions with an instructor who could provide targeted help with the exercises for that week [672].

Although this approach appears to show potential for teaching introductory programming, it is difficult to accommodate self-paced learning within the institutional course structures adopted by most universities. Where it is possible to use more self-paced learning approaches in online environments and within inverted classrooms that accommodate differentiated learning experiences, further research in this area has the opportunity to make a substantial contribution.

6.2.2 *Exploratory.*

Exploratory approaches encourage students to be creative and to drive the learning process. Problem-based learning uses complex open-ended questions to encourage students to develop problem-solving strategies. The problems may be set by the teachers or may be developed by students, as is often the case in project-based courses. Studio learning, by contrast, emphasises creativity and communication by asking students to present their solutions to others.

Problem-based learning. Although several studies using problem-based learning have reported increased motivation and social interactivity [34, 397, 483], and asserted the success of the approach [438], there is little consistent evidence of improvement in learning of content knowledge. Some case studies report learning gains in problem decomposition and program-testing strategy [469, 500] and overall improvement in pass rates [24] or in specific exam questions [540]. In other cases no difference in learning was observed [483], students without clear guidance struggled to solve the problems effectively [439], and concerns were raised about the level of theoretical insight acquired compared to those experiencing traditional lectures [397].

Studio learning. Studio-based learning takes inspiration from more creative disciplines such as architecture and fine art to embed a focus on creative design and public critique within an introductory programming course. This approach emphasises evaluation and communication in the design of a solution, and appears to be more effective when supported by a visualisation tool [289]. Students using a studio-based approach had higher self-efficacy than those in a more traditional classroom environment and were more favourably disposed towards peer learning, but both environments resulted in similar levels of content knowledge in final exams [282]. A three-year case study of studio-based learning reported that students enjoyed the social aspect of studio learning and found its open-ended nature highly engaging [541].

While instructors at a number of institutions have implemented and reported on one of these exploratory approaches, and some have conducted research on the outcomes, their efficacy will need to be evaluated far more widely and rigorously before there is any prospect that they will be widely adopted.

6.2.3 *Inverted Classrooms.*

Several variations of course design replace traditional lectures with other classtime activities such as practice exercises, collaborative problem solving, and teamwork. Such courses are often described as 'inverted' or 'flipped' classrooms.

The initial development costs for a well designed and supported inverted classroom can be quite high. One study estimates having spent approximately 600 hours to record the videos that students used to prepare for classes, and another 130 hours to prepare 32 worksheets for students to use in class [108].

The high cost of course development may be worthwhile, with reports of high levels of engagement [108], a greater sense of community [363], improved retention [192, 361], and improved performance in exams [191, 192, 361, 363].

Flipped classrooms typically have several graded components to help ensure that students complete all of the required work. One study observed that the flipped approach required students to devote more time to the course. Students completed more practice exercises and received more feedback in the flipped approach than in a more traditional classroom, which may explain the improved performance [191].

However, the additional work imposed on students may not always provide benefits. Quizzes used to determine whether students had completed the preparation before class meetings were reported to have no benefit [352], and well structured online courses may result in performance that is equivalent to the flipped approach [108].

Cognitive apprenticeship. Cognitive apprenticeship is informed by Bandura's social learning theory [44], which recognises the importance of learning through modelling the behaviour of others. Demonstrations by experts [564], worked examples [621], and individualised feedback (coaching) are characteristic of cognitive apprenticeship. The use of cognitive apprenticeship in computing classrooms has been associated with reduction in lectures and increased student agency, similarly to the flipped classroom model.

A variation of cognitive apprenticeship called extreme apprenticeship implemented at the University of Helsinki reduced lectures to a minimum and emphasised lab-based exercises with high levels of feedback [682]. This approach resulted in significant improvements in student understanding, and higher performance in subsequent courses [319, 351]. Jin and Corbett [306] showed that when worksheets that model expert problem solving are combined with an intelligent tutoring system that provides feedback on exercises, learning improves significantly compared with a standard presentation of material.

The cognitive apprenticeship delivery approach appears successful, but studies investigating the approach in a broader range of contexts are required to determine the extent to which these case studies can be generalised.

6.2.4 *Online Courses.*

Notwithstanding the recent growth of MOOCs, we found few papers dealing explicitly with online courses.

When a small private online course (SPOC) was introduced to complement the standard course content as an optional support structure, students liked the course, and the resources were well used without a negative impact on lecture attendance. The use of

badges to motivate students to complete online quizzes, and immediate feedback for programming exercises through automated assessment tools, were effective approaches to engage students [513].

The completion of practice exercises in an online textbook has been highly correlated with improved exam performance [585]. In a comparison between an online course and a flipped classroom using similar resources, completion of ungraded practice exercises was correlated with exam performance, but fewer students in the online course completed the exercises compared to those attending in person [107]. Given the choice of enrolling in a face-to-face or an online version of an introductory programming course, more than 80% of students chose the face-to-face course, based primarily on a preference for attending lectures, rather than other reasons such as prior experience or perceived workload [271].

6.3 Teaching Delivery

This section focuses on how teachers choose to deliver the curriculum. This includes the overall context in which introductory programming is taught, and the specific activities used to deliver the content. Figure 8 shows that publications focusing on delivery of teaching do not appear to be growing in number.

Table 4: Classification of papers focused on approaches taken to course delivery

Category	N	Description
Context	44	The 'flavour' of a course
Collaboration	53	Collaborative learning approaches
Techniques	59	Approaches, techniques and activities

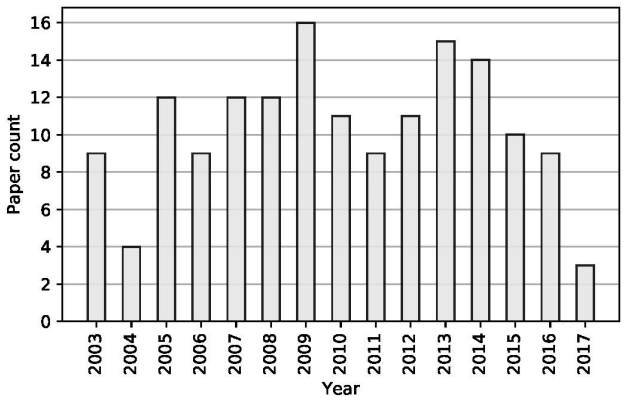

Figure 8: Number of publications focusing on teaching delivery per year

6.3.1 *Context.*

Course context encompasses the context of the examples, exercises, and assignments used to embed the content knowledge of programming, and has been referred to as the 'flavour' of the course [53]. The course context is perceived to have an impact on student perception of the discipline, and to affect student motivation, attitude, and level of engagement with programming. It is reported to affect enrolment in computing courses, and could impact on underrepresented groups [322, 437].

Teachers have explored a variety of contexts that connect introductory programming with real-world applications. It has been argued that the relevance and real-world application of computational processes are demonstrated effectively by tangible computing devices such as a Sifteo Cube [219], LEGO MindStorms [373], iRobot Creates [698], CSbots [364], and the Raspberry Pi [714]. Although such innovative contexts are seldom evaluated, there is some evidence that students enjoy using the devices [373], and this enjoyment may translate to greater participation and completion rates [364].

Ecology [713], mobile phones [602], networking [467], creative story-telling [310], real-world data analysis [26], creative art [227], mobile apps [403], and cross-disciplinary activities [42] have all been used to broaden interest in programming. Anecdotally, these contexts all provide positive experiences for students, but few formal evaluations explore their impact on students' learning.

Games are widely used to motivate students to learn programming [53, 130, 179, 333, 371, 688], and have the advantage of lending themselves to more creative open-ended projects. Although students prefer to have structured assignments [132], they enjoy the opportunity to be creative when supported by an appropriate framework [35, 301, 395], and can even be enlisted to develop games for the learning of programming [588]. However, there is some evidence that the course context is not as important as good structure and good support in the course [295].

Following the success of media computation for non-majors [233] there have been several reports of introductory programming courses that use image processing as a motivating context [165, 421, 542, 707].

Although there are very few studies that investigate the impact of context on students, Simon et al. [617] found a significant improvement in both retention rate and exam performance when a traditional introductory programming course was delivered using a media computation approach. However, a separate study of students in two different sections of a single course showed no differences in performance or retention when one section was delivered using a visual media context and the other using a more traditional programming context [422]. Comparing students taught in a robot programming context with those taught in a web programming context, Scott et al. [582] found that the former group spent significantly more time programming and that their programs showed more sophistication and functional coherence.

Despite the potential for students to be motivated by varying contexts, most publications provide case studies involving the context of delivery, rather than collecting and reporting data that evaluates the impact of the course on students. Further work to provide empirical evidence of the impact of course contexts would make a significant contribution to the community.

6.3.2 Collaboration.

Collaboration has several reported benefits in the introductory classroom environment, including improved motivation, improved understanding of content knowledge, and the development of soft skills such as teamwork and communication. In an educational environment where transferable skills are becoming increasingly important, collaborative activities offer opportunities to deliver a wide range of desirable outcomes.

The term 'contributing student pedagogy' [237], although not in widespread use, describes recent approaches in which students explicitly create material to be used by other students for learning. Examples include the creation of collaborative textbooks [67], sharing solutions [390], generating assessment questions (such as multiple choice questions) [157], and peer review [238].

Peer Review. Regularly engaging in peer review of programming code reportedly helps to correct low-level syntax errors, and promotes discussion of higher-level design and implementation issues [281]. Students engaged in reviews appear to be more effective and more engaged when the reviews are conducted collaboratively rather than individually [284].

Electronic voting. The use of electronic voting systems, often described as 'clickers', has risen in popularity as a way of making lectures more active. Although students report enthusiasm towards the use of clickers, many students participate only intermittently [144]. Peer instruction, which makes use of clickers as one component of a more structured pedagogy, has shown positive results. Students report that they like the approach [517], and it appears to improve their self-efficacy [731] and retention [519]. However, peer instruction is not uniformly successful; it relies on instructors to clearly communicate the purpose of the activities and to use grading policies that do not penalise incorrect answers during the peer instruction process; and it should be supported by other changes to course delivery, such as requiring students to spend additional time preparing before class [517]. Despite several positive outcomes resulting from peer instruction, there appears to be little evidence that it improves performance in final exams.

Pair programming. Pair programming has been described to be one of the few educational approaches that we know are effective [518]. Experiments comparing students engaged in solo programming with those engaged in pair programming have shown that students typically enjoy working in pairs [425, 427, 534, 712], although conflicts within pairs are frequently noted [425, 691, 716], giving rise to several studies investigating how best to pair students [533, 660, 691].

Students working in pairs produce work that is of higher quality than students working alone [244, 425, 427, 712], but the improved performance does not extend to individually assessed work [427, 534, 712, 716].

Pair programming may provide greater support for students with low academic performance [89, 90], resulting in increased pass rates [427, 534, 659, 708] and improved retention [115, 712], but provides little measurable benefit for more experienced students [716]. Several studies have also reported that the benefits of pair programming are more pronounced for women [89].

Distributed pair programming appears to work similarly to collocated pair programming, with students typically enjoying the process, producing code with higher quality and fewer defects, and being more likely to complete the course — but without evidence of improved performance in individual assessments [241, 725].

Other paired activities. An experiment comparing the use of think-pair-share (TPS) with a traditional lecture format in a single class showed that students learning with TPS believed that it helped them to learn. On a subsequent quiz the TPS group scored significantly better than the control group [347].

A study investigating the effects of peer tutoring on student confidence, attitude, and performance found no significant differences between students engaged in peer tutoring and those experiencing standard delivery of material [200]. Another study using the supplemental instruction (SI) approach, in which 'model' students guide additional learning sessions for their peers, reported no improvement in student performance [700].

Group activities. Cooperative learning activities that assign specific roles to each student in a team are reported to improve motivation, confidence, problem-solving ability [201], individual performance on exams [55], and performance in subsequent courses [225]. Although collaborative activities are easier to implement face to face, the benefits of working collaboratively with peers extend to distributed environments when supported by collaborative tools [20]. In a virtual classroom environment, Bower [85] found that peer review, program modification, and debugging tasks resulted in a high degree of engagement and interaction among peers. The creation of programs was also effective, but more sensitive to the task design and ability of the students.

Although collaborative learning is often used to improve content knowledge, there has been little explicit evaluation of soft skills developed through collaborative activities, skills such as communication, cooperation, commitment, work ethic and adaptability. While such skills may be time-consuming, or difficult, for teachers to effectively evaluate, peer assessment has been used successfully for this purpose [432, 433].

Summary. Evidence in the literature suggests that students enjoy the social interaction resulting from collaborative activities, and that working collaboratively has several benefits such as improved engagement, confidence, retention, and performance. Although pair programming is the most widely studied form of collaborative activity, increased individual performance appears to result more reliably from other forms. Instructors and researchers would benefit from further detailed case studies of effective collaborative activities for introductory programming.

6.3.3 Pedagogical Techniques.

In this section we survey a broad range of papers that discuss how instructors choose to teach specific topics in the curriculum and the types of activities that they use to teach these topics. We do not consider papers on methods that affect the structure of the whole course, which are more likely to be discussed in section 7. We also exclude papers on contextualisation and collaborative activities, which are discussed in sections 6.3.1 and 6.3.2.

Topic ordering. One of the basic pedagogical questions faced by instructors is the order in which topics should be introduced. Bruce [94] summarises a discussion that took place on the SIGCSE mailing list on whether to introduce procedural programming concepts before object-oriented programming concepts. While this relates to the bigger controversy on which programming paradigm to use in introductory programming, the discussion on the mailing list showed how much disagreement there is on topic ordering even among instructors who agree on introducing OO programming concepts in introductory programming. Although that discussion took place nearly 15 years ago, the controversy is still not settled. We found two papers that argued for specific topic orderings in introductory programming, but without providing empirical evidence. Barland [47] discusses how functions can be taught first when teaching Java and how this leverages what students already know about functions in mathematics. Bruce et al. [95] argue for introducing structural recursion before loops and arrays in an objects-first Java course, suggesting that this ordering provides more opportunity for reinforcing object-oriented concepts before students are exposed to concepts that can be dealt with in a non-object-oriented way. Paul [497] discusses the pedagogical implications of topic ordering in introductory programming courses.

Testing. Several works advocate that testing be taught early in introductory programming, with the expected benefits of promoting reflective programming instead of trial-and-error programming [91, 185, 469], improving understanding of programming concepts [261, 483], getting used to industry practices and tools [226], and boosting student confidence [704]. However, empirical evidence for these potential benefits in introductory programming is still limited. Papers describe the introduction of test-driven development (TDD) to introductory programming courses, but without providing any empirical evaluation [91, 697].

Edwards and Pérez-Quiñones [187] report that the Web-CAT tool has been used in several institutions for several years to evaluate the quality of student-written test cases in introductory and subsequent programming courses, but their evidence for the efficacy of TDD was gathered in non-introductory courses. Some evidence is provided for success in teaching novices how to write high quality test cases [663], for success in using TDD for teaching arrays [261], and for the usefulness of teaching mutation testing in a Pascal programming course [483].

Neto et al. [469] evaluated a teaching approach that requires students to develop test cases in a table-like manner, and found that they exhibited more thoughtful programming behaviour than when provided with feedback from an auto-grader.

On the other hand, some papers report that teaching TDD in introductory programming is an additional challenge, because adding a testing requirement can be overwhelming for novices [416], can be perceived by students as irrelevant or useless to their learning of programming [295], and can be at the expense of other material typically taught in introductory programming [663]. Some of the papers that argue for the early introduction of testing [226, 704] have the students use test cases written by instructors, which is pedagogically different from the TDD expectation that students will write their own test cases.

Exercises. The relatively large number of drill-and-practice tools mentioned in section 6.4.4 gives a hint of the central role that exercises play in the teaching style of many introductory programming instructors. Some instructors [580] take exercises a step further by designing their course in an exercise-intensive manner similar to the way that some mathematics courses are taught. The exercises used are generally either conventional coding exercises or multiple-choice concept questions, but other types of activity have also been proposed. Miller et al. [445] propose using a type of collaborative exercise based on creative thinking principles. Ginat and Shmalo [218] propose an activity type that creates cognitive conflict to directly address erroneous understanding of OOP concepts. Kortsarts and Kempner [345] present a set of non-intuitive probability experiments to be programmed by students, arguing that such exercises engage students and motivate them to learn. In a randomised experiment, Vihavainen et al. [681] found that adding supporting questions to self-explanation exercises improves student performance on similar exam questions. Lui et al. [390] describe how student solutions to lab exercises can be shared with the whole class to be used as worked examples, which could facilitate the acquisition of alternative schemas for solving problems. Denny et al. [157] showed in a randomised experiment that requiring students to create exercise questions in addition to solving exercises can bring about a greater improvement in their exam performance than simply solving exercises.

Teaching a process for programming. Several papers propose that students be taught a process for composing programs. Rubin [564] advocates live coding as a way to show the students the process that experts (such as the instructor) go through when they solve problems and write programs. In an experiment, Rubin found that students who were taught using live coding achieved higher grades on programming projects than students who were taught using static examples that are not modified live in front of the students. Bennedsen and Caspersen [62] propose using video recordings of experts as they program, supplementing live coding with the flexibility of allowing students to watch the process more than once and of presenting more complex problems than is typically possible in the limited class time.

On the other hand, Hu et al. [275, 276] advocate that students be explicitly taught a detailed programming process rather than being expected to infer one from examples. In one paper [275] they propose teaching goals and plans [624] using a visual notation implemented using blocks in the Scratch programming environment. They also propose a process that students can follow to move from goals and plans to the final program, and observed a statistically significant improvement in the performance of students on exam programming questions when taught using this method. In a later paper [276] they propose an iterative process with feedback for the intermediate steps, where unmerged plans can be executed separately before the whole program is built and executed.

Castro and Fisler [117] captured videos of students as they solved a programming problem in Racket and analysed how the students composed their solutions. These students had been taught to program using code templates [204], but Castro and Fisler found that students often used code templates without adjusting them, trying to decompose the problem on the fly around code they had already

written instead of carefully decomposing the problem in advance of writing code. They concluded that students should be explicitly taught schemas not just for code writing but also for problem decomposition.

Leveraging students' prior knowledge. To study the algorithmic knowledge that students have prior to learning any programming, Chen et al. [123] gave students a sorting problem on the first day of an introductory programming course. Although students showed no prior understanding of the notion of a data type, they were able to describe algorithms with sufficient detail. Their solutions made more use of post-test iteration (as in *do-until* loops) than of pre-test iteration (as in *while* loops). This knowledge could be used by instructors when introducing programming concepts.

Several authors attempt to leverage such prior knowledge. Blaheta [77] describes a vote-counting activity that can be used on the first day of an introductory programming course to introduce concepts about algorithms. Paul [496] uses the example of a digital music player to introduce object-oriented modelling and the difference between objects and references to objects. Gonzalez [224] describes activities using manipulatives and toys to introduce functions, message passing, memory concepts, and data types. Sanford et al. [572] interviewed ten experienced introductory programming instructors about their use of metaphors. The study analysed and categorised twenty different metaphors used by these instructors and found that some instructors used a single metaphor to explain several concepts while others used several metaphors to explain a single concept. They concluded that more work needs to be done to understand the limitations of using metaphors.

Many instructors, regardless of discipline, use analogies and metaphors as a way of connecting students' prior understanding of a familiar topic to a completely new topic. However, there is little empirical evidence for the effectiveness of using analogies in introductory programming. Cao et al. [109] evaluated experimentally the effectiveness of using analogies when teaching recursion, events, and multithreading. They measured short-term learning using clicker questions, long-term learning using exams and quizzes, and transfer using exam questions that require knowledge to be applied to new areas. Results showed potential benefit from analogies in the short term, but there was no strong evidence for benefit in the long term or in transfer to new areas.

Object-oriented design. Several papers describe techniques for teaching object-oriented design. Concepts such as abstraction and program decomposition can be introduced to students using large programming projects that are broken down to smaller pieces [318, 349, 626]. Keen and Mammen [318] discuss a term-long project in which students are given clear direction on program decomposition at early milestones and progressively less direction at later milestones. They found that a cohort taught using this technique showed less cyclomatic complexity in the code they wrote for a final assignment, compared to cohorts in previous years. Santos [574] describes a tool that allows object-oriented concepts to be taught using visual contexts such as board and card games. Students can manipulate object instances in these contexts and see the effect visually. Similarly, Montero et al. [450] found that students' understanding of basic object-oriented concepts improved when they used a visual environment that allows interaction with object instances and visual observation of the effect.

Visualisation. Beside object-oriented concepts, several papers report the use of visualisation tools in introductory programming for teaching topics such as recursion [22], sorting algorithms [670], roles of variables [10], and procedural programming concepts [8, 524]. Lahtinen and Ahoniemi [354] use a visualisation tool and an algorithm design activity to introduce students to computer science on their first day in the introductory programming course, allowing students with no programming experience to experiment directly with different issues related to algorithms. Hundhausen and Brown [287] use a visualisation tool in a studio teaching method, with students working in pairs to design an algorithm for a problem and construct a visualisation for their algorithm. The class then discuss the designed algorithms and their visualisations. The results of actually constructing visualisations rather than just watching or interacting with them were inconclusive, although the results for the effect of the overall studio teaching method were positive. On the other hand, AlZoubi et al. [22] evaluated a hybrid method for teaching recursion, where students perform different types of activity using a visualisation tool. They found that the number of students' activities in constructing visualisations correlated more strongly with learning gains than the number of activities in watching an animation or answering questions about it. Evaluating the use of visualisation tools in the classroom, Lattu et al. [362] observed 60 hours of assignment sessions and noted how students and instructors explained code to one another. They found that students and instructors very rarely used the visualisation tool that was at their disposal, instead using visual notation on the blackboard to trace program execution and show the structure of the manipulated data. They also used multiple computer windows simultaneously to talk about related pieces of code and to show how code relates to what happens at runtime. The study recommended that such observations should be taken into consideration when designing visualisation tools that are meant to be used in an explanatory learning setting as opposed to an exploratory one.

Video. Instructors have reported using multimedia [27, 430] and videos [113, 465, 591, 685] in introductory programming. Reasons for using videos included providing flexibility for students who cannot attend class in person [685], explaining extra material that is difficult to cover during class time [465], and dedicating class time for hands-on activities instead of lecturing [113]. Murphy and Wolff [465] note that instructors should not only care about the quality of their videos, but should also think carefully about how the videos are going to be used in the course. Their experience was disappointing, as a relatively small number of students used the videos which they spent a lot of time trying to perfect.

6.4 Tools

A variety of tools have been developed to support introductory programming. The majority of papers present new tools, but some report on evaluations of existing tools or comparative reviews of multiple tools. Table 5 shows the category, number of papers, and brief description of the papers in each of the following subsections.

Figure 9 shows the number of papers per year about tools, clearly showing a steady increase in the number of these papers.

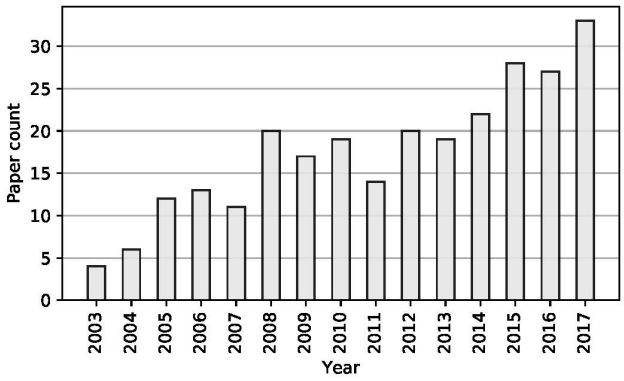

Figure 9: Number of publications per year focusing on teaching tools

Table 5: Classification of papers focused on tools

Category	N	Description
Reviews and research	16	Analysis of existing literature or broad research implications
Environments and editors	46	Programming environments, editors, IDEs
Libraries, APIs, and extensions	13	Libraries, APIs, programming language extensions
Learning programming	51	Tools to practice programming
Debugging and errors	12	Debugging, programming errors, compiler error messages
Design	13	Tools to practice problem-solving
Visualisation	24	Program and algorithm visualisation systems
Collaboration	13	Tools to help students collaborate
Games	23	Tools for developing games, games as learning objects
Evaluation	22	Evaluation of tools
Progress	21	Tools that monitor student progress

6.4.1 *Reviews and Research Considerations.*

The past 13 years have seen a number of reviews of programming tools, including three on programming languages and environments, as well as reviews of support tools, robots as tools, flowchart tools, visualisation tools, and learning tools. We list these below, then briefly discuss tools that have appeared in course census studies and tools that can support the replicability of research studies.

2004 *Environments:* Rongas et al. [560] propose a classification of 'learning tools and environments' for introductory programming students, broken down into IDEs, visualisation, virtual learning environments, and systems for submitting, managing, and testing of exercises. They include free and commercial tools, and provide guidelines for tool selection.

2005 *Languages and environments:* Kelleher and Pausch [320] present a taxonomy of languages and environments designed to make programming more accessible to novice programmers. The systems are organised by their primary goal (teaching programming or using programming to empower users) and then by the approach used in each system.

2005 *Environments:* Gross and Powers [229] summarise a representative collection of the assessments of novice programming environments, present a rubric for evaluating the quality of such assessments, and demonstrate the application of their rubric to the summarised works with the intent of helping inform future efforts in assessing novice programming environments.

2011 *Support tools:* Coull and Duncan [136] summarise the problems associated with learning and teaching introductory programming and discuss various support tools and techniques, indicating which of them have been evaluated. They derive ten requirements that a support tool should have.

2012 *Robots as tools:* Major et al. [405] conducted a systematic literature review on the use of robots as tools in introductory programming. They found that 75% of papers report robots to be an effective teaching tool. They note that most of these papers focus on the use of physical robots, and note the need for further research to assess the effectiveness of using simulated robots.

2015 *Flowchart tools:* Hooshyar et al. [266] surveyed flowchart-based programming environments aimed at novice programmers, identifying the basic structure employed in the majority of existing methods as well as key strengths and shortcomings.

2014 *Visualisation tools:* Sorva and Seppälä [634] carried out a comprehensive review of visualisation tools (64 pages and 200+ references), which we discuss further in section 6.4.7.

2017 *Learning tools:* Saito et al. [567] present a survey and taxonomy of 43 tools designed for use in introductory programming.

A 2003 census of introductory programming courses in Australia and New Zealand reports on the programming languages and environments used in 85 courses [153]. A similar survey of 35 institutions conducted in 2016 [419] showed a clear shift in language use from Java to Python.

Jadud and Henriksen [300] suggest that more computing education research studies should be easily replicable and that the tools used for data collection are often too specialised, unstable, or simply unavailable for use in replication studies. They present two tools to aid in the replication and extension of existing research regarding novice programmers, in addition to supporting new research. The first tool, specific to the BlueJ pedagogic programming environment, provides a starting point for replicating or extending existing studies of novice programmers learning Java. The second tool is a

portable standalone web server with a language-agnostic interface for storing data. The distinguishing feature of this server is that it is schema-free, meaning that it can easily support a wide range of data collection projects simultaneously with no reconfiguration required.

Notable gaps in the recent literature in terms of reviews of teaching tools include the areas of debugging and errors, tools to practice programming, design, student collaboration, games, evaluation of tools, student progress tools, and language extensions, APIs, and libraries. Among these areas there does appear to be emerging interest in debugging and errors, student collaboration, visualisation tools, and games.

6.4.2 Programming Environments and Editors.

The choice of programming environment for introductory programming courses is nearly as contentious as the choice of programming language. Some instructors choose a basic combination of command-line compiler such as javac or gcc coupled with a popular text editor such as Notepad++, while at the other extreme some choose popular industry-grade IDEs such as Eclipse. Between these extremes, many tools and environments have been developed specifically for introductory programming students, often with the aim of minimising features that are deemed unnecessary for beginners. In this section we discuss such pedagogical environments before discussing more general environments for mainstream languages, followed by work done in evaluating these environments. We conclude with a list of environments for beginning programmers, organised by language. We note that the landscape of environments and editors is constantly evolving, often driven by new platforms such as mobile phones, which hold promise for the creation of new editors, environments, and languages such as TouchDevelop [664].

Pedagogical environments. Pedagogical environments such as the Penumbra Eclipse plug-in [457] dispense with features not necessary for beginners. Others focus on being specifically suited for a particular approach or pedagogy; for example, BlueJ [342] is widely used for teaching Java in an objects-first approach. Others, such as the DrJava Eclipse Plug-in [545] and COALA [311], seek to provide suitable pedagogic environments while leveraging the power of industry-standard IDEs. Some, such as DrHJ [498], extend other pedagogical environments, in this case DrJava, to support specialised languages such as Habanero Java, a parallel programming language based on Java. The fact that many languages are tied to specific editors, environments, and/or contexts has been considered problematic by some, leading to tools that try to avoid these dependencies. Two such examples are Calico [79], a framework/environment that supports a variety of languages, pedagogical contexts, and physical devices, and Jeroo [571], a language, IDE, and simulator, inspired by Karel the Robot [98] and its descendants, that seeks to provide a smoother transition to Java or C++.

General environments for mainstream languages. Some systems seek to be as general, all-encompassing, and accessible as possible. A web-based system by Ng et al. [473] provides an all-encompassing online programming learning environment incorporating editing, compiling, debugging, and testing, as well as features supporting educator and student participation in learning activities such as coursework, feedback, and student progress monitoring. CodeLab [50], a web-based interactive programming exercise system for introductory programming classes, accommodates Python, Java, C++, C, C#, JavaScript, VB, and SQL. Rößling [561] presents a family of tools to support learning programming and a plan to integrate these tools into the Moodle learning management system.

Evaluations of environments. In addition to presenting new environments, many authors have evaluated their effectiveness. Comparing three popular environments for Python, Dillon et al. found that students struggled with low-assistive environments regardless of experience and confidence, that students were able to use moderately-assistive environments more effectively [166], and that the difficulty of transitioning between such environments may not be symmetrical [167]. Vihavainen et al. [680] studied novices writing their first lines of code in an industry-strength Java IDE (NetBeans) and found no evidence that learning to use the programming environment itself is hard. Despite this evidence, novice-friendly programming environments remain popular with instructors and students alike. McKay and Kölling [429] extended their previous cognitive modelling work by developing a new novice programming editor that is specifically designed to reduce intrusive bottlenecks in interaction design in some novice programming languages.

Here are some of the environments and editors that have been reported, categorised by the languages they support:

C: LearnCS! [375], unnamed systems [168, 452]
C++: CLIP [392]
Habanero Java: DrHJ [498]
Haskell: Helium [254]
Java: ALE [54] (Java-based platform for developing 2-D Android games), BlueJ [342], COALA [311], CodeMage [705], Decaf [58], DrJava Eclipse Plug-in [545], ELP [668], Gild [643], JGrasp [443], Jigsaw [100], Penumbra [457]
Jeroo: Jeroo [571]
Karel++: objectKarel [718]
Pascal: VIPER [3]
Python: CodeSkulptor [653], PyBlocks [516], Pythy [188]
Multiple Languages: AgentSheets [121], Calico [79], CodeLab [50], CloudCoder [491]

The vast array of environments used with introductory programming courses includes command-line compilers, industry-grade IDEs, and pedagogical environments specifically intended for learning. Some of the pedagogical environments work with mainstream languages while others are designed to work with teaching languages. Although some work has been done to evaluate these environments, it is still far from clear whether introductory programming students are better off with industry-grade environments working with mainstream languages, pedagogical environments working with mainstream languages, or pedagogical environments working with teaching languages.

With language and environment choice one of the most important decisions in teaching novice students, reliable and comprehensive evaluations in this area would be extremely valuable.

6.4.3 *Libraries, APIs, and Language Extensions.*

Along with the numerous and diverse programming environments used in introductory programming (discussed in section 6.4.2), a number of libraries, APIs, and language extensions have also been developed. These are important as they often seek to provide interaction and increase engagement with novice programming students. Papers in this group report on the use of robots [248, 440], graphics [222, 272, 415, 551, 566], and multimedia [447, 489].

A number of papers focus on bringing other engaging aspects to the novice programmer. RacketUI [239] allows students learning with the DrRacket pedagogical programming language to develop web forms without the need for a graphics library. Hamid [240] introduces a framework to facilitate the introduction of large, real-time, online data sources into introductory programming. Squint [466] is a Java library developed to support the use of event-driven programming and network applications in programming examples for introductory programming courses. Emerson [448], a plugin for the Sirikata open source virtual world platform, is designed for scripting objects in user-extensible virtual worlds such as Second Life, Active Worlds, and Open Wonderland, with the primary goal of making it easy for novice programmers to write and deploy interesting applications.

Most of the papers in this category seek to increase engagement with novices by providing easy access to robots, graphics/multimedia, or other interactive or stimulating devices and environments. It is important to consider whether this is enough for today's introductory programming students. Better integration of realistic data sets, virtual reality, and other hot-topic areas into introductory programming could help to engage more students, including those who are harder to engage.

6.4.4 *Tools for Learning Programming.*

One of the earliest focuses of tool developers was to build tools that help students to practise writing code. Over the years, these tools have evolved from grading to also providing feedback, and from presenting a fixed sequence of drill-and-practice problems to presenting problems adapted to the needs of the learner. These tools are only as effective as the feedback they provide, but provision of useful feedback is complicated by the fact that the coding solution to a problem is never unique.

Some programming tools have been designed to address specific troublesome programming concepts such as stack and heap visualisation [385], recursion [22], arrays [536], complex data types [441], exception handling [538], loops [206], linked lists [209], and object-oriented programming [172].

A number of different types of tool have been designed to help students learn programming in general:

- Assessment systems used to facilitate learning, e.g., Python Classroom Response System [732].
- Compiler assistants, which accept student solutions, present compiler error messages with additional information to help the student fix any syntax errors in the program, and collect usage data for analysis by the tool administrator.
- Program submission systems, which accept student solutions, may include a compiler assistant, use test cases to determine whether the solution is correct, and provide the student with minimal feedback to that effect, e.g., Online judge system (YOJ) [645].
- Drill-and-practice systems, which are program submission systems containing a host of built-in problems and a lesson plan, that is, an order in which those problems should be presented to the student, such as CodeWorkout [186], CodeWrite [161], CodeLab [51], Projekt Tomo [386], The Programmer's Learning Machine (PLM) [532], EduJudge [678], AlgoWeb [173], and a tool from Viope [114]. Some also include animation [561].
- Intelligent tutoring systems, which are drill-and-practice systems that provide context-sensitive feedback and adapt to the needs of the learner, such as L2Code [177, 728], ITAP [549], PHP ITS [699], FIT Java Tutor [230], CPP-Tutor [2], PROPL [359], and Java Programming Laboratory (JPL) [527].

Approaches used in tools to teach programming include the use of mental models [343], notional machines [73], custom languages [539], and deterministic finite automata [28]. Some provide automatically-generated hints [307] or narrative-based coaching [141], while others recommend a series of edits to transform a student's program into a target version [730].

Not all of these tools were designed to assist with code-writing. Other ways that tools have supported learning of programming include code-tracing [29, 126, 468], code style [128], Parsons puzzles [199], program analysis, either by generating system dependence graphs [19] or by having students identify beacons in code [250, 369], and authoring, either by capturing illustrative examples [543] or by generating sequences of program examples [74].

Some tools were developed to facilitate communication or collaboration, such as between the instructor and student [23] or between two or more students [86, 232, 243].

The current trends in tools for programming include addressing skills other than code-writing, and using intelligent tutoring technology and data-driven techniques to improve the feedback provided to the learner. In an ideal future, the developed tools will be evaluated for efficacy and disseminated for widespread use by other instructors.

6.4.5 *Debugging, Errors, and Error Messages.*

In this section we review tools to aid in debugging programs, tools that help with student errors, and tools that help students to interpret error messages. This is an emerging area, particularly the work on errors and error messages, with most of the papers we encountered being published since 2010.

Several tools have been developed to help novices debug their programs. Whyline is a tool that aids debugging by presenting the programmer with a set of questions based on the code [339]. The programmer selects questions and the system provides possible explanations based on several types of code analysis. One evaluation found that novices using Whyline were twice as fast as experts without it. Lam et al. [358] propose a hybrid system that provides a certain level of automated debugging combined with system-facilitated input from instructors. Noting that traditional debugging tools can be difficult for novices to learn, Hu and Chao [277] designed a system that provides students with debugging

scaffoldings designed to help them detect, find, and correct logic errors in programs by visualising execution results.

Many tools focus not on debugging but on student errors or on the error messages generated when errors arise. Most of these focus on compile-time (syntax and semantic) errors and error messages, but some work has also been done on runtime messages. A tool by Jackson et al. [298] logs all errors from students and faculty in order to investigate the discrepancies between the errors that instructors think students make and those that students actually make. HelpMeOut [251] is a social recommender system that helps with the debugging of error messages by suggesting solutions that peers have applied in the past. Bluefix [695] uses crowd-sourced feedback to help students with error diagnosis. Other tools focus on enhancing standard compiler error messages, with some studies reporting positive effects [56, 58] and others reporting no effect [158], leaving the effects of compiler error enhancement a currently open question.

Although there is much attention on compile-time syntax errors, many authors have developed tools to help students with warnings, runtime errors, and other types of error. Backstop is a tool that helps students interpret Java runtime error messages [463]. Dy and Rodrigo [178] developed a system that detects 'non-literal' Java errors, errors reported by the compiler that do not match the actual error. C-Helper is a static checker for C that provides more direct warning messages than gcc, and also alerts the novice programmer to 'latent errors', those for which commercial compilers do not report an error, but where the intentions of the programmer are contrary to the program [671].

The complexity of most debugging tools presents a barrier to their use by novices. Recently, substantial focus has been given to tools that reduce the need for debugging, either by attempting to reduce the number of errors committed by novices in the first place, or, when errors do arise, by providing more helpful error messages. Most of the attention in this area has focused on compile-time syntax or semantic errors, but runtime errors, warnings, and latent errors (not reported by the compiler) have also been explored. It is not yet clear whether these means of helping novices understand error messages are effective, and more work is needed in the area.

6.4.6 *Design Tools.*

Introductory programming courses typically set out to teach both problem solving — devising an algorithm for a given problem — and programming — converting the algorithm into correct program code in a selected programming language. Although problem solving is an important component of computational thinking, its teaching is seldom as clearly defined as the teaching of programming. Several tools have been built to help students learn problem solving, typically using one of three approaches:

- UML diagrams: These tools approach problem solving by means of building UML diagrams, an endeavour uniquely suited to object-oriented programming; examples are CIMEL ITS [455] and Green [17, 577].
- Flowcharts: These tools help the student construct a flowchart for the problem. Examples are FITS [269], FMAS [267], SITS [268], VisualLogic [231], and Progranimate [581]. The constructed flowchart may then be automatically converted to code. Some attempts have been made to compare these tools; for example, RAPTOR and VisualLogic [5].
- Pseudocode: These tools elicit the pseudocode in English; an example is PROPL [216, 359].

A tool has also been reported that can be used to capture the teacher's problem-solving actions and replay them for the student's benefit [210], and another uses regular expressions to translate novice programs into detailed textual algorithms [4].

Tools for problem solving are in their infancy. Given the increasing recognition of the importance of computational thinking in higher education, it is expected that more developers will focus on building and evaluating tools in this area.

6.4.7 *Visualisation Tools.*

This section focuses on visualisation tools intended for use by novices. We found two broad types, those that visualise programs and those that visualise algorithms. The program visualisation tools are dominated by Java, with only a handful of tools focusing on other languages. Algorithm visualisation is more often language-agnostic.

In 2013, Sorva et al. [632] published the first comprehensive review of visualisation systems intended for teaching beginners, providing descriptions of such systems from the preceding three decades and reviewing findings from their empirical evaluations. This review is intended to serve as a reference for the creators, evaluators, and users of educational program visualisation systems. It also revisits the issue of learner engagement, which has been identified as a potentially key factor in the success of educational software visualisation, and summarises what little is known about engagement in the context of program visualisation systems for beginners. Finally, they propose a refinement of the frameworks previously used by computing education researchers to rank types of learner engagement. Overall, their review illustrates that program visualisation systems for beginners are often short-lived research prototypes that support the user-controlled viewing of program animations, but they note a recent trend to support more engaging modes of user interaction. Their findings largely support the use of program visualisation in introductory programming education, but research to 2013 was deemed insufficient to support more nuanced conclusions with respect to learner engagement.

The Sorva et al. review also references a review on algorithm visualisation by Hundhausen et al. [288] that was not identified in our search because it was published in 2002; however, we mention it here for the interested reader. As the 2013 review by Sorva et al. is recent and comprehensive (64 pages and well over 200 references), in this section we do not discuss papers that were covered in that review and also identified in our search [66, 97, 228, 312, 330, 353, 362, 453, 535, 635]. The remaining papers can be classified as either program visualisation, predominantly in Java, or algorithm visualisation.

Program visualisation: Java. Jeliot [60] is a program animation system in which the visualisation is created completely automatically from the source code, so that it requires no effort to prepare or modify an animation, even on the fly during a class. JavaCHIME [650] is a graphical environment that allows users to examine classes and objects interactively and to execute individual methods without

the need for a testing class containing a main method. Users can also examine variables and methods and interactively alter the values of variables while testing the methods. PGT (Path Generation Tool) [334] visualises paths that correspond to statements in source code. EduVisor (EDUcational VISual Object Runtime) [451], which can be integrated with the Netbeans IDE, shows students the structure of their programs at runtime, seeking to incorporate knowledge accumulated from many prior projects. Using jGRASP [140], students are able to build dynamic visualisations that, in conjunction with the debugger, help them understand and correct bugs in their programs more effectively than using the debugger alone.

Program visualisation: other languages. Visualisation tools designed for other languages include PyLighter [81] for Python, CMeRun [197] for C++, and VIPER [3] for Pascal.

Algorithm visualisation. Although algorithm visualisation has a long history, seven of the eight papers in this subsection are from the past five years, suggesting that algorithm visualisation is enjoying a revitalisation, at least as regards introductory programming. Alvis Live! is an algorithm development and visualisation model where the line of algorithm code currently being edited is re-evaluated on every edit, leading to immediate syntactic feedback, along with immediate semantic feedback in the form of a visualisation [286]. Results suggest that the immediacy of the model's feedback can help novices to quickly identify and correct programming errors, and ultimately to develop semantically correct code. Other work in the area includes using visual tiling patterns for learning basic programming control and repetition structures [373], a tool for learning iterative control structures [412], flowchart-based algorithm visualisations [265], tools to help with visualising notional machines [145, 620], and a plugin for the MindXpres presentation tool to improve the visualisation of source code in teachers' presentations [558, 559].

Summary. Visualisation tools for novices fall into two categories, those that visualise program execution and those that visualise algorithms. The former are normally tied to a single language, generally Java, whereas most algorithm visualisation tools are language-agnostic. It is notable that most of the algorithm visualisation work we found has occurred in the past five years. The comprehensive 2013 review by Sorva [632] is commended to readers who wish to explore the area more deeply.

6.4.8 *Collaboration Tools.*

The power of learning to program in pairs or groups has long been recognised in computing education. Groupware technology is increasingly being used to facilitate learning, among learners who might be in the same location (collocated) or away from one another (distributed) and who might be present at the same time (synchronous) or at different times (asynchronous).

Various tools for collaboration have been reported in the literature and have been reviewed [724]. They fall into a few broad categories:

- Tools to help students write programs collaboratively, including distributed [86] as well as collocated and synchronous [212]; working on bite-sized programs [490] or objects-first programs [510]; and collaboration facilitated by similarities among student programs [72]
- Peer assessment [110] and peer review [151]
- Incorporating collaboration into existing environments such as Alice [11], BlueJ [207], and DrJava [625]

Other attempts to incorporate collaboration into introductory programming include using Facebook to seek immediate assistance while learning Java [356] and awarding points to students for helping with group learning [512].

With the availability of inexpensive and ubiquitous hardware (mobile phones, tablets, etc.) the desire for any-time, anywhere learning, and an increasingly interconnected world with a growing trend of crowdsourcing and just-in-time pairing of demand and supply, we expect collaboration tools to take on much greater prominence in the future of computing education.

6.4.9 *Tools Using Games.*

Games have probably been used as teaching tools since prehistoric times, so it is natural for the practice to be applied to the introductory programming context. Papers in this category have an explicit focus on digital games as a teaching tool. We do not include papers describing non-digital tools such as pen-and-paper games.

Some papers describe or evaluate approaches, tools, frameworks, or software development kits (SDKs) that can be used by students to program their own games or game-like software with only an introductory level of knowledge and experience. Sung et al. [646] present a casual game development framework to help instructors to incorporate game content into their introductory classes, and Frost [211] presents a Java package to simplify the development of 2D games for beginners. Though some papers present example games to showcase the motivational consequence of such frameworks, little empirical evaluation is reported. One systematic review found that incorporating game themes into introductory programming (for example, as assigned development tasks) did not improve pass rates to the same extent as other forms of intervention [679].

Educational games that can be used as learning objects or teaching aids have received much attention by researchers during the period of this review. An ITiCSE working group on game development for computing education reviewed over 100 games that taught computing concepts, 75% of which were classified as covering 'software development fundamentals' [309][1]. Other reviews also identify long lists of digital games that have been developed to teach programming concepts [675]. Several of these games have been empirically evaluated. Two notable examples are Wu's Castle [179] and Gidget [366]; however, there are too many to list here.

Though studies in this area tend to report promising results, they seldom firmly establish causality or determining the size of effects that are uniquely attributable to a particular tool. Studies specific to the introductory programming context tend not to apply validated methods of measurement or to compare tools to alternatives. There is thus little empirical evidence to support the efficacy of game

[1]The games considered by the 2016 ITiCSE Working Group on Game Development for Computer Science Education are listed at https://goo.gl/ZTJNi3.

development or of playing educational games in the introductory programming context. Future work should seek to establish such empirical evidence.

6.4.10 *Evaluation of Tools.*

Recent years have seen a growing emphasis on the evaluation of tools and pedagogical approaches in introductory computing. Approaches used for evaluation including anecdotal, survey-based, and controlled evaluation. In this section we list papers that deal with evaluation of tools in any of these forms.

Different categories of tools have been evaluated to date:

- Visual programming environments: ALICE, one of the more popular environments, has been evaluated often [131, 215, 475, 523]. More recently, mobile versions of visual environments have been compared [563]. A study has also investigated the effectiveness of the block interface common to block-based programming languages [525].
- Mini-languages used to teach introductory programming, such as Karel the Robot [98] and Logo [116]
- The use of robots such as LEGO Mindstorms has been reported to affect motivation [414, 435]
- Auto-graded coding exercise platforms — student behaviour when using them and their benefits for learning [120, 184, 589, 711]
- Interactive books and learning objects and their benefits [182, 515, 686]
- Integrated development environments and their aspects that might be beneficial for novice programmers [304, 548]
- Intelligent tutoring systems (ITS): their benefits [623], the role of tutorial dialog in ITS [710], and the use of automatic program repair in ITS [722].
- Tools for algorithm development: ALVIS [290] and MELBA [573], as opposed to tools for coding, such as Verificator [484, 485]

Tools have also been used in the evaluation of pedagogical approaches such as self-paced learning [480], explicit instruction of the roles of variables [601], and the quality of examples in textbooks [83]. Evaluations have also been made of various features of tools, including menu-based self-explanation [198], example-based learning [573], and tutorial dialogue [710], as well as how students learn with tools such as Visual Program Simulation [633].

Some welcome trends are an increase in the use of controlled evaluation and statistical techniques for analysis of data, multi-institutional evaluation of tools, and the use of learning theories to explain the results of evaluation.

6.4.11 *Student Progress.*

Some papers deal with tools that track student progress by automating various processes. Some systems track student progress in the traditional sense, assignment to assignment or day to day. Others track progress within assignments, or track more specific notions of progress, such as progression through programming activities based on compiler data. The use of tools to track student progress appears to be increasing, with the majority of the papers we report on being from the latter half of the search period, since 2010.

Wilcox [711] notes that many departments are automating key processes such as grading. After careful consideration of the effects of automation and how it should be applied, he presents results from a highly automated introductory programming course, concluding that carefully managed automation can increase student performance while saving teaching resources. Boada et al. [80] introduced a web-based system that reinforces lecture and laboratory sessions, making it possible to give personalised attention to students, assess student participation, and continuously monitor student progress. Evaluation with teachers and students was positive.

Alammary et al. [12] designed a Smart Lab system that allows instructors to monitor student programming activity via live code snapshots, compilation activity, and other metrics such as idle time. Kim et al. [324] introduced two systems that can together infer student progress from activities such as code revisions. The system was trialled with 1000 students and 240,000 source code revisions. Diana et al. [164] present a dashboard that not only logs and presents student progress data but can predict which students are in need. Their report suggests that student outcomes can be accurately predicted from student program states at various time points throughout a course. Presenting a tool to easily capture student affect data and demonstrate that accurate predictions of student performance are possible, Haden et al. [235] identify specific response patterns that can signal at-risk students. Wortman and Rheingans [717] use a visualisation tool to interpret the progress of students and thus to isolate the causes of decreased retention in their department. Similarly, ClockIt[479] unobtrusively monitors and logs student software development activities with the aim of reducing high attrition rates, and Retina [462] collects information about students' programming activities and provides useful and informative reports to both students and instructors based on the aggregation of that data. Retina can also make real-time recommendations to students, in order to help them quickly address some of the errors they make.

CodeBrowser [255] is a browser-side code snapshot analysis tool that allows easy entry of snapshot data. Its authors used this tool to uncover differences between students who passed and failed an introductory programming course. AutoLEP [690] combines static analysis of student programs with dynamic testing to help students test their programs and evaluate whether they meet their specifications. The system also encourages students to find and work through bugs, by providing automatic syntactic and structural checking and immediate detailed feedback. The authors provide evidence that this tool can improve students' learning experience and reduce the workload of the teaching staff. Rodrigo et al. [557] built and used several tools to study the behaviour of novice programmers, including what edits they make, when they compile their programs, and how they respond to errors, enabling educators to identify at-risk students and determine specific interventions. These tools include metrics on student compilation errors, time between compilations, Error Quotient [300], and successive compilations.

Rather than simply monitoring student success, some systems focus on minimising procrastination or helping students manage time more effectively. Pryor [526] introduced a system to monitor and quantify student procrastination in real time, and evaluated its use for evaluating new course implementations, material, and instructor strategies that might reduce procrastination. Yau and Joy [720] provided a context-aware and adaptive learning schedule

tool which offers students the most effective help depending on their context, preference, and time.

Not all tools in this category are software tools. Zimmerman and Rupakheti [730] introduced a framework for recognising the desired target for partially-written code and recommending a reliable series of edits to transform the input program into the target solution. Inspired by the formal code inspection process commonly used in the software industry, Hundhausen et al. [283] explored the use of pedagogical code reviews (PCRs). They found that PCRs not only improved the quality of students' code but also positively impacted students' communication and sense of community. However, a significant challenge was making code solutions and review results accessible to team members before, during, and after the team reviews. To overcome this, they presented an online environment specifically tailored to PCRs. Becker [57] introduced a new metric to measure the progress of novice programmers by measuring the frequency of repeated errors made while programming. This metric was designed to be less context-dependent than other metrics such as the Error Quotient [300], and could complement tools such as those discussed above by Rodrigo et al. [557]. Porter et al. [521] showed that student performance can be convincingly predicted using clicker data.

6.5 Infrastructure

Teaching infrastructure in introductory programming is a fairly broad term, including physical infrastructure such as labs and hardware, digital infrastructure such as software, and infrastructure related to human resources such as teaching assistants and mentors. This topic has too few papers to warrant a plot of their distribution over time.

Several studies [156, 683] focus on hiring undergraduates to participate as teaching assistants (TAs) in introductory programming courses, thus allowing upper-level students to contribute, take responsibility, and learn the content in greater depth. This approach provides several advantages, such as community building, participatory development of learning materials, high engagement and feedback, and increasing the sense of responsibility and interest [156, 683].

There is also growing interest in student mentoring and tutoring services in introductory programming research [175, 270]. Such services assist novices to engage and build up their confidence in programming. Mentorship and tutorship programs rely on volunteers under a well-defined framework from the university [175], and can be offered in an online mode [270]. These programs have proved to help novice student programmers to engage in, learn, and improve their programming, but the range of qualities of the mentors is an important element of these services [175].

Another research direction in teaching infrastructure is the investigation of hardware and software to support novices. Environments such as a tablet-PC-based classroom presentation system [341], Blackboard and Facebook [408], and a purpose-built mobile social learning environment [407], have been designed to support introductory programming courses. Such teaching infrastructures enable increased novice-to-novice interactions, help to identify misunderstandings and misconceptions of novice programmers, and address them via timely feedback [407]. However, there is a need for further work on how various hardware and software infrastructures can be integrated [408].

Another growing research direction identified in our study is that of electronic textbooks. Contemporary e-textbooks are interactive and incorporate a number of active components such as video, code editing and execution, and code visualisation [444]. Such textbooks are created with an open-source authoring system that allows the instructor to customise the content, offering low-cost and easily maintained solutions that support both the student and the teacher [195, 444]. Initial results indicate a promising infrastructure with good potential to support students' learning of introductory programming [195, 444, 515].

Overall, teaching assistants have been found extremely beneficial in introductory programming, supporting both students' learning and their engagement and identity building [156, 683]. Student mentoring and tutoring services are another positive infrastructure development [175, 270]. There are also studies focusing on hardware and software infrastructure to support novices [407, 408]. We found little research in the area of physical infrastructure (e.g., designing lab and studio spaces), or how infrastructure such as hackerspaces and makerspaces contribute to introductory programming.

Further research should consider the interplay of different infrastructures (physical, digital, and human-resources), and how those infrastructures contribute holistically to the learning experience of programming novices.

7 CURRICULUM

Anecdotally, some computing educators would argue that all introductory programming courses teach the same thing: the basis and basics of computer programming. From that perspective, there would be no need for papers that describe the curriculum of an introductory programming course. While most papers on introductory programming do indeed appear to take the curriculum as a given, there are others for which aspects of the curriculum are the clear focus. Indeed, descriptions of curriculum have long been a staple of computing education conferences, as explained by Valentine [676]: "Colleagues describe how their institution has tried a new curriculum, adopted a new language or put up a new course . . . [These] presentations serve an important function: we are a community of educators and sharing our successes (and failures) enriches the whole community." Table 6 summarises the papers that we found in this category, while figure 10 shows the number of papers per year focusing on curriculum.

Table 6: Classification of papers focused on curriculum

Category	N	Description
Curriculum	102	Curriculum in general
Languages	70	Choice of programming language
Paradigms	53	Choice of programming paradigm

Many curriculum-related papers describe an unconventional focus to the course itself; examples are UML modelling [61], design patterns [291], and problem-solving and algorithms [203, 256]. Others describe the decision to stream the students into either a simpler or a more advanced introductory course, and the consequences of

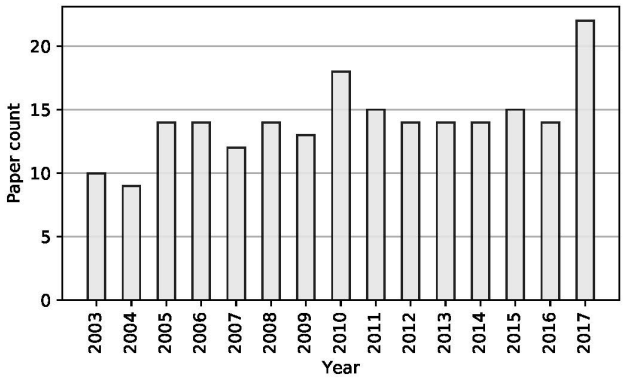

Figure 10: Number of publications per year focusing on curriculum

so doing [31, 331]. There are also courses that explicitly combine introductory programming with a distinctly different topic, such as biology [169] or law[622].

Emerging from the premise that 'X is very important, and should therefore be incorporated in the introductory programming course', a number of papers explain how to do that. Examples of X in such papers include databases [134], concurrency [96], networking [221], parallel programming [340], and information security [413, 522].

More formally, researchers have studied the concepts in introductory programming courses and the extent to which these concepts are covered. Most of these studies are fairly general, conducted either by surveys of academics [65, 260, 579] or by analysis of textbooks [68, 434]. These are supplemented by the occasional paper surveying students to explore the difficulty of specific topics such as recursion and iteration [446].

We found just two papers proposing the application of theory to the design of an introductory programming course. Sorva and Seppälä [634] illustrate how a number of design principles were applied to the design of a course, and Thota [662] describes a course design process based on the research method of phenomenography.

Notwithstanding these examples, the publications on introductory programming curriculum are overwhelmingly experience reports. The field would clearly benefit from more empirical research into the benefits or otherwise of adopting particular curricular choices.

7.1 Programming Languages

The programming language wars [639] have for decades played a major role in computing education publications, and were assured of a place in our search. Many papers argue for the use of specific programming languages in the introductory programming course, or describe how a course has been implemented with a particular programming language. Examples include ActionScript [139], Pascal [297], Scratch [550], App Inventor [689], and Karel++ [718]. Some researchers, not satisfied with any currently available languages, have devised programming languages of their own and used them to teach the course [78, 461]. Of particular interest in this area are languages, sometimes subsets of natural language, devised for students whose first language is not English; examples include Bengali [7], Japanese [481], and Spanish [482]. Figure 11 shows the number of papers per year focusing on programming languages.

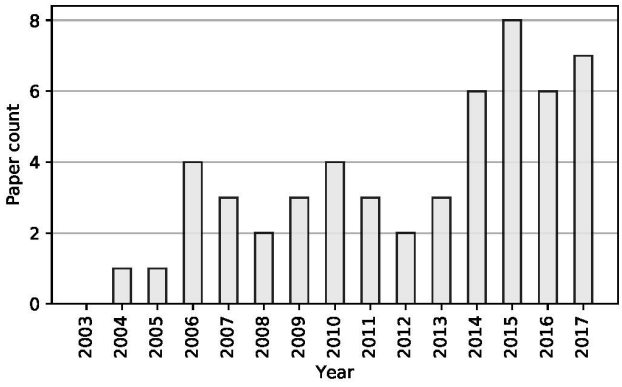

Figure 11: Number of publications per year focusing on programming languages

Given the perennial question of which language, if any, is best suited for the teaching of introductory programming, many papers describe empirical comparisons between languages, often finding no significant difference in learning outcomes, but sometimes finding differences in student satisfaction. Papers have compared Modula-2 with Java [296], C++ with Java [202], C++ with Python [33, 193], Python with Java [348], and Alice with Java [459].

Some authors address the language problem by using multiple languages in the same course. Their papers are often just descriptive in nature and positive in outlook [43, 402], but Martínez-Valdés et al. [418] somewhat refreshingly present their negative experience with the sequence of Scratch and Java.

While all of the language-based papers mentioned above have the potential to guide educators in their own choice of language, a number of authors make that guidance the focus of their papers. de Raadt et al. [152] discuss the industry and academic implications of the choice of language; Parker et al. [494] develop a number of criteria to be used in selecting a programming language; Mannila and de Raadt [410] offer an objective comparison of languages for the introductory programming course; and Stefik and Siebert [640] present an empirical investigation of the syntax of candidate languages. At a broader level, de Raadt et al. [153] and subsequently Mason and Simon [419] report on a longitudinal survey of programming languages, integrated development environments, paradigms, and other features of introductory programming courses in Australia and New Zealand over the full span of this literature review.

The choice of programming language is clearly a highly subjective one and is driven by many competing factors. It is clear that the literature will continue to host publications making the case for different programming language choices — and that no paper or series of papers will ever put an end to the debate, as no programming language will ever satisfy all of the conflicting requirements of an introductory programming course.

7.2 Programming Paradigms

Inextricably linked with the question of which programming language to use is the question of which programming paradigm to use. Some programming languages necessarily imply a corresponding paradigm: teaching with Scheme means teaching functional programming [638]. On the other hand, most object-oriented programming languages can be used either with the procedural paradigm or with the object-oriented paradigm, which are together known as the imperative paradigm. Figure 12 shows the distribution of papers relating to paradigm. Interestingly, although papers about choice of language appear to be relatively active over the past five years, papers focusing on paradigm are, if anything, shrinking in number.

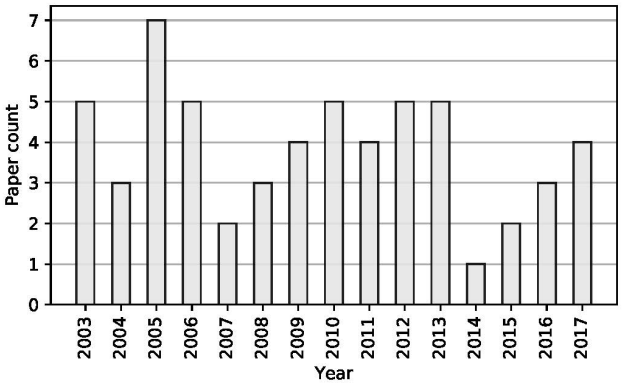

Figure 12: Number of publications per year focusing on paradigm

There are surveys on which paradigms are in use: Aleksić and Ivanović [14] note that while most European institutions support object-orientation, most of their introductory courses use the procedural paradigm. There are papers comparing the procedural and object-oriented paradigms [213, 544, 684], sometimes with the former preceding the latter in a single course [302]. There are papers comparing the functional and object-oriented paradigms [124, 246]. The object-oriented paradigm appears to dominate the field, and many papers offer guidelines for teaching within that paradigm [18, 82, 133, 379]. As with programming languages, each paradigm has its proponents, and after full consideration of all the papers it is by no means clear that any paradigm is superior to any other for the teaching of introductory programming.

8 ASSESSMENT

The design and approach to assessment in introductory programming is a large area of research. Table 7 summarises the number of papers in each category, and figure 13 shows the distribution of papers about assessment. The past two years have seen a substantial increase in publications, perhaps indicative of the growing use of learning analytics related to assessment of student learning.

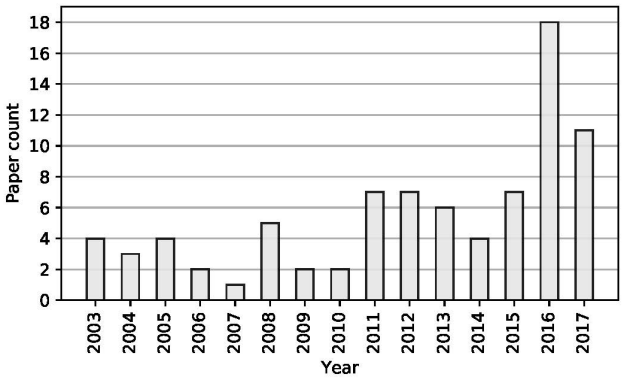

Figure 13: Number of publications per year focusing on assessment

Table 7: Classification of papers focused on assessment

Category	N	Description
Approach	16	Different ways that students are assessed
Design	21	Theories and models of assessment
Analysis	12	Exam questions and responses
Feedback	16	Tools and methods that enhance feedback
Tools	14	Automated and semi-automated assessment
Integrity	10	Academic integrity and plagiarism

8.1 Assessment Approaches

The form of assessment is recognised as a key determinant to how students approach their learning. We found a variety of approaches to exam and in-semester assessment of introductory programming.

Different approaches to the form and conduct of final examinations were reported. A comparative study by Jevinger and Von Hausswolff [305] found strengths and weaknesses in two distinct styles of exam, a computer-based (lab) exam comprising one large programming task and a paper-based exam where students are given a series of smaller tasks.

Lab exams are used as a way of providing a valid assessment of programming efficacy. Students and staff perceive the assessment to be fair and an accurate assessment of programming skills [63, 236, 715] and to improve student motivation [52]. Students working on computers produce more accurate code than on paper [360]. In a study by Haghighi and Sheard [236] involving both paper-based and computer-based assessment, students found the computer test environment to be more stressful; however, ten years later, Matthíasdóttir and Arnalds [420] found that students were more comfortable taking exams on a computer than on paper.

Exploring the use of a 'cheat sheet' of hand-written notes in a closed-book written exam, de Raadt [150] found that thoughtfully prepared cheat sheets were found to improve student performance and support students without incurring the disadvantages of open-book exams.

Different forms of group assessment have been discussed. Sindre et al. [618] report the use of open-ended group assignments and the

use of collaborative exams where students work in small groups to solve a set of problems. Collaborative exams were most beneficial to mid-performing students and worked best when group members had comparable levels of ability.

Cain and colleagues [102, 105, 546] report a portfolio assessment model where students undertake formative assessment throughout the semester and are assessed solely on a portfolio submitted at the end of semester; this approach was found to improve student performance and bring about high levels of student and staff satisfaction.

Different approaches to grading assessments have been reported. Rather than the typical practice of using a simple binary marking scale for multiple-choice questions, Simon [605] describes a marking scheme where part marks are allocated for some distractors. LeJeune [368] reports that a contract grading scheme used to support mastery learning was preferred by students over traditional grading schemes.

Despite the important role of assessment in how students approach learning, only one detailed study of the student perspective was found. A phenomenographic study of students' experiences of assessment by Riese [547] found that assessment as guidance, assessment as an opportunity to learn, and assessment as a way to communicate were important to students.

Much of the research on assessment approaches has focused on the validity and fairness of the assessment and on student performance. With assessment taking a variety of forms, there is potential for further research on how these different forms influence students' approach to learning programming.

8.2 Assessment Design

Designing assessment according to a theoretical basis can strengthen the validity and quality of the assessment process. Although we found many different approaches to designing assessment and many different assessment designs, we found few references to theory. An interview study of programming teachers by Sheard et al. [596] found variation in their approaches to developing final exams but little evidence that they based their designs on learning theories or models.

Some researchers explore the use of theories or taxonomies to improve the quality of assessment. Bloom's taxonomy was used by Anderson et al. [25] to devise a criterion-referenced grading scheme for students grouped into one of three levels according to their ability. This enabled assessments to be set to challenge the stronger students without leaving the weaker students to flounder [384]. Bloom's taxonomy was used to classify and review a set of exam questions, provide insights into the cognitive level of difficulty of the questions [661], and determine the appropriate level of challenge for test questions [174]. However, Shuhidan et al. [603] found that it was difficult to classify multiple-choice questions according to Bloom's taxonomy.

A number of studies use Biggs's SOLO taxonomy [76] to explore exam questions and students' responses. SOLO has been used by Shuhidan et al. [603] and Whalley et al. [701] to gain a deeper understanding of the programming process, and by Ginat and Menashe [217] and Sheard et al. [593] to assess the depth of programming knowledge and understanding. In a study by Castro and Fisler [118], SOLO was expanded to a multi-strand form to enable assessment of different performance levels with sets of programming design skills.

Software metrics have been explored as way to estimate and compare the difficulty of code-writing tasks. Whalley and Kasto [703] proposed that in an educational context software metrics should be used in conjunction with Bloom's taxonomy or SOLO to incorporate the cognitive levels required to answer the questions.

Biggs' model of constructive alignment [75] has been used by Cain and Woodward [104] as a foundation for the design of a portfolio assessment model.

We found examples of assessment designed to encourage a particular behaviour or a specific learning outcome. One case is an assessment approach by Edwards [185] designed to develop analytical skills through test-driven development, in which students are required to demonstrate program correctness through testing. A continuous assessment model developed by Duch Brown et al. [176] to encourage students to work consistently was found to have no impact on course success rates but to increase the workload of staff. Open-ended assignments were used by VanDeGrift [677] to encourage creativity, and Stevenson and Wagner [642] found that real-world assignments with an open-ended element allow opportunities for creativity while also providing challenge and interest for more advanced students.

In some cases assessment techniques are designed to assess specific learning outcomes. Memory diagrams were used by Holliday and Luginbuhl [263] as a tool to assess students' understanding of programming concepts, and a marking scheme was developed by de Raadt et al. [154] to assess the application and comprehension of programming strategies that were explicitly taught in a course.

Some studies have explored the use of specific question types. A new style of exam question, the Parsons problem [495], has been evaluated by Denny et al. [160], and versions of Soloway's rainfall problem [624] have been used and evaluated as exam questions by Simon [606] and Lakanen et al. [357]. Craig et al. [138] find that setting a question in a familiar context has no impact on student results.

In summary, we have found a number of examples of the use of SOLO and Bloom's taxonomies to investigate assessment instruments, but few examples of theory being used as a basis for an overall assessment approach. We see potential for further research in this area.

8.3 Assessment Analysis

The instrument used for assessment is clearly a key component of an assessment task, particularly an exam or a test. Analysis of the characteristics of assessment instruments can provide a detailed understanding of how students are assessed and of the alignment of the assessment with the learning outcomes.

The characteristics of assessment instruments have been analysed to understand how students are assessed. Detailed analyses of written exam papers from institutions in a number of countries explore the types of question used [608], the difficulty of the questions [610], and their complexity [595]. Questions suitable for benchmarking have been proposed and trialled in exams in multiple

institutions by Sheard, Simon, and colleagues [597, 612, 613]. Another analysis of exam papers from multiple institutions by Petersen et al. [506] finds that most questions focus on code writing and involve multiple concepts, making them challenging for novices. In a follow-up study, single-concept questions were trialled and were found to be predictors of overall performance on code-writing questions. A study by Zingaro et al. [733] found that these single-concept questions can simplify the marking process and provide more accurate feedback. A more recent study by Luxton-Reilly and Petersen [396] analyses exam questions to determine the syntax elements involved in each question, giving a detailed picture of what is assessed and confirming that questions typically involve multiple conceptual components.

A detailed analysis of student performance on multiple-choice exam questions was performed by Lister [380] with the aim of developing criteria for evaluating this type of question. In another study of multiple-choice questions, by Lulis and Freedman [391], instructors were able to consistently rate the level of difficulty of the questions, and found that it correlated with student performance on these questions.

Analysis of student responses to code-writing questions by Kasto et al. [317] found a relationship between the number of possible solutions and the question difficulty as measured by students' performance on these questions.

In summary, analysis of exam assessment instruments and student performance, often across multiple institutions, has revealed a wealth of information about this form of assessment. The style of exam papers varies greatly across courses, leaving an untapped potential for sharing questions and benchmarking student performance across courses and institutions.

8.4 Feedback on Assessment

High quality feedback as part of summative and formative assessment is regarded as a vital element in improving learning in introductory programming courses [487]. Feedback reinforces students' self-regulated learning, allowing them to set their own goals, monitor their performance according to these goals, and implement and adjust their learning strategies [486]. The majority of the identified papers focus on creating tools for providing automated feedback, a direct consequence of large-scale introductory programming classes [253]. The tools focus mainly on improving the accuracy of fault-finding through various methods [30, 253, 278, 619] and on improving the timeliness of feedback by automatically reusing feedback generated for previous solutions [252, 253].

A few works identify the need to create and provide individualised feedback to students. This can be achieved through the generation of hints from a predefined taxonomy as proposed by Suzuki et al. [647], by using a mixed-initiative approach that combines both teacher expertise and solution structures derived through program synthesis techniques as suggested by Head et al. [252], or by providing students with clear, plain-English correction rules that they can follow in order to correct their code, in effect enhancing compiler messages, as analysed in the study by Singh et al. [619]. We found only the study of Stegeman et al. [641] to focus on providing feedback on code quality, which was done through the use of a generic rubric. Most papers identify the need for continuous and timely feedback, which can potentially be achieved through summative in-class activities. One mechanism proposed by Cicirello [129] for continuous feedback uses unannounced 'pop' quizzes, which were found to improve students' performance.

While we found some works that focus on the design of effective feedback, such as the study by Ott et al. [487], the majority of papers focus on the use of a mixed-initiative approach for designing feedback and on ensuring that feedback is provided in diverse forms such as automated, manual, written, oral, or visual. In summary, despite recognition of the critical importance of high-quality feedback to improve the learning outcomes, motivation, and engagement of novice students, most of the work concentrates on the automated generation of feedback, generally by enhancing compiler messages or generating hints, approaches that have not been demonstrated to be effective.

8.5 Assessment Tools

The papers discussed in this section present and evaluate tools developed specifically for formative or summative assessment.

We identify two main trends in the design of assessment tools: the automation of assessment processes to reduce the cost of assessment, and the design of more informative assessment tools through the use of test suites. Automated or semi-automated tools are designed mainly to facilitate the marking process from the instructor's perspective, thereby reducing the cost of marking, especially in large introductory classes. Beyond this, the tools show significant variation in their development approach, using mutation analysis [1], program repair [492], or formal verification tools [687] among others.

Test cases or test suites that students can repeatedly query with their submission are used to improve the quality of the feedback given by the tool. This includes custom-made tools such as ProgTest [636] and industrial automated test suites [214]. As with the tools focused on automated marking, the methods for creating the tests vary significantly, including automated UML testing [259] and the use of reference implementations created by the lecturer to generate test cases [308]. The use of automated test suites supports research showing that students become frustrated when feedback is given after the final submission of the assignment, when it is too late for students to address it [477]. On the other hand, it is reported that feedback merely listing specific failed test cases can lead students to running resubmission cycles with small incremental changes, focused only on passing test cases and not on the software product as a whole [293].

A small number of papers consider the design of self-assessment tools or protocols or the implementation of tools to support student self-assessment. Self-assessment and reflection are critical pillars in the development of self-regulating learning behaviours, which are crucial for student success.

A number of tools focus on giving students feedback on their programming process. Bluefix [695], which can be integrated into the BlueJ IDE, provides programming students with crowd-sourced error diagnosis and repair feedback. NoobLab [470] is an online learning environment that attempts to provide an analogue of the traditional in-lab tutor-student dynamic that is specific to programming. Watson et al. [696] present a tool that combines implicit

and explicit feedback measures to provide real-time assistance in programming tasks. Hundhausen and Brown [285] report on an experimental study that investigated the impact of feedback self-selection on novice imperative programming. Their within-subjects design compared the impact of three different levels of feedback on syntactic and semantic correctness: no visual feedback at all; visual feedback, in the form of a visualisation of the program's execution state, provided on request when a 'run' button is hit; and visual feedback, in the form of a visualisation of the program's execution state, updated on every keystroke. Participants in the second and third treatments produced programs with significantly fewer syntactic and semantic errors than those given no visual feedback; however, no significant differences were found between the two visual feedback treatments, suggesting that the benefits of delivering a continuously updated visual representation of a program's execution may fail to justify the substantial costs of implementing such feedback.

PRAISE [151] facilitates anonymous review and delivers prompt feedback from multiple sources including peer-generated feedback. Alemán et al. [15] report statistically significant empirical results for an online multiple-choice question system integrated with a neural network that generates diagnostic feedback. RedPencil [9] seeks to provide more timely and targeted feedback than is possible with traditional learning management systems. Zapušek et al. [727] use machine learning techniques to automatically distinguish between 'basic' and 'advanced' styles with Python submissions. More generally, Boada et al. [80] and Yoo et al. [723] both present systems that broadly but specifically support teaching and learning programming, including features such as laboratory feedback and progress tracking.

Tool suitability has been analysed from the point of view of the effectiveness of enhanced compiler messages [509], or by focusing on the usefulness of automatically generated hints [511].

Our analysis finds that most tools focus on reducing the educators' workload by automating testing, by providing a platform for peers to provide feedback, or by using machine learning techniques to generate feedback. Possibly because the tools tend to be used at large scale, we found no comparisons of the effectiveness of personalised, educator-written feedback with that generated by tools in their various forms. There also appear to be opportunities for controlled experiments comparing the various ways of giving feedback so as to better understand the conditions under which they are most effective.

8.6 Academic Integrity

Plagiarism and other forms of cheating are a huge issue in most courses, impacting the students, the teachers, and the integrity of the learning and teaching environment. We found only a small number of papers that deal with academic integrity in introductory programming courses. There is a large corpus of literature on cheating and plagiarism in computing courses, particularly in relation to programming, but apparently few of them focus on the introductory course.

A comprehensive study of introductory programming courses [594] found 21 different strategies used by academics to reduce levels of cheating in their courses. These were targeted at reducing cheating through education, discouragement, making cheating difficult, and empowering students to take responsibility for their behaviour. The study found that academics often employ a range of strategies across these areas.

Most papers focus on tools or methods to detect plagiarism. An evaluation of 12 code-similarity detection tools and their suitability for use in introductory programming courses [449] considers the different techniques used by students to disguise their code. Although all tools showed merit, academic oversight was recommended when using the tools for plagiarism detection.

High levels of naturally occurring similarity between coding solutions make detection of plagiarism more difficult. An investigation of the level of natural similarity in different types of programming assessment [88] concludes that tasks with divergent solutions lead to more valid assessment. Improvement on existing methods for detecting code similarity was found by comparing low-level instructions rather than source code tokens [315] and by applying unsupervised learning techniques to features extracted from intermediate compilation phases [719].

An innovative technique for detection of plagiarism [146], based on the application of a digital watermark to students' submissions, was shown to be a useful technique for distinguishing between the supplier and the recipient when program copying was detected. In another study [654], discrepancies in grades between an invigilated test and a take-home assignment were used to check for cases where a student might have had unauthorised external help.

We found only one study that explored the student perspective to gain insights into violations of academic integrity. Hellas et al. [257] investigated a case of suspected cheating in take-home exams. Interviews with the students elicited three types of plagiarism behaviour: help-seeking, collaboration, and systematic cheating. Analysis of log file data collected while students completed the exam showed patterns that could potentially be used to indicate the occurrence of plagiarism.

The 2016 ITiCSE Working Group of Simon et al. [614] investigated academic integrity in relation to computing courses. A key finding of the study was that academics often take different approaches to prevent and deal with academic integrity violations depending on the level of the course. The small number of papers dealing with academic integrity in introductory programming courses is therefore a concern. Furthermore, the emphasis on detection rather than education indicates that this is an area in need of further research.

9 LIMITATIONS

The method of Kitchenham [335] for performing a review of literature in a relatively narrow area entails certain limitations, some of which might be amplified by our adaptation of the method to a far broader field. The main limitations of this review are:

- bias in the selection of databases, journals, and publications, due to possible subjectivity and lack of relevant information (although we did consider a broad range of relevant venues);
- bias in the search string because keywords are discipline- and language-specific;

- the possibility of false negatives, papers not identified by our search terms, estimated in section 3.2 to be about 15% of the papers found;
- the omission from consideration of papers that members of the working group were unable to access, estimated in section 3.4 to be about 1% of the papers found;
- bias in uniformity and potential inaccuracy in data extraction as it was performed by different authors (although all of them have expertise in computing education research);
- bias from interpretation of some findings, methods, or approaches, as some parts of the methodology from the selected studies were not described in detail.

The authors attempted to reduce these limitations by developing the research protocol and research questions in advance. The search terms were chosen for pragmatic reasons: the terms captured a substantial number of the papers from our sample set, without missing too many. However, selecting search terms is a balance between the numbers of false positives and false negatives. Choosing different terms would have resulted in some differences to our initial data set, particularly as there is a lack of standardisation in the keywords used by authors.

10 DISCUSSION AND CONCLUSIONS

While we have tried to be comprehensive in our subject matter, there are many potentially relevant papers that are not listed in this review, typically because other papers had reported similar things, or because their contribution was relatively small. Additionally, papers about topics that apply across the broader spectrum of computing are unlikely to refer specifically to introductory programming, so while they may inform the field, they are not discussed here. We are clearly not in a position to assert that this review covers all possible papers on introductory programming. However, we consider that our approach has identified a broad cross-section of the literature and should be regarded as a relatively large and objective sample of the work on introductory programming as at mid-2018.

10.1 The Focus of the Literature

Below we present the categories and subcategory descriptions of introductory programming papers that we reviewed. The largest number of papers that we reviewed focused on students, followed by teaching, then curriculum, and finally assessment. The number of papers corresponding to each category/description is in parentheses. Two subcategories are broken down further in Table 4 (Teaching — Techniques and activities used to teach) and Table 5 (Teaching — Tools that support teaching and learning).

- Students — Table 2 (539)
  - Content (313)
    - Measuring student ability (169)
    - Measurements of student activity (69)
    - Code reading, writing, debugging (58)
    - Models of student understanding (17)
  - Sentiment (184)
    - Student attitudes (105)
    - Measuring/improving engagement (61)
    - Experiences of programming (18)
  - Subgroups (42)
    - Women and minorities (25)
    - Students at risk of failing (17)
- Teaching — Table 3 (497)
  - Tools that support teaching and learning (254)
  - Techniques and activities used to teach (156)
  - Overall course structure (53)
  - Models, taxonomies, and other theories (19)
  - Institutional and environmental support (15)
- Curriculum — Table 6 (225)
  - Curriculum in general (102)
  - Choice of programming language (70)
  - Choice of programming paradigm (53)
- Assessment — Table 7 (89)
  - Theories and models of assessment (21)
  - Different ways that students are assessed (16)
  - Tools and methods that enhance feedback (16)
  - Automated and semi-automated assessment (14)
  - Exam questions and responses (12)
  - Academic integrity and plagiarism (10)

10.2 Developing Areas

While there is clear growth in the number of publications about introductory programming, this growth appears largely outside the ACM SIGCSE venues. Not all of the topic categories show an increase in publication numbers. Notably, publications that focus on theory (both theory related to students and theory related to teaching) are not growing at the rate of other topic areas. We also observe that the number of publications related to overall approaches to teaching delivery, teaching infrastructure, and curriculum are relatively stable. One interpretation of this trend is that there is less innovation or discussion in these areas, perhaps reflecting the growing maturity of the field with less scope for controversy, or perhaps reflecting increasing publications in areas that are 'easy' to publish about, such as the development of new tools, while avoiding more 'difficult' research areas that require substantial course and curriculum redesign.

10.3 Reported Evidence

While undertaking the review we became aware of several issues relating to the generalisability and validity of evidence provided by papers, which are discussed in this section.

A large proportion of the publications provide little detail about the context in which the report (activity/study/experience) is embedded. For a reader to determine if the results of the paper are transferable to their own teaching context, it is important to know details about the population of students and the teaching context. In many studies, the construct that was being measured was not formally defined, the operationalisation of the construct was not described, and effect sizes were not reported. For researchers, such reports do not have sufficient detail to replicate the studies, enabling the research community to obtain valid and reliable results, which in turn would allow for more informed recommendations for teaching practice.

For future publications, we recommend that authors report much more detailed information about the context in which they are operating, and, where possible, publicly archive detailed information about their syllabus, learning outcomes, infrastructure, teaching approaches, and population demographics, for the reference of future researchers.

A large proportion of the publications that we examined focus on tools developed by their authors. While much work has clearly gone into building tools to help students learn programming, comparatively little attention has been paid to evaluating the tools, or disseminating them for use beyond the institution where they were developed.

For tools or approaches that are used to identify areas of concern (such as predictors of success, student misconceptions, identification of at-risk students), there is a tendency to report on the success of the innovation, but there is not enough focus on what to do once an issue is identified: what remedies should be applied? Systematic studies are needed to further investigate whether the tools and approaches actually result in improved outcomes for students, rather than simply providing additional information for teachers.

We hope that this situation will improve, with the increased emphasis on rigorous evaluation in computing education research and the increased need for tools to help the burgeoning computing student population.

Despite many years of study and debate, some significant issues in teaching programming remain unresolved and probably unresolvable. Ideological differences over choice of language, paradigm (such as functional vs imperative), and organisational approach (such as objects early vs objects late) are obvious examples.

Although introductory programming courses are considered to cause significant levels of student anxiety and frustration, and student engagement levels in computing are benchmarked as among the lowest of any discipline [454], there are very few publications focusing on the student experience, particularly with respect to pastoral care. This is an important area that appears to be underexplored, and we feel that the community would benefit from further work in this area.

Finally, most publications involving the collection of data from students report that students enjoyed themselves and found the activity that was the focus of the paper to be interesting and engaging. This is likely to be at least partially due to publication bias, with only successful interventions being reported. However, it also suggests that a very wide range of activities and approaches can make introductory programming courses interesting and engaging. This contrasts with broader benchmarks of levels of engagement in computing and reported high dropout rates. Given that almost anything we try appears to work, these conflicting results suggest that the broader computing education community might benefit from increased engagement with SIGCSE and computing education conferences. The authors of most of the papers that we have examined appear to be teaching in an effective and engaging manner; with greater dissemination among teachers who are not part of the SIGCSE community, their experiences could benefit the entire computing education community.

The perception, and ultimately the success, of computing as a discipline could be substantially improved by continued reporting of case studies and experiences of engaging activities that teachers could adopt in their classrooms, and by increased dissemination encouraging the adoption of these approaches.

10.4 Contributions

In conclusion, this report presents insights from a systematic literature review in the area of introductory programming education over the past 15 years, making at least the following contributions:

- identifying the aspects of introductory programming have been the focus of publications;
- summarising the developments in research and practice with regard to introductory programming education over the past 15 years;
- highlighting the evidence that has been reported with regard to these developments;
- discussing how the findings can be used to design future introductory programming courses, as well as to advance research in the area of introductory programming.

ACKNOWLEDGMENTS

We are greatly appreciative of the guidance and advice offered by the anonymous reviewers of this report and by the ITiCSE working group chairs. Where we have failed to act on their suggestions, it is not because we disagree with the suggestions but because it was not practical to implement them.

REFERENCES

[1] Kalle Aaltonen, Petri Ihantola, and Otto Seppälä. 2010. Mutation analysis vs. code coverage in automated assessment of students' testing skills. In Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA '10). ACM, New York, NY, USA, 153–160. https://doi.org/10.1145/1869542.1869567

[2] Samy S. Abu Naser. 2009. Evaluating the effectiveness of the CPP-Tutor, an Intelligent Tutoring System for students learning to program in C++. Journal of Applied Sciences Research 5, 1 (2009), 109–114.

[3] Michał Adamaszek, Piotr Chrzaçstowski-Wachtel, and Anna Niewiarowska. 2008. VIPER a student-friendly visual interpreter of Pascal. In Informatics Education — Supporting Computational Thinking, Roland T. Mittermeir and Maciej M. Sysło (Eds.). Springer, Berlin, Heidelberg, 192–203. https://doi.org/10.1007/978-3-540-69924-8_18

[4] Abejide Ade-Ibijola, Sigrid Ewert, and Ian Sanders. 2014. Abstracting and narrating novice programs using regular expressions. In Proceedings of the Southern African Institute for Computer Scientist and Information Technologists Annual Conference 2014 on SAICSIT 2014 Empowered by Technology (SAICSIT '14). ACM, New York, NY, USA, Article 19, 19:19–19:28 pages. https://doi.org/10.1145/2664591.2664601

[5] Achla Agarwal, Krishna K. Agarwal, Leslie Fife, and M. Emre Celebi. 2016. Raptor and Visual Logic©: a comparison of flowcharting tools for CS0. J. Comput. Sci. Coll. 31, 4 (2016), 36–41. http://dl.acm.org/citation.cfm?id=2904127.2904132

[6] Alireza Ahadi, Raymond Lister, Shahil Lal, Juho Leinonen, and Arto Hellas. 2017. Performance and consistency in learning to program. In Proceedings of the Nineteenth Australasian Computing Education Conference (ACE '17). ACM, New York, NY, USA, 11–16. https://doi.org/10.1145/3013499.3013503

[7] Nova Ahmed, Arman Kamal, Adnan Nuruddin, and Syed Tanveer Jishan. 2016. My code in my native tone: Cha Script. In Proceedings of the Eighth International Conference on Information and Communication Technologies and Development (ICTD '16). ACM, New York, NY, USA, Article 28, 28:1–28:4 pages. https://doi.org/10.1145/2909609.2909611

[8] Tuukka Ahoniemi and Essi Lahtinen. 2007. Visualizations in preparing for programming exercise sessions. Electronic Notes in Theoretical Computer Science 178 (2007), 137–144. https://doi.org/10.1016/j.entcs.2007.01.043

[9] T. C. Ahren. 2005. Using online annotation software to provide timely feedback in an introductory programming course. In IEEE Frontiers in Education Conference (FIE '05). IEEE, T2H–1. https://doi.org/10.1109/FIE.2005.1611917

[10] Nouf M. Al-Barakati and Arwa Y. Al-Aama. 2009. The effect of visualizing roles of variables on student performance in an introductory programming course. In Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and

Technology in Computer Science Education (ITiCSE '09). ACM, New York, NY, USA, 228–232. https://doi.org/10.1145/1562877.1562949

[11] Ahmad Al-Jarrah and Enrico Pontelli. 2014. "AliCe-ViLlagE" Alice as a Collaborative Virtual Learning Environment. In IEEE Frontiers in Education Conference (FIE '14). IEEE, 1–9. https://doi.org/10.1109/FIE.2014.7044089

[12] Ali Alammary, Angela Carbone, and Judy Sheard. 2012. Implementation of a smart lab for teachers of novice programmers. In Proceedings of the Fourteenth Australasian Computing Education Conference — Volume 123 (ACE '12). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 121–130. http://dl.acm.org/citation.cfm?id=2483716.2483731

[13] Ahmed S. Alardawi and Agil M. Agil. 2015. Novice comprehension of object-oriented OO programs: an empirical study. In 2015 World Congress on Information Technology and Computer Applications (WCITCA). IEEE, 1–4. https://doi.org/10.1109/WCITCA.2015.7367057

[14] Veljko Aleksić and Mirjana Ivanović. 2016. Introductory programming subject in European higher education. Informatics in Education 15, 2 (2016), 163–182. https://doi.org/10.15388/infedu.2016.09

[15] José Luis Fernández Alemán, Dominic Palmer-Brown, and Chrisina Draganova. 2010. Evaluating student response driven feedback in a programming course. In 2010 10th IEEE International Conference on Advanced Learning Technologies. IEEE, 279–283. https://doi.org/10.1109/ICALT.2010.82

[16] Laura K. Alford, Mary Lou Dorf, and Valeria Bertacco. 2017. Student perceptions of their abilities and learning environment in large introductory computer programming courses. In ASEE Annual Conference and Exposition, Conference Proceedings, Vol. 2017-June. ASEE.

[17] Carl Alphonce and Blake Martin. 2005. Green: a pedagogically customizable round-tripping UML class diagram Eclipse plug-in. In Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange (eclipse '05). ACM, New York, NY, USA, 115–119. https://doi.org/10.1145/1117696.1117720

[18] Carl Alphonce and Phil Ventura. 2003. Using graphics to support the teaching of fundamental object-oriented principles in CS1. In Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '03). ACM, New York, NY, USA, 156–161. https://doi.org/10.1145/949344.949391

[19] Fatima AlShamsi and Ashraf Elnagar. 2009. JLearn-DG: Java learning system using dependence graphs. In Proceedings of the 11th International Conference on Information Integration and Web-based Applications &Amp; Services (iiWAS '09). ACM, New York, NY, USA, 633–637. https://doi.org/10.1145/1806338.1806458

[20] Maria Altebarmakian, Richard Alterman, Anna Yatskar, Kendall Harsch, and Antonella DiLillo. 2016. The microgenetic analysis of staged peer collaboration for introductory programming. In IEEE Frontiers in Education Conference (FIE '16). IEEE, 1–8. https://doi.org/10.1109/FIE.2016.7757613

[21] Raad A. Alturki. 2016. Measuring and improving student performance in an introductory programming course. Informatics in Education 15, 2 (2016), 183–204. https://doi.org/10.15388/infedu.2016.10

[22] Omar AlZoubi, Davide Fossati, Barbara Di Eugenio, Nick Green, Mehrdad Alizadeh, and Rachel Harsley. 2015. A hybrid model for teaching recursion. In Proceedings of the 16th Annual Conference on Information Technology Education (SIGITE '15). ACM, New York, NY, USA, 65–70. https://doi.org/10.1145/2808006.2808030

[23] Milan Amaratunga, Gayan Wickramasinghe, Milinda Deepal, Oshani Perera, Dilshan De Silva, and Samantha Rajapakse. 2013. An interactive programming assistance tool (iPAT) for instructors and novice programmers. In 2013 8th International Conference on Computer Science & Education. IEEE, 680–684. https://doi.org/10.1109/ICCSE.2013.6553995

[24] Ana Paula L. Ambrósio and Fábio M. Costa. 2010. Evaluating the impact of PBL and tablet PCs in an algorithms and computer programming course. In Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10). ACM, New York, NY, USA, 495–499. https://doi.org/10.1145/1734263.1734431

[25] Lorin W Anderson, David R Krathwohl, Peter W Airasian, Kathleen A Cruikshank, Richard E Mayer, Paul R Pintrich, James Raths, and Merlin C Wittrock. 2001. A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives (abridged ed.). Longman, White Plains, NY.

[26] Ruth E. Anderson, Michael D. Ernst, Robert Ordóñez, Paul Pham, and Ben Tribelhorn. 2015. A data programming CS1 course. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15). ACM, New York, NY, USA, 150–155. https://doi.org/10.1145/2676723.2677309

[27] Subashini Annamalai and Sobihatun Nur Abdul Salam. 2017. A multimedia approach towards learning C programming: a discussion from novice learners' perspective. Journal of Telecommunication, Electronic and Computer Engineering 9, 2-12 (2017), 99–103.

[28] D. F. Ninan S. A. Akinboro H. Abimbola Soriyan Anuoluwapo Ajayi, Emmanuel A. Olajubu. 2010. Development and testing of a graphical FORTRAN learning tool for novice programmers. Interdisciplinary Journal of Information, Knowledge, and Management 5 (2010), 277–291. https://doi.org/10.28945/1176

[29] Masayuki Arai and Tomomi Yamazaki. 2006. Design of a learning support system to aid novice programmers in obtaining the capability of tracing. In Proceedings of the Sixth International Conference on Advanced Learning Technologies (ICALT '06). IEEE, 396–397.

[30] Eliane Araujo, Matheus Gaudencio, Dalton Serey, and Jorge Figueiredo. 2016. Applying spectrum-based fault localization on novice's programs. In IEEE Frontiers in Education Conference (FIE '16). IEEE, 1–8. https://doi.org/10.1109/FIE.2016.7757727

[31] Glen Archer, Briana Bettin, Leonard Bohmann, Allison Carter, Christopher Cischke, Linda M Ott, and Leo Ureel. 2017. The impact of placement strategies on the success of students in introductory computer science. In IEEE Frontiers in Education Conference (FIE '17). IEEE, 1–9. https://doi.org/10.1109/FIE.2017.8190526

[32] Hazleen Aris. 2015. Improving students performance in introductory programming subject: a case study. In 10th International Conference on Computer Science & Education (ICCSE '15). IEEE, 657–662. https://doi.org/10.1109/ICCSE.2015.7250328

[33] Muhammad Ateeq, Hina Habib, Adnan Umer, and Muzammil Ul Rehman. 2014. C++ or Python? Which one to begin with: a learner's perspective. In International Conference on Teaching and Learning in Computing and Engineering (LaTiCE '14). IEEE, 64–69. https://doi.org/10.1109/LaTiCE.2014.20

[34] Nikolaos Avouris, Stefanos Kaxiras, Odysseas Koufopavlou, Kyriakos Sgarbas, and Polyxeni Stathopoulou. 2010. Teaching introduction to computing through a project-based collaborative learning approach. In 2010 14th Panhellenic Conference on Informatics. IEEE, 237–241. https://doi.org/10.1109/PCI.2010.13

[35] John Aycock, Etienne Pitout, and Sarah Storteboom. 2015. A game engine in pure Python for CS1: design, experience, and limits. In Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '15). ACM, New York, NY, USA, 93–98. https://doi.org/10.1145/2729094.2742590

[36] Mewati Ayub, Hapnes Toba, Steven Yong, and Maresha C. Wijanto. 2017. Modelling students' activities in programming subjects through educational data mining. Global Journal of Engineering Education 19, 3 (2017), 249–255.

[37] David Azcona and Alan F. Smeaton. 2017. Targeting at-risk students using engagement and effort predictors in an introductory computer programming course. In Data Driven Approaches in Digital Education. Springer International Publishing, 361–366. https://doi.org/10.1007/978-3-319-66610-5_27

[38] Shahdatunnaim Azmi, Noorminshah A Iahad, and Norasnita Ahmad. 2016. Attracting students' engagement in programming courses with gamification. In IEEE Conference on e-Learning, e-Management and e-Services (IC3e '16). 112–115. https://doi.org/10.1109/IC3e.2016.8009050

[39] Monica Babes-Vroman, Isabel Juniewicz, Bruno Lucarelli, Nicole Fox, Thu Nguyen, Andrew Tjang, Georgiana Haldeman, Ashni Mehta, and Risham Chokshi. 2017. Exploring gender diversity in CS at a large public R1 research university. In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17). ACM, New York, NY, USA, 51–56. https://doi.org/10.1145/3017680.3017773

[40] Suzan Badri, James Denholm-Price, and James Orwell. 2011. Layout for learning: designing an interface for students learning to program. In Proceedings of the 3rd International Conference on Computer Supported Education (CSEDU '11), Vol. 1. INSTICC, SciTePress, 324–332. https://doi.org/10.5220/0003346403240332

[41] Yu Bai, Liqian Chen, Gang Yin, Xinjun Mao, Ye Deng, Tao Wang, Yao Lu, and Huaimin Wang. 2017. Quantitative analysis of learning data in a programming course. In Database Systems for Advanced Applications. Springer International Publishing, 436–441. https://doi.org/10.1007/978-3-319-55705-2_37

[42] Bridget Baird and Christine Chung. 2010. Expanding CS1: applications across the liberal arts. J. Comput. Sci. Coll. 25, 6 (2010), 47–54. http://dl.acm.org/citation.cfm?id=1791129.1791139

[43] Olle Bälter and Duane A. Bailey. 2010. Enjoying Python, processing, and Java in CS1. ACM Inroads 1, 4 (2010), 28–32. https://doi.org/10.1145/1869746.1869758

[44] Albert Bandura. 1962. Social learning through imitation. In Nebraska Symposium on Motivation. Univer. Nebraska Press, Oxford, England, 211–274.

[45] Mousumi Banerjee, Michelle Capozzoli, Laura McSweeney, and Debajyoti Sinha. 1999. Beyond kappa: a review of interrater agreement measures. The Canadian Journal of Statistics / La Revue Canadienne de Statistique 27, 1 (1999), 3–23. http://www.jstor.org/stable/3315487

[46] Lecia J. Barker, Charlie McDowell, and Kimberly Kalahar. 2009. Exploring factors that influence computer science introductory course students to persist in the major. In Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE '09). ACM, New York, NY, USA, 153–157. https://doi.org/10.1145/1508865.1508923

[47] Ian Barland. 2008. Some methods for teaching functions first using Java. In Proceedings of the 46th Annual Southeast Regional Conference on XX (ACM-SE 46). ACM, New York, NY, USA, 256–259. https://doi.org/10.1145/1593105.1593172

[48] Glenda Barlow-Jones and Duan van der Westhuizen. 2017. Problem solving as a predictor of programming performance. In Communications in Computer and Information Science. Springer International Publishing, 209–216. https:

//doi.org/10.1007/978-3-319-69670-6_14

[49] Tiffany Barnes, Eve Powell, Amanda Chaffin, and Heather Lipford. 2008. Game2Learn: improving the motivation of CS1 students. In Proceedings of the 3rd International Conference on Game Development in Computer Science Education (GDCSE '08). ACM, New York, NY, USA, 1–5. https://doi.org/10.1145/1463673.1463674

[50] Valerie Barr and Deborah Trytten. 2016. Using turing's craft codelab to support CS1 students as they learn to program. ACM Inroads 7, 2 (2016), 67–75. https://doi.org/10.1145/2903724

[51] Jordan Barria-Pineda, Julio Guerra, Yun Huang, and Peter Brusilovsky. 2017. Concept-level knowledge visualization for supporting self-regulated learning. In Proceedings of the 22nd International Conference on Intelligent User Interfaces Companion (IUI '17 Companion). ACM, New York, NY, USA, 141–144. https://doi.org/10.1145/3030024.3038262

[52] João Paulo Barros, Luís Estevens, Rui Dias, Rui Pais, and Elisabete Soeiro. 2003. Using lab exams to ensure programming practice in an introductory programming course. In Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '03). ACM, New York, NY, USA, 16–20. https://doi.org/10.1145/961511.961519

[53] Jessica D. Bayliss and Sean Strout. 2006. Games as a "flavor" of CS1. In Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '06). ACM, New York, NY, USA, 500–504. https://doi.org/10.1145/1121341.1121498

[54] Jennifer Bayzick, Bradley Askins, Sharon Kalafut, and Michael Spear. 2013. Reading mobile games throughout the curriculum. In Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13). ACM, New York, NY, USA, 209–214. https://doi.org/10.1145/2445196.2445264

[55] Leland L. Beck, Alexander W. Chizhik, and Amy C. McElroy. 2005. Cooperative learning techniques in CS1: design and experimental evaluation. In Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '05). ACM, New York, NY, USA, 470–474. https://doi.org/10.1145/1047344.1047495

[56] Brett A. Becker. 2016. An effective approach to enhancing compiler error messages. In Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16). ACM, New York, NY, USA, 126–131. https://doi.org/10.1145/2839509.2844584

[57] Brett A. Becker. 2016. A new metric to quantify repeated compiler errors for novice programmers. In Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '16). ACM, New York, NY, USA, 296–301. https://doi.org/10.1145/2899415.2899463

[58] Brett A. Becker, Graham Glanville, Ricardo Iwashima, Claire McDonnell, Kyle Goslin, and Catherine Mooney. 2016. Effective compiler error message enhancement for novice programming students. Computer Science Education 26, 2-3 (2016), 148–175. https://doi.org/10.1080/08993408.2016.1225464

[59] Mordechai Ben-Ari. 2001. Constructivism in computer science education. Journal of Computers in Mathematics and Science Teaching 20, 1 (2001), 45–73.

[60] Mordechai Ben-Ari, Roman Bednarik, Ronit Ben-Bassat Levy, Gil Ebel, Andrés Moreno, Niko Myller, and Erkki Sutinen. 2011. A decade of research and development on program animation: the Jeliot experience. Journal of Visual Languages & Computing 22 (2011), 375–384. https://doi.org/10.1016/j.jvlc.2011.04.004

[61] Jens Bennedsen and Michael E. Caspersen. 2004. Programming in context: a model-first approach to CS1. In Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '04). ACM, New York, NY, USA, 477–481. https://doi.org/10.1145/971300.971461

[62] Jens Bennedsen and Michael E. Caspersen. 2005. Revealing the programming process. In Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '05). ACM, New York, NY, USA, 186–190. https://doi.org/10.1145/1047344.1047413

[63] Jens Bennedsen and Michael E. Caspersen. 2006. Abstraction ability as an indicator of success for learning object-oriented programming? SIGCSE Bull. 38, 2 (2006), 39–43. https://doi.org/10.1145/1138403.1138430

[64] Jens Bennedsen and Michael E. Caspersen. 2007. Failure rates in introductory programming. SIGCSE Bull. 39, 2 (2007), 32–36. https://doi.org/10.1145/1272848.1272879

[65] Jens Bennedsen and Carsten Schulte. 2007. What does "objects-first" mean?: An international study of teachers' perceptions of objects-first. In Proceedings of the Seventh Baltic Sea Conference on Computing Education Research - Volume 88 (Koli Calling '07). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 21–29. http://dl.acm.org/citation.cfm?id=2449323.2449327

[66] Jens Bennedsen and Carsten Schulte. 2010. BlueJ visual debugger for learning the execution of object-oriented programs? Trans. Comput. Educ. 10, 2, Article 8 (2010), 8:1–8:22 pages. https://doi.org/10.1145/1789934.1789938

[67] Chris Bennett. 2009. Student-authored wiki textbook in CS1. J. Comput. Sci. Coll. 24, 6 (2009), 50–56. http://dl.acm.org/citation.cfm?id=1529995.1530006

[68] Marc Berges and Peter Hubwieser. 2013. Concept specification maps: displaying content structures. In Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '13). ACM, New York, NY, USA, 291–296. https://doi.org/10.1145/2462476.2462503

[69] Marc Berges, Michael Striewe, Philipp Shah, Michael Goedicke, and Peter Hubwieser. 2016. Towards deriving programming competencies from student errors. In International Conference on Learning and Teaching in Computing and Engineering (LaTiCE '16). IEEE, 19–23. https://doi.org/10.1109/LaTiCE.2016.6

[70] Anders Berglund and Raymond Lister. 2010. Introductory Programming and the Didactic Triangle. In Proceedings of the Twelfth Australasian Conference on Computing Education — Volume 103 (ACE '10). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 35–44.

[71] Matthew Berland, Taylor Martin, Tom Benton, Carmen Petrick Smith, and Don Davis. 2013. Using learning analytics to understand the learning pathways of novice programmers. Journal of the Learning Sciences 22, 4 (2013), 564–599. https://doi.org/10.1080/10508406.2013.836655

[72] Matthew Berland, Carmen Petrick Smith, and Don Davis. 2013. Visualizing live collaboration in the classroom with AMOEBA. In Computer-Supported Collaborative Learning Conference, CSCL, Vol. 2. International Society of the Learning Sciences, 2–5. https://doi.dx.org/10.22318/cscl2013.2

[73] Michael Berry and Michael Kölling. 2016. Novis: a notional machine implementation for teaching introductory programming. In International Conference on Learning and Teaching in Computing and Engineering (LaTiCE '16). IEEE, 54–59. https://doi.org/10.1109/LaTiCE.2016.5

[74] Mária Bieliková. 2006. An adaptive web-based system for learning programming. International Journal of Continuing Engineering Education and Life-Long Learning 16, 1-2 (2006), 122–136. https://doi.org/10.1504/IJCEELL.2006.008922

[75] John Biggs. 1996. Enhancing teaching through constructive alignment. Higher education 32, 3 (1996), 347–364.

[76] John B. Biggs and Kevin F. Collis. 1982. Evaluating the quality of learning: the SOLO taxonomy (structure of the observed learning outcome). Academic Press.

[77] Don Blaheta. 2009. Democracy in the classroom: an exercise for the first days of CS1. In Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '09). ACM, New York, NY, USA, 36–39. https://doi.org/10.1145/1562877.1562895

[78] Ben Blake. 2010. BLAKE a language designed for programming I. Education and Information Technologies 15, 4 (2010), 277–291. https://doi.org/10.1007/s10639-010-9139-3

[79] Douglas Blank, Jennifer S. Kay, James B. Marshall, Keith O'Hara, and Mark Russo. 2012. Calico: a multi-programming-language, multi-context framework designed for computer science education. In Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12). ACM, New York, NY, USA, 63–68. https://doi.org/10.1145/2157136.2157158

[80] Imma Boada, Josep Soler, Ferran Prados, and Jordi Poch. 2004. A teaching/learning support tool for introductory programming courses. In Information Technology Based Proceedings of the Fifth International Conference on Higher Education and Training (ITHET '04). IEEE, 604–609. https://doi.org/10.1109/ITHET.2004.1358243

[81] Michael G. Boland and Curtis Clifton. 2009. Introducing PyLighter: dynamic code highlighter. In Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE '09). ACM, New York, NY, USA, 489–493. https://doi.org/10.1145/1508865.1509037

[82] Jürgen Börstler, Marie Nordström, Lena Kallin Westin, Jan-Erik Moström, and Johan Eliasson. 2008. Transitioning to OOP/Java — a never ending story. In Reflections on the Teaching of Programming: Methods and Implementations, Jens Bennedsen, Michael E. Caspersen, and Michael Kölling (Eds.). Springer, Berlin, Heidelberg, 80–97. https://doi.org/10.1007/978-3-540-77934-6_8

[83] Jürgen Börstler, Marie Nordström, and James H. Paterson. 2011. On the quality of examples in introductory Java textbooks. Trans. Comput. Educ. 11, 1, Article 3 (2011), 21 pages. https://doi.org/10.1145/1921607.1921610

[84] Nigel Bosch, Yuxuan Chen, and Sidney D'Mello. 2014. It's written on your face: detecting affective states from facial expressions while learning computer programming. In Intelligent Tutoring Systems. Springer International Publishing, 39–44. https://doi.org/10.1007/978-3-319-07221-0_5

[85] Matt Bower. 2007. Groupwork activities in synchronous online classroom spaces. In Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '07). ACM, New York, NY, USA, 91–95. https://doi.org/10.1145/1227310.1227345

[86] Kristy Elizabeth Boyer, August A. Dwight, R. Taylor Fondren, Mladen A. Vouk, and James C. Lester. 2008. A development environment for distributed synchronous collaborative programming. SIGCSE Bull. 40, 3 (2008), 158–162. https://doi.org/10.1145/1597849.1384315

[87] Kristy Elizabeth Boyer, Rachael S. Dwight, Carolyn S. Miller, C. Dianne Raubenheimer, Matthias F. Stallmann, and Mladen A. Vouk. 2007. A case for smaller class size with integrated lab for introductory computer science. In Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '07). ACM, New York, NY, USA, 341–345. https://doi.org/10.1145/1227310.1227430

[88] Steven Bradley. 2016. Managing plagiarism in programming assignments with blended assessment and randomisation. In Proceedings of the 16th Koli Calling International Conference on Computing Education Research (Koli Calling '16). ACM, New York, NY, USA, 21–30. https://doi.org/10.1145/2999541.2999560

[89] Grant Braught, L. Martin Eby, and Tim Wahls. 2008. The effects of pair-programming on individual programming skill. In Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '08). ACM, New York, NY, USA, 200–204. https://doi.org/10.1145/1352135.1352207

[90] Grant Braught, John MacCormick, and Tim Wahls. 2010. The benefits of pairing by ability. In Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10). ACM, New York, NY, USA, 249–253. https://doi.org/10.1145/1734263.1734348

[91] Tom Briggs and C. Dudley Girard. 2007. Tools and techniques for test-driven learning in CS1. J. Comput. Sci. Coll. 22, 3 (2007), 37–43. http://dl.acm.org/citation.cfm?id=1181849.1181854

[92] Neil C.C. Brown and Amjad Altadmri. 2014. Investigating novice programming mistakes: educator beliefs vs. student data. In Proceedings of the Tenth Annual Conference on International Computing Education Research (ICER '14). ACM, New York, NY, USA, 43–50. https://doi.org/10.1145/2632320.2632343

[93] Neil Christopher Charles Brown, Michael Kölling, Davin McCall, and Ian Utting. 2014. Blackbox: a large scale repository of novice programmers' activity. In Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14). ACM, New York, NY, USA, 223–228. https://doi.org/10.1145/2538862.2538924

[94] Kim B. Bruce. 2005. Controversy on how to teach CS 1: a discussion on the SIGCSE-members mailing list. SIGCSE Bull. 37, 2 (2005), 111–117. https://doi.org/10.1145/1083431.1083477

[95] Kim B. Bruce, Andrea Danyluk, and Thomas Murtagh. 2005. Why structural recursion should be taught before arrays in CS 1. In Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '05). ACM, New York, NY, USA, 246–250. https://doi.org/10.1145/1047344.1047430

[96] Kim B. Bruce, Andrea Danyluk, and Thomas Murtagh. 2010. Introducing concurrency in CS 1. In Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10). ACM, New York, NY, USA, 224–228. https://doi.org/10.1145/1734263.1734341

[97] Michael P Bruce-Lockhart and Theodore S Norvell. 2007. Developing mental models of computer programming interactively via the web. In IEEE Frontiers in Education Conference (FIE '07). IEEE, S3H–3–S3H–8. https://doi.org/10.1109/FIE.2007.4418051

[98] Peter Brusilovsky, Olena Shcherbinina, and Sergey Sosnovsky. 2004. Mini-languages for non-computer science majors: what are the benefits? Interactive Technology and Smart Education 1, 1 (2004), 21–28. https://doi.org/10.1108/17415650480000009

[99] Engin Bumbacher, Alfredo Sandes, Amit Deutsch, and Paulo Blikstein. 2013. Student coding styles as predictors of help-seeking behavior. In Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 856–859. https://doi.org/10.1007/978-3-642-39112-5_130

[100] Carl Burch. 2009. Jigsaw, a programming environment for Java in CS1. J. Comput. Sci. Coll. 24, 5 (2009), 37–43. http://dl.acm.org/citation.cfm?id=1516595.1516604

[101] Ignacio Cabrera, Jorge Villalon, and Jorge Chavez. 2017. Blending communities and team-based learning in a programming course. IEEE Transactions on Education 60 (2017), 288–295. Issue 4. https://doi.org/10.1109/TE.2017.2698467

[102] Andrew Cain. 2013. Developing assessment criteria for portfolio assessed introductory programming. In Proceedings of IEEE International Conference on Teaching, Assessment and Learning for Engineering (TALE '13). IEEE, 55–60. https://doi.org/10.1109/TALE.2013.6654399

[103] Andrew Cain. 2014. Factors influencing student learning in portfolio assessed introductory programming. In IEEE International Conference on Teaching, Assessment and Learning for Engineering (TALE '14). IEEE, 55–62. https://doi.org/10.1109/TALE.2014.7062585

[104] Andrew Cain and Clinton J Woodward. 2012. Toward constructive alignment with portfolio assessment for introductory programming. In Proceedings of IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE '12). IEEE, H1B–11–H1B–17. https://doi.org/10.1109/TALE.2012.6360322

[105] Andrew Cain, Clinton J Woodward, and Shannon Pace. 2013. Examining student progress in portfolio assessed introductory programming. In Proceedings of 2013 IEEE International Conference on Teaching, Assessment and Learning for Engineering (TALE). IEEE, 67–72. https://doi.org/10.1109/TALE.2013.6654401

[106] Ünal Çakıroğlu. 2014. Analyzing the effect of learning styles and study habits of distance learners on learning performances: a case of an introductory programming course. International Review of Research in Open and Distance Learning 15, 4 (2014), 161–185.

[107] Jennifer Campbell, Diane Horton, and Michelle Craig. 2016. Factors for success in online CS1. In Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '16). ACM, New York, NY, USA, 320–325. https://doi.org/10.1145/2899415.2899457

[108] Jennifer Campbell, Diane Horton, Michelle Craig, and Paul Gries. 2014. Evaluating an inverted CS1. In Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14). ACM, New York, NY, USA, 307–312. https://doi.org/10.1145/2538862.2538943

[109] Yingjun Cao, Leo Porter, and Daniel Zingaro. 2016. Examining the value of analogies in introductory computing. In Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16). ACM, New York, NY, USA, 231–239. https://doi.org/10.1145/2960310.2960313

[110] Antonella Carbonaro and Mirko Ravaioli. 2017. Peer assessment to promote deep learning and to reduce a gender gap in the traditional introductory programming course. Journal of E-Learning and Knowledge Society 13, 3 (2017), 121–129. https://doi.org/10.20368/1971-8829/1398

[111] Angela Carbone, John Hurst, Ian Mitchell, and Dick Gunstone. 2009. An exploration of internal factors influencing student learning of programming. In Proceedings of the Eleventh Australasian Conference on Computing Education — Volume 95 (ACE '09). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 25–34. http://dl.acm.org/citation.cfm?id=1862712.1862721

[112] Rachel Cardell-Oliver. 2011. How can software metrics help novice programmers?. In Proceedings of the Thirteenth Australasian Computing Education Conference — Volume 114 (ACE '11). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 55–62. http://dl.acm.org/citation.cfm?id=2459936.2459943

[113] Martin C. Carlisle. 2010. Using You Tube to enhance student class preparation in an introductory Java course. In Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10). ACM, New York, NY, USA, 470–474. https://doi.org/10.1145/1734263.1734419

[114] Jeffrey Carver and Lisa Hende. 2006. Viope as a tool for teaching introductory programming: an empirical investigation. In 19th Conference on Software Engineering Education & Training (CSEET'06). IEEE, 9–16. https://doi.org/10.1109/CSEET.2006.38

[115] Jeffrey C Carver, Lisa Henderson, Lulu He, Julia Hodges, and Donna Reese. 2007. Increased retention of early computer science and software engineering students using pair programming. In 20th Conference on Software Engineering Education & Training (CSEET'07). IEEE, 115–122. https://doi.org/10.1109/CSEET.2007.29

[116] Michael E. Caspersen and Henrik Bærbak Christensen. 2008. CS1: getting started. In Reflections on the Teaching of Programming: Methods and Implementations, Jens Bennedsen, Michael E. Caspersen, and Michael Kölling (Eds.). Springer, Berlin, Heidelberg, 130–141. https://doi.org/10.1007/978-3-540-77934-6_11

[117] Francisco Enrique Vicente Castro and Kathi Fisler. 2016. On the interplay between bottom-up and datatype-driven program design. In Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16). ACM, New York, NY, USA, 205–210. https://doi.org/10.1145/2839509.2844574

[118] Francisco Enrique Vicente Castro and Kathi Fisler. 2017. Designing a multi-faceted SOLO taxonomy to track program design skills through an entire course. In Proceedings of the 17th Koli Calling International Conference on Computing Education Research (Koli Calling '17). ACM, New York, NY, USA, 10–19. https://doi.org/10.1145/3141880.3141891

[119] A. T. Chamillard. 2011. Using a student response system in CS1 and CS2. In Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE '11). ACM, New York, NY, USA, 299–304. https://doi.org/10.1145/1953163.1953253

[120] Bruce Char. 2016. Automatic feedback systems for code: what can they tell the busy instructor? J. Comput. Sci. Coll. 31, 4 (2016), 87–93. http://dl.acm.org/citation.cfm?id=2904127.2904143

[121] Spyropoulos Charalampos, Vassilios Dagdilelis, and Georgios Evangelidis. 2005. Teaching object-oriented thinking to novice programmers using the AgentSheets environment. In IADIS International Conference on Cognition and Exploratory Learning in Digital Age (CELDA '05). IADIS, 343–348.

[122] Therese Charles, David Bustard, and Michaela Black. 2011. Experiences of promoting student engagement through game-enhanced learning. In Serious Games and Edutainment Applications. Springer London, 425–445. https://doi.org/10.1007/978-1-4471-2161-9_21

[123] Tzu-Yi Chen, Gary Lewandowski, Robert McCartney, Kate Sanders, and Beth Simon. 2007. Commonsense computing: using student sorting abilities to improve instruction. In Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '07). ACM, New York, NY, USA, 276–280. https://doi.org/10.1145/1227310.1227408

[124] Tzu-Yi Chen, Alvaro Monge, and Beth Simon. 2006. Relationship of early programming language to novice generated design. SIGCSE Bull. 38, 1 (2006), 495–499. https://doi.org/10.1145/1124706.1121496

[125] Donald Chinn, Judy Sheard, Angela Carbone, and Mikko-Jussi Laakso. 2010. Study habits of CS1 students: what do they do outside the classroom?. In Proceedings of the Twelfth Australasian Conference on Computing Education — Volume 103 (ACE '10). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 53–62. http://dl.acm.org/citation.cfm?id=1862219.1862229

[126] Chih-Yueh Chou and Peng-Fei Sun. 2013. An educational tool for visualizing students' program tracing processes. Computer Applications in Engineering

Education 21, 3 (2013), 432–438. https://doi.org/10.1002/cae.20488
[127] Yu-kai Chou. 2015. Actionable gamification: beyond points, badges, and leaderboards. Octalysis Group.
[128] Rohan Roy Choudhury, Hezheng Yin, and Armando Fox. 2016. Scale-driven automatic hint generation for coding style. In Intelligent Tutoring Systems. Springer, Cham, 122–132. https://doi.org/10.1007/978-3-319-39583-8_12
[129] Vincent A. Cicirello. 2009. On the role and effectiveness of pop quizzes in CS1. In Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE '09). ACM, New York, NY, USA, 286–290. https://doi.org/10.1145/1508865.1508971
[130] Daniel C Cliburn. 2006. The effectiveness of games as assignments in an introductory programming course. In IEEE Frontiers in Education Conference (FIE '06). IEEE, 6–10. https://doi.org/10.1109/FIE.2006.322314
[131] Daniel C Cliburn. 2008. Student opinions of Alice in CS1. In IEEE Frontiers in Education Conference (FIE '08). IEEE, T3B–1–T3B–6. https://doi.org/10.1109/FIE.2008.4720254
[132] Daniel C Cliburn, Susan M Miller, and Emma Bowring. 2010. Student preferences between open-ended and structured game assignments in CS1. In IEEE Frontiers in Education Conference (FIE '10). IEEE, F2H–1–F2H–5. https://doi.org/10.1109/FIE.2010.5673668
[133] Stephen Cooper, Wanda Dann, and Randy Pausch. 2003. Teaching objects-first in introductory computer science. In Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '03). ACM, New York, NY, USA, 191–195. https://doi.org/10.1145/611892.611966
[134] Malcolm Corney, Donna Teague, and Richard N. Thomas. 2010. Engaging students in programming. In Proceedings of the Twelfth Australasian Conference on Computing Education — Volume 103 (ACE '10). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 63–72. http://dl.acm.org/citation.cfm?id=1862219.1862230
[135] Evandro B. Costa, Baldoino Fonseca, Marcelo Almeida Santana, Fabrísia Ferreira de Araújo, and Joilson Rego. 2017. Evaluating the effectiveness of educational data mining techniques for early prediction of students' academic failure in introductory programming courses. Computers in Human Behavior 73 (2017), 247–256. https://doi.org/10.1016/j.chb.2017.01.047
[136] Natalie J Coull and Ishbel MM Duncan. 2011. Emergent requirements for supporting introductory programming. Innovations in Teaching and Learning in Information and Computer Sciences 10, 1 (2011), 78–85. https://doi.org/10.11120/ital.2011.10010078
[137] Michelle Craig and Andrew Petersen. 2016. Student difficulties with pointer concepts in C. In Proceedings of the Australasian Computer Science Week Multiconference (ACSW '16). ACM, New York, NY, USA, Article 8, 8:1–8:10 pages. https://doi.org/10.1145/2843043.2843348
[138] Michelle Craig, Jacqueline Smith, and Andrew Petersen. 2017. Familiar contexts and the difficulty of programming problems. In Proceedings of the 17th Koli Calling International Conference on Computing Education Research (Koli Calling '17). ACM, New York, NY, USA, 123–127. https://doi.org/10.1145/3141880.3141898
[139] Stewart Crawford and Elizabeth Boese. 2006. ActionScript: a gentle introduction to programming. J. Comput. Sci. Coll. 21, 3 (2006), 156–168. http://dl.acm.org/citation.cfm?id=1089182.1089203
[140] James Cross, Dean Hendrix, Larry Barowski, and David Umphress. 2014. Dynamic program visualizations: an experience report. In Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14). ACM, New York, NY, USA, 609–614. https://doi.org/10.1145/2538862.2538958
[141] Gilbert Cruz, Jacob Jones, Meagan Morrow, Andres Gonzalez, and Bruce Gooch. 2017. An AI system for coaching novice programmers. In Learning and Collaboration Technologies. Technology in Education. Springer International Publishing, 12–21. https://doi.org/10.1007/978-3-319-58515-4_2
[142] Kathryn Cunningham, Sarah Blanchard, Barbara Ericson, and Mark Guzdial. 2017. Using tracing and sketching to solve programming problems: replicating and extending an analysis of what students draw. In Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17). ACM, New York, NY, USA, 164–172. https://doi.org/10.1145/3105726.3106190
[143] Quintin Cutts, Emily Cutts, Stephen Draper, Patrick O'Donnell, and Peter Saffrey. 2010. Manipulating mindset to positively influence introductory programming performance. In Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10). ACM, New York, NY, USA, 431–435. https://doi.org/10.1145/1734263.1734409
[144] Quintin I. Cutts and Gregor E. Kennedy. 2005. Connecting learning environments using electronic voting systems. In Proceedings of the 7th Australasian Conference on Computing Education — Volume 42 (ACE '05). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 181–186. http://dl.acm.org/citation.cfm?id=1082424.1082447
[145] Marin Aglić Čuvić, Josip Maras, and Saša Mladenović. 2017. Extending the object-oriented notional machine notation with inheritance, polymorphism, and GUI events. In 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO '17). IEEE, 794–799. https://doi.org/10.23919/MIPRO.2017.7973530
[146] Charlie Daly and Jane Horgan. 2005. Patterns of plagiarism. In Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '05). ACM, New York, NY, USA, 383–387. https://doi.org/10.1145/1047344.1047473
[147] Sayamindu Dasgupta and Benjamin Mako Hill. 2017. Learning to code in localized programming languages. In Proceedings of the Fourth ACM Conference on Learning @ Scale (L@S '17). ACM, New York, NY, USA, 33–39. https://doi.org/10.1145/3051457.3051464
[148] Mark Davies and Joseph L. Fleiss. 1982. Measuring agreement for multinomial data. Biometrics 38, 4 (1982), 1047–1051. http://www.jstor.org/stable/2529886
[149] Suzanne L. Dazo, Nicholas R. Stepanek, Robert Fulkerson, and Brian Dorn. 2016. An empirical analysis of video viewing behaviors in flipped CS1 courses. In Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '16). ACM, New York, NY, USA, 106–111. https://doi.org/10.1145/2899415.2899468
[150] Michael de Raadt. 2012. Student created cheat-sheets in examinations: impact on student outcomes. In Proceedings of the Fourteenth Australasian Computing Education Conference — Volume 123 (ACE '12). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 71–76. http://dl.acm.org/citation.cfm?id=2483716.2483725
[151] Michael de Raadt, David Lai, and Richard Watson. 2007. An evaluation of electronic individual peer assessment in an introductory programming course. In Proceedings of the Seventh Baltic Sea Conference on Computing Education Research — Volume 88 (Koli Calling '07). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 53–64. http://dl.acm.org/citation.cfm?id=2449323.2449330
[152] Michael de Raadt, Richard Watson, and Mark Toleman. 2003. Language tug-of-war: industry demand and academic choice. In Proceedings of the Fifth Australasian Conference on Computing Education — Volume 20 (ACE '03). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 137–142. http://dl.acm.org/citation.cfm?id=858403.858420
[153] Michael de Raadt, Richard Watson, and Mark Toleman. 2004. Introductory programming: what's happening today and will there be any students to teach tomorrow?. In Proceedings of the Sixth Australasian Conference on Computing Education — Volume 30 (ACE '04). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 277–282. http://dl.acm.org/citation.cfm?id=979968.980005
[154] 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 (ACE '09). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 45–54. http://dl.acm.org/citation.cfm?id=1862712.1862723
[155] Adrienne Decker and Elizabeth Lane Lawley. 2013. Life's a game and the game of life: how making a game out of it can change student behavior. In Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13). ACM, New York, NY, USA, 233–238. https://doi.org/10.1145/2445196.2445269
[156] Adrienne Decker, Phil Ventura, and Christopher Egert. 2006. Through the looking glass: reflections on using undergraduate teaching assistants in CS1. In Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '06). ACM, New York, NY, USA, 46–50. https://doi.org/10.1145/1121341.1121358
[157] Paul Denny, Diana Cukierman, and Jonathan Bhaskar. 2015. Measuring the effect of inventing practice exercises on learning in an introductory programming course. In Proceedings of the 15th Koli Calling Conference on Computing Education Research (Koli Calling '15). ACM, New York, NY, USA, 13–22. https://doi.org/10.1145/2828959.2828967
[158] Paul Denny, Andrew Luxton-Reilly, and Dave Carpenter. 2014. Enhancing syntax error messages appears ineffectual. In Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education (ITiCSE '14). ACM, New York, NY, USA, 273–278. https://doi.org/10.1145/2591708.2591748
[159] Paul Denny, Andrew Luxton-Reilly, John Hamer, Dana B. Dahlstrom, and Helen C. Purchase. 2010. Self-predicted and actual performance in an introductory programming course. In Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '10). ACM, New York, NY, USA, 118–122. https://doi.org/10.1145/1822090.1822124
[160] Paul Denny, Andrew Luxton-Reilly, and Beth Simon. 2008. Evaluating a new exam question: Parsons problems. In Proceedings of the Fourth International Workshop on Computing Education Research (ICER '08). ACM, New York, NY, USA, 113–124. https://doi.org/10.1145/1404520.1404532
[161] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. 2011. Codewrite: supporting student-driven practice of Java. In Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE '11). ACM, New York, NY, USA, 471–476. https://doi.org/10.1145/1953163.1953299
[162] Sebastian Deterding, Dan Dixon, Rilla Khaled, and Lennart Nacke. 2011. From game design elements to gamefulness: defining "gamification". In Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments (MindTrek '11). ACM, New York, NY, USA, 9–15. https:

//doi.org/10.1145/2181037.2181040

[163] Adrian Devey and Angela Carbone. 2011. Helping first year novice programming students pass. In Proceedings of the Thirteenth Australasian Computing Education Conference — Volume 114 (ACE '11). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 135–144. http://dl.acm.org/citation.cfm?id=2459936.2459953

[164] Nicholas Diana, Michael Eagle, John Stamper, Shuchi Grover, Marie Bienkowski, and Satabdi Basu. 2017. An instructor dashboard for real-time analytics in interactive programming assignments. In Proceedings of the Seventh International Learning Analytics & Knowledge Conference (LAK '17). ACM, New York, NY, USA, 272–279. https://doi.org/10.1145/3027385.3027441

[165] Paul E. Dickson. 2011. Motivating students taking CS1 by using image manipulation in C and C+. J. Comput. Sci. Coll. 26, 6 (2011), 136–141. http://dl.acm.org/citation.cfm?id=1968521.1968551

[166] Edward Dillon, Monica Anderson, and Marcus Brown. 2012. Comparing feature assistance between programming environments and their "effect" on novice programmers. J. Comput. Sci. Coll. 27, 5 (2012), 69–77. http://dl.acm.org/citation.cfm?id=2168874.2168894

[167] Edward Dillon, Monica Anderson, and Marcus Brown. 2012. Comparing mental models of novice programmers when using visual and command line environments. In Proceedings of the 50th Annual Southeast Regional Conference (ACM-SE '12). ACM, New York, NY, USA, 142–147. https://doi.org/10.1145/2184512.2184546

[168] Daghan Dinç and Suzan Üsküdarli. 2009. A web environment to support teaching introductory programming. In 2009 Fourth International Conference on Internet and Web Applications and Services. IEEE, 578–582. https://doi.org/10.1109/ICIW.2009.93

[169] Zachary Dodds, Ran Libeskind-Hadas, and Eliot Bush. 2010. When CS 1 is Biology 1: crossdisciplinary collaboration as CS context. In Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '10). ACM, New York, NY, USA, 219–223. https://doi.org/10.1145/1822090.1822152

[170] Liam Doherty, J Shakya, M Jordanov, P Lougheed, D Brokenshire, S Rao, and VS Kumar. 2005. Recognizing opportunities for mixed-initiative interactions in novice programming. In AAAI Fall Symposium on Mixed-Initiative Problem-Solving Assistants. AAAI, 51–56.

[171] Brian Dorn and Allison Elliott Tew. 2015. Empirical validation and application of the computing attitudes survey. Computer Science Education 25, 1 (2015), 1–36. https://doi.org/10.1080/08993408.2015.1014142

[172] Brian Dorn and Dean Sanders. 2003. Using Jeroo to introduce object-oriented programming. In IEEE Frontiers in Education Conference (FIE '03), Vol. 1. IEEE, T4C–22–7 Vol.1. https://doi.org/10.1109/FIE.2003.1263372

[173] Ricardo Vargas Dorneles, Delcino Picinin Jr, and André Gustavo Adami. 2010. ALGOWEB: a web-based environment for learning introductory programming. In 10th IEEE International Conference on Advanced Learning Technologies (ICALT '10). IEEE, 83–85. https://doi.org/10.1109/ICALT.2010.30

[174] Mohsen Dorodchi, Nasrin Dehbozorgi, and Tonya K Frevert. 2017. "I wish I could rank my exam's challenge level!": An algorithm of Bloom's taxonomy in teaching CS1. In IEEE Frontiers in Education Conference (FIE '17). IEEE, 1–5. https://doi.org/10.1109/FIE.2017.8190523

[175] Daryl D'Souza, Margaret Hamilton, James Harland, Peter Muir, Charles Thevathayan, and Cecily Walker. 2008. Transforming learning of programming: a mentoring project. In Proceedings of the Tenth Conference on Australasian Computing Education — Volume 78 (ACE '08). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 75–84. http://dl.acm.org/citation.cfm?id=1379249.1379256

[176] Amalia Duch Brown, Joaquim Gabarró Vallès, Jordi Petit Silvestre, Maria Josep Blesa Aguilera, and María José Serna Iglesias. 2015. A cost-benefit analysis of continuous assessment. In Proceedings of the 7th International Conference on Computer Supported Education (CSEDU '15). SciTePress, 57–66.

[177] Mark S. Durrheim, Abejide Ade-Ibijola, and Sigrid Ewert. 2016. Code pathfinder: a stepwise programming e-tutor using plan mirroring. In Communications in Computer and Information Science. Springer International Publishing, 69–82. https://doi.org/10.1007/978-3-319-47680-3_7

[178] Thomas Dy and Ma. Mercedes Rodrigo. 2010. A detector for non-literal Java errors. In Proceedings of the 10th Koli Calling International Conference on Computing Education Research (Koli Calling '10). ACM, New York, NY, USA, 118–122. https://doi.org/10.1145/1930464.1930485

[179] Michael Eagle and Tiffany Barnes. 2009. Experimental evaluation of an educational game for improved learning in introductory computing. SIGCSE Bull. 41, 1 (2009), 321–325. https://doi.org/10.1145/1539024.1508980

[180] Alice H Eagly and Shelly Chaiken. 1998. Attitude structure and function. In The Handbook of Social Psychology. McGraw-Hill.

[181] Anna Eckerdal, Mikko-Jussi Laakso, Mike Lopez, and Amitrajit Sarkar. 2011. Relationship between text and action conceptions of programming: a phenomenographic and quantitative perspective. In Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education (ITiCSE '11). ACM, New York, NY, USA, 33–37. https://doi.org/10.1145/1999747.1999760

[182] Alex Edgcomb and Frank Vahid. 2014. Effectiveness of online textbooks vs. interactive web-native content. In Proceedings of the 2014 ASEE Annual Conference. ASEE.

[183] Alex Edgcomb, Frank Vahid, Roman Lysecky, and Susan Lysecky. 2017. Getting students to earnestly do reading, studying, and homework in an introductory programming class. In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17). ACM, New York, NY, USA, 171–176. https://doi.org/10.1145/3017680.3017732

[184] Alex Daniel Edgcomb, Frank Vahid, Roman Lysecky, and Susan Lysecky. 2017. An analysis of incorporating small coding exercises as homework in introductory programming courses. In Proceedings of the 2017 ASEE Annual Conference. ASEE.

[185] Stephen H. Edwards. 2004. Using software testing to move students from trial-and-error to reflection-in-action. In Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '04). ACM, New York, NY, USA, 26–30. https://doi.org/10.1145/971300.971312

[186] Stephen H. Edwards and Krishnan Panamalai Murali. 2017. CodeWorkout: short programming exercises with built-in data collection. In Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17). ACM, New York, NY, USA, 188–193. https://doi.org/10.1145/3059009.3059055

[187] Stephen H. Edwards and Manuel A. Pérez-Quiñones. 2007. Experiences using test-driven development with an automated grader. J. Comput. Sci. Coll. 22, 3 (2007), 44–50. http://dl.acm.org/citation.cfm?id=1181849.1181855

[188] Stephen H. Edwards, Daniel S. Tilden, and Anthony Allevato. 2014. Pythy: improving the introductory Python programming experience. In Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14). ACM, New York, NY, USA, 641–646. https://doi.org/10.1145/2538862.2538977

[189] Johan Eliasson, Lena Kallin Westin, and Marie Nordström. 2006. Investigating students' confidence in programming and problem solving. In IEEE Frontiers in Education Conference (FIE '06). IEEE, 22–27. https://doi.org/10.1109/FIE.2006.322490

[190] Allison Elliott Tew and Brian Dorn. 2013. The case for validated tools in computer science education research. Computer 46, 9 (2013), 60–66.

[191] Joelle Elmaleh and Venky Shankararaman. 2017. Improving student learning in an introductory programming course using flipped classroom and competency framework. In IEEE Global Engineering Education Conference (EDUCON '17). IEEE, 49–55. https://doi.org/10.1109/EDUCON.2017.7942823

[192] Ashraf Elnagar and Mahir Ali. 2012. A modified team-based learning methodology for effective delivery of an introductory programming course. In Proceedings of the 13th Annual Conference on Information Technology Education (SIGITE '12). ACM, New York, NY, USA, 177–182. https://doi.org/10.1145/2380552.2380604

[193] Richard J. Enbody, William F. Punch, and Mark McCullen. 2009. Python CS1 as preparation for C++ CS2. In Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE '09). ACM, New York, NY, USA, 116–120. https://doi.org/10.1145/1508865.1508907

[194] Kiran L. N. Eranki and Kannan M. Moudgalya. 2012. A collaborative approach to scaffold programming efficiency using spoken tutorials and its evaluation. In 8th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom '12). IEEE, 556–559. https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6450951

[195] Barbara J. Ericson, Kantwon Rogers, Miranda Parker, Briana Morrison, and Mark Guzdial. 2016. Identifying design principles for CS teacher ebooks through design-based research. In Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16). ACM, New York, NY, USA, 191–200. https://doi.org/10.1145/2960310.2960335

[196] Anthony Estey, Hieke Keuning, and Yvonne Coady. 2017. Automatically classifying students in need of support by detecting changes in programming behaviour. In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17). ACM, New York, NY, USA, 189–194. https://doi.org/10.1145/3017680.3017790

[197] Jim Etheredge. 2004. CMeRun: program logic debugging courseware for CS1/CS2 students. In Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '04). ACM, New York, NY, USA, 22–25. https://doi.org/10.1145/971300.971311

[198] Thomas R Etherington. 2016. Teaching introductory GIS programming to geographers using an open source Python approach. Journal of Geography in Higher Education 40, 1 (2016), 117–130. https://doi.org/10.1080/03098265.2015.1086981

[199] Geela Venise Firmalo Fabic, Antonija Mitrovic, and Kourosh Neshatian. 2017. Investigating the effectiveness of menu-based self-explanation prompts in a mobile Python tutor. In Artificial Intelligence in Education, Elisabeth André, Ryan Baker, Xiangen Hu, Ma. Mercedes T. Rodrigo, and Benedict du Boulay (Eds.). Springer International Publishing, Cham, 498–501. https://doi.org/10.1007/978-3-319-61425-0_49

[200] Lisa Facey-Shaw and Paul Golding. 2005. Effects of peer tutoring and attitude on academic performance of first year introductory programming students. In IEEE Frontiers in Education Conference (FIE '05). IEEE, S1E. https://doi.org/10.1109/FIE.2005.1612175

[201] Katrina Falkner and David S Munro. 2009. Easing the transition: a collaborative learning approach. In Proceedings of the Eleventh Australasian Conference on Computing Education — Volume 95 (ACE '09). Australian Computer Society, Inc., Darlinghurst, Australia, 65–74. http://dl.acm.org/citation.cfm?id=1862712.1862725

[202] Waleed Farag, Sanwar Ali, and Debzani Deb. 2013. Does language choice influence the effectiveness of online introductory programming courses?. In Proceedings of the 14th Annual ACM SIGITE Conference on Information Technology Education (SIGITE '13). ACM, New York, NY, USA, 165–170. https://doi.org/10.1145/2512276.2512293

[203] Rob Faux. 2006. Impact of preprogramming course curriculum on learning in the first programming course. IEEE Transactions on Education 49 (2006), 11–15. Issue 1. https://doi.org/10.1109/TE.2005.852593

[204] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. How to Design Programs: An Introduction to Programming and Computing. MIT Press.

[205] James B Fenwick Jr, Cindy Norris, Frank E Barry, Josh Rountree, Cole J Spicer, and Scott D Cheek. 2009. Another look at the behaviors of novice programmers. In Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE '09). ACM, New York, NY, USA, 296–300. https://doi.org/10.1145/1508865.1508973

[206] José Luis Fernández Alemán and Youssef Oufaska. 2010. SAMtool, a tool for deducing and implementing loop patterns. In Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '10). ACM, New York, NY, USA, 68–72. https://doi.org/10.1145/1822090.1822111

[207] Kasper Fisker, Davin McCall, Michael Kölling, and Bruce Quig. 2008. Group work support for the BlueJ IDE. In Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '08). ACM, New York, NY, USA, 163–168. https://doi.org/10.1145/1384271.1384316

[208] Fred Fonseca and Larry Spence. 2014. The karate kid method of problem based learning. In Innovative Practices in Teaching Information Sciences and Technology. Springer International Publishing, Cham., 9–17. https://doi.org/10.1007/978-3-319-03656-4_2

[209] Davide Fossati, Barbara Di Eugenio, Christopher Brown, and Stellan Ohlsson. 2008. Learning linked lists: experiments with the iList system. In Intelligent Tutoring Systems. Springer, Berlin, Heidelberg, 80–89. https://doi.org/10.1007/978-3-540-69132-7_13

[210] Patrice Frison. 2015. A teaching assistant for algorithm construction. In Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '15). ACM, New York, NY, USA, 9–14. https://doi.org/10.1145/2729094.2742588

[211] Daniel Frost. 2008. Ucigame, a Java library for games. In Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '08). ACM, New York, NY, USA, 310–314. https://doi.org/10.1145/1352135.1352243

[212] Luka Fürst and Viljan Mahnič. 2012. A cooperative development system for an interactive introductory programming course. World Transactions on Engineering and Technology Education 10, 2 (2012), 122–127.

[213] Judith Gal-Ezer, Tamar Vilner, and Ela Zur. 2009. Has the paradigm shift in CS1 a harmful effect on data structures courses: a case study. In Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE '09). ACM, New York, NY, USA, 126–130. https://doi.org/10.1145/1508865.1508909

[214] Jianxiong Gao, Bei Pang, and Steven S Lumetta. 2016. Automated feedback framework for introductory programming courses. In Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '16). ACM, New York, NY, USA, 53–58. https://doi.org/10.1145/2899415.2899440

[215] Ryan Garlick and Ebru Celikel Cankaya. 2010. Using Alice in CS1: a quantitative experiment. In Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '10). ACM, New York, NY, USA, 165–168. https://doi.org/10.1145/1822090.1822138

[216] Stuart Garner. 2007. A program design tool to help novices learn programming. In Australian Society for Computers in Learning in Tertiary Education Annual Conference (ASCILITE '07). Australasian Society for Computers in Learning in Tertiary Education, 321–324.

[217] David Ginat and Eti Menashe. 2015. SOLO taxonomy for assessing novices' algorithmic design. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15). ACM, New York, NY, USA, 452–457. https://doi.org/10.1145/2676723.2677311

[218] David Ginat and Ronit Shmalo. 2013. Constructive use of errors in teaching CS1. In Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13). ACM, New York, NY, USA, 353–358. https://doi.org/10.1145/2445196.2445300

[219] Mark Goadrich. 2014. Incorporating tangible computing devices into CS1. J. Comput. Sci. Coll. 29, 5 (2014), 23–31. http://dl.acm.org/citation.cfm?id=2600623.2600627

[220] Paul Golding, Lisa Facey-Shaw, and Vanesa Tennant. 2006. Effects of peer tutoring, attitude and personality on academic performance of first year introductory programming students. In IEEE Frontiers in Education Conference (FIE '06). IEEE, 7–12. https://doi.org/10.1109/FIE.2006.322662

[221] Michael H Goldwasser and David Letscher. 2007. Introducing network programming into a CS1 course. In Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '07). ACM, New York, NY, USA, 19–22. https://doi.org/10.1145/1268784.1268793

[222] Michael H Goldwasser and David Letscher. 2009. A graphics package for the first day and beyond. In Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE '09). ACM, New York, NY, USA, 206–210. https://doi.org/10.1145/1508865.1508945

[223] Anabela Jesus Gomes, Alvaro Nuno Santos, and António José Mendes. 2012. A study on students' behaviours and attitudes towards learning to program. In Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '12). ACM, New York, NY, USA, 132–137. https://doi.org/10.1145/2325296.2325331

[224] Gracielo Gonzalez. 2004. Constructivism in an introduction to programming course. J. Comput. Sci. Coll. 19 (2004), 299–305. http://dl.acm.org/citation.cfm?id=1050231.1050277

[225] Graciela Gonzalez. 2006. A systematic approach to active and cooperative learning in CS1 and its effects on CS2. In Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '06). ACM, New York, NY, USA, 133–137. https://doi.org/10.1145/1121341.1121386

[226] Morten Goodwin and Tom Drange. 2016. Teaching programming to large student groups through test driven development comparing established methods with teaching based on test driven development. In Proceedings of the 8th International Conference on Computer Supported Education (CSEDU '16), Vol. 1. SciTePress, 281–288.

[227] Ira Greenberg, Deepak Kumar, and Dianna Xu. 2012. Creative coding and visual portfolios for CS1. In Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12). ACM, New York, NY, USA, 247–252. https://doi.org/10.1145/2157136.2157214

[228] Paul Gries, Volodymyr Mnih, Jonathan Taylor, Greg Wilson, and Lee Zamparo. 2005. Memview: a pedagogically-motivated visual debugger. In IEEE Frontiers in Education Conference (FIE '05). IEEE, S1J–11. https://doi.org/10.1109/FIE.2005.1612204

[229] Paul Gross and Kris Powers. 2005. Evaluating assessments of novice programming environments. In Proceedings of the First International Workshop on Computing Education Research (ICER '05). ACM, New York, NY, USA, 99–110. https://doi.org/10.1145/1089786.1089796

[230] Sebastian Gross and Niels Pinkwart. 2015. Towards an integrative learning environment for Java programming. In IEEE 15th International Conference on Advanced Learning Technologies (ICALT '15). IEEE, 24–28. https://doi.org/10.1109/ICALT.2015.75

[231] Dee Gudmundsen, Lisa Olivieri, and Namita Sarawagi. 2011. Using Visual Logic ©: three different approaches in different courses — General Education, CS0, and CS1. J. Comput. Sci. Coll. 26, 6 (2011), 23–29. http://dl.acm.org/citation.cfm?id=1968521.1968529

[232] Minzhe Guo, Taolun Chai, and Kai Qian. 2010. Design of online runtime and testing environment for instant Java programming assessment. In 2010 Seventh International Conference on Information Technology: New Generations. IEEE, 1102–1106. https://doi.org/10.1109/ITNG.2010.227

[233] Mark Guzdial. 2003. A media computation course for non-majors. In Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '03). ACM, New York, NY, USA, 104–108. https://doi.org/10.1145/961511.961542

[234] Simo Haatainen, Antti-Jussi Lakanen, Ville Isomöttönen, and Vesa Lappalainen. 2013. A practice for providing additional support in CS1. In Learning and Teaching in Computing and Engineering (LaTiCE '13). IEEE, 178–183. https://doi.org/10.1109/LaTiCE.2013.39

[235] Patricia Haden, Dale Parsons, Krissi Wood, and Joy Gasson. 2017. Student affect in CS1: insights from an easy data collection tool. In Proceedings of the 17th Koli Calling International Conference on Computing Education Research (Koli Calling '17). ACM, New York, NY, USA, 40–49. https://doi.org/10.1145/3141880.3141881

[236] Pari Delir Haghighi and Judy Sheard. 2005. Summative computer programming assessment using both paper and computer. In Towards Sustainable and Scalable Educational Innovations Informed by the Learning Sciences (ICCE '05). IOS Press, 67–75.

[237] John Hamer, Quintin Cutts, Jana Jackova, Andrew Luxton-Reilly, Robert McCartney, Helen Purchase, Charles Riedesel, Mara Saeli, Kate Sanders, and Judithe Sheard. 2008. Contributing student pedagogy. SIGCSE Bull. 40, 4 (2008), 194–212. https://doi.org/10.1145/1473195.1473242

[238] John Hamer, Helen C Purchase, Paul Denny, and Andrew Luxton-Reilly. [n. d.]. Quality of Peer Assessment in CS1. 27–36. https://doi.org/10.1145/1584322.

1584327
[239] Nadeem Abdul Hamid. 2012. Automated web-based user interfaces for novice programmers. In Proceedings of the 50th Annual Southeast Regional Conference (ACM-SE '12). ACM, New York, NY, USA, 42–47. https://doi.org/10.1145/2184512.2184523
[240] Nadeem Abdul Hamid. 2016. A generic framework for engaging online data sources in introductory programming courses. In Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '16). ACM, New York, NY, USA, 136–141. https://doi.org/10.1145/2899415.2899437
[241] Brian Hanks. 2005. Student performance in CS1 with distributed pair programming. In Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '05). ACM, New York, NY, USA, 316–320. https://doi.org/10.1145/1067445.1067532
[242] Brian Hanks. 2006. Student attitudes toward pair programming. In Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITICSE '06). ACM, New York, NY, USA, 113–117. https://doi.org/10.1145/1140124.1140156
[243] Brian Hanks. 2008. Problems encountered by novice pair programmers. J. Educ. Resour. Comput. 7, 4, Article 2 (2008), 2:1–2:13 pages. https://doi.org/10.1145/1316450.1316452
[244] Brian Hanks, Charlie McDowell, David Draper, and Milovan Krnjajic. 2004. Program quality with pair programming in CS1. In Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '04). ACM, New York, NY, USA, 176–180. https://doi.org/10.1145/1007996.1008043
[245] Brian Hanks, Laurie Murphy, Beth Simon, Renée McCauley, and Carol Zander. 2009. CS1 students speak: advice for students by students. In Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE '09). ACM, New York, NY, USA, 19–23. https://doi.org/10.1145/1508865.1508875
[246] Michael R Hansen and Jens Thyge Kristensen. 2008. Experiences with functional programming in an introductory curriculum. In Reflections on the Teaching of Programming, Jens Bennedsen, Michael E. Caspersen, and Michael Kölling (Eds.). Springer, Berlin, Heidelberg, 30–46. https://doi.org/10.1007/978-3-540-77934-6_4
[247] Brian K Hare. 2013. Classroom interventions to reduce failure & withdrawal in CS1: a field report. J. Comput. Sci. Coll. 28, 5 (2013), 228–235. http://dl.acm.org/citation.cfm?id=2458569.2458618
[248] Douglas Harms. 2011. Personal robots in CS1: implementing the Myro API in Java. In Proceedings of the 12th International Conference on Computer Systems and Technologies (CompSysTech '11). ACM, New York, NY, USA, 552–557. https://doi.org/10.1145/2023607.2023699
[249] Brian Harrington and Ayaan Chaudhry. 2017. TrAcademic: improving participation and engagement in CS1/CS2 with gamified practicals. In Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17). ACM, New York, NY, USA, 347–352. https://doi.org/10.1145/3059009.3059052
[250] Nathan Harris and Charmain Cilliers. 2006. A program beacon recognition tool. In 2006 7th International Conference on Information Technology Based Higher Education and Training (ITHET '06). IEEE, 216–225. https://doi.org/10.1109/ITHET.2006.339767
[251] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R Klemmer. 2010. What would other programmers do: suggesting solutions to error messages. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10). ACM, New York, NY, USA, 1019–1028. https://doi.org/10.1145/1753326.1753478
[252] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. 2017. Writing reusable code feedback at scale with mixed-initiative program synthesis. In Proceedings of the Fourth ACM Conference on Learning @ Scale (L@S '17). ACM, New York, NY, USA, 89–98. https://doi.org/10.1145/3051457.3051467
[253] David Heaney and Charlie Daly. 2004. Mass production of individual feedback. In Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '04). ACM, New York, NY, USA, 117–121. https://doi.org/10.1145/1007996.1008029
[254] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. 2003. Helium, for learning Haskell. In Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell (Haskell '03). ACM, New York, NY, USA, 62–71. https://doi.org/10.1145/871895.871902
[255] Kenny Heinonen, Kasper Hirvikoski, Matti Luukkainen, and Arto Vihavainen. 2014. Using CodeBrowser to seek differences between novice programmers. In Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14). ACM, New York, NY, USA, 229–234. https://doi.org/10.1145/2538862.2538981
[256] James Heliotis and Richard Zanibbi. 2011. Moving away from programming and towards computer science in the CS first year. J. Comput. Sci. Coll. 26, 3 (2011), 115–125. http://dl.acm.org/citation.cfm?id=1859159.1859183
[257] Arto Hellas, Juho Leinonen, and Petri Ihantola. 2017. Plagiarism in take-home exams: help-seeking, collaboration, and systematic cheating. In Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17). ACM, New York, NY, USA, 238–243. https://doi.org/10.1145/3059009.3059065
[258] Joseph Henrich, Steven J Heine, and Ara Norenzayan. 2010. The weirdest people in the world? Behavioral and Brain Sciences 33, 2-3 (2010), 61–83.
[259] Pavel Herout and Premysl Brada. 2015. Duck Testing Enhancements for Automated Validation of Student Programmes. In Proceedings of the 7th International Conference on Computer Supported Education – Volume 1 (CSEDU '15). SciTePress, Portugal, 228–234. https://doi.org/10.5220/0005412902280234
[260] Matthew Hertz and Sarah Michele Ford. 2013. Investigating factors of student learning in introductory courses. In Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13). ACM, New York, NY, USA, 195–200. https://doi.org/10.1145/2445196.2445254
[261] Michael Hilton and David S Janzen. 2012. On teaching arrays with test-driven learning in WebIDE. In Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '12). ACM, New York, NY, USA, 93–98. https://doi.org/10.1145/2325296.2325322
[262] Amanda M Holland-Minkley and Thomas Lombardi. 2016. Improving engagement in introductory courses with homework resubmission. In Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16). ACM, New York, NY, USA, 534–539. https://doi.org/10.1145/2839509.2844576
[263] Mark A Holliday and David Luginbuhl. 2004. CS1 assessment using memory diagrams. In Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '04). ACM, New York, NY, USA, 200–204. https://doi.org/10.1145/971300.971373
[264] Lars Josef Höök and Anna Eckerdal. 2015. On the bimodality in an introductory programming course: an analysis of student performance factors. In International Conference on Learning and Teaching in Computing and Engineering (LaTiCE '15). IEEE, 79–86. https://doi.org/10.1109/LaTiCE.2015.25
[265] Dainal Hooshyar, Rodina Binti Ahmad, Mohd Hairul Nizam Md Nasir, and Wong Ching Mun. 2014. Flowchart-based approach to aid novice programmers: a novel framework. In 2014 International Conference on Computer and Information Sciences (ICCOINS). IEEE, 1–5. https://doi.org/10.1109/ICCOINS.2014.6868826
[266] Danial Hooshyar, Rodina Binti Ahmad, Mohd Hairul Nizam Md Nasir, Shahaboddin Shamshirband, and Shi-Jinn Horng. 2015. Flowchart-based programming environments for improving comprehension and problem-solving skill of novice programmers: a survey. International Journal of Advanced Intelligence Paradigms 7, 1 (2015), 24–56.
[267] Danial Hooshyar, Rodina Binti Ahmad, Ram Gopal Raj, Mohd Hairul Nizam Md Nasir, Moslem Yousef, Shi-Jinn Horng, and Jože Rugelj. 2015. A flowchart-based multi-agent system for assisting novice programmers with problem solving activities. Malaysian Journal of Computer Science 28, 2 (2015), 132–151.
[268] Danial Hooshyar, Rodina Binti Ahmad, Moslem Yousefi, Moein Fathi, Shi-Jinn Horng, and Heuiseok Lim. 2018. SITS: a solution-based intelligent tutoring system for students' acquisition of problem-solving skills in computer programming. Innovations in Education and Teaching International 55, 3 (2018), 325–335. https://doi.org/10.1080/14703297.2016.1189346
[269] Danial Hooshyar, Rodina Binti Ahmad, Moslem Yousefi, FD Yusop, and S-J Horng. 2015. A flowchart-based intelligent tutoring system for improving problem-solving skills of novice programmers. Journal of Computer Assisted Learning 31, 4 (2015), 345–361. https://doi.org/10.1111/jcal.12099
[270] V Horner and P Gouws. 2016. E-tutoring support for undergraduate students learning computer programming at the university of South Africa. In Proceedings of the Computer Science Education Research Conference 2016 (CSERC '16). ACM, New York, NY, USA, 29–36. https://doi.org/10.1145/2998551.2998557
[271] Diane Horton, Jennifer Campbell, and Michelle Craig. 2016. Online CS1: who enrols, why, and how do they do?. In Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16). ACM, New York, NY, USA, 323–328. https://doi.org/10.1145/2839509.2844578
[272] David Hovemeyer and David Babcock. 2009. Using terminal window graphics in CS1. J. Comput. Sci. Coll. 24, 3 (2009), 151–158. http://dl.acm.org/citation.cfm?id=1409873.1409902
[273] David Hovemeyer, Arto Hellas, Andrew Petersen, and Jaime Spacco. 2016. Control-flow-only abstract syntax trees for analyzing students' programming progress. In Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16). ACM, New York, NY, USA, 63–72. https://doi.org/10.1145/2960310.2960326
[274] Wen Chin Hsu and Scott W Plunkett. 2016. Attendance and grades in learning programming classes. In Proceedings of the Australasian Computer Science Week Multiconference (ACSW '16). ACM, New York, NY, USA, Article 4, 4:1–4:6 pages. https://doi.org/10.1145/2843043.2843061
[275] Minjie Hu, Michael Winikoff, and Stephen Cranefield. 2012. Teaching novice programming using goals and plans in a visual notation. In Proceedings of

the Fourteenth Australasian Computing Education Conference — Volume 123 (ACE '12). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 43–52. http://dl.acm.org/citation.cfm?id=2483716.2483722

[276] Minjie Hu, Michael Winikoff, and Stephen Cranefield. 2013. A process for novice programming using goals and plans. In Proceedings of the Fifteenth Australasian Computing Education Conference — Volume 136 (ACE '13). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 3–12. http://dl.acm.org/citation.cfm?id=2667199.2667200

[277] Yun-Jen Hu and Po-Yao Chao. 2015. A simulation-based learning environment for learning debugging. In Proceedings of the 23rd International Conference on Computers in Education (ICCE '15). 310–312.

[278] Chenn-Jung Huang, Chen Chun-Hua, Luo Yun-Cheng, Chen Hong-Xin, and Yi-Ta Chuang. 2008. Developing an intelligent diagnosis and assessment e-learning tool for introductory programming. Journal of Educational Technology & Society 11, 4 (2008), 139–157.

[279] JL Huff and HR Clements. 2017. The hidden person within the frustrated student: an interpretative phenomenological analysis of a student's experience in a programming course. In American Society for Engineering Education Annual Conference & Exposition. ASEE.

[280] Bowen Hui and Shannon Farvolden. 2017. How can learning analytics improve a course?. In Proceedings of the 22nd Western Canadian Conference on Computing Education (WCCCE '17). ACM, New York, NY, USA, Article 1, 6 pages. https://doi.org/10.1145/3085585.3085586

[281] Christopher Hundhausen, Anukrati Agrawal, Dana Fairbrother, and Michael Trevisan. 2009. Integrating pedagogical code reviews into a CS 1 course: an empirical study. In Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE '09). ACM, New York, NY, USA, 291–295. https://doi.org/10.1145/1508865.1508972

[282] Christopher Hundhausen, Anukrati Agrawal, Dana Fairbrother, and Michael Trevisan. 2010. Does studio-based instruction work in CS 1?: an empirical comparison with a traditional approach. In Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10). ACM, New York, NY, USA, 500–504. https://doi.org/10.1145/1734263.1734432

[283] Christopher Hundhausen, Anukrati Agrawal, and Kyle Ryan. 2010. The design of an online environment to support pedagogical code reviews. In Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10). ACM, New York, NY, USA, 182–186. https://doi.org/10.1145/1734263.1734324

[284] Christopher D Hundhausen, Pawan Agarwal, and Michael Trevisan. 2011. Online vs. face-to-face pedagogical code reviews: an empirical comparison. In Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE '11). ACM, New York, NY, USA, 117–122. https://doi.org/10.1145/1953163.1953201

[285] Christopher D Hundhausen and Jonathan Lee Brown. 2007. An experimental study of the impact of visual semantic feedback on novice programming. Journal of Visual Languages & Computing 18, 6 (2007), 537–559. https://doi.org/10.1016/j.jvlc.2006.09.001

[286] Christopher D Hundhausen and Jonathan L Brown. 2007. What you see is what you code: a "live" algorithm development and visualization environment for novice learners. Journal of Visual Languages & Computing 18, 1 (2007), 22–47. https://doi.org/10.1016/j.jvlc.2006.03.002

[287] Christopher D Hundhausen and Jonathan L Brown. 2008. Designing, visualizing, and discussing algorithms within a CS 1 studio experience: an empirical study. Computers & Education 50, 1 (2008), 301–326. https://doi.org/10.1016/j.compedu.2006.06.002

[288] Christopher D Hundhausen, Sarah A Douglas, and John T Stasko. 2002. A meta-study of algorithm visualization effectiveness. Journal of Visual Languages & Computing 13, 3 (2002), 259–290. https://doi.org/10.1006/jvlc.2002.0237

[289] Christopher D Hundhausen, Sean Farley, and Jonathan Lee Brown. 2006. Can direct manipulation lower the barriers to programming and promote positive transfer to textual programming? An experimental study. In IEEE Symposium on Visual Languages and Human-Centric Computing, 2006 (VL/HCC'06). IEEE, 157–164. https://doi.org/10.1109/VLHCC.2006.12

[290] Christopher D Hundhausen, N Hari Narayanan, and Martha E Crosby. 2008. Exploring studio-based instructional models for computing education. In Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '08). ACM, New York, NY, USA, 392–396. https://doi.org/10.1145/1352135.1352271

[291] Jacqueline Hundley. 2008. A review of using design patterns in CS1. In Proceedings of the 46th Annual Southeast Regional Conference (ACM-SE 46). ACM, New York, NY, USA, 30–33. https://doi.org/10.1145/1593105.1593113

[292] Jacqueline Hundley and Winard Britt. 2009. Engaging students in software development course projects. In The Fifth Richard Tapia Celebration of Diversity in Computing Conference: Intellect, Initiatives, Insight, and Innovations (TAPIA '09). ACM, New York, NY, USA, 87–92. https://doi.org/10.1145/1565799.1565820

[293] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of Recent Systems for Automatic Assessment of Programming Assignments. In Proceedings of the 10th Koli Calling International Conference on Computing Education Research (Koli Calling '10). ACM, New York, NY, USA, 86–93. https://doi.org/10.1145/1930464.1930480

[294] Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H Edwards, Essi Isohanni, Ari Korhonen, Andrew Petersen, Kelly Rivers, Miguel Ángel Rubio, Judy Sheard, Bronius Skupas, Jaime Spacco, Claudia Szabo, and Daniel Toll. 2015. Educational data mining and learning analytics in programming: literature review and case studies. In Proceedings of the 2015 ITiCSE on Working Group Reports (ITICSE-WGR '15). ACM, New York, NY, USA, 41–63. https://doi.org/10.1145/2858796.2858798

[295] Ville Isomöttönen and Vesa Lappalainen. 2012. CSI with games and an emphasis on TDD and unit testing: piling a trend upon a trend. ACM Inroads 3, 3 (2012), 62–68. https://doi.org/10.1145/2339055.2339073

[296] Mirjana Ivanović, Zoran Budimac, Miloš Radovanović, and Miloš Savić. 2015. Does the choice of the first programming language influence students' grades?. In Proceedings of the 16th International Conference on Computer Systems and Technologies (CompSysTech '15). ACM, New York, NY, USA, 305–312. https://doi.org/10.1145/2812428.2812448

[297] Janusz Jablonowski. 2004. Some remarks on teaching of programming. In Proceedings of the 5th International Conference on Computer Systems and Technologies (CompSysTech '04). ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/1050330.1050419

[298] James Jackson, Michael Cobb, and Curtis Carver. 2005. Identifying top Java errors for novice programmers. In IEEE Frontiers in Education Conference (FIE '05). IEEE, T4C–T4C. https://doi.org/10.1109/FIE.2005.1611967

[299] Matthew C Jadud. 2006. Methods and tools for exploring novice compilation behaviour. In Proceedings of the Second International Workshop on Computing Education Research (ICER '06). ACM, New York, NY, USA, 73–84. https://doi.org/10.1145/1151588.1151600

[300] Matthew C Jadud and Poul Henriksen. 2009. Flexible, reusable tools for studying novice programmers. In Proceedings of the Fifth International Workshop on Computing Education Research Workshop (ICER '09). ACM, New York, NY, USA, 37–42. https://doi.org/10.1145/1584322.1584328

[301] Siti Robaya Jantan and Syed Ahmad Aljunid. 2012. An experimental evaluation of scaffolded educational games design for programming. In IEEE Conference on Open Systems (ICOS '12). IEEE, 1–6. https://doi.org/10.1109/ICOS.2012.6417631

[302] Ambikesh Jayal, Stasha Lauria, Allan Tucker, and Stephen Swift. 2011. Python for teaching introductory programming: a quantitative evaluation. ITALICS Innovations in Teaching and Learning in Information and Computer Sciences 10, 1 (2011), 86–90. https://doi.org/10.11120/ital.2011.10010086

[303] Mehdi Jazayeri. 2015. Combining mastery learning with project-based learning in a first programming course: an experience report. In Proceedings of the 37th International Conference on Software Engineering — Volume 2 (ICSE '15). IEEE Press, Piscataway, NJ, USA, 315–318. http://dl.acm.org/citation.cfm?id=2819009.2819059

[304] Jam Jenkins, Evelyn Brannock, Thomas Cooper, Sonal Dekhane, Mark Hall, and Michael Nguyen. 2012. Perspectives on active learning and collaboration: javaw-ide in the classroom. In Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12). ACM, New York, NY, USA, 185–190. https://doi.org/10.1145/2157136.2157194

[305] Åse Jevinger and Kristina Von Hausswolff. 2016. Large programming task vs questions-and-answers examination in Java introductory courses. In International Conference on Learning and Teaching in Computing and Engineering (LaTiCE '16). IEEE, 154–161. https://doi.org/10.1109/LaTiCE.2016.25

[306] Wei Jin and Albert Corbett. 2011. Effectiveness of cognitive apprenticeship learning (CAL) and cognitive tutors (CT) for problem solving using fundamental programming concepts. In Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE '11). ACM, New York, NY, USA, 305–310. https://doi.org/10.1145/1953163.1953254

[307] Wei Jin, Albert Corbett, Will Lloyd, Lewis Baumstark, and Christine Rolka. 2014. Evaluation of guided-planning and assisted-coding with task relevant dynamic hinting. In International Conference on Intelligent Tutoring Systems. Springer International Publishing, Cham., 318–328. https://doi.org/10.1007/978-3-319-07221-0_40

[308] Chris Johnson. 2012. SpecCheck: automated generation of tests for interface conformance. In Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '12). ACM, New York, NY, USA, 186–191. https://doi.org/10.1145/2325296.2325343

[309] Chris Johnson, Monica McGill, Durell Bouchard, Michael K Bradshaw, Víctor A Bucheli, Laurence D Merkle, Michael James Scott, Z Sweedyk, J Ángel Velázquez-Iturbide, Zhiping Xiao, and Ming Zhang. 2016. Game development for computer science education. In Proceedings of the 2016 ITiCSE Working Group Reports (ITiCSE '16). ACM, New York, NY, USA, 23–44. https://doi.org/10.1145/3024906.3024908

[310] Mary Elizabeth "ME" Jones, Melanie Kisthardt, and Marie A Cooper. 2011. Interdisciplinary teaching: introductory programming via creative writing. In Proceedings of the 42nd ACM Technical Symposium on Computer Science

Education (SIGCSE '11). ACM, New York, NY, USA, 523–528. https://doi.org/10.1145/1953163.1953313

[311] Francisco Jurado, Ana I Molina, Miguel A Redondo, Manuel Ortega, Adam Giemza, Lars Bollen, and Heinz Ulrich Hoppe. 2009. Learning to program with COALA, a distributed computer assisted environment. Journal of Universal Computer Science 15, 7 (2009), 1472–1485.

[312] Erkki Kaila, Teemu Rajala, Mikko-Jussi Laakso, and Tapio Salakoski. 2009. Effects, experiences and feedback from studies of a program visualization tool. Informatics in Education 8, 1 (2009), 17–34.

[313] Geetha Kanaparan, Rowena Cullen, and David Mason. 2013. Self-efficacy and engagement as predictors of student programming performance. In PACIS 2013 Proceedings, Vol. 282. AISeL, 12. http://aisel.aisnet.org/pacis2013/282

[314] Pertti Kansanen. 1999. Teaching as teaching-studying-learning interaction. Scandinavian Journal of Educational Research 43, 1 (1999), 81–89.

[315] Oscar Karnalim. 2016. Detecting source code plagiarism on introductory programming course assignments using a bytecode approach. In International Conference on Information & Communication Technology and Systems (ICTS '16). IEEE, 63–68. https://doi.org/10.1109/ICTS.2016.7910274

[316] Petra Kastl, Oliver Krisch, and Ralf Romeike. 2017. 3D printing as medium for motivation and creativity in computer science lessons. In International Conference on Informatics in Schools: Situation, Evolution, and Perspectives. Springer International Publishing, 27–36. https://doi.org/10.1007/978-3-319-71483-7_3

[317] Nadia Kasto, Jacqueline Whalley, Anne Philpott, and David Whalley. 2014. Solution spaces. In Proceedings of the Sixteenth Australasian Computing Education Conference — Volume 148 (ACE '14). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 133–137. http://dl.acm.org/citation.cfm?id=2667490.2667506

[318] Aaron Keen and Kurt Mammen. 2015. Program decomposition and complexity in CS1. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15). ACM, New York, NY, USA, 48–53. https://doi.org/10.1145/2676723.2677219

[319] Hansi Keijonen, Jaakko Kurhila, and Arto Vihavainen. 2013. Carry-on effect in extreme apprenticeship. In IEEE Frontiers in Education Conference (FIE '13). IEEE, 1150–1155. https://doi.org/10.1109/FIE.2013.6685011

[320] Caitlin Kelleher and Randy Pausch. 2005. Lowering the barriers to programming: a taxonomy of programming environments and languages for novice programmers. ACM Comput. Surv. 37, 2 (2005), 83–137. https://doi.org/10.1145/1089733.1089734

[321] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2016. Towards a systematic review of automated feedback generation for programming exercises. In Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '16). ACM, New York, NY, USA, 41–46. https://doi.org/10.1145/2899415.2899422

[322] Nazish Zaman Khan and Andrew Luxton-Reilly. 2016. Is computing for social good the solution to closing the gender gap in computer science?. In Proceedings of the Australasian Computer Science Week Multiconference (ACSW '16). ACM, New York, NY, USA, Article 17, 17:1–17:5 pages. https://doi.org/10.1145/2843043.2843069

[323] Hassan Khosravi and Kendra M.L. Cooper. 2017. Using learning analytics to investigate patterns of performance and engagement in large classes. In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17). ACM, New York, NY, USA, 309–314. https://doi.org/10.1145/3017680.3017711

[324] Suin Kim, Jae Won Kim, Jungkook Park, and Alice Oh. 2016. Elice: an online CS education platform to understand how students learn programming. In Proceedings of the Third ACM Conference on Learning @ Scale (L@S '16). ACM, New York, NY, USA, 225–228. https://doi.org/10.1145/2876034.2893420

[325] Päivi Kinnunen and Lauri Malmi. 2006. Why students drop out CS1 course?. In Proceedings of the Second International Workshop on Computing Education Research (ICER '06). ACM, New York, NY, USA, 97–108. https://doi.org/10.1145/1151588.1151604

[326] Paivi Kinnunen and Beth Simon. 2010. Experiencing programming assignments in CS1: the emotional toll. In Proceedings of the Sixth International Workshop on Computing Education Research (ICER '10). ACM, New York, NY, USA, 77–86. https://doi.org/10.1145/1839594.1839609

[327] Päivi Kinnunen and Beth Simon. 2011. CS majors' self-efficacy perceptions in CS1: results in light of social cognitive theory. In Proceedings of the Seventh International Workshop on Computing Education Research (ICER '11). ACM, New York, NY, USA, 19–26. https://doi.org/10.1145/2016911.2016917

[328] Päivi Kinnunen and Beth Simon. 2012. My program is ok—am I? Computing freshmen's experiences of doing programming assignments. Computer Science Education 22, 1 (2012), 1–28. https://doi.org/10.1080/08993408.2012.655091

[329] Päivi Kinnunen and Beth Simon. 2012. Phenomenography and grounded theory as research methods in computing education research field. Computer Science Education 22, 2 (2012), 199–218. https://doi.org/10.1080/08993408.2012.692928

[330] Stephen Kirby, Benjamin Toland, and Catherine Deegan. 2010. Program visualisation tool for teaching programming in C. In Proceedings of the International Conference on Education, Training and Informatics (ICETI'10). 457–461.

[331] Michael S. Kirkpatrick and Chris Mayfield. 2017. Evaluating an alternative CS1 for students with prior programming experience. In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17). ACM, New York, NY, USA, 333–338. https://doi.org/10.1145/3017680.3017759

[332] Paul A Kirschner. 2017. Stop propagating the learning styles myth. Computers & Education 106 (2017), 166–171.

[333] Gabor Kiss and Zuzanna Arki. 2017. The influence of game-based programming education on the algorithmic thinking. Procedia—Social and Behavioral Sciences 237 (2017), 613–617. https://doi.org/10.1016/j.sbspro.2017.02.020

[334] Yoshihiro Kita, Tetsuro Katayama, and Shigeyuki Tomita. 2007. Implementation and evaluation of an automatic visualization tool "PGT" for programming education. In 5th ACIS International Conference on Software Engineering Research, Management & Applications (SERA 2007). IEEE, 213–220. https://doi.org/10.1109/SERA.2007.92

[335] Barbara Kitchenham. 2004. Procedures for performing systematic reviews. Keele, UK, Keele University 33, 2004 (2004), 1–26.

[336] Antti Knutas, Jouni Ikonen, Uolevi Nikula, and Jari Porras. 2014. Increasing collaborative communications in a programming course with gamification: a case study. In Proceedings of the 15th International Conference on Computer Systems and Technologies (CompSysTech '14). ACM, New York, NY, USA, 370–377. https://doi.org/10.1145/2659532.2659620

[337] A. J. Ko. 2003. Preserving non-programmers' motivation with error-prevention and debugging support tools. In IEEE Symposium on Human Centric Computing Languages and Environments, 2003. Proceedings. 2003. IEEE, 271–272. https://doi.org/10.1109/HCC.2003.1260245

[338] Andrew J. Ko. 2009. Attitudes and self-efficacy in young adults' computing autobiographies. In IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '09). IEEE, 67–74. https://doi.org/10.1109/VLHCC.2009.5295297

[339] Andrew J. Ko and Brad A. Myers. 2008. Debugging reinvented: asking and answering why and why not questions about program behavior. In Proceedings of the 30th International Conference on Software Engineering (ICSE '08). ACM, New York, NY, USA, 301–310. https://doi.org/10.1145/1368088.1368130

[340] Yousun Ko, Bernd Burgstaller, and Bernhard Scholz. 2013. Parallel from the beginning: the case for multicore programming in the computer science undergraduate curriculum. In Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13). ACM, New York, NY, USA, 415–420. https://doi.org/10.1145/2445196.2445320

[341] Kimberle Koile and David Singer. 2006. Improving learning in CS1 via tablet-pc-based in-class assessment. In Proceedings of the Second International Workshop on Computing Education Research (ICER '06). ACM, New York, NY, USA, 119–126. https://doi.org/10.1145/1151588.1151607

[342] Michael Kölling. 2008. Using BlueJ to introduce programming. In Reflections on the Teaching of Programming, Jens Bennedsen, Michael E. Caspersen, and Michael Kölling (Eds.). Springer, Berlin, Heidelberg, 98–115. https://doi.org/10.1007/978-3-540-77934-6_9

[343] Steven Kollmansberger. 2010. Helping students build a mental model of computation. In Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '10). ACM, New York, NY, USA, 128–131. https://doi.org/10.1145/1822090.1822127

[344] Mario Konecki, Sandra Lovrenčić, and Matija Kaniški. 2016. Using real projects as motivators in programming education. In 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO '16). IEEE, 883–886. https://doi.org/10.1109/MIPRO.2016.7522264

[345] Yana Kortsarts and Yulia Kempner. 2012. Enriching introductory programming courses with non-intuitive probability experiments component. In Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '12). ACM, New York, NY, USA, 128–131. https://doi.org/10.1145/2325296.2325330

[346] Aditi Kothiyal, Rwitajit Majumdar, Sahana Murthy, and Sridhar Iyer. 2013. Effect of think-pair-share in a large CS1 class: 83% sustained engagement. In Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research (ICER '13). ACM, New York, NY, USA, 137–144. https://doi.org/10.1145/2493394.2493408

[347] Aditi Kothiyal, Sahana Murthy, and Sridhar Iyer. 2014. Think-pair-share in a large CS1 class: does learning really happen?. In Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education (ITiCSE '14). ACM, New York, NY, USA, 51–56. https://doi.org/10.1145/2591708.2591739

[348] Theodora Koulouri, Stanislao Lauria, and Robert D. Macredie. 2014. Teaching introductory programming: a quantitative evaluation of different approaches. Trans. Comput. Educ. 14, 4, Article 26 (2014), 26:1–26:28 pages. https://doi.org/10.1145/2662412

[349] Svetlana Kouznetsova. 2007. Using bluej and blackjack to teach object-oriented design concepts in CS1. J. Comput. Sci. Coll. 22, 4 (2007), 49–55. http://dl.acm.org/citation.cfm?id=1229637.1229646

[350] Feng-Yang Kuo, Wen-Hsiung Wu, and Cathy S Lin. 2013. An investigation of self-regulatory mechanisms in learning to program Visual Basic. Journal of Educational Computing Research 49, 2 (2013), 225–247. https://doi.org/10.2190/EC.49.2.f

[351] Jaakko Kurhila and Arto Vihavainen. 2011. Management, structures and tools to scale up personal advising in large programming courses. In Proceedings of the 2011 Conference on Information Technology Education (SIGITE '11). ACM, New York, NY, USA, 3–8. https://doi.org/10.1145/2047594.2047596

[352] Lisa L. Lacher and Mark C. Lewis. 2015. The effectiveness of video quizzes in a flipped class. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15). ACM, New York, NY, USA, 224–228. https://doi.org/10.1145/2676723.2677302

[353] Essi Lahtinen and Tuukka Ahoniemi. 2007. Annotations for defining interactive instructions to interpreter based program visualization tools. Electronic Notes in Theoretical Computer Science 178 (2007), 121–128. https://doi.org/10.1016/j.entcs.2007.01.041

[354] Essi Lahtinen and Tuukka Ahoniemi. 2009. Kick-start activation to novice programming — a visualization-based approach. Electronic Notes in Theoretical Computer Science 224 (2009), 125–132. https://doi.org/10.1016/j.entcs.2008.12.056

[355] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. 2005. A study of the difficulties of novice programmers. In Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '05). ACM, New York, NY, USA, 14–18. https://doi.org/10.1145/1067445.1067453

[356] Chien-Hung Lai, Wei-Ching Lin, Bin-Shyan Jong, and Yen-Teh Hsia. 2013. Java assist learning system for assisted learning on facebook. In Learning and Teaching in Computing and Engineering (LaTiCE '13). IEEE, 77–82. https://doi.org/10.1109/LaTiCE.2013.10

[357] Antti-Jussi Lakanen, Vesa Lappalainen, and Ville Isomöttönen. 2015. Revisiting rainfall to explore exam questions and performance on CS1. In Proceedings of the 15th Koli Calling Conference on Computing Education Research (Koli Calling '15). ACM, New York, NY, USA, 40–49. https://doi.org/10.1145/2828959.2828970

[358] Maria S. W. Lam, Eric Y. K. Chan, Victor C. S. Lee, and Y. T. Yu. 2008. Designing an automatic debugging assistant for improving the learning of computer programming. In International Conference on Hybrid Learning and Education (ICHL '08). Springer, Berlin, Heidelberg, 359–370. https://doi.org/10.1007/978-3-540-85170-7_32

[359] H Chad Lane and Kurt VanLehn. 2004. A dialogue-based tutoring system for beginning programming. In Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference (FLAIRS '04), Vol. 2. AAAI, 449–454. http://www.aaai.org/Papers/FLAIRS/2004/Flairs04-078.pdf

[360] Vesa Lappalainen, Antti-Jussi Lakanen, and Harri Högmander. 2017. Towards computer-based exams in CS1. In Proceedings of the 9th International Conference on Computer Supported Education (CSEDU '17), Vol. 2. SciTePress, 125–136.

[361] Patricia Lasserre and Carolyn Szostak. 2011. Effects of team-based learning on a CS1 course. In Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education (ITiCSE '11). ACM, New York, NY, USA, 133–137. https://doi.org/10.1145/1999747.1999787

[362] Matti Lattu, Veijo Meisalo, and Jorma Tarhio. 2003. A visualisation tool as a demonstration aid. Computers & Education 41, 2 (2003), 133–148. https://doi.org/10.1016/S0360-1315(03)00032-0

[363] Celine Latulipe, N. Bruce Long, and Carlos E. Seminario. 2015. Structuring flipped classes with lightweight teams and gamification. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15). ACM, New York, NY, USA, 392–397. https://doi.org/10.1145/2676723.2677240

[364] Tom Lauwers, Illah Nourbakhsh, and Emily Hamner. 2009. CSbots: design and deployment of a robot designed for the CS1 classroom. In Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE '09). ACM, New York, NY, USA, 428–432. https://doi.org/10.1145/1508865.1509017

[365] Kris M.Y. Law, Victor C.S. Lee, and Y.T. Yu. 2010. Learning motivation in e-learning facilitated computer programming courses. Computers & Education 55, 1 (2010), 218–228. https://doi.org/10.1016/j.compedu.2010.01.007

[366] Michael J Lee, Faezeh Bahmani, Irwin Kwan, Jilian LaFerte, Polina Charters, Amber Horvath, Fanny Luor, Jill Cao, Catherine Law, Michael Beswetherick, et al. 2014. Principles of a debugging-first puzzle game for computing education. In IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2014). IEEE, 57–64. https://doi.org/10.1109/VLHCC.2014.6883023

[367] Juho Leinonen, Leo Leppänen, Petri Ihantola, and Arto Hellas. 2017. Comparison of time metrics in programming. In Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17). ACM, New York, NY, USA, 200–208. https://doi.org/10.1145/3105726.3106181

[368] Noel LeJeune. 2010. Contract grading with mastery learning in CS 1. J. Comput. Sci. Coll. 26, 2 (2010), 149–156. http://dl.acm.org/citation.cfm?id=1858583.1858604

[369] Ronald Leppan, Charmain Cilliers, and Marinda Taljaard. 2007. Supporting CS1 with a program beacon recognition tool. In Proceedings of the 2007 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries (SAICSIT '07). ACM, New York, NY, USA, 66–75. https://doi.org/10.1145/1292491.1292499

[370] Leo Leppänen, Juho Leinonen, Petri Ihantola, and Arto Hellas. 2017. Predicting academic success based on learning material usage. In Proceedings of the 18th Annual Conference on Information Technology Education (SIGITE '17). ACM, New York, NY, USA, 13–18. https://doi.org/10.1145/3125659.3125695

[371] Chuck Leska and John Rabung. 2005. Refactoring the CS1 course. J. Comput. Sci. Coll. 20, 3 (2005), 6–18. http://dl.acm.org/citation.cfm?id=1040196.1040199

[372] Colleen M. Lewis, Ken Yasuhara, and Ruth E. Anderson. 2011. Deciding to major in computer science: a grounded theory of students' self-assessment of ability. In Proceedings of the Seventh International Workshop on Computing Education Research (ICER '11). ACM, New York, NY, USA, 3–10. https://doi.org/10.1145/2016911.2016915

[373] Hai-Ning Liang, Charles Fleming, Ka Lok Man, and Tammam Tillo. 2013. A first introduction to programming for first-year students at a Chinese university using LEGO MindStorms. In Proceedings of IEEE International Conference on Teaching, Assessment and Learning for Engineering (TALE '13). IEEE, 233–238. https://doi.org/10.1109/TALE.2013.6654435

[374] Soohyun Nam Liao, Daniel Zingaro, Michael A. Laurenzano, William G. Griswold, and Leo Porter. 2016. Lightweight, early identification of at-risk CS1 students. In Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16). ACM, New York, NY, USA, 123–131. https://doi.org/10.1145/2960310.2960315

[375] Derrell Lipman. 2014. LearnCS!: a new, browser-based C programming environment for CS1. J. Comput. Sci. Coll. 29, 6 (2014), 144–150. http://dl.acm.org/citation.cfm?id=2602724.2602752

[376] Alex Lishinski, Aman Yadav, and Richard Enbody. 2017. Students' emotional reactions to programming projects in introduction to programming: measurement approach and influence on learning outcomes. In Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17). ACM, New York, NY, USA, 30–38. https://doi.org/10.1145/3105726.3106187

[377] Alex Lishinski, Aman Yadav, Richard Enbody, and Jon Good. 2016. The influence of problem solving abilities on students' performance on different assessment tasks in CS1. In Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16). ACM, New York, NY, USA, 329–334. https://doi.org/10.1145/2839509.2844596

[378] Alex Lishinski, Aman Yadav, Jon Good, and Richard Enbody. 2016. Learning to program: gender differences and interactive effects of students' motivation, goals, and self-efficacy on performance. In Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16). ACM, New York, NY, USA, 211–220. https://doi.org/10.1145/2960310.2960329

[379] Raymond Lister. 2004. Teaching Java first: experiments with a pigs-early pedagogy. In Proceedings of the Sixth Australasian Conference on Computing Education — Volume 30 (ACE '04). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 177–183. http://dl.acm.org/citation.cfm?id=979968.979992

[380] Raymond Lister. 2005. One small step toward a culture of peer review and multi-institutional sharing of educational resources: a multiple choice exam for first semester programming students. In Proceedings of the 7th Australasian Conference on Computing Education — Volume 42 (ACE '05). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 155–164. http://dl.acm.org/citation.cfm?id=1082424.1082444

[381] Raymond Lister. 2011. Concrete and other neo-piagetian forms of reasoning in the novice programmer. In Proceedings of the Thirteenth Australasian Computing Education Conference — Volume 114 (ACE '11). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 9–18. http://dl.acm.org/citation.cfm?id=2459936.2459938

[382] Raymond Lister, Elizabeth S Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, et al. 2004. A multi-national study of reading and tracing skills in novice programmers. In Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR '04). ACM, New York, NY, USA, 119–150. https://doi.org/10.1145/1044550.1041673

[383] Raymond Lister, Colin Fidge, and Donna Teague. 2009. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. In Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '09). ACM, New York, NY, USA, 161–165. https://doi.org/10.1145/1562877.1562930

[384] Raymond Lister and John Leaney. 2003. Introductory programming, criterion-referencing, and Bloom. In Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '03). ACM, New York, NY, USA, 143–147. https://doi.org/10.1145/611892.611954

[385] Stanislav Litvinov, Marat Mingazov, Vladislav Myachikov, Vladimir Ivanov, Yuliya Palamarchuk, Pavel Sozonov, and Giancarlo Succi. 2017. A tool for visualizing the execution of programs and stack traces especially suited for novice programmers. In Proceedings of the 12th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2017). Springer International Publishing, 235–240.

[386] Matija Lokar and Matija Pretnar. 2015. A low overhead automated service for teaching programming. In Proceedings of the 15th Koli Calling Conference on Computing Education Research (Koli Calling '15). ACM, New York, NY, USA, 132–136. https://doi.org/10.1145/2828959.2828964

[387] Dastyni Loksa and Andrew J. Ko. 2016. The role of self-regulation in programming problem solving process and success. In Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16). ACM, New York, NY, USA, 83–91. https://doi.org/10.1145/2960310.2960334

[388] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between reading, tracing and writing skills in introductory programming. In Proceedings of the Fourth International Workshop on Computing Education Research (ICER '08). ACM, New York, NY, USA, 101–112. https://doi.org/10.1145/1404520.1404531

[389] Ellie Lovellette, John Matta, Dennis Bouvier, and Roger Frye. 2017. Just the numbers: an investigation of contextualization of problems for novice programmers. In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17). ACM, New York, NY, USA, 393–398. https://doi.org/10.1145/3017680.3017726

[390] Andrew K. Lui, Yannie H. Y. Cheung, and Siu Cheung Li. 2008. Leveraging students' programming laboratory work as worked examples. SIGCSE Bull. 40, 2 (2008), 69–73. https://doi.org/10.1145/1383602.1383638

[391] Evelyn Lulis and Reva Freedman. 2011. Validating an instructor rating scale for the difficulty of CS1 test items in C++. J. Comput. Sci. Coll. 27, 2 (2011), 85–91. http://dl.acm.org/citation.cfm?id=2038836.2038847

[392] Harri Luoma, Essi Lahtinen, and Hannu-Matti Järvinen. 2007. CLIP, a command line interpreter for a subset of C++. In Proceedings of the Seventh Baltic Sea Conference on Computing Education Research — Volume 88 (Koli Calling '07). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 199–202. http://dl.acm.org/citation.cfm?id=2449323.2449351

[393] Andrew Luxton-Reilly. 2016. Learning to program is easy. In Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '16). ACM, New York, NY, USA, 284–289. https://doi.org/10.1145/2899415.2899432

[394] Andrew Luxton-Reilly, Brett A. Becker, Yingjun Cao, Roger McDermott, Claudio Mirolo, Andreas Mühling, Andrew Petersen, Kate Sanders, Simon, and Jacqueline Whalley. 2017. Developing assessments to determine mastery of programming fundamentals. In Proceedings of the 2017 ITiCSE Conference on Working Group Reports (ITiCSE-WGR '17). ACM, New York, NY, USA, 47–69. https://doi.org/10.1145/3174781.3174784

[395] Andrew Luxton-Reilly and Paul Denny. 2009. A simple framework for interactive games in CS1. In Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE '09). ACM, New York, NY, USA, 216–220. https://doi.org/10.1145/1508865.1508947

[396] Andrew Luxton-Reilly and Andrew Petersen. 2017. The compound nature of novice programming assessments. In Proceedings of the Nineteenth Australasian Computing Education Conference (ACE '17). ACM, New York, NY, USA, 26–35. https://doi.org/10.1145/3013499.3013500

[397] Marianne Lykke, Mayela Coto, Sonia Mora, Niels Vandel, and Christian Jantzen. 2014. Motivating programming students by problem based learning and LEGO robots. In IEEE Global Engineering Education Conference (EDUCON '14). IEEE, 544–555. https://doi.org/10.1109/EDUCON.2014.6826146

[398] Linxiao Ma, John Ferguson, Marc Roper, Isla Ross, and Murray Wood. 2009. Improving the mental models held by novice programmers using cognitive conflict and Jeliot visualisations. In Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '09). ACM, New York, NY, USA, 166–170. https://doi.org/10.1145/1562877.1562931

[399] Linxiao Ma, John Ferguson, Marc Roper, and Murray Wood. 2007. Investigating the viability of mental models held by novice programmers. In Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '07). ACM, New York, NY, USA, 499–503. https://doi.org/10.1145/1227310.1227481

[400] Linxiao Ma, John Ferguson, Marc Roper, and Murray Wood. 2011. Investigating and improving the models of programming concepts held by novice programmers. Computer Science Education 21, 1 (2011), 57–80. https://doi.org/10.1080/08993408.2011.554722

[401] Sandra Madison and James Gifford. 2002. Modular programming: novice misconceptions. Journal of Research on Technology in Education 34, 3 (2002), 217–229. https://doi.org/10.1080/15391523.2002.10782346

[402] Qusay H. Mahmoud, Wlodek Dobosiewicz, and David Swayne. 2004. Redesigning introductory computer programming with HTML, JavaScript, and Java. In Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '04). ACM, New York, NY, USA, 120–124. https://doi.org/10.1145/971300.971344

[403] Qusay H Mahmoud and Pawel Popowicz. 2010. A mobile application development approach to teaching introductory programming. In IEEE Frontiers in Education Conference (FIE '10). IEEE, T4F–1–T4F–6. https://doi.org/10.1109/FIE.2010.5673608

[404] Mirna Carelli Oliveira Maia, Dalton Serey, and Jorge Figueiredo. 2017. Learning styles in programming education: a systematic mapping study. In IEEE Frontiers in Education Conference (FIE '17). IEEE, 1–7. https://doi.org/10.1109/FIE.2017.8190465

[405] L. Major, T. Kyriacou, and O. P. Brereton. 2012. Systematic literature review: teaching novices programming using robots. In IET Software, Vol. 6. IET, 502–513. Issue 6. https://doi.org/10.1049/iet-sen.2011.0125

[406] David J. Malan. 2010. Reinventing CS50. In Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10). ACM, New York, NY, USA, 152–156. https://doi.org/10.1145/1734263.1734316

[407] Mercy Maleko, Margaret Hamilton, and Daryl D'Souza. 2012. Novices' perceptions and experiences of a mobile social learning environment for learning of programming. In Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '12). ACM, New York, NY, USA, 285–290. https://doi.org/10.1145/2325296.2325364

[408] Mercy Maleko, Dip Nandi, Margaret Hamilton, Daryl D'Souza, and James Harland. 2013. Facebook versus Blackboard for supporting the learning of programming in a fully online course: the changing face of computing education. In Learning and Teaching in Computing and Engineering (LaTiCE '13). IEEE, 83–89. https://doi.org/10.1109/LaTiCE.2013.31

[409] Lauri Malmi, Judy Sheard, Simon, Roman Bednarik, Juha Helminen, Päivi Kinnunen, Ari Korhonen, Niko Myller, Juha Sorva, and Ahmad Taherkhani. 2014. Theoretical underpinnings of computing education research: what is the evidence?. In Proceedings of the Tenth Annual Conference on International Computing Education Research (ICER '14). ACM, New York, NY, USA, 27–34. https://doi.org/10.1145/2632320.2632358

[410] Linda Mannila and Michael de Raadt. 2006. An objective comparison of languages for teaching introductory programming. In Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006 (Koli Calling '06). ACM, New York, NY, USA, 32–37. https://doi.org/10.1145/1315803.1315811

[411] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Measuring the effectiveness of error messages designed for novice programmers. In Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE '11). ACM, New York, NY, USA, 499–504. https://doi.org/10.1145/1953163.1953308

[412] Jakeline Marcos-Abed. 2014. Using a COAC# for CS1. In Proceedings of the Western Canadian Conference on Computing Education (WCCCE '14). ACM, New York, NY, USA, Article 10, 10:1–10:3 pages. https://doi.org/10.1145/2597959.2597971

[413] Stefanie A. Markham. 2009. Expanding security awareness in introductory computer science courses. In Information Security Curriculum Development Conference (InfoSecCD '09). ACM, New York, NY, USA, 27–31. https://doi.org/10.1145/1940976.1940984

[414] Stefanie A. Markham and K. N. King. 2010. Using personal robots in CS1: experiences, outcomes, and attitudinal influences. In Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '10). ACM, New York, NY, USA, 204–208. https://doi.org/10.1145/1822090.1822148

[415] Tanya Markow, Eugene Ressler, and Jean Blair. 2006. Catch that speeding turtle: latching onto fun graphics in CS1. Ada Lett. XXVI, 3 (2006), 29–34. https://doi.org/10.1145/1185875.1185648

[416] Will Marrero and Amber Settle. 2005. Testing first: emphasizing testing in early programming courses. In Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '05). ACM, New York, NY, USA, 4–8. https://doi.org/10.1145/1067445.1067451

[417] Chris Martin, Janet Hughes, and John Richards. 2017. Learning experiences in programming: the motivating effect of a physical interface. In Proceedings of the 9th International Conference on Computer Supported Education, Vol. 1. SCITEPRESS, 162–172.

[418] José Alfredo Martínez-Valdés, J. Ángel Velázquez-Iturbide, and Raquel Hijón-Neira. 2017. A (relatively) unsatisfactory experience of use of Scratch in CS1. In Proceedings of the 5th International Conference on Technological Ecosystems for Enhancing Multiculturality (TEEM '17). ACM, New York, NY, USA, Article 8, 8:1–8:7 pages. https://doi.org/10.1145/3144826.3145356

[419] Raina Mason and Simon. 2017. Introductory programming courses in Australasia in 2016. In Proceedings of the Nineteenth Australasian Computing Education Conference (ACE '17). ACM, New York, NY, USA, 81–89. https://doi.org/10.1145/3013499.3013512

[420] Ásrún Matthíasdóttir and Hallgrímur Arnalds. 2016. E-assessment: students' point of view. In Proceedings of the 17th International Conference on Computer

Systems and Technologies (CompSysTech '16). ACM, New York, NY, USA, 369–374. https://doi.org/10.1145/2983468.2983497

[421] Sarah Matzko and Timothy A. Davis. 2006. Teaching CS1 with graphics and C. In Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITICSE '06). ACM, New York, NY, USA, 168–172. https://doi.org/10.1145/1140124.1140170

[422] Bruce A. Maxwell and Stephanie R. Taylor. 2017. Comparing outcomes across different contexts in CS1. In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17). ACM, New York, NY, USA, 399–403. https://doi.org/10.1145/3017680.3017757

[423] Brendan McCane, Claudia Ott, Nick Meek, and Anthony Robins. 2017. Mastery learning in introductory programming. In Proceedings of the Nineteenth Australasian Computing Education Conference (ACE '17). ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/3013499.3013501

[424] Renée McCauley, Christopher Starr, Walter Pharr, RoxAnn Stalvey, and George Pothering. 2006. Is CS1 better with the same lecture and lab instructor? SIGCSE Bull. 38, 2 (2006), 54–60. https://doi.org/10.1145/1138403.1138433

[425] Ian McChesney. 2016. Three years of student pair programming: action research insights and outcomes. In Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16). ACM, New York, NY, USA, 84–89. https://doi.org/10.1145/2839509.2844565

[426] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In ITiCSE Working Group Reports (ITiCSE-WGR 2001). ACM, New York, NY, USA, 125–180. https://doi.org/10.1145/572139.572181

[427] Charlie McDowell, Linda Werner, Heather E. Bullock, and Julian Fernald. 2003. The impact of pair programming on student performance, perception and persistence. In Proceedings of the 25th International Conference on Software Engineering (ICSE '03). IEEE Computer Society, Washington, DC, USA, 602–607. http://dl.acm.org/citation.cfm?id=776816.776899

[428] Aidan McGowan, Philip Hanna, and Neil Anderson. 2016. Teaching programming: understanding lecture capture YouTube analytics. In Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '16). ACM, New York, NY, USA, 35–40. https://doi.org/10.1145/2899415.2899421

[429] Fraser McKay and Michael Kölling. 2013. Predictive modelling for HCI problems in novice program editors. In Proceedings of the 27th International BCS Human Computer Interaction Conference (BCS-HCI '13). British Computer Society, Swinton, UK, UK, Article 35, 35:1–35:6 pages. http://dl.acm.org/citation.cfm?id=2578048.2578092

[430] Jim McKeown. 2004. The use of a multimedia lesson to increase novice programmers' understanding of programming array concepts. J. Comput. Sci. Coll. 19, 4 (2004), 39–50. http://dl.acm.org/citation.cfm?id=1050231.1050236

[431] Dawn McKinney and Leo F. Denton. 2004. Houston, we have a problem: there's a leak in the CS1 affective oxygen tank. In Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '04). ACM, New York, NY, USA, 236–239. https://doi.org/10.1145/971300.971386

[432] Dawn McKinney and Leo F. Denton. 2005. Affective assessment of team skills in agile CS1 labs: the good, the bad, and the ugly. In Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '05). ACM, New York, NY, USA, 465–469. https://doi.org/10.1145/1047344.1047494

[433] Dawn McKinney and Leo F. Denton. 2006. Developing collaborative skills early in the CS curriculum in a laboratory environment. In Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '06). ACM, New York, NY, USA, 138–142. https://doi.org/10.1145/1121341.1121387

[434] Kirby McMaster, Brian Rague, Samuel Sambasivam, and Stuart Wolthuis. 2016. Coverage of CS1 programming concepts in C++ and Java textbooks. In IEEE Frontiers in Education Conference (FIE '16). IEEE, 1–8. https://doi.org/10.1109/FIE.2016.7757618

[435] William Isaac McWhorter and Brian C. O'Connor. 2009. Do LEGO® Mindstorms® motivate students in CS1?. In Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE '09). ACM, New York, NY, USA, 438–442. https://doi.org/10.1145/1508865.1509019

[436] R. P. Medeiros, G. L. Ramalho, and T. P. Falcão. 2018. A systematic literature review on teaching and learning introductory programming in higher education. IEEE Transactions on Education (2018), 1–14. https://doi.org/10.1109/TE.2018.2864133

[437] Paola Medel and Vahab Pournaghshband. 2017. Eliminating gender bias in computer science education materials. In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17). ACM, New York, NY, USA, 411–416. https://doi.org/10.1145/3017680.3017794

[438] M. Dee Medley. 2007. Inquiry-based learning in CS1. J. Comput. Sci. Coll. 23, 2 (2007), 209–215. http://dl.acm.org/citation.cfm?id=1292428.1292464

[439] Andréa Mendonça, Clara de Oliveira, Dalton Guerrero, and Evandro Costa. 2009. Difficulties in solving ill-defined problems: a case study with introductory computer programming students. In IEEE Frontiers in Education Conference (FIE '09). IEEE, 1–6. https://doi.org/10.1109/FIE.2009.5350628

[440] Alexander Mentis, Charles Reynolds, Donald Abbott-McCune, and Benjamin Ring. 2009. Cementing abstraction with a concrete application: a focused use of robots in CS1. In Proceedings of ASEE Annual Conference and Exposition. 14.

[441] László Menyhárt and Gáborné Pap. 2014. Presentation of improved version of guide application for teaching programming fundamentals. In Proceedings on 7th International Multi-Conference on Engineering and Technological Innovation (IMETI '14). IIIS, 77–82.

[442] Michael A. Miljanovic and Jeremy S. Bradbury. 2017. Robobug: a serious game for learning debugging techniques. In Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17). ACM, New York, NY, USA, 93–100. https://doi.org/10.1145/3105726.3106173

[443] Alexander Miller, Stuart Reges, and Allison Obourn. 2017. jGRASP: a simple, visual, intuitive programming environment for CS1 and CS2. ACM Inroads 8, 4 (2017), 53–58. https://doi.org/10.1145/3148562

[444] Bradley N. Miller and David L. Ranum. 2012. Beyond PDF and epub: toward an interactive textbook. In Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '12). ACM, New York, NY, USA, 150–155. https://doi.org/10.1145/2325296.2325335

[445] L. D. Miller, Leen-Kiat Soh, Vlad Chiriacescu, Elizabeth Ingraham, Duane F. Shell, and Melissa Patterson Hazley. 2014. Integrating computational and creative thinking to improve learning and performance in CS1. In Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14). ACM, New York, NY, USA, 475–480. https://doi.org/10.1145/2538862.2538940

[446] Claudio Mirolo. 2012. Is iteration really easier to learn than recursion for CS1 students?. In Proceedings of the Ninth Annual International Conference on International Computing Education Research (ICER '12). ACM, New York, NY, USA, 99–104. https://doi.org/10.1145/2361276.2361296

[447] Ananya Misra, Douglas Blank, and Deepak Kumar. 2009. A music context for teaching introductory computing. In Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '09). ACM, New York, NY, USA, 248–252. https://doi.org/10.1145/1562877.1562955

[448] Behram F.T. Mistree, Bhupesh Chandra, Ewen Cheslack-Postava, Philip Levis, and David Gay. 2011. Emerson: accessible scripting for applications in an extensible virtual world. In Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2011). ACM, New York, NY, USA, 77–90. https://doi.org/10.1145/2048237.2048247

[449] Phatludi Modiba, Vreda Pieterse, and Bertram Haskins. 2016. Evaluating plagiarism detection software for introductory programming assignments. In Proceedings of the Computer Science Education Research Conference 2016 (CSERC '16). ACM, New York, NY, USA, 37–46. https://doi.org/10.1145/2998551.2998558

[450] Susana Montero, Paloma Díaz, David Díez, and Ignacio Aedo. 2010. Dual instructional support materials for introductory object-oriented programming: classes vs. objects. In IEEE Education Engineering Conference (EDUCON '10). IEEE, 1929–1934. https://doi.org/10.1109/EDUCON.2010.5492438

[451] Jan Moons and Carlos De Backer. 2009. Rationale behind the design of the EduVisor software visualization component. Electronic Notes in Theoretical Computer Science 224 (2009), 57–65. https://doi.org/10.1016/j.entcs.2008.12.049

[452] Anmol More, Jitendra Kumar, and VG Renumol. 2011. Web based programming assistance tool for novices. In IEEE International Conference on Technology for Education (T4E '11). IEEE, 270–273. https://doi.org/10.1109/T4E.2011.55

[453] Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. 2004. Visualizing programs with Jeliot 3. In Proceedings of the Working Conference on Advanced Visual Interfaces (AVI '04). ACM, New York, NY, USA, 373–376. https://doi.org/10.1145/989863.989928

[454] Michael Morgan, Jane Sinclair, Matthew Butler, Neena Thota, Janet Fraser, Gerry Cross, and Jana Jackova. 2017. Understanding international benchmarks on student engagement: awareness and research alignment from a computer science perspective. In Proceedings of the 2017 ITiCSE Conference on Working Group Reports (ITiCSE-WGR '17). ACM, New York, NY, USA, 1–24. https://doi.org/10.1145/3174781.3174782

[455] Sally H. Moritz, Fang Wei, Shahida M. Parvez, and Glenn D. Blank. 2005. From objects-first to design-first with multimedia and intelligent tutoring. In Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '05). ACM, New York, NY, USA, 99–103. https://doi.org/10.1145/1067445.1067475

[456] Barbara Moskal, Deborah Lurie, and Stephen Cooper. 2004. Evaluating the effectiveness of a new instructional approach. In Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '04). ACM, New York, NY, USA, 75–79. https://doi.org/10.1145/971300.971328

[457] Frank Mueller and Antony L. Hosking. 2003. Penumbra: an Eclipse plugin for introductory programming. In Proceedings of the 2003 OOPSLA Workshop on Eclipse Technology eXchange (Eclipse '03). ACM, New York, NY, USA, 65–68. https://doi.org/10.1145/965660.965674

[458] Andreas Mühling, Peter Hubwieser, and Marc Berges. 2015. Dimensions of programming knowledge. In International Conference on Informatics in Schools: Situation, Evolution, and Perspectives (ISSEP 2015). Springer, Cham, 32–44. https://doi.org/10.1007/978-3-319-25396-1_4

[459] Paul Mullins, Deborah Whitfield, and Michael Conlon. 2009. Using Alice 2.0 as a first language. J. Comput. Sci. Coll. 24, 3 (2009), 136–143. http://dl.acm.org/citation.cfm?id=1409873.1409900

[460] Jonathan P. Munson and Elizabeth A. Schilling. 2016. Analyzing novice programmers' response to compiler error messages. J. Comput. Sci. Coll. 31, 3 (2016), 53–61. http://dl.acm.org/citation.cfm?id=2835377.2835386

[461] Surya Muntha and Larry Morell. 2006. Adding object-orientation to Genesis. J. Comput. Sci. Coll. 21, 5 (2006), 101–106. http://dl.acm.org/citation.cfm?id=1127351.1127369

[462] Christian Murphy, Gail Kaiser, Kristin Loveland, and Sahar Hasan. 2009. Retina: helping students and instructors based on observed programming activities. In Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE '09). ACM, New York, NY, USA, 178–182. https://doi.org/10.1145/1508865.1508929

[463] Christian Murphy, Eunhee Kim, Gail Kaiser, and Adam Cannon. 2008. Backstop: a tool for debugging runtime errors. In Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '08). ACM, New York, NY, USA, 173–177. https://doi.org/10.1145/1352135.1352193

[464] Laurie Murphy and Lynda Thomas. 2008. Dangers of a fixed mindset: implications of self-theories research for computer science education. In Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '08). ACM, New York, NY, USA, 271–275. https://doi.org/10.1145/1384271.1384344

[465] Laurie Murphy and David Wolff. 2009. Creating video podcasts for CS1: lessons learned. J. Comput. Sci. Coll. 25, 1 (2009), 152–158. http://dl.acm.org/citation.cfm?id=1619221.1619252

[466] Thomas P. Murtagh. 2007. Squint: barely visible library support for CS1. In Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '07). ACM, New York, NY, USA, 526–530. https://doi.org/10.1145/1227310.1227489

[467] Thomas P. Murtagh. 2007. Weaving CS into CS1: a doubly depth-first approach. In Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '07). ACM, New York, NY, USA, 336–340. https://doi.org/10.1145/1227310.1227429

[468] Greg L. Nelson, Benjamin Xie, and Andrew J. Ko. 2017. Comprehension first: evaluating a novel pedagogy and tutoring system for program tracing in CS1. In Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17). ACM, New York, NY, USA, 2–11. https://doi.org/10.1145/3105726.3106178

[469] Vicente Lustosa Neto, Roberta Coelho, Larissa Leite, Dalton S. Guerrero, and Andrea P. Mendonça. 2013. POPT: a problem-oriented programming and testing approach for novice students. In Proceedings of the 2013 International Conference on Software Engineering (ICSE '13). IEEE Press, Piscataway, NJ, USA, 1099–1108. http://dl.acm.org/citation.cfm?id=2486788.2486939

[470] Paul Neve, Gordon Hunter, David Livingston, and James Orwell. 2012. NoobLab: an intelligent learning environment for teaching programming. In Proceedings of the The 2012 IEEE/WIC/ACM International Joint Conferences on Web Intelligence and Intelligent Agent Technology — Volume 03 (WI-IAT '12). IEEE Computer Society, Washington, DC, USA, 357–361. https://doi.org/10.1109/WI-IAT.2012.218

[471] Tia Newhall, Lisa Meeden, Andrew Danner, Ameet Soni, Frances Ruiz, and Richard Wicentowski. 2014. A support program for introductory CS courses that improves student performance and retains students from underrepresented groups. In Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14). ACM, New York, NY, USA, 433–438. https://doi.org/10.1145/2538862.2538923

[472] Paul Newton and Stuart Shaw. 2014. Validity in educational and psychological assessment. Sage.

[473] Sin Chun Ng, Steven O Choy, Reggie Kwan, and SF Chan. 2005. A web-based environment to improve teaching and learning of computer programming in distance education. In International Conference on Web-Based Learning (ICWL '05). Springer, Berlin, Heidelberg, 279–290. https://doi.org/10.1007/11528043_28

[474] Grace Ngai, Winnie W.Y. Lau, Stephen C.F. Chan, and Hong-va Leong. 2010. On the implementation of self-assessment in an introductory programming course. SIGCSE Bull. 41, 4 (2010), 85–89. https://doi.org/10.1145/1709424.1709453

[475] Thuy-Linh Nguyen, Dip Nandi, and Geoff Warburton. 2011. Alice in online and on-campus environments — how well is it received?. In Proceedings of Information Systems Educators Conference (ISECON '11).

[476] Uolevi Nikula, Orlena Gotel, and Jussi Kasurinen. 2011. A motivation guided holistic rehabilitation of the first programming course. Trans. Comput. Educ. 11, 4, Article 24 (2011), 24:1–24:38 pages. https://doi.org/10.1145/2048931.2048935

[477] Uolevi Nikula, Orlena Gotel, and Jussi Kasurinen. 2011. A motivation guided holistic rehabilitation of the first programming course. Trans. Comput. Educ. 11, 4, Article 24 (Nov. 2011), 24:1–24:38 pages. https://doi.org/10.1145/2048931.2048935

[478] Keith Nolan and Susan Bergin. 2016. The role of anxiety when learning to program: a systematic review of the literature. In Proceedings of the 16th Koli Calling International Conference on Computing Education Research (Koli Calling '16). ACM, New York, NY, USA, 61–70. https://doi.org/10.1145/2999541.2999557

[479] Cindy Norris, Frank Barry, James B. Fenwick Jr., Kathryn Reid, and Josh Rountree. 2008. ClockIt: collecting quantitative data on how beginning software developers really work. In Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '08). ACM, New York, NY, USA, 37–41. https://doi.org/10.1145/1384271.1384284

[480] Jeff Offutt, Paul Ammann, Kinga Dobolyi, Chris Kauffmann, Jaime Lester, Upsorn Praphamontripong, Huzefa Rangwala, Sanjeev Setia, Pearl Wang, and Liz White. 2017. A novel self-paced model for teaching programming. In Proceedings of the Fourth ACM Conference on Learning @ Scale (L@S '17). ACM, New York, NY, USA, 177–180. https://doi.org/10.1145/3051457.3053978

[481] Ken Okada, Manabu Sugiura, Yoshiaki Matsuzawa, Megumi Araki, and Hajime Ohiwa. 2008. Programming in Japanese for literacy education. In History of Computing and Education 3 (HCE3). Springer, US, 171–176. https://doi.org/10.1007/978-0-387-09657-5_12

[482] Osvaldo Luiz Oliveira, Ana Maria Monteiro, and Norton Trevisan Roman. 2013. Can natural language be utilized in the learning of programming fundamentals?. In IEEE Frontiers in Education Conference (FIE '13). IEEE, 1851–1856. https://doi.org/10.1109/FIE.2013.6685157

[483] Rafael AP Oliveira, Lucas BR Oliveira, Bruno BP Cafeo, and Vinicius HS Durelli. 2015. Evaluation and assessment of effects on exploring mutation testing in programming courses. In IEEE Frontiers in Education Conference (FIE '15). IEEE, 1–9. https://doi.org/10.1109/FIE.2015.7344051

[484] Tihomir Orehovački, Danijel Radošević, and Mladen Konecki. 2012. Acceptance of Verificator by information science students. In Proceedings of the 34th International Conference on Information Technology Interfaces (ITI '12). IEEE, 223–230. https://doi.org/10.2498/iti.2012.0451

[485] Tihomir Orehovački, Danijel Radošević, and Mladen Konecki. 2014. Perceived quality of Verificator in teaching programming. In 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO '14). IEEE, 643–648. https://doi.org/10.1109/MIPRO.2014.6859646

[486] Claudia Ott, Anthony Robins, Patricia Haden, and Kerry Shephard. 2015. Illustrating performance indicators and course characteristics to support students' self-regulated learning in CS1. Computer Science Education 25, 2 (2015), 174–198. https://doi.org/10.1080/08993408.2015.1033129

[487] Claudia Ott, Anthony Robins, and Kerry Shephard. 2016. Translating principles of effective feedback for students into the CS1 context. Trans. Comput. Educ. 16, 1, Article 1 (2016), 1:1–1:27 pages. https://doi.org/10.1145/2737596

[488] Özcan Özyurt and Hacer Özyurt. 2016. Using Facebook to enhance learning experiences of students in computer programming at introduction to programming and algorithm course. Computer Applications in Engineering Education 24 (2016), 546–554. https://doi.org/10.1002/cae.21730

[489] James Dean Palmer. 2013. Computer Science I with Flare. J. Comput. Sci. Coll. 28, 4 (2013), 94–100. http://dl.acm.org/citation.cfm?id=2458539.2458557

[490] James Dean Palmer, Joseph Flieger, and Eddie Hillenbrand. 2011. JavaGrinder: a web-based platform for teaching early computing skills. In Proceedings of ASEE Annual Conference and Exposition. ASEE, 15.

[491] Andrei Papancea, Jaime Spacco, and David Hovemeyer. 2013. An open platform for managing short programming exercises. In Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research (ICER '13). ACM, New York, NY, USA, 47–52. https://doi.org/10.1145/2493394.2493401

[492] Sagar Parihar, Ziyaan Dadachanji, Praveen Kumar Singh, Rajdeep Das, Amey Karkare, and Arnab Bhattacharya. 2017. Automatic grading and feedback using program repair for introductory programming courses. In Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17). ACM, New York, NY, USA, 92–97. https://doi.org/10.1145/3059009.3059026

[493] Myung Ah Park. 2010. Designing CS1 as an eye-opening tool to the utility of computer science and a research-initiating tool. J. Comput. Sci. Coll. 25, 4 (2010), 44–51. http://dl.acm.org/citation.cfm?id=1734797.1734805

[494] Kevin R Parker, Thomas A Ottaway, and Joseph T Chao. 2006. Criteria for the selection of a programming language for introductory courses. International Journal of Knowledge and Learning 2, 1-2 (2006), 119–139.

[495] Dale Parsons and Patricia Haden. 2006. Parson's Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses. In Proceedings of the 8th Australasian Conference on Computing Education - Volume 52 (ACE '06). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 157–163. http://dl.acm.org/citation.cfm?id=1151869.1151890

[496] Jody Paul. 2006. Leveraging students knowledge: introducing CS 1 concepts. J. Comput. Sci. Coll. 22, 1 (2006), 246–252. http://dl.acm.org/citation.cfm?id=1181811.1181846

[497] Jody Paul. 2006. "What first?" Addressing the critical initial weeks of CS-1. In IEEE Frontiers in Education Conference (FIE '06). IEEE, 1–5. https://doi.org/10.1109/FIE.2006.322382

[498] Jarred Payne, Vincent Cavé, Raghavan Raman, Mathias Ricken, Robert Cartwright, and Vivek Sarkar. 2011. DrHJ: a lightweight pedagogic IDE for Habanero Java. In Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ '11). ACM, New York, NY, USA, 147–150. https://doi.org/10.1145/2093157.2093180

[499] Patrick Peachock, Nicholas Iovino, and Bonita Sharif. 2017. Investigating eye movements in natural language and C++ source code — a replication experiment. In International Conference on Augmented Cognition. Springer, Cham, 206–218. https://doi.org/10.1007/978-3-319-58628-1_17

[500] Janice L. Pearce, Mario Nakazawa, and Scott Heggen. 2015. Improving problem decomposition ability in CS1 through explicit guided inquiry-based instruction. J. Comput. Sci. Coll. 31, 2 (2015), 135–144. http://dl.acm.org/citation.cfm?id=2831432.2831453

[501] Arnold Pears, Stephen Seidman, Crystal Eney, Päivi Kinnunen, and Lauri Malmi. 2005. Constructing a core literature for computing education research. ACM SIGCSE Bulletin 37, 4 (2005), 152–161.

[502] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. 2007. A survey of literature on the teaching of introductory programming. In ITiCSE Working Group Reports (ITiCSE-WGR '07). ACM, New York, NY, USA, 204–223. https://doi.org/10.1145/1345443.1345441

[503] Štefan Pero. 2014. How to detect programming skills of students?. In European Summit on Immersive Education. Springer, 63–72. https://doi.org/10.1007/978-3-319-22017-8_6

[504] Markeya S. Peteranetz, Abraham E. Flanigan, Duane F. Shell, and Leen-Kiat Soh. 2016. Perceived instrumentality and career aspirations in CS1 courses: change and relationships with achievement. In Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16). ACM, New York, NY, USA, 13–21. https://doi.org/10.1145/2960310.2960320

[505] Andrew Petersen, Michelle Craig, Jennifer Campbell, and Anya Tafliovich. 2016. Revisiting why students drop CS1. In Proceedings of the 16th Koli Calling International Conference on Computing Education Research (Koli Calling '16). ACM, New York, NY, USA, 71–80. https://doi.org/10.1145/2999541.2999552

[506] Andrew Petersen, Michelle Craig, and Daniel Zingaro. 2011. Reviewing CS1 exam question content. In Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE '11). ACM, New York, NY, USA, 631–636. https://doi.org/10.1145/1953163.1953340

[507] Andrew Petersen, Jaime Spacco, and Arto Vihavainen. 2015. An exploration of error quotient in multiple contexts. In Proceedings of the 15th Koli Calling Conference on Computing Education Research (Koli Calling '15). ACM, New York, NY, USA, 77–86. https://doi.org/10.1145/2828959.2828966

[508] Raymond Pettit, John Homer, Roger Gee, Susan Mengel, and Adam Starbuck. 2015. An empirical study of iterative improvement in programming assignments. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15). ACM, New York, NY, USA, 410–415. https://doi.org/10.1145/2676723.2677279

[509] Raymond S. Pettit, John Homer, and Roger Gee. 2017. Do enhanced compiler error messages help students?: Results inconclusive.. In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17). ACM, New York, NY, USA, 465–470. https://doi.org/10.1145/3017680.3017768

[510] Andrew M Phelps, Christopher A Egert, and Kevin J Bierre. 2005. MUPPETS: multi-user programming pedagogy for enhancing traditional study: an environment for both upper and lower division students. In IEEE Frontiers in Education Conference (FIE '05). IEEE, S2H–8. https://doi.org/10.1109/FIE.2005.1612247

[511] Phitchaya Mangpo Phothilimthana and Sumukh Sridhara. 2017. High-coverage hint generation for massive courses: do automated hints help CS1 students?. In Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17). ACM, New York, NY, USA, 182–187. https://doi.org/10.1145/3059009.3059058

[512] Dinh Dong Phuong, Yusuke Yokota, Fumiko Harada, and Hiromitsu Shimakawa. 2010. Graining and filling understanding gaps for novice programmers. In International Conference on Education and Management Technology (ICEMT '10). IEEE, 60–64. https://doi.org/10.1109/ICEMT.2010.5657544

[513] Marco Piccioni, Christian Estler, and Bertrand Meyer. 2014. Spoc-supported introduction to programming. In Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education (ITiCSE '14). ACM, New York, NY, USA, 3–8. https://doi.org/10.1145/2591708.2591759

[514] Chris Piech, Mehran Sahami, Daphne Koller, Steve Cooper, and Paulo Blikstein. 2012. Modeling how students learn to program. In Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12). ACM, New York, NY, USA, 153–160. https://doi.org/10.1145/2157136.2157182

[515] Kerttu Pollari-Malmi, Julio Guerra, Peter Brusilovsky, Lauri Malmi, and Teemu Sirkiä. 2017. On the value of using an interactive electronic textbook in an introductory programming course. In Proceedings of the 17th Koli Calling International Conference on Computing Education Research (Koli Calling '17). ACM, New York, NY, USA, 168–172. https://doi.org/10.1145/3141880.3141890

[516] M. Poole. 2017. Extending the design of a blocks-based Python environment to support complex types. In IEEE Blocks and Beyond Workshop (B&B '17). IEEE, 1–7. https://doi.org/10.1109/BLOCKS.2017.8120400

[517] Leo Porter, Dennis Bouvier, Quintin Cutts, Scott Grissom, Cynthia Lee, Robert McCartney, Daniel Zingaro, and Beth Simon. 2016. A multi-institutional study of peer instruction in introductory computing. In Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16). ACM, New York, NY, USA, 358–363. https://doi.org/10.1145/2839509.2844642

[518] Leo Porter, Mark Guzdial, Charlie McDowell, and Beth Simon. 2013. Success in introductory programming: what works? Commun. ACM 56, 8 (2013), 34–36. https://doi.org/10.1145/2492007.2492020

[519] Leo Porter and Beth Simon. 2013. Retaining nearly one-third more majors with a trio of instructional best practices in CS1. In Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13). ACM, New York, NY, USA, 165–170. https://doi.org/10.1145/2445196.2445248

[520] Leo Porter and Daniel Zingaro. 2014. Importance of early performance in CS1: two conflicting assessment stories. In Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14). ACM, New York, NY, USA, 295–300. https://doi.org/10.1145/2538862.2538912

[521] Leo Porter, Daniel Zingaro, and Raymond Lister. 2014. Predicting student success using fine grain clicker data. In Proceedings of the Tenth Annual Conference on International Computing Education Research (ICER '14). ACM, New York, NY, USA, 51–58. https://doi.org/10.1145/2632320.2632354

[522] Vahab Pournaghshband. 2013. Teaching the security mindset to CS1 students. In Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13). ACM, New York, NY, USA, 347–352. https://doi.org/10.1145/2445196.2445299

[523] Kris Powers, Stacey Ecott, and Leanne M. Hirshfield. 2007. Through the looking glass: teaching CS0 with alice. In Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '07). ACM, New York, NY, USA, 213–217. https://doi.org/10.1145/1227310.1227386

[524] Kellie Price and Suzanne Smith. 2014. Improving student performance in CS1. J. Comput. Sci. Coll. 30, 2 (2014), 157–163. http://dl.acm.org/citation.cfm?id=2667432.2667454

[525] Thomas W. Price and Tiffany Barnes. 2015. Comparing textual and block interfaces in a novice programming environment. In Proceedings of the Eleventh Annual International Conference on International Computing Education Research (ICER '15). ACM, New York, NY, USA, 91–99. https://doi.org/10.1145/2787622.2787712

[526] Mitchell Pryor. 2012. Real-time monitoring of student procrastination in a PSI first-year programming course. In Proceedings of ASEE Annual Conference and Exposition. ASEE, 14.

[527] Wayne Pullan, Steven Drew, and Steven Tucker. 2013. An integrated approach to teaching introductory programming. In Second International Conference on E-Learning and E-Technologies in Education (ICEEE '13). IEEE, 81–86. https://doi.org/10.1109/ICeLeTE.2013.6644352

[528] William Punch, Richard Enbody, Colleen McDonough, and Jon Sticklen. 2010. Measuring the effect of intervening early for academically at risk students in a CS1 course. ASEE.

[529] Sandeep Purao, Maung Sein, Hallgeir Nilsen, and Even Åby Larsen. 2017. Setting the pace: experiments with Keller's PSI. IEEE Transactions on Education 60 (2017), 97–104. Issue 2. https://doi.org/10.1109/TE.2016.2588460

[530] Yizhou Qian and James Lehman. 2017. Students' misconceptions and other difficulties in introductory programming: a literature review. ACM Trans. Comput. Educ. 18, 1, Article 1 (2017), 1:1–1:24 pages. https://doi.org/10.1145/3077618

[531] Keith Quille, Natalie Culligan, and Susan Bergin. 2017. Insights on gender differences in CS1: a multi-institutional, multi-variate study.. In Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17). ACM, New York, NY, USA, 263–268. https://doi.org/10.1145/3059009.3059048

[532] Martin Quinson and Gérald Oster. 2015. A teaching system to learn programming: the programmer's learning machine. In Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '15). ACM, New York, NY, USA, 260–265. https://doi.org/10.1145/2729094.2742626

[533] Alex Radermacher, Gursimran Walia, and Richard Rummelt. 2012. Assigning student programming pairs based on their mental model consistency: an initial investigation. In Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12). ACM, New York, NY, USA, 325–330. https://doi.org/10.1145/2157136.2157236

[534] Alex D. Radermacher and Gursimran S. Walia. 2011. Investigating the effective implementation of pair programming: an empirical investigation. In Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE '11). ACM, New York, NY, USA, 655–660. https://doi.org/10.1145/1953163.1953346

[535] Teemu Rajala, Mikko-Jussi Laakso, Erkki Kaila, and Tapio Salakoski. 2007. VILLE: a language-independent program visualization tool. In Proceedings of the Seventh Baltic Sea Conference on Computing Education Research — Volume 88 (Koli Calling '07). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 151–159. http://dl.acm.org/citation.cfm?id=2449323.2449340

[536] R.Z. Ramli, A.Y. Kapi, and N. Osman. 2015. Visualization makes array easy. In Proceedings of the 2015 International Conference on Testing and Measurement: Techniques and Applications (TMTA '15). CRC Press, 381–384.

[537] Justus J Randolph, George Julnes, Erkki Sutinen, and Steve Lehman. 2008. A methodological review of computer science education research. Journal of Information Technology Education: Research 7 (2008), 135–162.

[538] R. Rashkovits and I. Lavy. 2011. Students' strategies for exception handling. Journal of Information Technology Education: Research 10, 1 (2011), 183–207. https://doi.org/10.28945/1500

[539] Andrew Ray. 2012. Evolving the usage of LEGO robots in CS1 to facilitate high-level problem solving. In Proceedings of the IASTED International Conference on Computers and Advanced Technology in Education (CATE '12). ICTA Press, 91–98. https://doi.org/10.2316/P.2012.774-034

[540] Saquib Razak. 2013. A case for course capstone projects in CS1. In Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13). ACM, New York, NY, USA, 693–698. https://doi.org/10.1145/2445196.2445398

[541] Susan Reardon and Brendan Tangney. 2014. Smartphones, studio-based learning, and scaffolding: helping novices learn to program. Trans. Comput. Educ. 14, 4, Article 23 (2014), 23:1–23:15 pages. https://doi.org/10.1145/2677089

[542] Samuel A. Rebelsky, Janet Davis, and Jerod Weinman. 2013. Building knowledge and confidence with mediascripting: a successful interdisciplinary approach to CS1. In Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13). ACM, New York, NY, USA, 483–488. https://doi.org/10.1145/2445196.2445342

[543] Dale Reed, Sam John, Ryan Aviles, and Feihong Hsu. 2004. CFX: finding just the right examples for CS1. In Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '04). ACM, New York, NY, USA, 363–367. https://doi.org/10.1145/971300.971426

[544] Stuart Reges. 2006. Back to basics in CS1 and CS2. SIGCSE Bull. 38, 1 (2006), 293–297. https://doi.org/10.1145/1124706.1121432

[545] Charles Reis and Robert Cartwright. 2004. Taming a professional IDE for the classroom. In Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '04). ACM, New York, NY, USA, 156–160. https://doi.org/10.1145/971300.971357

[546] Jake Renzella and Andrew Cain. 2017. Supporting better formative feedback in task-oriented portfolio assessment. In IEEE 6th International Conference on Teaching, Assessment, and Learning for Engineering (TALE '17). IEEE, 360–367. https://doi.org/10.1109/TALE.2017.8252362

[547] Emma Riese. 2017. Students' experience and use of assessment in an online introductory programming course. In International Conference on Learning and Teaching in Computing and Engineering (LaTiCE '17). IEEE, 30–34. https://doi.org/10.1109/LaTiCE.2017.13

[548] Peter C. Rigby and Suzanne Thompson. 2005. Study of novice programmers using Eclipse and Gild. In Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange (Eclipse '05). ACM, New York, NY, USA, 105–109. https://doi.org/10.1145/1117696.1117718

[549] Kelly Rivers and Kenneth R. Koedinger. 2015. Data-driven hint generation in vast solution spaces: a self-improving Python programming tutor. International Journal of Artificial Intelligence in Education 27, 1 (2015), 37–64. https://doi.org/10.1007/s40593-015-0070-z

[550] Mona Rizvi, Thorna Humphries, Debra Major, Heather Lauzun, and Meghan Jones. 2011. A new CS0 course for at-risk majors. In 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T '11). IEEE, 314–323. https://doi.org/10.1109/CSEET.2011.5876101

[551] Eric Roberts and Keith Schwarz. 2013. A portable graphics library for introductory CS. In Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '13). ACM, New York, NY, USA, 153–158. https://doi.org/10.1145/2462476.2465590

[552] Michael Robey, Brian R Von Konsky, Jim Ivins, Susan J Gribble, Allan Loh, and David Cooper. 2006. Student self-motivation: lessons learned from teaching first year computing. In IEEE Frontiers in Education Conference (FIE '06). IEEE, 6–11. https://doi.org/10.1109/FIE.2006.322363

[553] Anthony Robins. 2010. Learning edge momentum: a new account of outcomes in CS1. Computer Science Education 20, 1 (2010), 37–71. https://doi.org/10.1080/08993401003612167

[554] 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

[555] Ma. Mercedes T. Rodrigo and Ryan S.J.d. Baker. 2009. Coarse-grained detection of student frustration in an introductory programming course. In Proceedings of the Fifth International Workshop on Computing Education Research Workshop (ICER '09). ACM, New York, NY, USA, 75–80. https://doi.org/10.1145/1584322.1584332

[556] Ma. Mercedes T. Rodrigo, Ryan S. Baker, Matthew C. Jadud, Anna Christine M. Amarra, Thomas Dy, Maria Beatriz V. Espejo-Lahoz, Sheryl Ann L. Lim, Sheila A.M.S. Pascua, Jessica O. Sugay, and Emily S. Tabanao. 2009. Affective and behavioral predictors of novice programmer achievement. In Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '09). ACM, New York, NY, USA, 156–160. https://doi.org/10.1145/1562877.1562929

[557] Maria Mercedes T Rodrigo, Emily Tabanao, Ma Beatriz E Lahoz, and Matthew C Jadud. 2009. Analyzing online protocols to characterize novice Java programmers. Philippine Journal of Science 138, 2 (2009), 177–190.

[558] Reinout Roels, Paul Meştereagă, and Beat Signer. 2016. An interactive source code visualisation plug-in for the MindXpres presentation platform. In Communications in Computer and Information Science. Springer, Cham, 169–188. https://doi.org/10.1007/978-3-319-29585-5_10

[559] Reinout Roels, Paul Meźtereagź, and Beat Signer. 2015. Towards enhanced presentation-based teaching of programming: an interactive source code visualisation approach. In Proceedings of the 7th International Conference on Computer Supported Education—Volume 1 (CSEDU '15). SciTePress, 98–107.

[560] Timo Rongas, Arto Kaarna, and Heikki Kalviainen. 2004. Classification of computerized learning tools for introductory programming courses: learning approach. In IEEE International Conference on Advanced Learning Technologies (ICALT '04). IEEE, 678–680. https://doi.org/10.1109/ICALT.2004.1357618

[561] Guido Rößling. 2010. A family of tools for supporting the learning of programming. Algorithms 3, 2 (2010), 168–182. https://doi.org/10.3390/a3020168

[562] Janet Rountree, Nathan Rountree, Anthony Robins, and Robert Hannah. 2005. Observations of student competency in a CS1 course. In Proceedings of the 7th Australasian Conference on Computing Education — Volume 42 (ACE '05). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 145–149. http://dl.acm.org/citation.cfm?id=1082424.1082442

[563] Krishnendu Roy, William C Rousse, and David B DeMeritt. 2012. Comparing the mobile novice programming environments: App Inventor for Android vs. GameSalad. In IEEE Frontiers in Education Conference (FIE '12). IEEE, 1–6. https://doi.org/10.1109/FIE.2012.6462363

[564] Marc J. Rubin. 2013. The effectiveness of live-coding to teach introductory programming. In Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13). ACM, New York, NY, USA, 651–656. https://doi.org/10.1145/2445196.2445388

[565] Miguel Angel Rubio, Rocio Romero-Zaliz, Carolina Mañoso, and P Angel. 2014. Enhancing an introductory programming course with physical computing modules. In IEEE Frontiers in Education Conference (FIE '14). IEEE, 1–8. https://doi.org/10.1109/FIE.2014.7044153

[566] Mark F. Russo. 2017. Doodlepad: next-gen event-driven programming for CS1. J. Comput. Sci. Coll. 32, 4 (2017), 99–105. http://dl.acm.org/citation.cfm?id=3055338.3055356

[567] Daisuke Saito, Ayana Sasaki, Hironori Washizaki, Yoshiaki Fukazawa, and Yusuke Muto. 2017. Program learning for beginners: survey and taxonomy of programming learning tools. In IEEE 9th International Conference on Engineering Education (ICEED '17). IEEE, 137–142. https://doi.org/10.1109/ICEED.2017.8251181

[568] A Sajana, Kamal Bijlani, and R Jayakrishnan. 2015. An interactive serious game via visualization of real life scenarios to learn programming concepts. In 6th International Conference on Computing, Communication and Networking Technologies (ICCCNT) (ICCCNT '15). IEEE, 1–8. https://doi.org/10.1109/ICCCNT.2015.7395173

[569] Norsaremah Salleh, Emilia Mendes, John Grundy, and Giles St. J. Burch. 2010. The effects of neuroticism on pair programming: an empirical study in the higher education context. In Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '10). ACM, New York, NY, USA, Article 22, 22:1–22:10 pages. https://doi.org/10.1145/1852786.1852816

[570] Norsaremah Salleh, Emilia Mendes, John Grundy, and Giles St. J Burch. 2010. An empirical study of the effects of conscientiousness in pair programming using the five-factor personality model. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering — Volume 1 (ICSE '10). ACM, New York, NY, USA, 577–586. https://doi.org/10.1145/1806799.1806883

[571] Dean Sanders and Brian Dorn. 2003. Jeroo: a tool for introducing object-oriented programming. In Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '03). ACM, New York, NY, USA, 201–204. https://doi.org/10.1145/611892.611968

[572] Joseph P. Sanford, Aaron Tietz, Saad Farooq, Samuel Guyer, and R. Benjamin Shapiro. 2014. Metaphors we teach by. In Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14). ACM, New York, NY, USA, 585–590. https://doi.org/10.1145/2538862.2538945

[573] Loé Sanou, Sybille Caffiau, Patrick Girard, and Laurent Guittet. 2008. Example usage evaluation for the learning of programming using the MELBA environment. In IADIS Multi Conference on Computer Science and Information Systems; Proceedings of Interfaces and Human Computer Interaction 2008 (MCCSIS '08).

ICTA Press, 35–42.

[574] André L. Santos. 2012. An open-ended environment for teaching Java in context. In Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '12). ACM, New York, NY, USA, 87–92. https://doi.org/10.1145/2325296.2325320

[575] Linda J. Sax, Kathleen J. Lehman, and Christina Zavala. 2017. Examining the enrollment growth: non-cs majors in CS1 courses. In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17). ACM, New York, NY, USA, 513–518. https://doi.org/10.1145/3017680.3017781

[576] Pablo Schoeffel, Raul Sidnei Wazlawick, and Vinicius Ramos. 2017. Impact of pre-university factors on the motivation and performance of undergraduate students in software engineering. In IEEE 30th Conference on Software Engineering Education and Training (CSEE&T '17). IEEE, 266–275. https://doi.org/10.1109/CSEET.2017.50

[577] Joachim Schramm, Sven Strickroth, Nguyen-Thinh Le, and Niels Pinkwart. 2012. Teaching UML skills to novice programmers using a sample solution based intelligent tutoring system. In Proceedings of the 25th International Florida Artificial Intelligence Research Society Conference (FLAIRS '12). AAAI Press, 472–477.

[578] Ivonne Schröter, Jacob Krüger, Janet Siegmund, and Thomas Leich. 2017. Comprehending studies on program comprehension. In Proceedings of the 25th International Conference on Program Comprehension (ICPC '17). IEEE Press, Piscataway, NJ, USA, 308–311. https://doi.org/10.1109/ICPC.2017.9

[579] Carsten Schulte and Jens Bennedsen. 2006. What do teachers teach in introductory programming?. In Proceedings of the Second International Workshop on Computing Education Research (ICER '06). ACM, New York, NY, USA, 17–28. https://doi.org/10.1145/1151588.1151593

[580] Jorg Schulze, Matthias Langrich, and Antje Meyer. 2007. The success of the demidovich-principle in undergraduate C# programming education. In IEEE Frontiers In Education Conference (FIE '07). IEEE, F4C–7–F4C–12. https://doi.org/10.1109/FIE.2007.4418090

[581] Andrew Scott, Mike Watkins, and Duncan McPhee. 2008. E-learning for novice programmers; a dynamic visualisation and problem solving tool. In 3rd International Conference on Information and Communication Technologies: From Theory to Applications (ICTTA '08). IEEE, 1–6. https://doi.org/10.1109/ICTTA.2008.4529966

[582] Michael James Scott, Steve Counsell, Stanislao Lauria, Stephen Swift, Allan Tucker, Martin Shepperd, and Gheorghita Ghinea. 2015. Enhancing practice and achievement in introductory programming with a robot Olympics. IEEE Transactions on Education 58 (2015), 249–254. Issue 4. https://doi.org/10.1109/TE.2014.2382567

[583] Michael James Scott and Gheorghita Ghinea. 2014. Measuring enrichment: the assembly and validation of an instrument to assess student self-beliefs in CS1. In Proceedings of the Tenth Annual Conference on International Computing Education Research (ICER '14). ACM, New York, NY, USA, 123–130. https://doi.org/10.1145/2632320.2632350

[584] Michael J Scott and Gheorghita Ghinea. 2014. On the domain-specificity of mindsets: the relationship between aptitude beliefs and programming practice. IEEE Transactions on Education 57 (2014), 169–174. Issue 3. https://doi.org/10.1109/TE.2013.2288700

[585] P. Seeling. 2016. Switching to blend-Ed: effects of replacing the textbook with the browser in an introductory computer programming course. In IEEE Frontiers in Education Conference (FIE '16). IEEE, 1–5. https://doi.org/10.1109/FIE.2016.7757620

[586] Otto Seppälä, Petri Ihantola, Essi Isohanni, Juha Sorva, and Arto Vihavainen. 2015. Do we know how difficult the rainfall problem is?. In Proceedings of the 15th Koli Calling Conference on Computing Education Research (Koli Calling '15). ACM, New York, NY, USA, 87–96. https://doi.org/10.1145/2828959.2828963

[587] Amber Settle, John Lalor, and Theresa Steinbach. 2015. Reconsidering the impact of CS1 on novice attitudes. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15). ACM, New York, NY, USA, 229–234. https://doi.org/10.1145/2676723.2677235

[588] Olga Shabalina, Christos Malliarakis, Florica Tomos, and Peter Mozelius. 2017. Game-based learning for learning to program: from learning through play to learning through game development. In European Conference on Games Based Learning 2017 (ECGBL '17), Vol. 11. Academic Conferences and Publishing International Limited, 571–576.

[589] Steven C. Shaffer and Mary Beth Rosson. 2013. Increasing student success by modifying course delivery based on student submission data. ACM Inroads 4, 4 (2013), 81–86. https://doi.org/10.1145/2537753.2537778

[590] Ritu Sharma, Haifeng Shen, and Robert Goodwin. 2016. Voluntary participation in discussion forums as an engagement indicator: an empirical study of teaching first-year programming. In Proceedings of the 28th Australian Conference on Computer-Human Interaction (OzCHI '16). ACM, New York, NY, USA, 489–493. https://doi.org/10.1145/3010915.3010967

[591] Jason H Sharp and Leah A Schultz. 2013. An exploratory study of the use of video as an instructional tool in an introductory C# programming course. Information Systems Education Journal 11, 6 (2013), 33.

[592] Alan Shaw. 2011. Using a collaborative programming methodology to incorporate social computing projects into introductory computer science courses. In Eighth International Conference on Information Technology: New Generations (ITNG '11). IEEE, 7–11. https://doi.org/10.1109/ITNG.2011.9

[593] Judy Sheard, Angela Carbone, Raymond Lister, Beth Simon, Errol Thompson, and Jacqueline L. Whalley. 2008. Going SOLO to assess novice programmers. In Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '08). ACM, New York, NY, USA, 209–213. https://doi.org/10.1145/1384271.1384328

[594] Judy Sheard, Simon, Matthew Butler, Katrina Falkner, Michael Morgan, and Amali Weerasinghe. 2017. Strategies for maintaining academic integrity in first-year computing courses. In Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17). ACM, New York, NY, USA, 244–249. https://doi.org/10.1145/3059009.3059064

[595] Judy Sheard, Simon, Angela Carbone, Donald Chinn, Tony Clear, Malcolm Corney, Daryl D'Souza, Joel Fenwick, James Harland, Mikko-Jussi Laakso, and Donna Teague. 2013. How difficult are exams?: a framework for assessing the complexity of introductory programming exams. In Proceedings of the Fifteenth Australasian Computing Education Conference — Volume 136 (ACE '13). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 145–154. http://dl.acm.org/citation.cfm?id=2667199.2667215

[596] Judy Sheard, Simon, Angela Carbone, Daryl D'Souza, and Margaret Hamilton. 2013. Assessment of programming: pedagogical foundations of exams. In Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '13). ACM, New York, NY, USA, 141–146. https://doi.org/10.1145/2462476.2465586

[597] Judy Sheard, Simon, Julian Dermoudy, Daryl D'Souza, Minjie Hu, and Dale Parsons. 2014. Benchmarking a set of exam questions for introductory programming. In Proceedings of the Sixteenth Australasian Computing Education Conference — Volume 148 (ACE '14). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 113–121. http://dl.acm.org/citation.cfm?id=2667490.2667504

[598] Judy Sheard, Simon, Margaret Hamilton, and Jan Lönnberg. 2009. Analysis of research into the teaching and learning of programming. In Proceedings of the Fifth International Workshop on Computing Education Research Workshop (ICER '09). ACM, New York, NY, USA, 93–104. https://doi.org/10.1145/1584322.1584334

[599] Lee Sheldon. 2011. The multiplayer classroom: designing coursework as a game. Cengage Learning.

[600] Duane F. Shell, Leen-Kiat Soh, Abraham E. Flanigan, and Markeya S. Peteranetz. 2016. Students' initial course motivation and their achievement and retention in college CS1 courses. In Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16). ACM, New York, NY, USA, 639–644. https://doi.org/10.1145/2839509.2844606

[601] Nianfeng Shi, Zhiyu Min, and Ping Zhang. 2017. Effects of visualizing roles of variables with animation and IDE in novice program construction. Telematics and Informatics 34, 5 (2017), 743–754. https://doi.org/10.1016/j.tele.2017.02.005

[602] Dermot Shinners-Kennedy and David J. Barnes. 2011. The novice programmer's "device to think with". In Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE '11). ACM, New York, NY, USA, 511–516. https://doi.org/10.1145/1953163.1953310

[603] Shuhaida Shuhidan, Margaret Hamilton, and Daryl D'Souza. 2009. A taxonomic study of novice programming summative assessment. In Proceedings of the Eleventh Australasian Conference on Computing Education — Volume 95 (ACE '09). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 147–156. http://dl.acm.org/citation.cfm?id=1862712.1862734

[604] T. Y. Sim. 2015. Exploration on the impact of online supported methods for novice programmers. In IEEE Conference on e-Learning, e-Management and e-Services (IC3e) (IC3e '15). IEEE, 158–162. https://doi.org/10.1109/IC3e.2015.7403505

[605] Simon. 2011. Assignment and sequence: why some students can't recognise a simple swap. In Proceedings of the 11th Koli Calling International Conference on Computing Education Research (Koli Calling '11). ACM, New York, NY, USA, 10–15. https://doi.org/10.1145/2094131.2094134

[606] Simon. 2013. Soloway's rainfall problem has become harder. In Learning and Teaching in Computing and Engineering (LaTiCE '13). IEEE, 130–135. https://doi.org/10.1109/LaTiCE.2013.44

[607] Simon, Angela Carbone, Michael de Raadt, Raymond Lister, Margaret Hamilton, and Judy Sheard. 2008. Classifying computing education papers: process and results. In Proceedings of the Fourth International Workshop on Computing Education Research (ICER '08). ACM, New York, NY, USA, 161–172. https://doi.org/10.1145/1404520.1404536

[608] Simon, Donald Chinn, Michael de Raadt, Anne Philpott, Judy Sheard, Mikko-Jussi Laakso, Daryl D'Souza, James Skene, Angela Carbone, Tony Clear, Raymond Lister, and Geoff Warburton. 2012. Introductory programming: examining the exams. In Proceedings of the Fourteenth Australasian Computing Education Conference — Volume 123 (ACE '12). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 61–70. http://dl.acm.org/citation.cfm?

id=2483716.2483724
[609] Simon, Alison Clear, Janet Carter, Gerry Cross, Atanas Radenski, Liviana Tudor, and Eno Tõnisson. 2015. What's in a Name?: International Interpretations of Computing Education Terminology. In Proceedings of the 2015 ITiCSE on Working Group Reports (ITICSE-WGR '15). ACM, New York, NY, USA, 173–186. https://doi.org/10.1145/2858796.2858803
[610] Simon, Daryl D'Souza, Judy Sheard, James Harland, Angela Carbone, and Mikko-Jussi Laakso. 2012. Can computing academics assess the difficulty of programming examination questions?. In Proceedings of the 12th Koli Calling International Conference on Computing Education Research (Koli Calling '12). ACM, New York, NY, USA, 160–163. https://doi.org/10.1145/2401796.2401822
[611] Simon, Mike Lopez, Ken Sutton, and Tony Clear. 2009. Surely we must learn to read before we learn to write!. In Proceedings of the Eleventh Australasian Conference on Computing Education — Volume 95 (ACE '09). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 165–170. http://dl.acm.org/citation.cfm?id=1862712.1862736
[612] Simon, Judy Sheard, Daryl D'Souza, Peter Klemperer, Leo Porter, Juha Sorva, Martijn Stegeman, and Daniel Zingaro. 2016. Benchmarking introductory programming exams: how and why. In Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '16). ACM, New York, NY, USA, 154–159. https://doi.org/10.1145/2899415.2899473
[613] Simon, Judy Sheard, Daryl D'Souza, Peter Klemperer, Leo Porter, Juha Sorva, Martijn Stegeman, and Daniel Zingaro. 2016. Benchmarking introductory programming exams: some preliminary results. In Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16). ACM, New York, NY, USA, 103–111. https://doi.org/10.1145/2960310.2960337
[614] Simon, Judy Sheard, Michael Morgan, Andrew Petersen, Amber Settle, Jane Sinclair, Gerry Cross, and Charles Riedesel. 2016. Negotiating the maze of academic integroty in computing education. In ITiCSE Working Group Reports (ITiCSE-WGR '16). ACM, New York, NY, USA, 57–80. https://doi.org/10.1145/3024906.3024910
[615] Beth Simon, Sue Fitzgerald, Renée McCauley, Susan Haller, John Hamer, Brian Hanks, Michael T. Helmick, Jan Erik Moström, Judy Sheard, and Lynda Thomas. 2007. Debugging assistance for novices: a video repository. In Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR '07). ACM, New York, NY, USA, 137–151. https://doi.org/10.1145/1345443.1345437
[616] Beth Simon, Brian Hanks, Laurie Murphy, Sue Fitzgerald, Renée McCauley, Lynda Thomas, and Carol Zander. 2008. Saying isn't necessarily believing: influencing self-theories in computing. In Proceedings of the Fourth International Workshop on Computing Education Research (ICER '08). ACM, New York, NY, USA, 173–184. https://doi.org/10.1145/1404520.1404537
[617] Beth Simon, Päivi Kinnunen, Leo Porter, and Dov Zazkis. 2010. Experience report: CS1 for majors with media computation. In Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '10). ACM, New York, NY, USA, 214–218. https://doi.org/10.1145/1822090.1822151
[618] Guttorm Sindre, Steinar Line, and Ottar V. Valvåg. 2003. Positive experiences with an open project assignment in an introductory programming course. In Proceedings of the 25th International Conference on Software Engineering (ICSE '03). IEEE Computer Society, Washington, DC, USA, 608–613. http://dl.acm.org/citation.cfm?id=776816.776900
[619] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13). ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/2491956.2462195
[620] Teemu Sirkiä. 2016. Jsvee & Kelmu: creating and tailoring program animations for computing education. In IEEE Working Conference on Software Visualization (VISSOFT '16). IEEE, 36–45. https://doi.org/10.1109/VISSOFT.2016.24
[621] Ben Skudder and Andrew Luxton-Reilly. 2014. Worked examples in computer science. In Proceedings of the Sixteenth Australasian Computing Education Conference — Volume 148 (ACE '14). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 59–64. http://dl.acm.org/citation.cfm?id=2667490.2667497
[622] Robert H. Sloan, Cynthia Taylor, and Richard Warner. 2017. Initial experiences with a CS + Law introduction to computer science (CS 1). In Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17). ACM, New York, NY, USA, 40–45. https://doi.org/10.1145/3059009.3059029
[623] Leen-Kiat Soh. 2006. Incorporating an intelligent tutoring system into CS1. In Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '06). ACM, New York, NY, USA, 486–490. https://doi.org/10.1145/1121341.1121494
[624] Elliot Soloway. 1986. Learning to program = learning to construct mechanisms and explanations. *Commun. ACM* 29, 9 (1986), 850–858.
[625] Hosung Song. 2010. Online shared editing for introductory programming courses. In Proceedings of 2nd International Conference on Computer Supported Education — Volume 1 (CSEDU '10), Vol. 1. INSTICC, SciTePress, 489–492. https://doi.org/10.5220/0002860904890492
[626] Raja Sooriamurthi. 2009. Introducing abstraction and decomposition to novice programmers. In Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '09). ACM, New York, NY, USA, 196–200. https://doi.org/10.1145/1562877.1562939
[627] Juha Sorva. 2007. Students' understandings of storing objects. In Proceedings of the Seventh Baltic Sea Conference on Computing Education Research - Volume 88 (Koli Calling '07). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 127–135. http://dl.acm.org/citation.cfm?id=2449323.2449337
[628] Juha Sorva. 2008. The same but different students' understandings of primitive and object variables. In Proceedings of the 8th Koli Calling International Conference on Computing Education Research (Koli Calling '08). ACM, New York, NY, USA, 5–15. https://doi.org/10.1145/1595356.1595360
[629] Juha Sorva. 2010. Reflections on threshold concepts in computer programming and beyond. In Proceedings of the 10th Koli Calling International Conference on Computing Education Research (Koli Calling '10). ACM, New York, NY, USA, 21–30. https://doi.org/10.1145/1930464.1930467
[630] Juha Sorva. 2013. Notional machines and introductory programming education. Trans. Comput. Educ. 13, 2, Article 8 (2013), 8:1–8:31 pages. https://doi.org/10.1145/2483710.2483713
[631] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A review of generic program visualization systems for introductory programming education. Trans. Comput. Educ. 13, 4, Article 15 (2013), 15:1–15:64 pages. https://doi.org/10.1145/2490822
[632] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A review of generic program visualization systems for introductory programming education. Trans. Comput. Educ. 13, 4, Article 15 (2013), 15:1–15:64 pages. https://doi.org/10.1145/2490822
[633] Juha Sorva, Jan Lönnberg, and Lauri Malmi. 2013. Students' ways of experiencing visual program simulation. Computer Science Education 23, 3 (2013), 207–238. https://doi.org/10.1080/08993408.2013.807962
[634] Juha Sorva and Otto Seppälä. 2014. Research-based design of the first weeks of CS1. In Proceedings of the 14th Koli Calling International Conference on Computing Education Research (Koli Calling '14). ACM, New York, NY, USA, 71–80. https://doi.org/10.1145/2674683.2674690
[635] Juha Sorva and Teemu Sirkiä. 2010. UUhistle: a software tool for visual program simulation. In Proceedings of the 10th Koli Calling International Conference on Computing Education Research (Koli Calling '10). ACM, New York, NY, USA, 49–54. https://doi.org/10.1145/1930464.1930471
[636] Draylson Micael de Souza, Seiji Isotani, and Ellen Francine Barbosa. 2015. Teaching novice programmers using ProgTest. International Journal of Knowledge and Learning 10, 1 (2015), 60–77. https://doi.org/10.1504/IJKL.2015.071054
[637] Jaime Spacco, Paul Denny, Brad Richards, David Babcock, David Hovemeyer, James Moscola, and Robert Duvall. 2015. Analyzing student work patterns using programming exercise data. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15). ACM, New York, NY, USA, 18–23. https://doi.org/10.1145/2676723.2677297
[638] Michael Sperber and Marcus Crestani. 2012. Form over function: teaching beginners how to construct programs. In Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming (Scheme '12). ACM, New York, NY, USA, 81–89. https://doi.org/10.1145/2661103.2661113
[639] Andreas Stefik and Stefan Hanenberg. 2014. The programming language wars: questions and responsibilities for the programming language community. In Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014). ACM, New York, NY, USA, 283–299. https://doi.org/10.1145/2661136.2661156
[640] Andreas Stefik and Susanna Siebert. 2013. An empirical investigation into programming language syntax. Trans. Comput. Educ. 13, 4, Article 19 (2013), 19:1–19:40 pages. https://doi.org/10.1145/2534973
[641] Martijn Stegeman, Erik Barendsen, and Sjaak Smetsers. 2016. Designing a rubric for feedback on code quality in programming courses. In Proceedings of the 16th Koli Calling International Conference on Computing Education Research (Koli Calling '16). ACM, New York, NY, USA, 160–164. https://doi.org/10.1145/2999541.2999555
[642] Daniel E. Stevenson and Paul J. Wagner. 2006. Developing real-world programming assignments for CS1. In Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITICSE '06). ACM, New York, NY, USA, 158–162. https://doi.org/10.1145/1140124.1140167
[643] Margaret-Anne Storey, Daniela Damian, Jeff Michaud, Del Myers, Marcellus Mindel, Daniel German, Mary Sanseverino, and Elizabeth Hargreaves. 2003. Improving the usability of Eclipse for novice programmers. In Proceedings of the 2003 OOPSLA Workshop on Eclipse Technology eXchange (Eclipse '03). ACM, New York, NY, USA, 35–39. https://doi.org/10.1145/965660.965668
[644] Leigh Ann Sudol-DeLyser, Mark Stehlik, and Sharon Carver. 2012. Code comprehension problems as learning events. In Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '12). ACM, New York, NY, USA, 81–86. https://doi.org/10.1145/2325296.

2325319

[645] Hui Sun, Bofang Li, and Min Jiao. 2014. Yoj: an online judge system designed for programming courses. In 2014 9th International Conference on Computer Science & Education. IEEE, 812–816. https://doi.org/10.1109/ICCSE.2014.6926575

[646] Kelvin Sung, Rob Nash, and Jason Pace. 2016. Building casual game SDKs for teaching CS1/2: a case study. J. Comput. Sci. Coll. 32, 1 (2016), 129–143. http://dl.acm.org/citation.cfm?id=3007225.3007253

[647] Ryo Suzuki, Gustavo Soares, Elena Glassman, Andrew Head, Loris D'Antoni, and Björn Hartmann. 2017. Exploring the design space of automatically synthesized hints for introductory programming assignments. In Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems (CHI EA '17). ACM, New York, NY, USA, 2951–2958. https://doi.org/10.1145/3027063.3053187

[648] John Sweller. 1994. Cognitive load theory, learning difficulty, and instructional design. Learning and instruction 4, 4 (1994), 295–312.

[649] Emily S. Tabanao, Ma. Mercedes T. Rodrigo, and Matthew C. Jadud. 2011. Predicting at-risk novice Java programmers through the analysis of online protocols. In Proceedings of the Seventh International Workshop on Computing Education Research (ICER '11). ACM, New York, NY, USA, 85–92. https://doi.org/10.1145/2016911.2016930

[650] Pallavi Tadepalli and H. Conrad Cunningham. 2004. JavaCHIME: Java class hierarchy inspector and method executer. In Proceedings of the 42nd Annual Southeast Regional Conference (ACM-SE 42). ACM, New York, NY, USA, 152–157. https://doi.org/10.1145/986537.986572

[651] Anya Tafliovich, Jennifer Campbell, and Andrew Petersen. 2013. A student perspective on prior experience in CS1. In Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13). ACM, New York, NY, USA, 239–244. https://doi.org/10.1145/2445196.2445270

[652] Yasuhiro Takemura, Hideo Nagumo, Kuo-Li Huang, and Hidekuni Tsukamoto. [n. d.]. Assessing the learners' motivation in the e-learning environments for programming education. In International Conference on Web-Based Learning (ICWL '07), Howard Leung, Frederick Li, Rynson Lau, and Qing Li (Eds.). Springer, Berlin, Heidelberg, 355–366. https://doi.org/10.1007/978-3-540-78139-4_32

[653] Terry Tang, Scott Rixner, and Joe Warren. 2014. An environment for learning interactive programming. In Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14). ACM, New York, NY, USA, 671–676. https://doi.org/10.1145/2538862.2538908

[654] Donna Teague. 2009. A people-first approach to programming. In Proceedings of the Eleventh Australasian Conference on Computing Education — Volume 95 (ACE '09). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 171–180. http://dl.acm.org/citation.cfm?id=1862712.1862737

[655] Donna Teague, Malcolm Corney, Alireza Ahadi, and Raymond Lister. 2013. A qualitative think aloud study of the early neo-piagetian stages of reasoning in novice programmers. In Proceedings of the Fifteenth Australasian Computing Education Conference — Volume 136 (ACE '13). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 87–95. http://dl.acm.org/citation.cfm?id=2667199.2667209

[656] Donna Teague and Raymond Lister. 2014. Longitudinal think aloud study of a novice programmer. In Proceedings of the Sixteenth Australasian Computing Education Conference — Volume 148 (ACE '14). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 41–50. http://dl.acm.org/citation.cfm?id=2667490.2667495

[657] Donna Teague and Raymond Lister. 2014. Manifestations of preoperational reasoning on similar programming tasks. In Proceedings of the Sixteenth Australasian Computing Education Conference — Volume 148 (ACE '14). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 65–74. http://dl.acm.org/citation.cfm?id=2667490.2667498

[658] Donna Teague and Raymond Lister. 2014. Programming: reading, writing and reversing. In Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education (ITiCSE '14). ACM, New York, NY, USA, 285–290. https://doi.org/10.1145/2591708.2591712

[659] D. Teague and P. Roe. 2009. Learning to program : from pear-shaped to pairs. In Proceedings of the 1st International Conference on Computer Supported Education (CSEDU '09), Vol. 2. SciTePress, 151–158.

[660] Lynda Thomas, Mark Ratcliffe, and Ann Robertson. 2003. Code warriors and code-a-phobes: a study in attitude and pair programming. In Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '03). ACM, New York, NY, USA, 363–367. https://doi.org/10.1145/611892.612007

[661] Errol Thompson, Andrew Luxton-Reilly, Jacqueline L. Whalley, Minjie Hu, and Phil Robbins. 2008. Bloom's taxonomy for CS assessment. In Proceedings of the Tenth Conference on Australasian Computing Education — Volume 78 (ACE '08). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 155–161. http://dl.acm.org/citation.cfm?id=1379249.1379265

[662] Neena Thota. 2014. Programming course design: phenomenographic approach to learning and teaching. In International Conference on Learning and Teaching in Computing and Engineering (LaTiCE '14). IEEE, 125–132. https://doi.org/10.1109/LaTiCE.2014.30

[663] Veronika Thurner and Axel Böttcher. 2015. An "objects first, tests second" approach for software engineering education. In IEEE Frontiers in Education Conference (FIE '15). IEEE, 1–5. https://doi.org/10.1109/FIE.2015.7344027

[664] Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, Manuel Fahndrich, Judith Bishop, Arjmand Samuel, and Tao Xie. 2012. The future of teaching programming is on mobile devices. In Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '12). ACM, New York, NY, USA, 156–161. https://doi.org/10.1145/2325296.2325336

[665] Lisa Torrey. 2011. Student interest and choice in programming assignments. J. Comput. Sci. Coll. 26, 6 (2011), 110–116. http://dl.acm.org/citation.cfm?id=1968521.1968545

[666] Gloria Childress Townsend. 2009. Using a groupware system in CS1 to engage introverted students. In Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '09). ACM, New York, NY, USA, 16–20. https://doi.org/10.1145/1562877.1562889

[667] V Javier Traver. 2010. On compiler error messages: what they say and what they mean. Advances in Human-Computer Interaction 2010 (2010), 26. https://doi.org/10.1155/2010/602570

[668] Nghi Truong, Peter Bancroft, and Paul Roe. 2003. A web based environment for learning to program. In Proceedings of the 26th Australasian Computer Science Conference — Volume 16 (ACSC '03). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 255–264. http://dl.acm.org/citation.cfm?id=783106.783135

[669] Hidekuni Tsukamoto, Hideo Nagumo, Yasuhiro Takemura, and Kenichi Matsumoto. 2009. Comparative analysis of 2D games and artwork as the motivation to learn programming. In IEEE Frontiers in Education Conference (FIE '09). IEEE, 1–6. https://doi.org/10.1109/FIE.2009.5350451

[670] Georgi Tuparov, Daniela Tuparova, and Vladimir Jordanov. 2014. Teaching sorting and searching algorithms through simulation-based learning objects in an introductory programming course. Procedia — Social and Behavioral Sciences 116 (2014), 2962–2966. https://doi.org/10.1016/j.sbspro.2014.01.688

[671] Kota Uchida and Katsuhiko Gondow. 2016. C-helper: C latent-error static/heuristic checker for novice programmers. In Proceedings of the 8th International Conference on Computer Supported Education (CSEDU '16), Vol. 1. SciTePress, 321–329.

[672] Timothy Urness. 2017. A hybrid open/closed lab for CS 1. In Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17). ACM, New York, NY, USA, 46–51. https://doi.org/10.1145/3059009.3059014

[673] Ian Utting, Allison Elliott Tew, Mike McCracken, Lynda Thomas, Dennis Bouvier, Roger Frye, James Paterson, Michael Caspersen, Yifat Ben-David Kolikant, Juha Sorva, and Tadeusz Wilusz. 2013. A fresh look at novice programmers' performance and their teachers' expectations. In Proceedings of the ITiCSE Working Group Reports Conference on Innovation and Technology in Computer Science Education-working Group Reports (ITiCSE -WGR '13). ACM, New York, NY, USA, 15–32. https://doi.org/10.1145/2543882.2543884

[674] Evgenia Vagianou. 2006. Program working storage: a beginner's model. In Proceedings of the 6th Baltic Sea Conference on Computing Education Research (Koli Calling '06). ACM, New York, NY, USA, 69–76. https://doi.org/10.1145/1315803.1315816

[675] Adilson Vahldick, António José Mendes, and Maria José Marcelino. 2014. A review of games designed to improve introductory computer programming competencies. In IEEE Frontiers in Education Conference (FIE '14). IEEE, 1–7. https://doi.org/10.1109/FIE.2014.7044114

[676] David W. Valentine. 2004. CS Educational Research: A Meta-analysis of SIGCSE Technical Symposium Proceedings. In Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '04). ACM, New York, NY, USA, 255–259. https://doi.org/10.1145/971300.971391

[677] Tammy VanDeGrift. 2015. Supporting creativity and user interaction in CS 1 homework assignments. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15). ACM, New York, NY, USA, 54–59. https://doi.org/10.1145/2676723.2677250

[678] Elena Verdú, Luisa M Regueras, María J Verdú, José P Leal, Juan P de Castro, and Ricardo Queirós. 2012. A distributed system for learning programming on-line. Computers & Education 58, 1 (2012), 1–10. https://doi.org/10.1016/j.compedu.2011.08.015

[679] Arto Vihavainen, Jonne Airaksinen, and Christopher Watson. 2014. A systematic review of approaches for teaching introductory programming and their influence on success. In Proceedings of the Tenth Annual Conference on International Computing Education Research (ICER '14). ACM, New York, NY, USA, 19–26. https://doi.org/10.1145/2632320.2632349

[680] Arto Vihavainen, Juha Helminen, and Petri Ihantola. 2014. How novices tackle their first lines of code in an IDE: analysis of programming session traces. In Proceedings of the 14th Koli Calling International Conference on Computing Education Research (Koli Calling '14). ACM, New York, NY, USA, 109–116. https://doi.org/10.1145/2674683.2674692

[681] Arto Vihavainen, Craig S. Miller, and Amber Settle. 2015. Benefits of self-explanation in introductory programming. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15). ACM, New York, NY, USA, 284–289. https://doi.org/10.1145/2676723.2677260

[682] Arto Vihavainen, Matti Paksula, and Matti Luukkainen. 2011. Extreme apprenticeship method in teaching programming for beginners. In Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE '11). ACM, New York, NY, USA, 93–98. https://doi.org/10.1145/1953163.1953196

[683] Arto Vihavainen, Thomas Vikberg, Matti Luukkainen, and Jaakko Kurhila. 2013. Massive increase in eager TAs: experiences from extreme apprenticeship-based CS1. In Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '13). ACM, New York, NY, USA, 123–128. https://doi.org/10.1145/2462476.2462508

[684] Tamar Vilner, Ela Zur, and Judith Gal-Ezer. 2007. Fundamental concepts of CS1: procedural vs. object oriented paradigm — a case study. In Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '07). ACM, New York, NY, USA, 171–175. https://doi.org/10.1145/1268784.1268835

[685] Tamar Vilner, Ela Zur, and Ronit Sagi. 2012. Integrating video components in CS1. In Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12). ACM, New York, NY, USA, 123–128. https://doi.org/10.1145/2157136.2157176

[686] Giovanni Vincenti, James Braman, and J. Scott Hilberg. 2013. Teaching introductory programming through reusable learning objects: a pilot study. J. Comput. Sci. Coll. 28, 3 (2013), 38–45. http://dl.acm.org/citation.cfm?id=2400161.2400172

[687] Milena Vujošević-Janičić, Mladen Nikolić, Dušan Tošić, and Viktor Kuncak. 2013. Software verification and graph similarity for automated evaluation of students' assignments. Information and Software Technology 55, 6 (2013), 1004–1016. https://doi.org/10.1016/j.infsof.2012.12.005

[688] Hong Wang. 2010. Teaching CS1 with Python GUI game programming. In AIP Conference Proceedings, Vol. 1247. AIP, 253–260. https://doi.org/10.1063/1.3460234

[689] K. Wang. 2015. Enhancing the teaching of CS 1 by programming mobile apps in MIT App Inventor. In ASEE Annual Conference & Exposition (ASEE '15). 26.671.1 – 26.671.9.

[690] Tiantian Wang, Xiaohong Su, Peijun Ma, Yuying Wang, and Kuanquan Wang. 2011. Ability-training-oriented automated assessment in introductory programming course. Computers & Education 56, 1 (2011), 220–226. https://doi.org/10.1016/j.compedu.2010.08.003

[691] Kera Z. B. Watkins and Maurice J. Watkins. 2009. Towards minimizing pair incompatibilities to help retain under-represented groups in beginning programming courses using pair programming. J. Comput. Sci. Coll. 25, 2 (2009), 221–227. http://dl.acm.org/citation.cfm?id=1629036.1629071

[692] Christopher Watson and Frederick W.B. Li. 2014. Failure rates in introductory programming revisited. In Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education (ITiCSE '14). ACM, New York, NY, USA, 39–44. https://doi.org/10.1145/2591708.2591749

[693] Christopher Watson, Frederick WB Li, and Jamie L Godwin. 2013. Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In IEEE 13th International Conference on Advanced Learning Technologies (ICALT '13). IEEE, 319–323. https://doi.org/10.1109/ICALT.2013.99

[694] Christopher Watson, Frederick W.B. Li, and Jamie L. Godwin. 2014. No tests required: comparing traditional and dynamic predictors of programming success. In Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14). ACM, New York, NY, USA, 469–474. https://doi.org/10.1145/2538862.2538930

[695] Christopher Watson, Frederick W. B. Li, and Jamie L. Godwin. 2012. BlueFix: using crowd-sourced feedback to support programming students in error diagnosis and repair. In International Conference on Web-Based Learning (ICWL '12), Elvira Popescu, Qing Li, Ralf Klamma, Howard Leung, and Marcus Specht (Eds.). Springer, Berlin, Heidelberg, 228–239. https://doi.org/10.1007/978-3-642-33642-3_25

[696] Christopher Watson, Frederick W. B. Li, and Rynson W. H. Lau. 2011. Learning programming languages through corrective feedback and concept visualisation. In International Conference on Web-Based Learning, Howard Leung, Elvira Popescu, Yiwei Cao, Rynson W. H. Lau, and Wolfgang Nejdl (Eds.). Springer, Berlin, Heidelberg, 11–20. https://doi.org/10.1007/978-3-642-25813-8_2

[697] Carol A Wellington, Thomas H Briggs, and C Dudley Girard. 2007. Experiences using automated 4ests and 4est driven development in computer 9cience i. In Agile 2007 (AGILE '07). IEEE, 106–112. https://doi.org/10.1109/AGILE.2007.27

[698] Briana Lowe Wellman, James Davis, and Monica Anderson. 2009. Alice and robotics in introductory CS courses. In The Fifth Richard Tapia Celebration of Diversity in Computing Conference: Intellect, Initiatives, Insight, and Innovations (TAPIA '09). ACM, New York, NY, USA, 98–102. https://doi.org/10.1145/1565799.1565822

[699] Dinesha Weragama and Jim Reye. 2013. The PHP intelligent tutoring system. In Artificial Intelligence in Education, H. Chad Lane, Kalina Yacef, Jack Mostow, and Philip Pavlik (Eds.). Springer, Berlin, Heidelberg, 583–586. https://doi.org/10.1007/978-3-642-39112-5_64

[700] Lena Kallin Westin and M Nordstrom. 2004. Teaching OO concepts—a new approach. In IEEE Frontiers in Education Conference (FIE '04). IEEE, F3C–6–11. https://doi.org/10.1109/FIE.2004.1408620

[701] Jacqueline Whalley, Tony Clear, Phil Robbins, and Errol Thompson. 2011. Salient elements in novice solutions to code writing problems. In Proceedings of the Thirteenth Australasian Computing Education Conference — Volume 114 (ACE '11). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 37–46. http://dl.acm.org/citation.cfm?id=2459936.2459941

[702] Jacqueline Whalley and Nadia Kasto. 2013. Revisiting models of human conceptualisation in the context of a programming examination. In Proceedings of the Fifteenth Australasian Computing Education Conference — Volume 136 (ACE '13). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 67–76. http://dl.acm.org/citation.cfm?id=2667199.2667207

[703] Jacqueline Whalley and Nadia Kasto. 2014. How difficult are novice code writing tasks?: A software metrics approach. In Proceedings of the Sixteenth Australasian Computing Education Conference — Volume 148 (ACE '14). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 105–112. http://dl.acm.org/citation.cfm?id=2667490.2667503

[704] Jacqueline L. Whalley and Anne Philpott. 2011. A unit testing approach to building novice programmers' skills and confidence. In Proceedings of the Thirteenth Australasian Computing Education Conference — Volume 114 (ACE '11). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 113–118. http://dl.acm.org/citation.cfm?id=2459936.2459950

[705] S. J. Whittall, W. A. C. Prashandi, G. L. S. Himasha, D. I. De Silva, and T. K. Suriyawansa. 2017. CodeMage: educational programming environment for beginners. In 9th International Conference on Knowledge and Smart Technology (KST '17). IEEE, 311–316. https://doi.org/10.1109/KST.2017.7886101

[706] David .M. Whittinghill, David B. Nelson, K. Andrew R. Richards, and Charles A. Calahan. 2014. Improving the affective element in introductory programming coursework for the "non programmer" student. In Proceedings of ASEE Annual Conference and Exposition. ASEE, 11.

[707] Richard Wicentowski and Tia Newhall. 2005. Using image processing projects to teach CS1 topics. In Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '05). ACM, New York, NY, USA, 287–291. https://doi.org/10.1145/1047344.1047445

[708] E Wiebe, Laurie Williams, Julie Petlick, Nachiappan Nagappan, Suzanne Balik, Carol Miller, and Miriam Ferzli. 2003. Pair programming in introductory programming labs. In Proceedings of ASEE Annual Conference. ASEE, 3503–3514.

[709] Susan Wiedenbeck, Xiaoning Sun, and Thippaya Chintakovid. 2007. Antecedents to end users' success in learning to program in an introductory programming course. In IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '07). IEEE, 69–72. https://doi.org/10.1109/VLHCC.2007.8

[710] Joseph B. Wiggins, Joseph F. Grafsgaard, Kristy Elizabeth Boyer, Eric N. Wiebe, and James C. Lester. 2016. Do you think you can? the influence of student self-efficacy on the effectiveness of tutorial dialogue for computer science. International Journal of Artificial Intelligence in Education 27, 1 (2016), 130–153. https://doi.org/10.1007/s40593-015-0091-7

[711] Chris Wilcox. 2015. The role of automation in undergraduate computer science education. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15). ACM, New York, NY, USA, 90–95. https://doi.org/10.1145/2676723.2677226

[712] Laurie Williams, Charlie McDowell, Nachiappan Nagappan, Julian Fernald, and Linda Werner. 2003. Building pair programming knowledge through a family of experiments. In Proceedings of International Symposium on Empirical Software Engineering (ISESE '03). IEEE, 143–152. https://doi.org/10.1109/ISESE.2003.1237973

[713] Michael Wirth. 2009. Ecological footprints as case studies in programming. In IEEE Toronto International Conference Science and Technology for Humanity (TIC-STH '09). IEEE, 188–193. https://doi.org/10.1109/TIC-STH.2009.5444510

[714] Michael Wirth and Judi McCuaig. 2014. Making programs with the Raspberry PI. In Proceedings of the Western Canadian Conference on Computing Education (WCCCE '14). ACM, New York, NY, USA, Article 17, 17:1–17:5 pages. https://doi.org/10.1145/2597959.2597970

[715] Denise Woit and David Mason. 2003. Effectiveness of online assessment. In Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '03). ACM, New York, NY, USA, 137–141. https://doi.org/10.1145/611892.611952

[716] Krissi Wood, Dale Parsons, Joy Gasson, and Patricia Haden. 2013. It's never too early: pair programming in CS1. In Proceedings of the Fifteenth Australasian Computing Education Conference — Volume 136 (ACE '13). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 13–21. http://dl.acm.org/citation.cfm?id=2667199.2667201

[717] Dana Wortman and Penny Rheingans. 2007. Visualizing trends in student performance across computer science courses. In Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '07).

ACM, New York, NY, USA, 430–434. https://doi.org/10.1145/1227310.1227458

[718] Stelios Xinogalos, Maya Satratzemi, and Vassilios Dagdilelis. 2006. An introduction to object-oriented programming with a didactic microworld: objectKarel. Computers & Education 47, 2 (2006), 148–171. https://doi.org/10.1016/j.compedu.2004.09.005

[719] Jitendra Yasaswi, Sri Kailash, Anil Chilupuri, Suresh Purini, and C. V. Jawahar. 2017. Unsupervised learning based approach for plagiarism detection in programming assignments. In Proceedings of the 10th Innovations in Software Engineering Conference (ISEC '17). ACM, New York, NY, USA, 117–121. https://doi.org/10.1145/3021460.3021473

[720] Jane Yau and Mike Joy. 2007. Architecture of a context-aware and adaptive learning schedule for learning Java. In Seventh IEEE International Conference on Advanced Learning Technologies (ICALT '07). IEEE, 252–256. https://doi.org/10.1109/ICALT.2007.72

[721] Leelakrishna Yenigalla, Vinayak Sinha, Bonita Sharif, and Martha Crosby. 2016. How novices read source code in introductory courses on programming: an eye-tracking experiment. In Foundations of Augmented Cognition: Neuroergonomics and Operational Neuroscience, Dylan D. Schmorrow and Cali M. Fidopiastis (Eds.). Springer International Publishing, Cham, 120–131. https://doi.org/10.1007/978-3-319-39952-2_13

[722] Jooyong Yi, Umair Z. Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A feasibility study of using automated program repair for introductory programming assignments. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '17). ACM, New York, NY, USA, 740–751. https://doi.org/10.1145/3106237.3106262

[723] Jungsoon P Yoo, Suk Jai Seo, and Sung K Yoo. 2004. Designing an adaptive tutor for CS-I laboratory. In Proceedings of the International Conference on Internet Computing (IC'04), Vol. 1. 459–464.

[724] Nurliana Yusri, Ain Zulika, Sharifah Mashita Syed-Mohamad, and Nur'Aini Abdul Rashid. 2014. Tools for teaching and learning programming: a review and proposed tool. Frontiers in Artificial Intelligence and Applications 265 (2014), 859–872. https://doi.org/10.3233/978-1-61499-434-3-859

[725] Nick Z. Zacharis. 2011. Measuring the effects of virtual pair programming in an introductory programming Java course. IEEE Transactions on Education 54 (2011), 168–170. Issue 1. https://doi.org/10.1109/TE.2010.2048328

[726] Carol Zander, Lynda Thomas, Beth Simon, Laurie Murphy, Renée McCauley, Brian Hanks, and Sue Fitzgerald. 2009. Learning styles: novices decide. In Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '09). ACM, New York, NY, USA, 223–227. https://doi.org/10.1145/1562877.1562948

[727] Matej Zapušek, Martin Možina, Ivan Bratko, Jože Rugelj, and Matej Guid. 2014. Designing an interactive teaching tool with ABML knowledge refinement loop. In Intelligent Tutoring Systems, Stefan Trausan-Matu, Kristy Elizabeth Boyer, Martha Crosby, and Kitty Panourgia (Eds.). Springer International Publishing, Cham, 575–582. https://doi.org/10.1007/978-3-319-07221-0_73

[728] Ramon Zatarain-Cabada, M. L. Barrón-Estrada, J. Moisés Osorio-Velásquez, L. Zepeda-Sánchez, and Carlos A. Reyes-García. 2008. L2Code: an author environment for hybrid and personalized programming learning. In Hybrid Learning and Education, Joseph Fong, Reggie Kwan, and Fu Lee Wang (Eds.). Springer, Berlin, Heidelberg, 338–347. https://doi.org/10.1007/978-3-540-85170-7_30

[729] Daniela Zehetmeier, Anne Brüggemann-Klein, Axel Böttcher, and Veronika Thurner. 2016. A concept for interventions that address typical error classes in programming education. In IEEE Global Engineering Education Conference (EDUCON '16). IEEE, 545–554. https://doi.org/10.1109/EDUCON.2016.7474605

[730] Kurtis Zimmerman and Chandan R Rupakheti. 2015. An automated framework for recommending program elements to novices (n). In Proceedings of 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15). IEEE, 283–288. https://doi.org/10.1109/ASE.2015.54

[731] Daniel Zingaro. 2014. Peer instruction contributes to self-efficacy in CS1. In Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14). ACM, New York, NY, USA, 373–378. https://doi.org/10.1145/2538862.2538878

[732] Daniel Zingaro, Yuliya Cherenkova, Olessia Karpova, and Andrew Petersen. 2013. Facilitating code-writing in PI classes. In Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13). ACM, New York, NY, USA, 585–590. https://doi.org/10.1145/2445196.2445369

[733] Daniel Zingaro, Andrew Petersen, and Michelle Craig. 2012. Stepping up to integrative questions on CS1 exams. In Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12). ACM, New York, NY, USA, 253–258. https://doi.org/10.1145/2157136.2157215

[734] Daniel Zingaro and Leo Porter. 2016. Impact of student achievement goals on CS1 outcomes. In Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16). ACM, New York, NY, USA, 279–296. https://doi.org/10.1145/2839509.2844553

[735] Imran A. Zualkernan. 2007. Using Soloman-Felder learning style index to evaluate pedagogical resources for introductory programming classes. In Proceedings of the 29th International Conference on Software Engineering (ICSE '07). IEEE Computer Society, Washington, DC, USA, 723–726. https://doi.org/10.1109/ICSE.2007.96