Raspberry Pi Foundation

Teaching programming in schools: A review of approaches and strategies

Raspberry Pi Foundation
Jane Waite and Sue Sentance

November 2021

Teaching programming in schools: A review of approaches and strategies

Raspberry Pi Foundation
Jane Waite and Sue Sentance
November 2021

Published in November 2021
by the Raspberry Pi Foundation

www.raspberrypi.org

ISSN 2514-586X (19)

Table of contents

Introduction

Globally, computer science (CS) education has been generating increasing interest as a school subject in the last few years. Programming is a key part of computer science and computing; it is a skill that cannot sit separately from the theoretical components of computing. Rather, programming is the application of concepts that often are hard to understand until they are put into practice. If your program does not do as you intended, then it is likely you have not understood a computer science concept. This leads us to the conclusion that 'practice' is not simply skill reinforcement, it is the route to understanding.
It is in this context, and with the knowledge that programming has been found to be "difficult" by many learners in undergraduate education, that we have drawn together some of the research on how to teach programming, which we refer to as programming pedagogy. We must acknowledge that we still lack evidence in the field of teaching and learning for primary and secondary school students, as programming research is often conducted with older learners in university or with small numbers of students in particular contexts where generalisation cannot be guaranteed. Therefore, we have chosen approaches with emerging evidence and that merit further investigation.
To teach any subject requires good teaching skills, knowledge about the subject being taught, and specific knowledge — known as pedagogical content knowledge — that a teacher gains about how to teach a particular topic, to their students, in the learning context, at a given moment in time. When reading our report, you might wish to think carefully about which combination of instructional approaches is likely to ensure that learning is accessible for all your students.
Research into teaching computer science at university level has mirrored its delivery: there is a long history of investigations into both content and associated pedagogy. For research related to younger learners, focus in the 1980s was on the teaching and learning of IT skills, on work related to Logo (a programming language aimed at younger pupils (Papert, 1980)), and on how teachers might leverage new technology-based teaching aids. As outlined by the 2017 Royal Society report *After the reboot*, research into how computer science can be taught in primary and secondary classrooms is as yet very limited, having only recently started to emerge (The Royal Society, 2017). We need continued research to consider many aspects of teaching computing, including:

- Why programming is difficult
- How to teach it effectively
- How to motivate and encourage students
- What contexts and classroom tools work best
- What roles vocabulary and tools play
- What computational thinking is and how can it be effectively embedded

Some consensus is emerging regarding research questions, but computing education is a new field, and much of the underlying research is in its early stages; the reliability of current evidence may perhaps be restricted due to the limitations of the studies from which it was gained. These studies have often been conducted with short time frames and small numbers of learners, in informal rather than classroom settings, and without robust means for pre- and post-assessment of interventions. Nevertheless, the evidence we have gives us a starting point.

Teaching programming: Approaches and techniques

Computer programming is now part of the curriculum in schools in England and many other countries. Although it is not necessarily the primary focus of the curriculum, it is the area of computing that many teachers find most difficult to teach, and also the one into which the most computing education research has been conducted.

1. Classroom strategies

1. Classroom strategies

In this section, we consider some well-researched classroom strategies that teachers can use to teach programming in schools. These include:

- Pair programming
- Peer instruction
- Live coding
- PRIMM
- Worked examples
- Subgoal labelling
- Reading and tracing code
- Pattern-oriented instruction
- Targeted tasks, e.g. debugging, sabotage, annotation, fill in the gaps, Parson's Problems

1.1. Pair programming

Used in industry and education, pair programming is a collaborative approach where two people simultaneously work on a single software development project. Swapping roles regularly, one person (the driver) has control of the mouse and keyboard, and the other (the navigator) continuously collaborates by reviewing the code written and keeping track of work done against the design (McDowell, Werner, Bullock, & Fernald, 2006).

Few studies have examined pair programming in primary and secondary education. Most research has been done with university students, producing evidence that pair programming leads to increased learning and improved code quality, with the caveat that careful implementation is needed to ensure success (Hanks, Fitzgerald, McCauley, Murphy, & Zander, 2011; Salleh, Mendes, & Grundy, 2011; Umapathy & Ritzhaupt, 2017).

One ten-year, school-based research programme in the US concluded that pair work had advantages over solo programming for building programming knowledge and computational thinking (Denner, Werner, Campe, & Ortiz, 2014) but that the greatest increases in knowledge occurred for confident partners who were paired with a friend who had comparatively less programming knowledge (Werner et al., 2013). The team behind this programme also reported subtle differences in approaches to collaboration related to ethnicity (Ruvalcaba, Werner, & Denner, 2016). However, the programme's studies were performed with relatively small numbers of pupils, as was a study that found that secondary school girls preferred pair programming (Liebenberg, Mentz, & Breed, 2012).

In opposition to these positive conclusions on pair programming, research at a summer school coding course concluded that pair programming resulted in less work being completed and no increase in the overall progression of learning (Lewis, 2011). However, there were significant differences in the implementation of the pair programming between the summer school study and the class-based studies: in the summer school study, new partners were assigned every day by the teacher; in the classroom programme, learners were involved in pair assignment and worked together throughout a project. Moreover, roles were swapped every 5 minutes in the summer school study and every 20 minutes in the classroom programme.

Pair programming is thus a plausible method for engaging students in programming, with evidence that it can improve teaching and learning. However, care needs to be taken with implementing this approach with social dynamics, power struggles, friendship dynamics, confidence with computers, inequity of roles, how students talk to each other, the structure of tasks, and teacher intervention all potentially impacting interactions and learning (Lewis & Shah, 2015; Shah & Lewis, 2019; Denner, Green, & Campe, 2021). Further research with larger numbers of students in different contexts with carefully controlled interventions is needed to provide robust recommendations for classroom practice.

1.2. Peer instruction

Peer instruction (PI) is not simply peers teaching each other — it is a specific peer-to-peer teaching approach championed in university physics courses, and there is evidence that it increases students' learning (Crouch & Mazur, 2001). In class, learners are provided with carefully constructed, concept-based, multiple-choice questions, which are based on pre-lesson reading. Learners independently consider the questions and give their answer (vote) using flashcards or an online voting system. They then share their responses with their peers and discuss their thinking before re-submitting their answer (re-vote). The teacher reviews learners' first and second answers and, if needed, provides further support after the second answers before moving on to the next question.

The popularity and success of PI in university physics courses have led to it being used in other areas, including undergraduate programming classes. In undergraduate computer science courses, the introduction of PI (using electronic clickers) has been reported to lead to improvements in student retention (Porter & Simon, 2013), self-efficacy (Zingaro, 2014), in-class learning (Taylor et al., 2018), and longer-term exam outcomes (Zingaro & Porter, 2015). However, despite students saying they like PI and despite better short-term learning gains, a recent review of studies into PI reported little evidence of improved final examination performance. The review authors commented on the importance of the pre-lesson reading and the educators' explanation of the purpose of the PI activities for a greater likelihood of success of the approach (Luxton-Reilly et al., 2018).

Fewer studies on the impact of PI have been conducted in school settings, and fewer still in classroom computing contexts. In a Czech high-school physics class case study, teachers reported a preference for flashcards over electronic voting, and learners reported improved learning (Šestáková, 2016). The authors of a US study with five high-school physics classes reported that, despite improved learner outcomes, it was unclear whether this improvement was due to PI or to other aspects of the learning scenario such as student ability, teacher differences, or increased familiarity with question types (Cummings & Roberts, 2008).

In classroom settings, teachers sometimes use some of the constituent components of PI, e.g. carefully constructed, concept-related questions, flipped learning, and cooperative learning such as 'think, pair, share' (Lyman, 1981).

Think, pair, share has a long history of research involving younger learners, with evidence of its positive impact on pupil contribution (Rowe, 1986) and motivation, and on teachers' opportunities for assessment (Cooper & Robinson, 2000). In undergraduate programming research, think, pair, share has been found to positively impact learning outcomes (Kothiyal, Murthy, & Iyer, 2014). On the other hand, a comparison of grade 5 and 6 (*n*=108) students learning binary numbers using an unplugged approach or a think, pair, share approach found no difference in learning outcomes (Thies & Vahrenhold, 2016).

Peer instruction appears to be useful for teaching programming, and it could be used for all aspects of computing teaching.

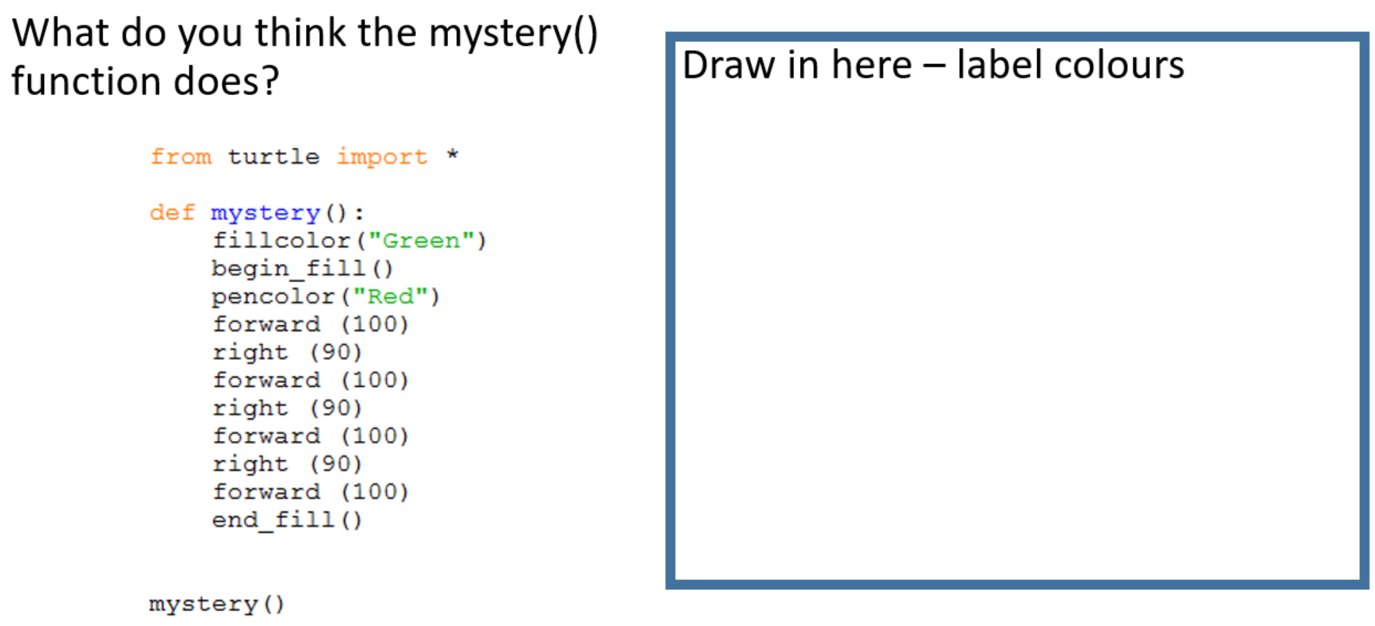
In Morrison and colleagues' recent review of undergraduate computer science research with a focus on broadening participation for women, PI was found to be implemented in a wide variety of ways and often associated with other interventions, making it difficult to draw conclusions about this approach by itself. However, the majority of PI studies included in the review reported some form of positive affective (attitude, motivation, engagement, identity formation), cognitive (academic performance, learning performance), or population (retention in field, graduation rates, employability, culture) outcome, often for all students not just for women. The authors suggest that educators try collaborative learning to broaden participation, especially PI, but caveat this with advice to carefully structure activities, include student training on how to be a good partner or team member, and look out for microaggressions and biased behaviour (Morrison et al., 2021). While studies of PI in university settings have promising results, further work is needed to investigate PI itself, its components, its outcomes, and other collaborative and peer-to-peer forms of learning in classroom settings.

1.3. Live coding

Modelling is a form of in-class demonstration where students observe as a teacher completes an activity whilst talking through their thought process. This brings an apprenticeship approach to teaching, and in the teaching of programming, it is also referred to as live coding (Rubin, 2013) (not to be confused with 'live coding' as a form of performance art). Pupil interaction may be introduced into this approach by asking learners what to do next at various points in the activity, and by asking them to spot mistakes. Modelling is also used in the teaching of other subjects, e.g. English: teachers may model writing as a direct form of instruction with little student involvement, or they may engage learners in active joint composition (shared writing) (Swartz, Klein, & Shook, 2001; Cremin & Baker, 2010).

To support learning through live coding, two things are essential. First, the teacher must carefully select appropriate examples for teaching new concepts, consolidating understanding, or addressing existing or potential misconceptions. Second, live coding should reveal the thinking of the demonstrator: what the teacher says as they 'think aloud' is crucial to the effectiveness of live coding.

In research with undergraduates, live coding has been compared to learning from static code, and it was found to be as good as, if not better than, letting learners read code examples, especially for helping learners approach larger coding assignments by demonstrating good programming habits (Rubin, 2013). Additional modelling can be provided by video recordings of experts programming (Bennedsen & Caspersen, 2005). In research into the creation of a


What do you think the mystery() function does?

```
from turtle import *

def mystery():
    fillcolor("Green")
    begin_fill()
    pencolor("Red")
    forward (100)
    right (90)
    forward (100)
    right (90)
    forward (100)
    right (90)
    forward (100)
    end_fill()


mystery()
```

Draw in here – label colours

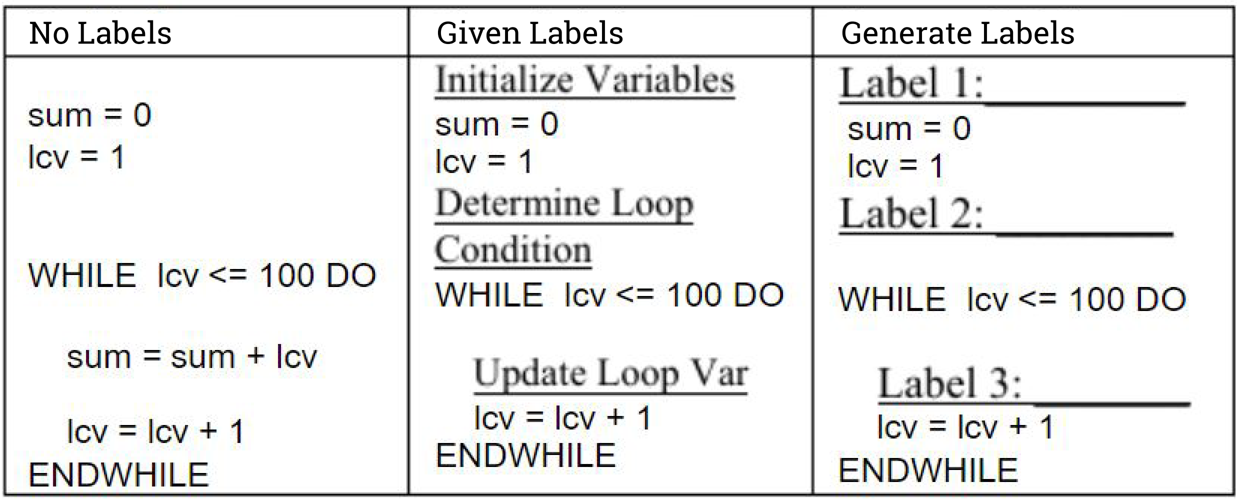
Figure 1: An example Predict activity (Sentance, Waite, & Kalia, 2019, p.478).

primary school maths and programming curriculum (ScratchMaths), live coding was noted as a technique used by more experienced teachers to supplement other approaches, and the researchers concluded that this was likely to lead to deep learning (Benton, Hoyles, Kalas, & Noss, 2017). In a recent review of teaching and learning of computational thinking through programming, the importance of encouraging learners to 'think aloud' was emphasised, as was the role of demonstration to model the problem-solving process (Lye & Koh, 2014).

1.4. PRIMM

PRIMM (Predict, Run, Investigate, Modify, Make), developed by Sue Sentance, is a pedagogy that has been evidenced to improve the learning of programming in classrooms (Sentance & Waite, 2017; Sentance, Waite, & Kalia, 2019). Building upon the findings of several other research groups, the pedagogy includes a sequence of instructional approaches and an emphasis on teachers and students talking about programming (Sentance & Waite, 2021).

The first stage is Predict and is centred around students reading a high-quality sample program that was created by their teacher, or a resource developer, which exemplifies the learning objectives. See Figure 1 for an example Predict activity. Learning to read code has been proven to be an essential first step needed before students write code (Lister, Fidge, & Teague, 2009). Importantly, students do not spend time typing in the sample program used at the Predict stage, rather they are given the program and spend time reading and talking about it. Following prediction, the code is Run, the next stage



No Labels	Given Labels	Generate Labels

sum = 0
lcv = 1
 	
WHILE lcv <= 100 DO
    sum = sum + lcv
    lcv = lcv + 1
ENDWHILE
	Initialize Variables
	sum = 0
	lcv = 1
	Determine Loop Condition
	WHILE lcv <= 100 DO
	    Update Loop Var
	    lcv = lcv + 1
	ENDWHILE
		Label 1: ________
		sum = 0
		lcv = 1
		Label 2: ________
		WHILE lcv <= 100 DO
		    Label 3: ________
		    lcv = lcv + 1
		ENDWHILE

Figure 2: Partial worked example formatted with no labels, given labels, and placeholders for student-generated labels (Morrison, Margulieux, Ericson, & Guzdial, 2016).

of PRIMM. Then, students move on to the Investigate stage, which requires them to answer carefully constructed questions that draw out important learning points. Teachers and resource developers who design Investigate stage questions are encouraged to use the Block Model (Schulte, 2008) to help them create their questions. The Block Model provides a holistic view of programs, from the detail of individual commands to what the overall program achieves.

The last two stages of PRIMM are Modify and Make; these stages particularly build on the Use–Modify–Create model (Lee et al., 2011) that has become popular in helping students take ownership of the products they make. Throughout the stages, classroom discussion is built into the process and fosters a socially rich experience of learning to program (Sentance, Waite, & Kalia, 2019).

1.5. Worked examples and subgoal labelling

As well as using sample programs for students to predict what the program will do, teachers also use sample programs when they model how to write code based on worked examples. Worked examples can be provided to students for them to learn about concepts, processes, and features of programming environments, such as the concept of iteration, the process of development, the role of variables, and tools for debugging.

A further enhancement of the use of worked examples is subgoal modelling: meaningful labels are added to worked examples to visually group steps into subgoals, highlighting the structure of code (see Figure 2). Students can be given code with subgoals, or they can be asked to add subgoals. Research indicates that students given code

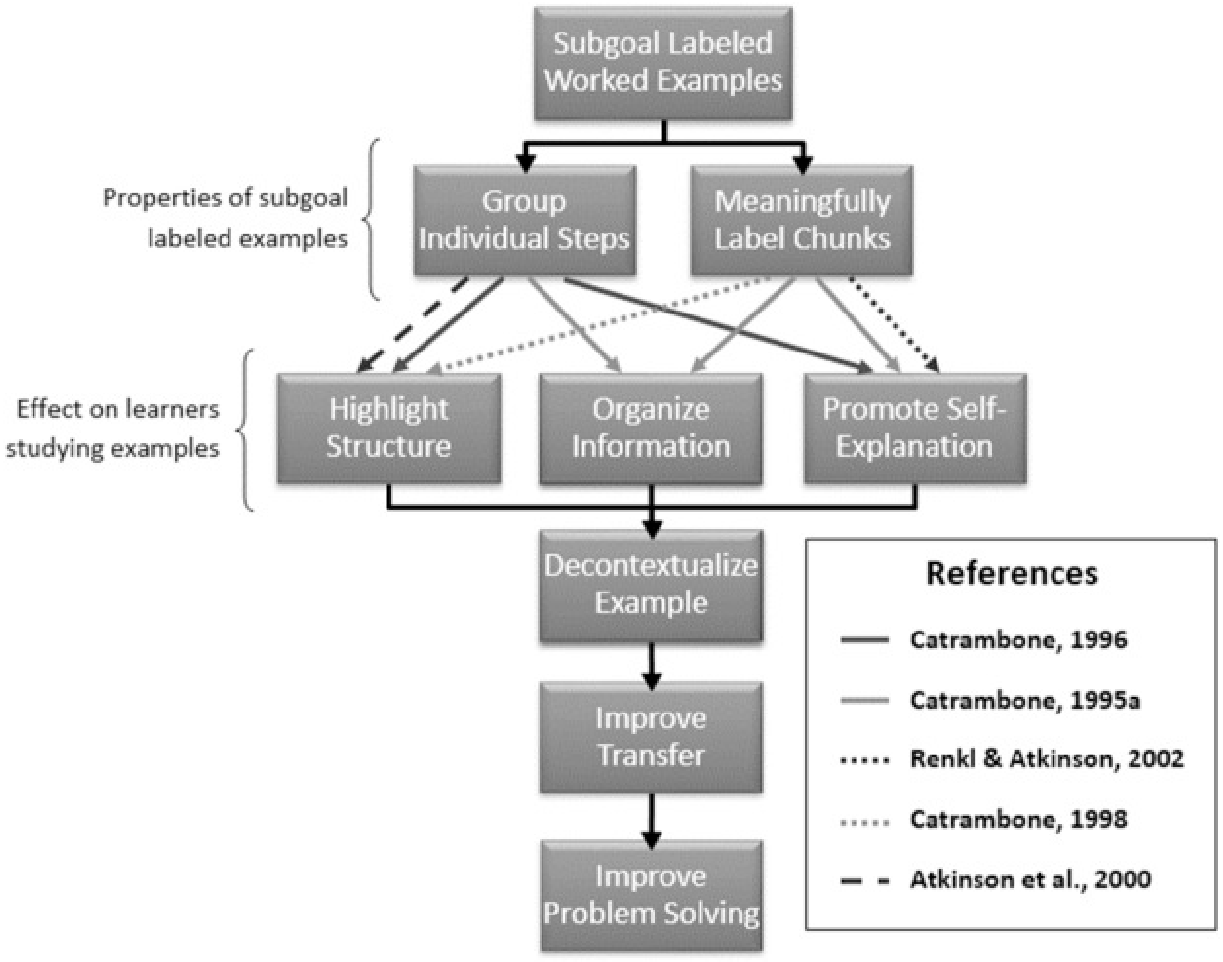


Figure 3: Diagram of how subgoal labelled worked examples can help learners improve problem solving performance. The "properties of subgoal labelled examples" level describes the physical characteristics of subgoal labels. The "effect on learners studying examples" level describes how these characteristics help the learners use effective learning strategies (Margulieux & Catrambone, 2016, p.60).

including subgoals perform significantly better on programming tasks than students given code without subgoals, and than students who are asked to add subgoals (Margulieux & Catrambone, 2016; Morrison, Margulieux, Ericson, & Guzdial, 2016). The researchers involved in these studies suggest that these results may be due to subgoals reducing cognitive load as students do not have to concern themselves with the extraneous load of the incidental information of the context. Instead, as shown in Figure 3, students can use the abstracted subgoal labels, which provide meaningfully labelled chunks and grouped steps, to work with the structure of the program that is already organised and promotes self explanation (Margulieux & Catrambone, 2016; Morrison, Margulieux, Ericson, & Guzdial, 2016).

Most research on worked examples with subgoals has been done in university settings. Recently, Margulieux, Morrison, Franke, and Ramilison (2020) redesigned a resource aimed at 15–18 years olds, adding subgoal labels to code.org[1] resources in an Advanced Placement programming course. The authors compared the performance of students who used the original unit to students using the redesigned unit and found some positive effects on outcomes. Students learning with subgoals performed no better on knowledge-based assessment but performed better on problem solving questions, wrote more on open ended questions, and continued to use subgoals after the course. Teachers working with students on the redesigned activities were surveyed and suggested that struggling students found subgoals the most useful (Margulieux, Morrison, Franke, & Ramilison, 2020).

1.6. Reading and tracing code

Substantial research in university settings has found that learning to read code is an essential part of learning to program (Lister, Fidge, & Teague, 2009; Lopez, Whalley, Robbins, & Lister, 2008; Venables, Tan, & Lister, 2009), with evidence suggesting that novices must be able to read 50% of their code (tracing code accuracy) before they can independently and confidently write code (Lister, Fidge, & Teague, 2009). Tracing is the skill by which one predicts the order in which programmed commands will be executed, including working out data values at each point in the program. A path of learning to support programming development has been suggested that requires learners to know about basic data structures and programming constructs before being required to trace code, which then leads to activities that involve explaining and writing code (Lopez, Whalley, Robbins, & Lister, 2008). Teague and Lister also found that using a carefully scaffolded sequence incorporating very small tasks with single elements for code reading and tracing led to increased programming knowledge for university students (Teague & Lister, 2014a).

Other researchers suggested a similar approach for primary school learners working with route-based programming, using a sequence of activities moving from reading and interpreting lines of code to eventually reading an entire program and predicting what it will do (Gujberova & Kalas, 2013). However, the challenge with this approach is to identify what stage a student is at, and to ensure they are given the right tasks and time to master skills before moving on (Teague & Lister, 2014b).

Eye-tracking has been used to investigate how students learn to read code and how this might change as they become more experienced, with evidence suggesting that experts read code less linearly than novices (Busjahn et al., 2015).

In a study with much younger learners, Dwyer et al. (2015) reported unintended affordances of visually complex block-based programming environments. For example, some students predicted a sprite's movement based on its visual appearance, such as where it was 'looking', rather than using the code associated with the sprite. Other students were unaware that there

[1] https://code.org/ accessed 11 November 2021

was code associated with the sprite, not realising they needed to click on the object to see the code. The authors recommended using explicit instruction to help students learn about a programming environment's different features, e.g. that the user interface/ execution area and code editing area work independently and together (Dwyer et al., 2015).

Some research has focused on developing tools and processes to teach how to trace and support the act of tracing programs. For example, tools to draw learners' attention to significant code features (beacons) have been used with undergraduates (Leppan, Cilliers, & Taljaard, 2007). In school settings, processes that systemise the teaching of tracing have also been devised, e.g. TRACS[2], a methodology developed in Scotland (Donaldson & Cutts, 2018).

More widely, approaches to teaching programming often include elements of code reading and tracing; for example, PRIMM includes carefully chosen code examples for students to read as the code exemplifies certain concepts or skills to be learned (Sentance & Waite, 2017).

1.7. Pattern-oriented instruction

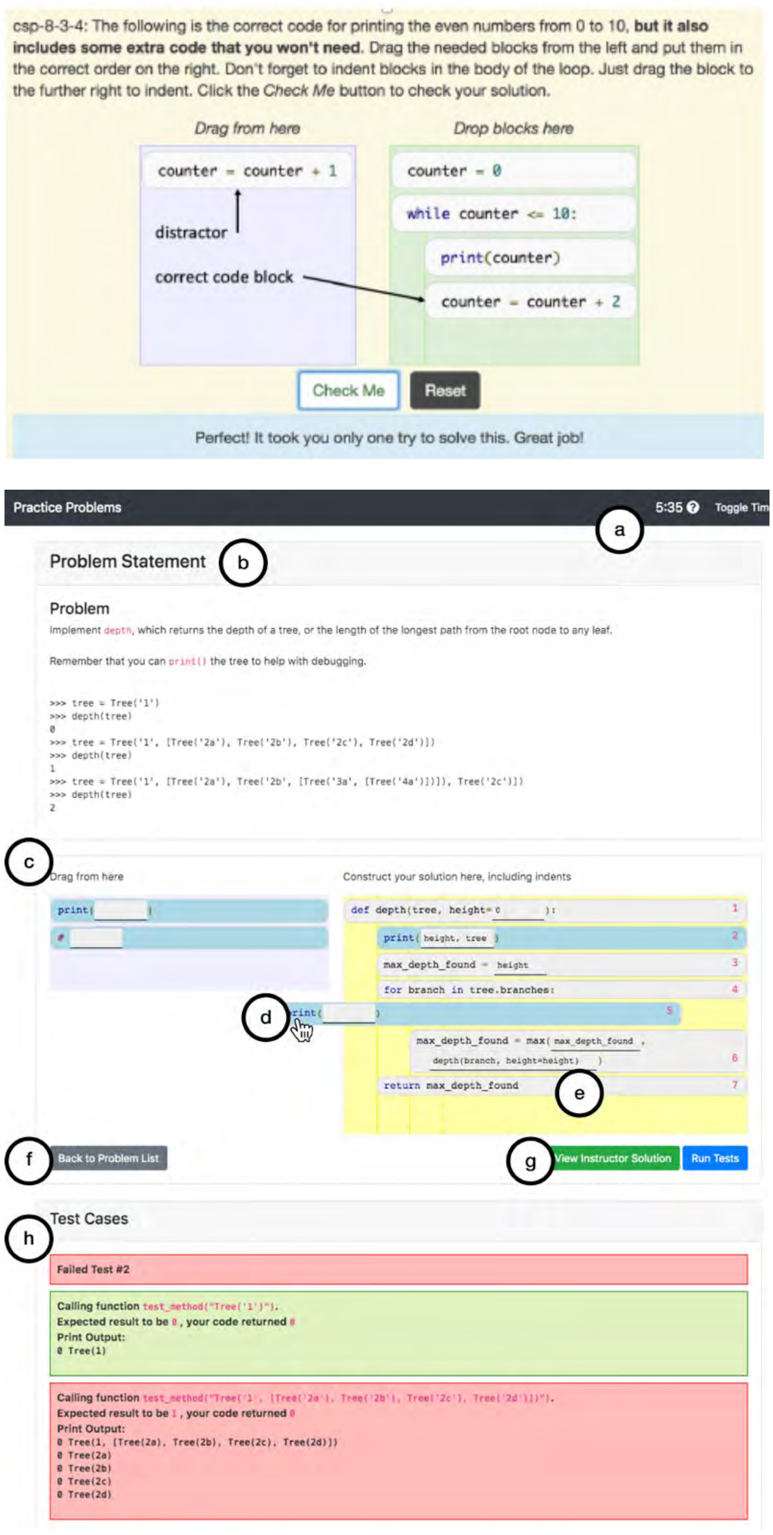
Many programming lessons require students to assemble programming commands into programs to achieve a particular purpose. As an intermediary step, some instructional approaches draw students' attention to commonly used patterns of commands. Students learn about and then re-use these patterns. Pattern-oriented instruction (POI) is one such approach.

POI was developed by Orna Muller and is thought to reduce students' cognitive load as students can think about the pattern as a 'black box' that meets a particular goal (Muller, 2005). The approach has been reported to help undergraduate computer science students learn how to better break problems down into parts and build up potential solutions (Muller, Ginat, & Haberman, 2007).

In POI, lessons are carefully planned to introduce students to lots of examples of a pattern and examples become more complex over time. Students are required to look for similarities and differences in the application of patterns and to discover how patterns are misused and to think about alternative patterns that might solve the same task (Muller, Haberman, & Averbuch, 2004).

POI often involves much student talk, as learners discuss different ways to solve problems. Using common patterns to teach programming has been used in universities for some time (e.g. Beck, Thomas, Drake, East, & Wallingford, 1996) and has been suggested for games development in school contexts (Repenning et al., 2015; Barnes et al., 2017). However, more work is needed to establish what patterns are most useful for different types of programs, in what programming languages, and what a pattern-based progression might look like for students.

[2] https://trace.dcs.gla.ac.uk/planc/tag/tracs/ accessed 12 November 2021



csp-8-3-4: The following is the correct code for printing the even numbers from 0 to 10, **but it also includes some extra code that you won't need.** Drag the needed blocks from the left and put them in the correct order on the right. Don't forget to indent blocks in the body of the loop. Just drag the block to the further right to indent. Click the *Check Me* button to check your solution.

```
counter = counter + 1
```

```
counter = 0
while counter <= 10:
    print(counter)
    counter = counter + 2
```

Perfect! It took you only one try to solve this. Great job!

Problem Statement

Problem

Implement `depth`, which returns the depth of a tree, or the length of the longest path from the root node to any leaf.

Remember that you can `print()` the tree to help with debugging.

```
>>> tree = Tree('1')
>>> depth(tree)
0
>>> tree = Tree('1', [Tree('2a'), Tree('2b'), Tree('2c'), Tree('2d')])
>>> depth(tree)
1
>>> tree = Tree('1', [Tree('2a'), Tree('2b', [Tree('3a', [Tree('4a')])]), Tree('2c')])
>>> depth(tree)
2
```

Drag from here

```
print(      )
#
```

Construct your solution here, including indents

```
def depth(tree, height= 0 ):
    print( height, tree )
    max_depth_found = height
    for branch in tree.branches:
    print(      )
        max_depth_found = max( max_depth_found ,
         depth(branch, height=height) )
    return max_depth_found
```

Test Cases

Failed Test #2

Calling function `test_method("Tree('1')")`.
Expected result to be 0 , your code returned 0
Print Output:

```
0 Tree(1)
```

Calling function `test_method("Tree('1', [Tree('2a'), Tree('2b'), Tree('2c'), Tree('2d')])")`.
Expected result to be 1 , your code returned 0
Print Output:

```
0 Tree(1, [Tree(2a), Tree(2b), Tree(2c), Tree(2d)])
0 Tree(2a)
0 Tree(2b)
0 Tree(2c)
0 Tree(2d)
```

Figure 4, top: A two-dimensional Parson's Problem with the solution on the right and a distractor on the left (Ericson, McCall, & Cunningham, 2019, p.1). Bottom: A nearly correct Faded Parson's Problem finding the depth of a tree. (a) Optional timer. (b) Problem description. (c) Faded Parson's Problem interface; participants can drag blocks between the bin (left) and the solution (right). (d) An optional print block being dragged to the right. (e) A blank that has been filled in with code by the student. (f) Students can navigate back to the exercise list or (g) run tests on their current solution. After "effort-completing" an exercise, they can view the instructor solution (g). (h) Descriptive test case results up to the first failed test (Weinman, Fox, & Hearst, 2021, p.6).

1.8. Targeted tasks (e.g. debugging, sabotage, annotation, fill in the gaps, Parson's Problems)

In this section, we group a range of other classroom activities together and are giving them the collective name **targeted tasks**. We define targeted tasks as those that focus students on specific learning goals. Such focused activities have been suggested to be particularly important for the teaching of the more difficult concepts such as programming initialisation, variables and loops, and assignment, which need to be explicitly taught (Hubwieser, Armoni, Giannakos, & Mittermeir, 2014; Meerbaum-Salant, Armoni, & Ben-Ari, 2013) and within a carefully considered progression of learning experiences (Seiter & Foreman, 2013; Falkner & Vivian, 2015; Dwyer, Hill, Carpenter, Harlow, & Franklin, 2014).

There are a wide range of potential targeted tasks that can be used as learning activities to highlight students' alternate conceptions or exemplify programming concepts. Some examples are spotting concepts, recalling facts or examples, changing aspects of programs, grouping and classifying example work, comparing solutions, following instructions, decomposing solutions, checking and improving work.

More specifically, students can predict what code will do, match designs to programs, investigate and fix buggy code, or sabotage code for their peers to fix. Students can be asked to annotate code with an explanation of what the code is intended to do. Parts of code can be removed and students asked to fill the gaps.

One particular format of targeted tasks is that of Parson's Problems; these provide learners with all the code required, but in sections, and with the sections in the wrong order (Parsons & Hadon, 2006). There are many variants of Parson's Problems, such as including superfluous lines of code with common syntactic or semantic errors to act as distractors (Ericson, Margulieux, & Rick, 2017) (see Figure 4, top), faded Parson's Problems where students increasingly complete some lines of code (Weinman, Fox, & Hearst, 2020) (see Figure 4, bottom), and adaptive Parson's Problems, which dynamically control problem difficulty based on a student's performance (Ericson, Foley, & Rick, 2018). Predominantly studied in undergraduate settings, Parson's Problems have been suggested to be particularly effective to help students with tasks that are not unusual (Haynes & Ericson, 2021), help students understand patterns in programs (Weinman et al., 2021), and to improve student engagement (Ericson et al., 2017). Notably, Parson's Problems have been compared to code reading and tracing activities with students being found to make more progress when using Parson's Problems (Ericson et al., 2017). However, a review of studies on Parson's Problems calls for more research due to a lack of replicated research (Du, Luxton-Reilly, & Denny, 2020).

The selection of tasks, along with how they are presented and scaffolded, within a progression of learning is important for both teachers' expectations as well as student confidence. The tasks need to be matched to each student's current level of knowledge, skills, and understanding. Such tasks can also be used for assessment purposes, but we do not cover assessment in any detail here.

2. Contexts and environments for learning programming

2. Contexts and environments for learning programming

In this section, we consider the ways in which programming can be taught, including the type of language and the classroom context, with focus on:

- Physical computing
- Block-based programming
- Project-based learning
- Programming unplugged
- Games

2.1. Physical computing

Using different contexts for teaching programming may inspire learners' interest, and some contexts appear to be more motivational than others. A common finding from research is that physical computing projects are particularly motivational to pupils (Garneli, Giannakos, & Chorianopoulos, 2015).

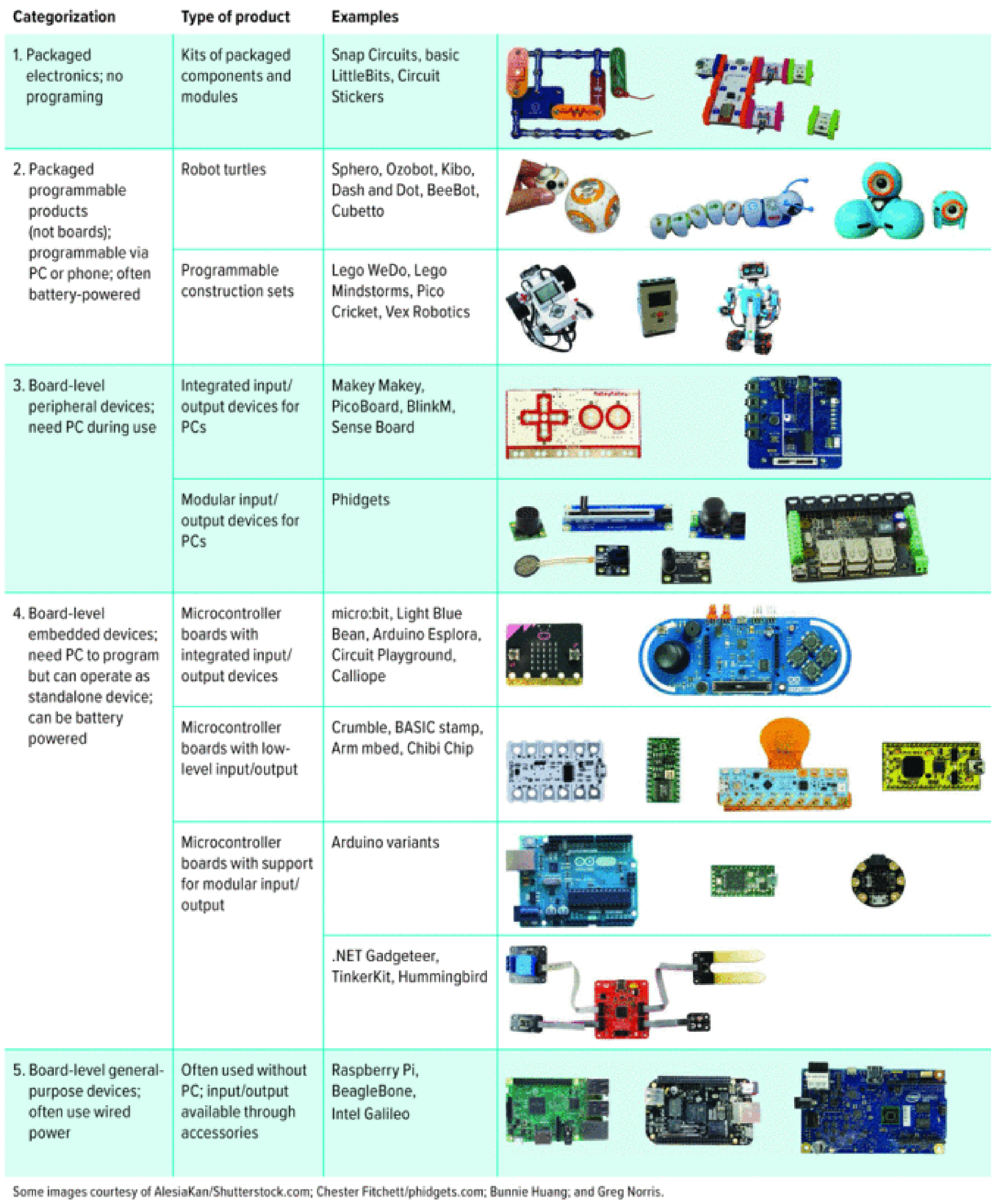
There are many different types of physical computing devices. As shown in Figure 5, devices can include packaged electronics with no programming required, programmable robots and construction sets, programmable boards with integrated or external input and output devices that need a PC during use, battery-powered embedded programmable boards, which can operate without a PC, and general-purpose programmable boards that often use wired power (Hodges, Sentance, Finney, & Ball, 2020).

Particularly popular, and inspired by Papert, the programmable robots of the 1970s and 1980s (such as the Roamer[3] and the Bee-Bot[4]) have been used to teach primary programming for some time. Similarly, small general-purpose programmable boards such as the Raspberry Pi[5], microcontrollers such as the Arduino[6], and similar products have been used in education for many years.

The recent development of low-cost educational microcontrollers and block-based programming languages has renewed interest in physical computing, exerting pressure on, and providing new opportunities for, teachers to incorporate physical computing and robotics into their teaching and learning activities. However, there is limited robust empirical evidence on the pedagogies to use, or on the impact of using physical computing on teaching and learning (Toh, Causo, Tzuo, Chen, & Yeo, 2016). Moreover, a multitude of high-profile resources for physical computing projects is now available. This abundance of options, combined with the lack of evidence-based teaching approaches, means there is a risk that schools will invest in resources that they cannot use effectively, or that have little impact on pupil progress; the product rather than the learning objectives may become the focus.

Research into the pedagogy to use with the current range of resources has started to emerge, and studies suggest sometimes conflicting approaches. Pre-teaching skills is advised by some, but a just-in-time approach is recommended by others; some researchers advocate for an exploratory approach (Przybylla & Romeike, 2014), while others promote following a design process (Bers, Flannery, Kazakoff, & Sullivan, 2014); often a range of targeted tasks, such as debugging activities

[3] http://www.valiant-technology.com/uk/pages/roamertoohome.php?cat=8&8 accessed 11 November 2021
[4] https://www.tts-group.co.uk/bee-bot-classroom-set-/1017264.html?gclid=Cj0KCQjwt-6LBhDlARIsAIPRQcK5CujcblAajX9coOmbvKArDyFtUIpvWIomY2n4W4wIGZFKSHrta4aAj2TEALw_wcB accessed 11 November 2021
[5] https://www.raspberrypi.com accessed 11 November 2021
[6] https://www.arduino.cc/ accessed 11 November 2021



Categorization	Type of product	Examples
1. Packaged electronics; no programing	Kits of packaged components and modules	Snap Circuits, basic LittleBits, Circuit Stickers
2. Packaged programmable products (not boards); programmable via PC or phone; often battery-powered	Robot turtles	Sphero, Ozobot, Kibo, Dash and Dot, BeeBot, Cubetto
	Programmable construction sets	Lego WeDo, Lego Mindstorms, Pico Cricket, Vex Robotics
3. Board-level peripheral devices; need PC during use	Integrated input/output devices for PCs	Makey Makey, PicoBoard, BlinkM, Sense Board
	Modular input/output devices for PCs	Phidgets
4. Board-level embedded devices; need PC to program but can operate as standalone device; can be battery powered	Microcontroller boards with integrated input/output devices	micro:bit, Light Blue Bean, Arduino Esplora, Circuit Playground, Calliope
	Microcontroller boards with low-level input/output	Crumble, BASIC stamp, Arm mbed, Chibi Chip
	Microcontroller boards with support for modular input/output	Arduino variants
		.NET Gadgeteer, TinkerKit, Hummingbird
5. Board-level general-purpose devices; often use wired power	Often used without PC; input/output available through accessories	Raspberry Pi, BeagleBone, Intel Galileo

Some images courtesy of AlesiaKan/Shutterstock.com; Chester Fitchett/phidgets.com; Bunnie Huang; and Greg Norris.

Figure 5: Suggested classification of physical computing devices (Hodges et al., 2020, p.26).

and code tracing work, is proposed (Kafai et al., 2014; Major, Kyriacou, & Brereton, 2012). A study by Kalelioglu and Sentance (2020) found that teachers commonly used demonstrations/live coding, pair programming, tinkering, copying programs, and explaining code verbally in their physical computing lessons.

When code tracing and debugging, students use their understanding of the notional machine to help them predict what will happen when the program executes. Recently, physical computing in high-school contexts has been used to study how teachers introduce notional machines in their explanations, role play, analogies, and metaphors (Jayathirtha & Kafai, 2021). Physical computing contexts have also been used to investigate culturally relevant pedagogy, including introducing handcrafting electronic textiles in non-formal (e.g. museums, after-school) and classroom settings, with promising results for broadening participation (e.g. Buchholz, Shively, Peppler, & Wohlwend, 2014; Kafai et al., 2014; Kafai et al., 2019).

2.2. Block-based programming

Sometimes called block-based, visual, or graphical programming languages, these languages use graphical images to represent programming commands. These easy-to-use languages are used not only with the youngest learners in formal and non-formal learning contexts, but also with older students in formal introductory programming lessons.

Block-based languages and their programming environments provide a range of affordances over and above text-based languages. Affordances include not requiring students to memorise and type in commands, not requiring students to deal with unfamiliar and sometimes confusing characters such as {}, [], and ==, and presenting natural language type block labels. Commands are often grouped by colour to give hints about their shared purpose, and shapes dynamically change their size to signal the scope of the command. Common shapes indicate which combinations of programming objects are allowed and provide an environment that allows quick and easy program-building (Bau, Gray, Kelleher, Sheldon, & Turbak, 2017; Weintrop, Killen, Munzar, & Franke, 2019).

Available since the 1990s, educational block-based languages have been developed to be easy to get started with, but also to be powerful enough to create advanced programs. Block-based languages such as Alice (Cooper, Dann, & Pausch, 2000), Scratch (Resnick et al., 2009), and Blockly (Fraser, 2015) have been suggested to be the most appropriate type of programming environment for young learners, such as those at primary (K–5) schools, with a prediction that this will remain so for the foreseeable future (Kölling, 2015).

As well as being heralded as improving students' outcomes from primary students to undergraduate contexts (Franklin et al., 2017; Grover & Basu, 2017; Price & Barnes, 2015; Weintrop & Wilensky, 2017; Malan & Leitner, 2007), block-based languages have been suggested to increase student interest in computing (Lewis, 2010; Maloney, Peppler, Kafai, Resnick, & Rusk, 2008).

Despite their popularity across settings, identifying the features of block-based programming languages that have the greatest impact on student outcomes and interest is an open question. In a recent large-scale research study comparing US high-school students' understanding of block-based versus text-based pseudocode, students were found to perform better with the block-based versions of the same questions (Weintrop et al., 2019). This is despite the block-based pseudocode not including many of the features of block-based programming languages that have been attributed as the main affordances of this form of programming language. The pseudocode was not colour coded, did not use natural-language type labels, and was not dynamic.

With an ever-growing number of educational block-based languages used in an ever-growing range of educational contexts, teachers must decide which language is best for their learners both in terms of their current level of expertise and how this will support their next steps in learning. However, which features of block-based languages and their programming environments are most important for these different users is unclear.

2.3. Project-based learning

Advice to provide opportunities for learner autonomy in school is not new (Rose, 2009). Autonomy increases intrinsic motivation as learners take ownership and pride in their work (Deci, 1971). Self-determination theory suggests that motivation is elicited and sustained by the three basic needs of autonomy, competence, and relatedness (Deci & Ryan, 1985, 2000). Project-based learning (Thomas, 2000), problem-based learning (Savery & Duffy, 1995), and inquiry-based learning (Edelson, Gordin, & Pea, 1999) vary in their definitions (Thomas, 2000); however, all incorporate the essential features of autonomy, ownership, and realism as learners are provided with choices of what to investigate and how to run their project.

Construction and constructionism are associated with these types of project-, problem-, and inquiry-based approaches as learners make things (or knowledge) through active exploration (Papert, 1980) and where the products made are meaningful in some way to the maker (Kafai & Resnick, 1996).

Criticism has been levelled at purely autonomous learning scenarios as learners left entirely to their own devices may develop alternate conceptions, ineffective mental models, or learn little, with the suggestion that a structured, guided approach is preferential (Mayer, 2004; Meerbaum-Salant et al., 2013; Clement & Merriman, 1988). Lye and Koh, in their review of teaching and learning of computational thinking through programming, found that construction (creating programs) with scaffolding was the most popular approach used by teachers (Lye & Koh, 2014). Finding the right level of scaffolding is not easy, as evidenced in a recent teacher survey where some respondents said they wanted to increase student autonomy and others wanted to reduce it (Rich et al., 2018). What seems to be important is to provide sufficient scaffolding to ensure that competence needs are met, and at the same time to provide opportunities for autonomy and relatedness.

2.4. Programming unplugged

Originally developed to raise awareness of, and enthusiasm in, computer science, unplugged activities teach about computing without a computer (Bell, Alexander, Freeman, & Grimley, 2009). Concepts such as abstraction, data representation, binary, and sorting algorithms can be introduced or further developed to deepen learning in this way (Rodriguez, Kennicutt, Rader, & Camp, 2017). Unplugged approaches include the use of stories, role play, magic, analogies, and metaphors (Curzon & McOwan, 2017; Curzon, 2013). Despite unplugged activities being named as a popular instructional method by teachers (Sentance & Csizmadia, 2017) and being claimed to be an effective teaching approach (Berry et al., 2015; Computer Science Teachers Association, 2011), research evidence on their effectiveness on learning outcomes is mixed (Bell et al., 2009; Curzon, 2013; Thies & Vahrenhold, 2016).

Stories, factual or fictional, can be used to provide real-world or imaginary contexts to introduce new and unfamiliar concepts. For

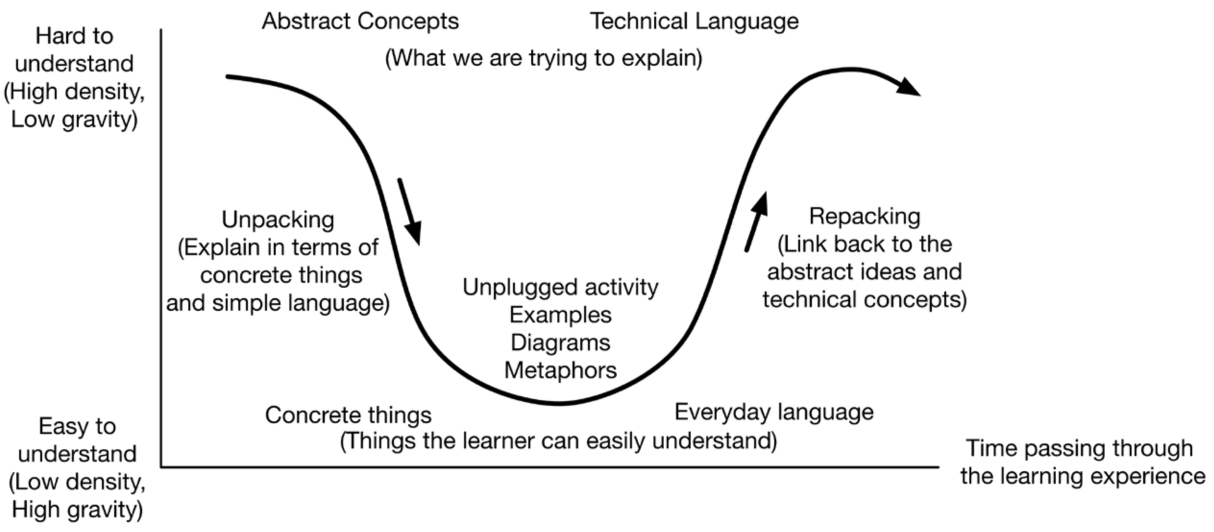

Hard to understand (High density, Low gravity)
Abstract Concepts
Technical Language
(What we are trying to explain)
Unpacking (Explain in terms of concrete things and simple language)
Unplugged activity
Examples
Diagrams
Metaphors
Repacking (Link back to the abstract ideas and technical concepts)
Easy to understand (Low density, High gravity)
Concrete things
Everyday language
(Things the learner can easily understand)
Time passing through the learning experience

Figure 6: Traversing a semantic wave (Waite, Maton, Curzon, & Tuttiett, 2019, p.3).

example, algorithm development has been introduced using the true story of a person with locked-in syndrome who developed a set of rules to communicate through blinking (Curzon, 2013).

Role play can provide a physical enactment of a complex concept. For example, acting out a bubble-sort breaks down the process into individual steps and highlights features that might otherwise be difficult to envisage (Katai, Toth, & Adorjani, 2014). Role play can also be used to help learners design new products, as they step through and try out their ideas. For example, when learning how to program programmable toys, students can ‘play turtle’ to help them understand the way the machine works, as they embody and execute the steps of their solution (Papert, 1980).

Analogies and metaphors can be used to introduce new concepts by using the learner's knowledge of other concepts as a springboard to make links and build new understanding. However, this requires teachers to have a depth of understanding of: the concept being introduced; the concept being compared against; the learners’ understanding of the comparative concept; the progression of learners' understanding of the analogy or metaphor; and potential misconceptions associated with the developing mental model.

A common analogy in computing is the explanation that a variable is like a box, for which research has uncovered a range of misconceptions (Hermans, Swidan, Aivaloglou, & Smit, 2018). However, there is limited research on the use of other analogies and metaphors in classroom settings within current curriculum progression. For example, it is not clear what impact there is on young children’s mental model of a computer if they are informed that a CPU is a brain or that some computer files are viruses that replicate and damage other computers.

Semantic waves, a sociological knowledge-building theory, have been suggested to be a useful pedagogical tool for planning and

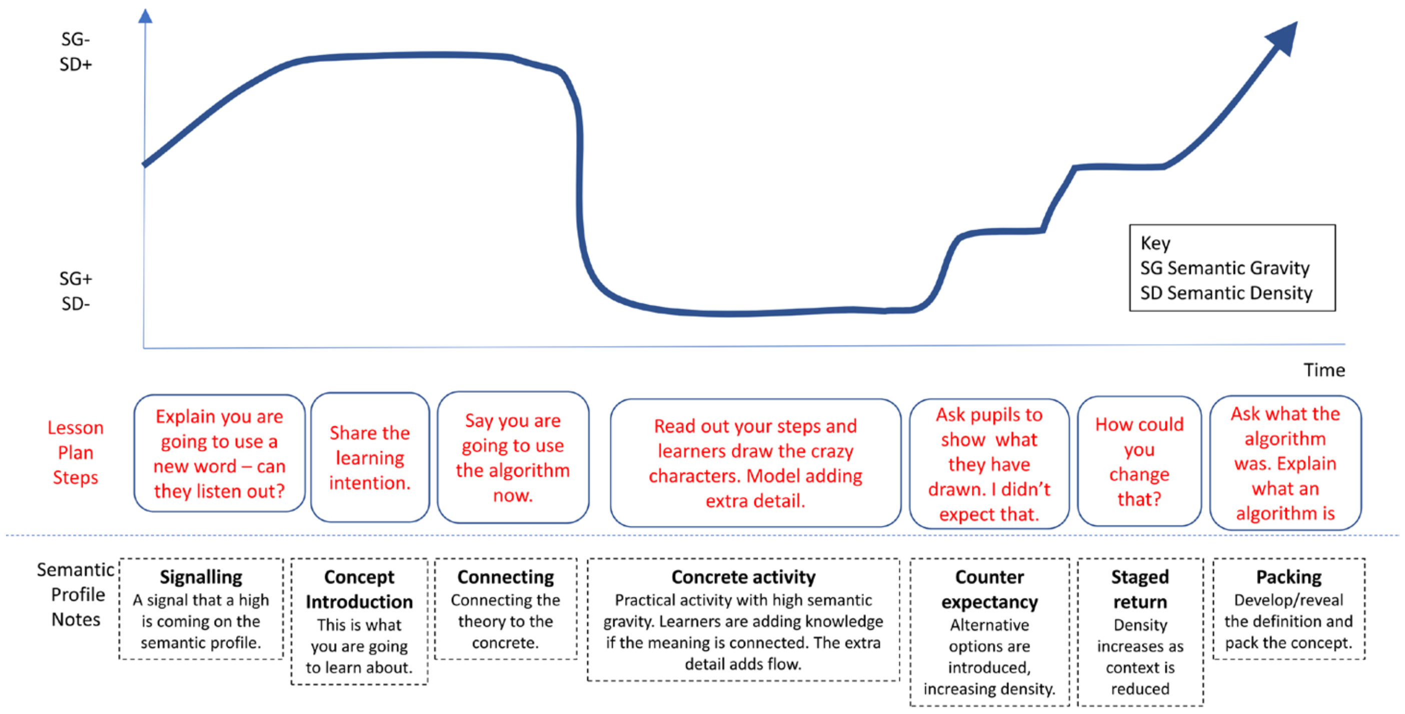

SG-
SD+
SG+
SD-
Key
SG Semantic Gravity
SD Semantic Density
Time
Lesson Plan Steps
Explain you are going to use a new word – can they listen out?
Share the learning intention.
Say you are going to use the algorithm now.
Read out your steps and learners draw the crazy characters. Model adding extra detail.
Ask pupils to show what they have drawn. I didn't expect that.
How could you change that?
Ask what the algorithm was. Explain what an algorithm is
Semantic Profile Notes
Signalling A signal that a high is coming on the semantic profile.
Concept Introduction This is what you are going to learn about.
Connecting Connecting the theory to the concrete.
Concrete activity Practical activity with high semantic gravity. Learners are adding knowledge if the meaning is connected. The extra detail adds flow.
Counter expectancy Alternative options are introduced, increasing density.
Staged return Density increases as context is reduced
Packing Develop/reveal the definition and pack the concept.

Figure 7: Semantic profile for the Crazy Characters lesson plan introduction (Waite et al., 2019, p.5).

evaluating the concept-rich, yet practically applied subject of computer science. It has been used to study why unplugged approaches may be more or less effective (Waite et al., 2019; Curzon, Waite, Maton, & Donohue, 2020). Simply put, in this theory, contexts and concept vocabulary are mapped over time as a profile. A wave shape to the profile, as shown in Figure 6, has been associated with successful knowledge building across subject areas from biology to dance as abstract concepts and familiar contexts and vocabulary are successfully connected for prior learning (unpacking) and for new learning (repacking) (Maton, 2013; Maton, Hood, & Shay, 2016). Figure 7 shows the semantic profile of an unplugged learning activity, showing how the lesson is taught over time, including staged repacking, and opportunities to improve the unpacking phase (Waite et al., 2019). Further research is needed to investigate semantic profiling and unplugged activities, and their long-term impact, as well as whether the approach is useful more generally in computer science education.

2.5. Games

Similarly to physical computing, game creation as a context for learning how to program has been cited as being motivational for students (Repenning et al., 2015). However, as with research into physical computing pedagogy, the evidence for game creation being beneficial is often not robust (Kafai & Burke, 2015).

One notable example of a games context being used to teach programming is the work of the Scalable Design Team (Repenning et al., 2015). This group have developed software

(AgentSheets & AgentCubes[7]), curricula, and a framework, which requires learners to design and program games as a precursor to designing and programming simulations for science and other subjects. Rather than basing teaching on objectives related to students learning about programming constructs such as sequence, selection, and repetition, the curricula focus on common patterns used in creating simulations. Patterns such as ‘generation’, ‘absorption’, ‘diffusion’, and ‘transportation’ are exemplified and used to drive the learning objectives. Also, a Use–Modify–Create type approach with much group work and a focus on pupil ownership of work is used. The team have reported success in terms of both pupils’ learning and motivation (Repenning et al., 2015).

[7] https://agentsheets.com/ accessed 11 November 2021

3. Supporting learners

3. Supporting learners

Using the strategies outlined in Section 1 and contexts in Section 2 will help students learn how to program. In this section, we consider what the research says about how we can further support learners, particularly those who have difficulties with programming. We look at:

- Addressing potential and common difficulties and alternative conceptions (misconceptions)
- Cognitive apprenticeship
- Developing metacognition around abstraction
- Include design
- Focus on vocabulary and language
- Scaffolding and a blended approach
- Developing problem-solving skills (computational thinking)

3.1. Addressing potential and common difficulties and alternative conceptions

Several school-based studies have suggested that more difficult programming concepts, such as initialisation, variables, loops, and assignment need to be explicitly taught (Grover & Basu, 2017; Meerbaum-Salant et al., 2013). Studies of alternative conceptions, sometimes called misconceptions[8], have mostly been undertaken in higher education rather than schools, and include what concepts are judged as difficult or not and how to approach them (Du Boulay, 1986; Veerasamy, D'Souza, & Laakso, 2016), with far fewer studies focusing on younger students (Gal-Ezer & Zur, 2004; Hermans et al., 2018). Sorva, in his PhD research, noted over 150 potential programming misconceptions (Sorva, 2012). A lack of teacher knowledge can contribute to development of misconceptions, as well as limited dissemination of approaches and tools that can reduce misconceptions and difficulties (Qian & Lehman, 2017; Sorva, 2018).

The relationship between difficulties and misconceptions has been evidenced by several studies. Students who have alternate conceptions about the operation of various constructs (e.g. in-built functions, parameter passing, nested if statements, for loops, using lists) have been found to make mistakes in related knowledge-based tasks and coding activities (Veerasamy, D'Souza, & Laakso, 2016). For example, learners may have difficulty using variables, because they have the misconception that variables hold more than one value, formed due to a mental model based on the 'variable as a box' metaphor (Hermans et al., 2018).

Shared alternate conceptions about vocabulary have also been noted. These arise from learners thinking that terminology that exists across subjects, including mathematics and familiar English terms, has a shared meaning (Qian & Lehman, 2017; Sorva, 2018). For example, learners may believe that the symbol "=" means the same thing in maths and programming, or that a variable in science is the same in some ways as a variable in computer science.

There is a long history of research around misconceptions that relate to the notional machine (Du Boulay, 1986). Simply put, researchers have claimed that there are misconceptions based on learners' inaccurate or incomplete understanding of how a computer works and how it executes the code for a specific programming language. Another similar, commonly held false belief is claimed to be that

[8] Misconceptions is a term that is often used in education literature when discussing a learner's evolving understanding and the points at which this understanding (mental model) deviates in some way from what was expected or planned. Misconceptions implies there is a problem, a faulty mental model, and something that needs to be addressed and overcome. Alternate or alternative conceptions are terms also used in this field and are interpreted by some as synonyms for misconceptions or they can be seen as having less of a negative connotation; this allows them to be used in a way that accepts that learners' mental models will develop along different lines moving towards a planned view. Other terms used in this area include preconceptions, naive beliefs and theories, alternative beliefs and frameworks. We use the term misconception where studies have used this term and do not differentiate or analyse the study authors' view of the term.

novices attribute computers with an innate ability to sort out errors in students' code (Pea, 1986).

Several approaches have been found to counter alternate conceptions where students believe that longer programs are more inefficient than shorter programs, or that more variables mean less efficiency. These approaches include earlier teaching of related concepts, work on underpinning vocabulary, and introduction of tasks that directly address the misconception (Gal-Ezer & Zur, 2004). Such activities are targeted tasks that pre-empt or directly address and rectify misconceptions, including asking learners to compare programs line by line or create new versions (Gal-Ezer & Zur, 2004).

Some approaches for directly tackling difficulties may in themselves cause misconceptions. For example, using the box analogy to help learners understand variables can introduce limited and faulty mental models (Qian & Lehman, 2017). Research suggests that teachers should carefully assess learners' understanding to reveal the details of their current mental models, enabling them to work out what approach might be best to help their learners overcome difficulties and make progress (Qian & Lehman, 2017; Sorva, 2018).

3.2. Cognitive apprenticeship

Cognitive apprenticeship is a concept introduced by Collins, Brown, and Newman back in 1987 and refers to the way that novices gain expert skills by observing and then practising expert activity (Collins, Brown, & Newman, 1987). Some teaching approaches associated with cognitive apprenticeship are modelling, coaching, scaffolding, student articulation, reflection, and exploration.

As a form of cognitive apprenticeship, collaborative learning through pupil-to-pupil support, such as with Digital Leaders, appears to provide opportunities for peer apprenticeship. However, the effectiveness of this approach regarding learner outcomes has not yet been formally evaluated. Passey (2014) highlighted the benefits of Digital Leaders for teachers and pupils, as technological support was provided. Still, the author recommended further research on the balance of activities undertaken and the outcomes and perceptions for all pupils engaged in the programme. Similarly, the need for research into the impact on pupils who are providing support in collaborative learning approaches has also been raised (Ching & Kafai, 2008).

In their review of teaching and learning of computational thinking through programming, Lye and Koh found that authentic contexts with scaffolding and reflection activities appeared to be most successful, but the authors advised that no one pedagogical solution is appropriate for all classes. They suggested using a number of approaches that fall under the umbrella of cognitive apprenticeship, including much scaffolding at the start of projects, modelling, and studying, modifying, and extending code samples (Lye & Koh, 2014).

3.3. Developing metacognition around abstraction

Abstraction has been cited as the cornerstone of computer science (Wing, 2008) and although it has been argued that it is not a skill that is unique to computer science (Ubiquity staff, 2007), there appears to be a consensus that being able to use and understand abstractions is a fundamental aspect of learning to program (Barr & Stephenson, 2011; Tedre & Denning, 2016).

Several frameworks have been suggested that support teachers, and their students, to build

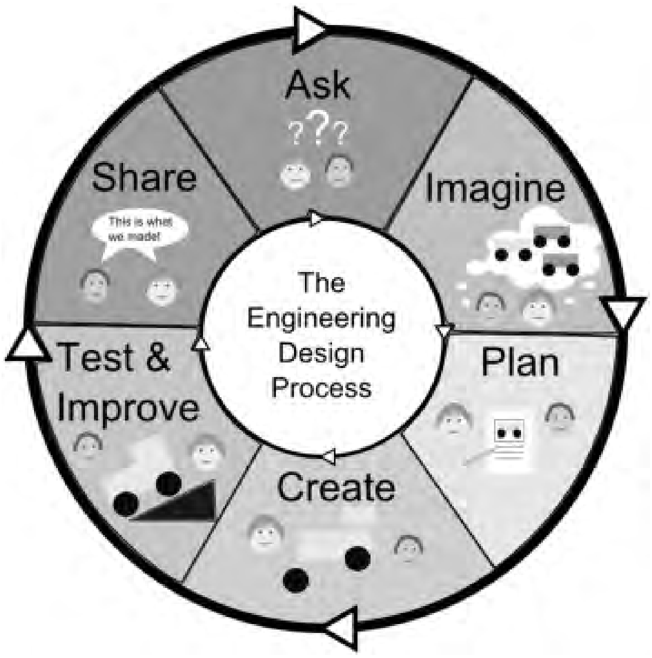


Figure 8: An illustration of the simplified engineering design process (Bers et al., 2014, p.155).

mental models about abstractions related to programming, which will help them to teach and learn how to program.

Established through research with university students, the Abstraction Transition Taxonomy (AT) divides student knowledge and practices in learning to program into three levels: Code, Computer Science (CS) Speak, and English; AT also describes the transitions between these levels. An example transition goal given by the study is "Given a technical description (CS Speak) of how to achieve a goal, choose code that will accomplish that goal" (Cutts, Esper, Fecho, Foster, & Simon, 2012).

For the understanding of algorithms by university students, researchers have defined Levels of Abstraction (LOA), a framework similar to AT but with four levels: execution of code, code, object, and goals (Perrenet & Kaasenbrood, 2006). Armoni (2013) further developed this framework for high-school students, in which the 'object' level was renamed 'algorithm' level to support teacher and pupil understanding, and transitions across the levels were also defined.

Armoni and Statter successfully used this adapted LOA framework in high schools (Statter & Armoni, 2016), and reported that learners using the framework showed improvements in attendance, algorithm development, algorithm creation, ability to explain solutions, and understanding of initialisation, with more improvement by girls than boys (Statter & Armoni, 2017). To support primary classrooms, the LOA levels have been further renamed as running the code, code, design, and task (Waite, Curzon, Marsh, Sentance, & Hawden-Bennett, 2018).

The importance of students being able to move from the 'task' level to the 'code' level and vice versa is linked with advice that learners would benefit from being able to draw on existing templates or plans that solve a certain type of problem. This ability to abstract a task into such templates was noted in expert programmers and a recommendation was made that novice programmers would benefit from being taught 'learning templates' as well as a process for problem solving (Lokkila et al., 2016).

3.4. Include design

Teaching the process of problem solving is not a new requirement in computing (Soloway, 1986; Robins, Rountree, & Rountree, 2003) and yet it appears to be rarely addressed as a goal in curricula (Rich, Strickland, & Franklin, 2017) or included in resources for teachers to use (Falkner & Vivian, 2015).

For younger learners, a simplified engineering design process, see Figure 8, has been

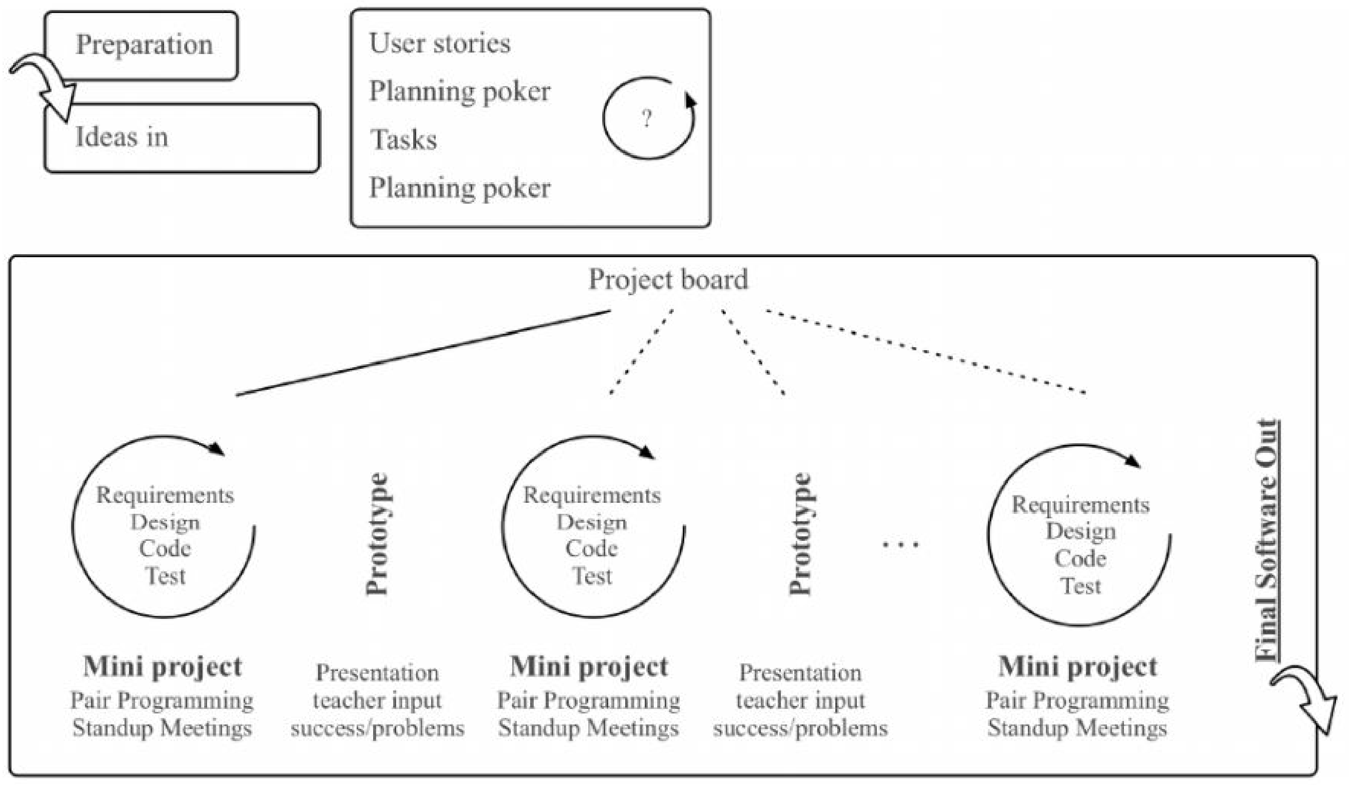


Figure 9: Agile model for projects in computing education (AMoPCE) (Romeike & Göttel, 2012, p.55).

suggested to support program development; this process includes phases of ask, imagine, plan, create, test and improve, and share (Bers et al., 2014). However, difficulties of implementing design in primary classrooms have been found to include student resistance to design, a lack of time to do design, a lack of teacher and pupil expertise in design, conflicting pedagogy choices, a lack of teaching resources, and confusion over what an algorithm is (Waite, Curzon, Marsh, & Sentance, 2020).

In industry, the classic approach to software development is a waterfall: the software requirements are gathered; these are then analysed; then a design is created; the design is implemented as code; the code is tested and finally delivered. Other development approaches are now popular, including test-driven design and iterative methods. Simply put, iterative methods run through a similar cycle to the waterfall for parts of the solution instead of the whole, and these parts are combined as the development moves along. Another professional software development approach is agile methodology. Several studies have reported on the use of agile methodologies in high schools, including development of an agile process for school use (see Figure 9) (Romeike & Göttel, 2012) and with indications of increased code quality, student motivation (Missiroli, Russo, & Ciancarini, 2016), and student self-sufficiency (Kastl, Kiesmüller, & Romeike, 2016).

However, the design process used by students learning to program is considered to be different to the process used by professional programmers, because students are likely to explore more than experts do, and they don't have to consider issues such as how a software system might change in the longer term or how it might be reused. Researchers have suggested the use of two consecutive cycles for teaching students problem solving in programming: an exploration process cycle in which students analyse and understand programs, and a design process cycle in which students design and construct programs (Schulte, Magenheim, Müller, & Budde, 2017). Teachers employing this approach need to consider which cycle learners are moving through and how to nimbly move students from one cycle to the other and back again. This design–exploration cycle has not yet been used in practice and further work is needed to explore it in action.

Besides using industry methodologies, other approaches for teaching how to design have been suggested that are included in overall structured problem-, process-, or project-based approaches to running projects in which students learn how to program.

In higher education research into Problem-Based Learning (PBL), improved student motivation and increased generic design skills have been reported (Nuutila, Törmä, & Malmi, 2005). Middle school education researchers have claimed Process-Oriented Guided Inquiry Learning (PoGiL) to be effective in teaching computing including design (Griffin, Pirmann, & Gray, 2016). The authors of a study comparing a Project-Based Learning (PjBL) strategy, a traditional learning strategy, and a game-development strategy reported that the PjBL students completed their activity with fewer mistakes, while the traditional group experimented with more complex concepts, although not always successfully (Garneli, Giannakos, & Chorianopoulos, 2015).

Design and software life cycle teaching is incorporated explicitly, to different degrees, in models and frameworks for teaching programming, but further research is needed to evaluate what design objectives should be included and which approach is best to improve design expertise for pupils in classroom settings.

3.5. Focus on vocabulary and language

Research connecting concept development to speech has a long history. Sapir remarked in 1921: "The birth of a new concept is invariably foreshadowed by a more or less strained or extended use of old linguistic material; the concept does not attain to individual and independent life until it has found a distinctive linguistic embodiment" (Sapir, 1921, para. 15). Student performance has a clear association with an understanding of subject-related vocabulary (Espin & Foegen, 1996), and explicit instruction including integration, repetition, and meaningful use is cited as being essential for vocabulary development (Beck, McKeown, & Kucan, 2013; Nagy, 1988). Incidental, or topical, experiences from general listening and reading in other contexts have improved learner progress in developing conceptual understanding (Carlisle, Fleming, & Gudbrandsen, 2000).

In maths, a subject where conceptual understanding is bound to vocabulary (Capraro, Capraro, & Rupley, 2010), topical word learning can be problematical as the colloquial meaning of terms can be different to the mathematical meaning, and some terms are unique to maths (Dunston & Tyminski, 2013). Computing poses similar problems, as the terminology can be unfamiliar or have different meanings to more general use. For example, the words bit, bug, memory, and cloud have different meanings in computing to their general use. More technical words such as algorithm may have subtly different meanings across school subjects and a range of definitions within computing (Diethelm & Goshler, 2015).

Not all educators may be aware of the confusion that these 'reused', or differing definition terms pose for those new to computing. Conversely, teachers may be unaware of the nuanced differences and misconceptions that may arise from exploiting apparent analogies of common terms. Therefore, although key vocabulary is essential to develop understanding, it must be introduced with an awareness of the potential mental models and alternate conceptions that may emerge. Further research is needed to explore learners' understanding of terms, teachers' use of terms, and the impact of inconsistent use of terms (Diethelm & Goshler, 2015).

International studies and curricula have been developed that focus on the importance of pupil talk and vocabulary. In the US, Grover and Pea (2013) developed a discourse-intensive curriculum, whereby the significance of terminology was emphasised, and activities developed that required learners to rehearse and use key terms, verifying and constructing personal understanding through social interactions. Working on Israeli secondary computing materials, Armoni has developed a framework with carefully constructed levels. The importance of using different vocabulary to distinguish between the development of the algorithm and its implementation as code was suggested as essential for developing conceptual understanding (Armoni, 2013; Statter & Armoni, 2016). Work with university learners in Scotland on the language used in solving computing tasks paints a similar picture, with the importance of talk and the use of English–computer science talk being key for the development of understanding (Cutts et al., 2012).

Sentance and Waite (2021) synthesised discourse frameworks associated with the study of talk in general teaching and learning to analyse talk in high-school programming classrooms where the PRIMM pedagogy was being used. The authors developed a generic theoretical model for planning and evaluating talk in the programming classroom (see Figure 10) and found several key factors that enhanced discourse. Key factors included encouraging talk through classroom routines, using questions and explanations, including goals on vocabulary, and careful design of learning contexts, including using example code, activity structure, and the student's own code to stimulate talk (Sentance & Waite, 2021).

Applying theories related to classroom talk, Zakaria et al. have designed and investigated a structured feedback intervention for teachers to use to support students doing shared programming tasks. Comparing the dialogue and activity of six pairs of students, aged 10 to 11 years old, from classes with and without the intervention, the authors reported promising results in productive collaboration and discourse such as increased exploratory talk including more justification and an increase in shared alternative ideas. The authors reported that further work is needed to refine the feedback framework and larger and more diverse sample sizes are needed to validate the approach (Zakaria et al., 2021). The impact on learning outcomes also needs to be investigated.

3.6. Supporting learning and a blended approach

Three broad theories, or approaches to learning, have been noted as prevalent in research studies on teaching programming in classrooms (Waite, 2017):

- Exploration
- Problem solving and making (Papert, 1980)
- Direct instruction (Dreyfus & Dreyfus, 1980)

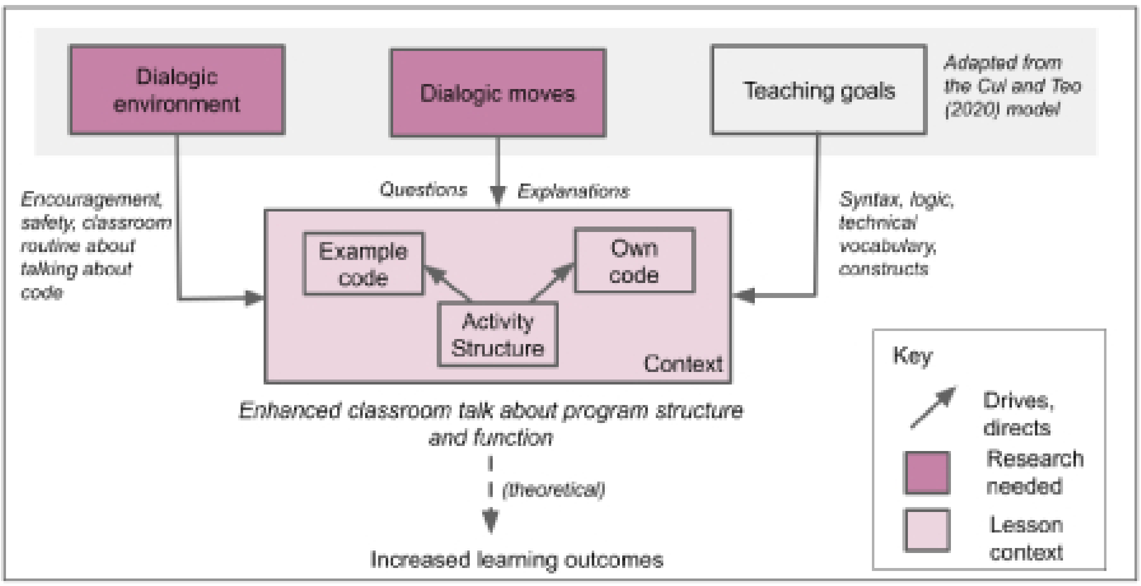

Dialogic environment
Dialogic moves
Teaching goals
Adapted from the Cui and Teo (2020) model
Encouragement, safety, classroom routine about talking about code
Questions
Explanations
Syntax, logic, technical vocabulary, constructs
Example code
Own code
Activity Structure
Context
Enhanced classroom talk about program structure and function
(theoretical)
Increased learning outcomes
Key
Drives, directs
Research needed
Lesson context

Figure 10: Talk in the programming classroom (Sentance & Waite, 2021, p.13).

Within these approaches, the degree of control pupils have over what they are learning about can vary. Generally speaking, direct instruction gives the teachers more control of the learning objectives, whereas with the exploration and making activities the students have more control. However, even in problem solving and making and exploration activities, a teacher can give pupils more or less freedom with the task at hand and can constrain the learning, for example, by limiting resources available or by framing the task.

Associated with providing support is the idea of scaffolding, which is used in education to describe both the micro-level scaffolding of teachers interacting with students in lessons and also the macro-level scaffolding of planning lesson goals and the organisation of learning tasks (Hammond & Gibbons, 2001). A simple example of a continuum of scaffolding has been suggested to provide initial guidance for teachers to understand their choices better; the Computer Science Student-Centred Instructional Continuum (CS-SCIC) includes broad categories of instructional approaches such as copying code, targeted tasks, shared coding, project-based, inquiry-based, and tinkering. This has been successfully used in England and the USA to support teacher professional development (Waite & Liebe, 2021). CS-SCIC reflects the tension between exploration, making, and direct teaching, but does not advocate any approach or order of use of approaches over any others. It simply gives teachers a way to talk about their choices with the expectation that teachers will create a sequence of learning experiences (Waite & Liebe, 2021).

In creating learning experiences, rather than just using one approach to teach programming, some advocate a blended approach encompassing a range of approaches (Grover, Pea, & Cooper, 2015). Which approaches should be included, and in what order, is suggested by studies, and sometimes the advice provided can be combined, but sometimes there are conflicting views. Some research suggests a controlled progression of more direct teaching approaches should be planned for teaching programming, particularly for more difficult concepts (Hubwieser et al., 2014).

Curricula often include exploration activities at different points in a sequence and progression of learning (Grover, Pea, & Cooper, 2015; Meerbaum-Salant et al., 2013; Repenning et al., 2015). Exploration might be used to: introduce new concepts; revise or consolidate ideas; pre-empt, address or challenge misconceptions; or provide opportunities for extension and creativity.

Hansen, Hansen, Dwyer, Harlow, & Franklin (2016) used the Universal Design for Learning (UDL) framework to underpin their US grades 4 to 6 computing curriculum, which focuses on differentiation to support all learners. By incorporating a carefully constructed learning progression and tasks moving from simple to complex (Franklin et al., 2016), a range of instructional strategies are included, such as unplugged activities, modelling, small group work, a sandbox to try out new skills, as well as more targeted tasks to teach specific concepts.

In their review of the teaching and learning of computational thinking through programming, Lye and Koh emphasise constructionism. Still, they advise that no one pedagogical solution is appropriate for all classes. They suggest scaffolding at the start of projects, studying, modifying, and extending code samples, as well as recommending that demonstrations, tutorials, and debugging tools be used (Lye & Koh, 2014). They also suggest the use of Use–Modify–Create (Lee et al., 2011) as well as problem and project-based learning, using authentic contexts, with a 'just-in-time' approach to present new concepts as and when needed, and scaffolding and reflection activities emphasising that students ought to be 'thinking-doing and not just doing' (Lye & Koh, 2014).
In suggesting their concepts, practices, and perspectives, Brennan and Resnick (2012) have proposed a framework for learning block-based programming. However, this framework does not prescribe in detail a classroom pedagogy for teaching programming. Insight from a blended approach that may reflect the framework is offered by the Creative Curriculum authored by Brennan, Balch, and Chung (2014), where concepts are introduced through a series of projects. Instructional techniques, including minimally guided exploration, guided exploration with suggested blocks, debugging activities, reusing and remixing example code snippets, and learners creating their own work with support from examples of code (Scratch cards and example Scratch projects), are included as well as design journals to record ideas and provide a means to share and reflect on learning.

It seems likely that a blended approach is most effective. However, further research is needed to investigate the effectiveness of each instructional approach in different contexts, in particular orders, for different learners at different points in their progression (Webb, Repenning, & Koh, 2012).

3.7. Develop generic problem-solving skills (computational thinking)

Definitions of computational thinking vary. Some definitions include programming concepts (such as sequence, repetition, and events), others a range of problem-solving practices (such as logical reasoning, algorithmic thinking, abstraction, decomposition, and evaluation), and some include generic skills (such as collaboration and questioning). Complicating this, popular views and definitions of the topic seem to change over time.
The link between aspects of computational

thinking ability and aspects of general problem-solving ability have been correlated through recent research comparing the results of a computational thinking test to standard psychometric tests (Román-González, Pérez-González, & Jiménez-Fernández, 2017). However, this research focused on only one narrow view of computational thinking. Which components of computational thinking are most useful for learners and what instructional approaches to teach computational thinking are most effective is still unproven.

Despite this uncertainty of the impact of computational thinking or its components (Tedre & Denning, 2016; Curzon, Bell, Waite, & Dorling, 2019), there are a wide range of resources and approaches for inclusion of computational thinking in teaching and learning of programming (Falkner & Vivian, 2015). Based on a knowledge of computational thinking, educators are advised by some that they can draw out points of learning in existing lesson activities or incorporate specific computational thinking activities in schemes of work teaching programming (Curzon et al., 2019).

Any ongoing research on computing pedagogy requires a review of what computational thinking is viewed as at the considered point in time, how it impacts teaching and learning, and its role within programming pedagogy.

4. Conclusion

4. Conclusion

In this report, we have described research that teachers can use to support their teaching of programming in schools. We have covered a range of classroom strategies such as reading code and pair programming, contexts in which programming may be taught, and how to support students. This is a substantial research area, so there may be some omissions. We have focused on topics and strategies that are particularly applicable to teachers in the classroom, based on our experience.

If you’re a teacher, you will probably already use a toolkit of approaches to support your teaching; programming is no different and this review may help you with your choice of instructional strategy, and selecting different techniques according to the needs of your learners to ensure all students make progress.

Robust pedagogical content knowledge for programming in schools is only just starting to emerge, unlike other subjects where "how to teach it" has been more widely researched. Over time, we hope to see more evidence emerge for the approaches outlined here as we move towards an even more informed and evidence-based view of "what works in computing".

References

Armoni, M. (2013). On Teaching Abstraction in Computer Science to Novices. *Journal of Computers in Mathematics and Science Teaching, 32*(3), 265–284. **https://www.learntechlib.org/p/41271/**

Barnes, J., Hoover, A. K., Fatehi, B., Moreno-León, J., Smith, G., & Harteveld, C. (2017). Exploring emerging design patterns in student-made climate change games. *Proceedings of the 12th International Conference on the Foundations of Digital Games, 64*, 1–6. **https://doi.org/10.1145/3102071.3116224**

Barr, V., & Stephenson, C. (2011). Bringing computational thinking to K-12: what is involved and what is the role of the computer science education community? *ACM Inroads, 2*(1), 48–54. **https://doi.org/10.1145/1929887.1929905**

Bau, D., Gray, J., Kelleher, C., Sheldon, J., & Turbak, F. (2017). Learnable programming: Blocks and beyond. *Communications of the ACM*, 60(6), 72–80. **https://doi.org/10.1145/3015455**

Beck, I. L., McKeown, M. G., & Kucan, L. (2013). *Bringing words to life: Robust vocabulary instruction*. Guilford Press.

Beck, W., Thomas, S. R., Drake, J., East, J. P., & Wallingford, E. (1996). Pattern Based Programming Instruction. *1996 Annual Conference Proceedings*, 1.349.1–1.349.10. **https://peer.asee.org/6228**

Bell, T., Alexander, J., Freeman, I., & Grimley, M. (2009). Computer science unplugged: School students doing real computing without computers. *The New Zealand Journal of Applied Computing and Information Technology, 13*(1), 20–29.

Bennedsen, J., & Caspersen, M. E. (2005). Revealing the programming process. *ACM SIGCSE Bulletin, 37*(1), 186–190. **https://doi.org/10.1145/1047124.1047413**

Benton, L., Hoyles, C., Kalas, I., & Noss, R. (2017). Bridging primary programming and mathematics: Some findings of design research in England. *Digital Experiences in Mathematics Education, 3*(2), 115–138. **https://link.springer.com/article/10.1007/s40751-017-0028-x**

Berry, M., Woollard, J., Hughes, P., Chippendale, J., Ross, Z., & Waite, J. (2015). *Barefoot computing resources*. Available at **https://www.barefootcomputing.org/primary-computing-resources** (accessed 23 September 2021)

Bers, M., Flannery, L., Kazakoff, E. R., & Sullivan, A. (2014). Computational thinking and tinkering: Exploration of an early childhood robotics curriculum. *Computers & Education, 72*, 145–157. **https://doi.org/10.1016/j.compedu.2013.10.020**

Brennan, K., Balch, C., & Chung, M. (2014). *Creative computing*. Harvard Graduate School of Education. Available at

https://creativecomputing.gse.harvard.edu/guide/curriculum.html (accessed 6 October 2021)

Brennan, K., & Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. *Proceedings of the 2012 Annual Meeting of the American Educational Research Association*. Available at **https://web.media.mit.edu/~kbrennan/files/Brennan_Resnick_AERA2012_CT.pdf** (accessed 6 October 2021)

Buchholz, B., Shively, K., Peppler, K., & Wohlwend, K. (2014). Hands On, Hands Off: Gendered Access in Crafting and Electronics Practices. *Mind, Culture, and Activity, 21*(4), 278–297. **https://doi.org/10.1080/10749039.2014.939762**

Busjahn, T., Bednarik, R., Begel, A., Crosby, M., Paterson, J. H., Schulte, C., Sharif, B., & Tamm, S. (2015). Eye movements in code reading: relaxing the linear order. *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension (ICPC '15)*, pp.255–265. **https://doi.org/10.1109/ICPC.2015.36**

Capraro, R. M., Capraro, M. M., & Rupley, W. H. (2010). Semantics and Syntax: A Theoretical Model for How Students May Build Mathematical Misunderstandings. *Journal of Mathematics Education, 3*(2), 58–66. Available at **http://educationforatoz.org/images/4.Robert_M._Capraro%2C_Mary_Margaret_Capraro%2C_William_H._Rupley.pdf** (accessed 23 November 2021)

Carlisle, J. F., Fleming, J. E., & Gudbrandsen, B. (2000). Incidental word learning in science classes. *Contemporary Educational Psychology, 25*(2), 184–211. **https://doi.org/10.1006/ceps.1998.1001**

Ching, C. C., & Kafai, Y. B. (2008). Peer pedagogy: Student collaboration and reflection in a learning- through-design project. *Teachers College Record, 110*(12), 2601–2632.

Clement, D. H., & Merriman, S. (1988). Componential developments in Logo programming and environments. In R. E. Mayer (Ed.), *Teaching and learning computer programming* (pp. 13–54). Erlbaum. **https://psycnet.apa.org/record/1988-98743-002**

Collins, A., Brown, J., & Newman, S. (1987). Cognitive Apprenticeship: Teaching the Craft of Reading, Writing, and Mathematics. *Technical Report No. 403*. Center for the Study of Reading, University of Illinois at Urbana-Champaign. Available at **https://www.ideals.illinois.edu/bitstream/handle/2142/17958/ctrstreadtechrepv01987i00403_opt.pdf** (accessed 5 November 2021)

Computer Science Teachers Association (2011). *Computational Thinking Teacher Resources 2nd Edition*. Available at **https://cdn.iste.org/www-root/2020-10/ISTE_CT_Teacher_Resources_2ed.pdf?_ga=2.156427001.378398415.1633422919-869434791.1633422919** (accessed 5 October 2021)

Cooper, J. L., & Robinson, P. (2000). Getting started: informal small-group strategies in large classes. *New Directions for Teaching & Learning, 2000*(81), 17–24. **https://doi.org/10.1002/tl.8102**

Cooper, S., Dann, W., & Pausch, R. (2000). Alice: a 3-D tool for introductory programming concepts. *Journal of Computing Sciences in Colleges, 15*(5), 107–116. Available at **https://dl.acm.org/doi/10.5555/364133.364161** (accessed 6 October 2021)

Cremin, T., & Baker, S. (2010). Exploring teacher-writer identities in the classroom: Conceptualising the struggle. *English Teaching: Practice and Critique, 9*(3), 8–25. Available at **https://edlinked.soe.waikato.ac.nz/journal/files/etpc/files/2010v9n3art1.pdf** (accessed 29 October 2021)

Crouch, C. H., & Mazur, E. (2001). Peer instruction: Ten years of experience and results. *American Journal of Physics, 69*(9), 970–977. **https://doi.org/10.1119/1.1374249**

Cummings, K., & Roberts, S. G. (2008). A study of peer instruction methods with high school physics students. *AIP Conference Proceedings, 1064*(1), 103–106. **https://doi.org/10.1063/1.3021227**

Curzon, P. (2013). cs4fn and computational thinking unplugged. *Proceedings of the 8th Workshop in Primary and Secondary Computing Education*, pp.47–50. **https://doi.org/10.1145/2532748.2611263**

Curzon, P., Bell, T., Waite, J., & Dorling, M. (2019). Computational thinking. In S. A. Fincher & A. V. Robins (Eds.), *The Cambridge Handbook of Computing Education Research* (pp.513–546). Cambridge University Press. Available at **https://qmro.qmul.ac.uk/xmlui/handle/123456789/57010** (accessed 6 October 2021)

Curzon, P., & McOwan, P. W. (2017). *The power of computational thinking: Games, magic and puzzles to help you become a computational thinker*. World Scientific.

Curzon, P., Waite, J., Maton, K., & Donohue, J. (2020). Using semantic waves to analyse the effectiveness of unplugged computing activities. *Proceedings of the 15th Workshop on Primary and Secondary Computing Education, 18*, 1–10. **https://doi.org/10.1145/3421590.3421606**

Cutts, Q., Esper, S., Fecho, M., Foster, S. R., & Simon, B. (2012). The abstraction transition taxonomy: developing desired learning outcomes through the lens of situated cognition. *Proceedings of the 9th Annual International Conference on International Computing Education Research*, pp.63–70. **https://doi.org/10.1145/2361276.2361290**

Deci, E. L. (1971). Effects of externally mediated rewards on intrinsic motivation. *Journal of Personality and Social Psychology, 18*(1), 105–115. **https://doi.org/10.1037/h0030644**

Deci, E., & Ryan, R. M. (1985). *Intrinsic motivation and self-determination in human behavior.* Springer Science & Business Media.

Deci, E. L., & Ryan, R. M. (2000). The "what?" and "why?" of goal pursuits:

Human needs and the self-determination of behavior. *Psychological Inquiry, 11*(4), 227–268. **https://doi.org/10.1207/S15327965PLI1104_01**

Denner, J., Green, E., & Campe, S. (2021). Learning to program in middle school: How pair programming helps and hinders intrepid exploration. *Journal of the Learning Sciences*, in press **https://doi.org/10.1080/10508406.2021.1939028**

Denner, J., Werner, L., Campe, S., & Ortiz, E. (2014). Pair programming: Under what conditions is it advantageous for middle school students? *Journal of Research on Technology in Education, 46*(3), 277–296. **https://doi.org/10.1080/15391523.2014.888272**

Diethelm, I., & Goschler, J. (2015). Questions on spoken language and terminology for teaching computer science. *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, pp.21–26. **https://doi.org/10.1145/2729094.2742600**

Donaldson, P., & Cutts, Q. (2018). Flexible low-cost activities to develop novice code comprehension skills in schools. *Proceedings of the 13th Workshop on Primary and Secondary Computing Education*, p.19. **https://doi.org/10.1145/3265757.3265776**

Dreyfus, S. E., & Dreyfus, H. L. (1980). *A five-stage model of the mental activities involved in directed skill acquisition*. California University Berkeley Operations Research Center. Available at **https://apps.dtic.mil/sti/citations/ADA084551** (accessed 6 October 2021)

Du, Y., Luxton-Reilly, A., & Denny, P. (2020). A Review of Research on Parsons Problems. *Proceedings of the Twenty-Second Australasian Computing Education Conference (ACE'20)*, pp.195–202. **https://doi.org/10.1145/3373165.3373187**

Du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research, 2*(1), 57–73. **https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9**

Dunston, P. J., & Tyminski, A. M. (2013). What's the Big Deal about Vocabulary? *Mathematics Teaching in the Middle School, 1*, 38–45. **http://www.jstor.org/stable/10.5951/mathteacmiddscho.19.1.0038**

Dwyer, H., Hill, C., Carpenter, S., Harlow, D., & Franklin, D. (2014). Identifying elementary students' pre- instructional ability to develop algorithms and step-by-step instructions. *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, pp.511–516. **https://doi.org/10.1145/2538862.2538905**

Dwyer, H., Hill, C., Hansen, A., Iveland, A., Franklin, D., & Harlow, D. (2015). Fourth Grade Students Reading Block-Based Programs: Predictions, Visual Cues, and Affordances. *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, pp.111–119. **https://doi.org/10.1145/2787622.2787729**

Edelson, D. C., Gordin, D. N., & Pea, R. D. (1999). Addressing the Challenges of Inquiry-

Based Learning Through Technology and Curriculum Design. *Journal of the Learning Sciences, 8*(3–4), 391–450. **https://doi.org/10.1080/10508406.1999.9672075**

Ericson, B. J., Foley, J. D., & Rick, J. (2018). Evaluating the Efficiency and Effectiveness of Adaptive Parsons Problems. *Proceedings of the 2018 ACM Conference on International Computing Education Research (ICER '18)*, pp.60–68. **https://doi.org/10.1145/3230977.3231000**

Ericson, B. J., Margulieux, L. E., & Rick, J. (2017). Solving Parson's problems versus fixing and writing code. *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*, pp.20–29. **https://doi.org/10.1145/3141880.3141895**

Ericson, B., McCall, A., & Cunningham, K. (2019). Investigating the Affect and Effect of Adaptive Parsons Problems. *Proceedings of the 19th Koli Calling International Conference on Computing Education Research*, pp.1–10. **https://doi.org/10.1145/3364510.3364524**

Espin, C. A., & Foegen, A. (1996). Validity of general outcome measures for predicting secondary students' performance on content-area tasks. *Exceptional Children, 62*, 497–514. **https://doi.org/10.1177/001440299606200602**

Falkner, K., & Vivian, R. (2015). A review of computer science resources for learning and teaching with K-12 computing curricula: An Australian case study. *Computer Science Education, 25*, 390–429. **https://doi.org/10.1080/08993408.2016.1140410**

Franklin, D., Hill, C., Dwyer, H. A., Hansen, A. K., Iveland, A., & Harlow, D. B. (2016). Initialization in Scratch: Seeking Knowledge Transfer. *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pp.217–222. **https://doi.org/10.1145/2839509.2844569**

Franklin, D., Skifstad, G., Rolock, R., Mehrotra, I., Ding, V., Hansen, A., Weintrop, D., & Harlow, D. (2017). Using Upper-Elementary Student Performance to Understand Conceptual Sequencing in a Blocks-based Curriculum. *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pp.231–236. **https://doi.org/10.1145/3017680.3017760**

Fraser, N. (2015). Ten things we've learned from Blockly. *IEEE Blocks and Beyond Workshop (Blocks and Beyond) 2015*, pp.49–50. **https://doi.org/10.1109/BLOCKS.2015.7369000**

Gal-Ezer, J., & Zur, E. (2004). The efficiency of algorithms—misconceptions. *Computers & Education, 42*(3), 215–226. **https://doi.org/10.1016/j.compedu.2003.07.004**

Garneli, V., Giannakos, M. N., & Chorianopoulos, K. (2015). Computing education in K-12 schools: A review of the literature. *2015 IEEE Global Engineering Education Conference (EDUCON)*, pp.543–551. **https://doi.org/10.1109/EDUCON.2015.7096023**

Griffin, J., Pirmann, T., & Gray, B. (2016). Two Teachers, Two Perspectives on CS Principles. *Proceedings of the 47th ACM Technical Symposium on Computing Science*

Education, pp.461–466. **https://doi.org/10.1145/2839509.2844630**

Grover, S., & Basu, S. (2017). Measuring Student Learning in Introductory Block-Based Programming: Examining Misconceptions of Loops, Variables, and Boolean Logic. *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pp. 267–272. **https://doi.org/10.1145/3017680.3017723**

Grover, S., & Pea, R. (2013). Using a discourse-intensive pedagogy and android's app inventor for introducing computational concepts to middle school students. *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, pp.723–728. **https://doi.org/10.1145/2445196.2445404**

Grover, S., Pea, R., & Cooper, S. (2015). Designing for deeper learning in a blended computer science course for middle school students. *Computer Science Education, 25*, 199–237. **https://doi.org/10.1080/08993408.2015.1033142**

Gujberova, M., & Kalas, I. (2013). Designing productive gradations of tasks in primary programming education. *Proceedings of the 8th Workshop in Primary and Secondary Computing Education*, pp.108–117. **https://dl.acm.org/doi/10.1145/2532748.2532750**

Hammond, J., & Gibbons, P. (2001). What is Scaffolding? In J. Hammond (Ed.), *Scaffolding: Teaching and learning in language and literacy education*, pp.1–14. Primary English Teaching Association. Available at **https://eric.ed.gov/?id=ED45644**7 (accessed 15 October 2021)

Hanks, B., Fitzgerald, S., McCauley, R., Murphy, L., & Zander, C. (2011). Pair programming in education: a literature review. *Computer Science Education, 21*(2), 135–173. **https://doi.org/10.1080/08993408.2011.579808**

Hansen, A., Hansen, E., Dwyer, H., Harlow, D., & Franklin, D. (2016). Differentiating for Diversity: Using Universal Design for Learning in Elementary Computer Science Education. *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pp.376–381.

Haynes, C. C., & Ericson, B. J. (2021). Problem-Solving Efficiency and Cognitive Load for Adaptive Parsons Problems vs. Writing the Equivalent Code. *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems,* 60. **https://doi.org/10.1145/3411764.3445292**

Hermans, F., Swidan, A., Aivaloglou, E., & Smit, M. (2018). Thinking out of the box: comparing metaphors for variables in programming education. *Proceedings of the 13th Workshop on Primary and Secondary Computing Education*, pp.1–8. **https://doi.org/10.1145/3265757.3265765**

Hodges, S., Sentance, S., Finney, J., & Ball, T. (2020). Physical Computing: A Key Element of Modern Computer Science Education. *Computer, 53*(4), 20–30. **https://doi.org/10.1109/MC.2019.2935058**

Hubwieser, P., Armoni, M., Giannakos, M. N., & Mittermeir, R. T. (2014). Perspectives and visions of computer science education in primary and secondary (K-12) schools. *ACM Transactions on Computing Education, 14*(2), 7. **https://doi.org/10.1145/2602482**

Jayathirtha, G., & Kafai, Y. (2021). Notional Machines in a Semester-long Introductory Physical Computing High School Unit. *Proceedings of the 17th ACM Conference on International Computing Education Research*, pp.448–449. **https://doi.org/10.1145/3446871.3469796**

Kafai, Y. B., & Burke, Q. (2015). Constructionist gaming: Understanding the benefits of making games for learning. *Educational Psychologist, 50*(4), 313–334. **https://doi.org/10.1080/00461520.2015.1124022**

Kafai, Y. B., Fields, D. A., Lui, D. A., Walker, J. T., Shaw, M. S., Jayathirtha, G., Nakajima, T. M., Goode, J., & Giang, M. T. (2019). Stitching the Loop with Electronic Textiles: Promoting Equity in High School Students' Competencies and Perceptions of Computer Science. *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pp.1176–1182. **https://doi.org/10.1145/3287324.3287426**

Kafai, Y. B., Lee, E., Searle, K., Fields, D., Kaplan, E., & Lui, D. (2014). A crafts-oriented approach to computing in high school: Introducing computational concepts, practices, and perspectives with electronic textiles. *ACM Transactions on Computing Education, 14*(1), 1. **https://doi.org/10.1145/2576874**

Kafai, Y. B., & Resnick, M. (1996). *Constructionism in Practice: Designing, Thinking, and Learning in a Digital World*. Lawrence Erlbaum Associates.

Kalelioglu, F., & Sentance, S. (2020). Teaching with physical computing in school: the case of the micro:bit. *Education Information Technology, 25*, 2577–2603. **https://doi.org/10.1007/s10639-019-10080-8**

Kastl, P., Kiesmüller, U., & Romeike, R. (2016). Starting out with Projects: Experiences with Agile Software Development in High Schools. *Proceedings of the 11th Workshop in Primary and Secondary Computing Education*, pp.60–65. **https://doi.org/10.1145/2978249.2978257**

Katai, Z., Toth, L., & Adorjani, A. K. (2014). Multi-Sensory Informatics Education. *Informatics in Education, 13*(2), 225–240. **https://www.learntechlib.org/p/158137/**

Kölling, M. (2015). Lessons from the Design of Three Educational Programming Environments: Blue, BlueJ and Greenfoot. *International Journal of People-Oriented Programming, 4*(1), 5–32. **https://doi.org/10.4018/IJPOP.2015010102**

Kothiyal, A., Murthy, S., & Iyer. S. (2014). Think-pair-share in a large CS1 class: does learning really happen? *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education (ITiCSE '14)*, pp.51–56. **https://doi.org/10.1145/2591708.2591739**

Lee, I., Martin, F., Denner, J., Coulter, B., Allan, W., Erickson, J., Malyn-Smith, J., & Werner. L. (2011). Computational thinking for youth in practice. *ACM Inroads, 2*(1), 32–37. **https://doi.org/10.1145/1929887.1929902**

Leppan, R., Cilliers, C., & Taljaard, M. (2007). Supporting CS1 with a program beacon recognition tool. *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)*, pp.66–75. **https://doi.org/10.1145/1292491.1292499**

Lewis, C. M. (2010). How programming environment shapes perception, learning and goals: Logo vs. Scratch. *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, pp.346–350. **https://doi.org/10.1145/1734263.1734383**

Lewis, C. M. (2011). Is pair programming more effective than other forms of collaboration for young students? *Computer Science Education, 21*(2), 105–134. **https://doi.org/10.1080/08993408.2011.579805**

Lewis, C. M., & Shah, N. (2015). How equity and inequity can emerge in pair programming. *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, pp.41–50. **https://doi.org/10.1145/2787622.2787716**

Liebenberg, J., Mentz, E., & Breed, B. (2012). Pair programming and secondary school girls' enjoyment of programming and the subject Information Technology (IT). *Computer Science Education, 22*(3), 219–236. **https://doi.org/10.1080/08993408.2012.713180**

Lister, R., Fidge, C., & Teague, D. (2009). Further Evidence of a Relationship Between Explaining, Tracing and Writing Skills in Introductory Programming. *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '09)*, pp.161–165. **https://doi.org/10.1145/1595496.1562930**

Lokkila, E., Rajala, T., Veerasamy, A., Enges-Pyykönen, P., Laakso, M. J., & Salakoski, T. (2016). How students' programming process differs from experts – a case study with a robot programming exercise. *EDULEARN16 Proceedings of the 8th International Conference on Education and New Learning Technologies*, pp.1555–1562. **http://dx.doi.org/10.21125/edulearn.2016.1308**

Lopez, M., Whalley, J., Robbins, P., & Lister, R. (2008). Relationships between reading, tracing and writing skills in introductory programming. *Proceedings of the fourth international workshop on computing education research*, pp.101–112. **https://doi.org/10.1145/1404520.1404531**

Luxton-Reilly, A., Albluwi, I., Becker, B. A., Giannakos, M., Kumar, A. N., Ott, L., Paterson, J., Scott, M. J., Sheard, J., & Szabo, C. (2018). A review of introductory programming research 2003–2017. *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, pp.342–343. **https://doi.org/10.1145/3197091.3205841**

Lye, S. Y., & Koh, J. H. L. (2014). Review on teaching and learning of computational thinking through programming: What is next for K-12? *Computers in Human Behavior, 41*, 51–61. **https://doi.org/10.1016/j.chb.2014.09.012**

Lyman, F. T. (1981). The responsive classroom discussion: The inclusion of all students. *Mainstreaming Digest, 109*, 113. Available at **https://www.scienceopen.com/document?vid=347f5de1-c4d9-46e9-a869-fc37cf19383b** (accessed 6 October 2021)

Major, L., Kyriacou, T., & Brereton, O. P. (2012). Systematic literature review: Teaching novices programming using robots. *IET Software, 6*(6), 502–513. **https://doi.org/10.1049/iet-sen.2011.0125**

Malan, D. J., & Leitner, H. H. (2007). Scratch for budding computer scientists. *ACM SIGCSE Bulletin, 39*(1), 223–227. **https://doi.org/10.1145/1227504.1227388**

Maloney, J. H., Peppler, K., Kafai, Y., Resnick, M., & Rusk, N. (2008). Programming by choice: Urban youth learning programming with Scratch. *ACM SIGCSE Bulletin, 40*(1), 367–371. **https://doi.org/10.1145/1352322.1352260**

Margulieux, L. E., & Catrambone, R. (2016). Improving problem solving with subgoal labels in expository text and worked examples. *Learning and Instruction, 42*, 58–71. **https://doi.org/10.1016/j.learninstruc.2015.12.002**

Margulieux, L. E., Morrison, B. B., Franke, B., & Ramilison, H. (2020). Effect of Implementing Subgoals in Code.org's Intro to Programming Unit in Computer Science Principles. *ACM Transactions on Computing Education, 20*(4), 1–24. **https://doi.org/10.1145/3415594**

Maton, K. (2013). Making semantic waves: a key to cumulative knowledge-building. *Linguistics and Education, 24*, 8–22.

Maton, K., Hood, S., & Shay, S. (2016). *Knowledge-building: educational studies in legitimation code theory*. Routledge.

Mayer, R. E. (2004). Should there be a three-strikes rule against pure discovery learning? *American Psychologist, 59*(1), 14–19. **https://doi.org/10.1037/0003-066x.59.1.14**

McDowell, C., Werner, L., Bullock, H., & Fernald, J. (2006). Pair programming improves student retention, confidence, and program quality. *Communications of the ACM, 49*(8), 90–95. **https://doi.org/10.1145/1145287.1145293**

Meerbaum-Salant, O., Armoni, M., Ben-Ari, M. (2013). Learning computer science concepts with Scratch. *Computer Science Education, 23*, 239–264. **https://doi.org/10.1080/08993408.2013.832022**

Missiroli, M., Russo, D., & Ciancarini, P. (2016). Learning Agile software development in high school: an investigation. *Proceedings of the 38th International Conference on Software Engineering Companion*, pp.293–302. **https://ieeexplore.ieee.org/document/7883313**

Morrison, B. B., Margulieux, L. E., Ericson, B., & Guzdial, M. (2016). Subgoals help students solve Parsons problems. *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pp.42–47. **https://doi.org/10.1145/2839509.2844617**

Morrison, B. B., Quinn, B., Bradley, S., Buffardi, K., Harrington, B., Hu, H., Kallia, M., McNeill, F., Ola, O., Parker, M. C., Rosato, J., & Waite. J. (2021). Chronicling the Evidence for Broadening Participation. *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 2 (ITiCSE 2021)*, pp.601–602. **https://doi.org/10.1145/3456565.3461441**

Muller, O. (2005). Pattern oriented instruction and the enhancement of analogical reasoning. *Proceedings of the first international workshop on computing education research (ICER '05)*, pp.57–67. **https://doi.org/10.1145/1089786.1089792**

Muller, O., Ginat, D., & Haberman, B. (2007). Pattern-oriented instruction and its influence on problem decomposition and solution construction. *Proceedings of the 12th annual SIGCSE conference on innovation and technology in computer science education (ITiCSE '07)*, pp.151–155. **https://doi.org/10.1145/1268784.1268830**

Muller, O., Haberman, B., & Averbuch, H. (2004). (An almost) pedagogical pattern for pattern-based problem-solving instruction. *Proceedings of the 9th annual SIGCSE conference on innovation and technology in computer science education (ITiCSE '04)*, pp.102–106. **https://doi.org/10.1145/1007996.1008025**

Nagy, W. E. (1988). *Teaching vocabulary to improve reading comprehension*. ERIC. Available at **https://eric.ed.gov/?id=ED29847**1 (accessed 5 October 2021)

Nuutila, E., Törmä, S., & Malmi, L. (2005). PBL and computer programming—the seven steps method with adaptations. *Computer Science Education, 15*(2), 123–142. **https://doi.org/10.1080/08993400500150788**

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. Basic Books, Inc.

Parsons, D., & Haden, P. (2006). Parson's Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses. *Proceedings of the 8th Australasian Conference on Computing Education (ACE '06)*, pp.157–163.

Passey, D. (2014). Intergenerational learning practices—Digital leaders in schools. *Education and Information Technologies, 19*(3), 473–494. **https://doi.org/10.1007/s10639-014-9322-z**

Pea, R. D. (1986). Language-independent conceptual "bugs" in novice programming. *Journal of Educational Computing Research, 2*, 25–36. **https://doi.org/10.2190/689T-1R2A-X4W4-29J2**

Perrenet, J., & Kaasenbrood, E. (2006). Levels of abstraction in students' understanding of the concept of algorithm: the qualitative perspective. *ACM SIGCSE Bulletin, 38*(3), 270–274. **https://doi.org/10.1145/1140124.1140196**

Porter, L., & Simon, B. (2013). Retaining nearly one-third more majors with a trio of instructional best practices in CS1. *Proceedings of the 44th ACM technical symposium on computer science education*, pp.165–170. **https://doi.org/10.1145/2445196.2445248**

Price, T. W., & Barnes, T. (2015). Comparing Textual and Block Interfaces in a Novice Programming Environment. *Proceedings of the Eleventh Annual International Conference on International Computing Education Research (ICER '15)*, pp.91–99. **https://doi.org/10.1145/2787622.2787712**

Przybylla, M., & Romeike, R. (2014). Physical computing in computer science education. *Proceedings of the 9th Workshop in Primary and Secondary Computing Education*, pp.136–137. **https://doi.org/10.1145/2670757.2670782**

Qian, Y., & Lehman, J. (2017). Students' misconceptions and other difficulties in introductory programming: a literature review. *ACM Transactions on Computing Education, 18*, 1. **https://doi.org/10.1145/3077618**

Repenning, A., Webb, D. C., Koh, K. H., Nickerson, H., Miller, S. B., Brand, C., Horses, I. H. M., Basawapatna, A., Gluck, F., Grover, R., Gutierrez, K., & Repenning, N. (2015). Scalable Game Design: A Strategy to Bring Systemic Computer Science Education to Schools through Game Design and Simulation Creation. *ACM Transactions on Computing Education, 15*(2), 1–31. **https://doi.org/10.1145/2700517**

Resnick, M., Silverman, B., Kafai, Y., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., & Silver, J. (2009). Scratch: Programming for all. *Communications of the ACM, 52*(11), 60–67. **https://doi.org/10.1145/1592761.1592779**

Rich, P. J., Browning, S. F., Perkins, M., Shoop, T., Yoshikawa, E., & Belikov, O. M. (2018). Coding in K-8: International Trends in Teaching Elementary/Primary Computing. *TechTrends, 63*, 311–329. **http://dx.doi.org/10.13140/RG.2.2.29782.14409/1**

Rich, K., Strickland, C., & Franklin, D. (2017). A Literature Review through the Lens of Computer Science Learning Goals Theorized and Explored in Research. *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pp.495–500. **https://doi.org/10.1145/3017680.3017772**

Robins, A., Rountree, J., Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education, 13*, 137–172. **https://doi.org/10.1076/csed.13.2.137.14200**

Rodriguez, B., Kennicutt, S., Rader, C., & Camp, T. (2017). Assessing Computational Thinking in CS Unplugged Activities. *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pp. 501–506. **https://doi.org/10.1145/3017680.3017779**

Román-González, M., Pérez-González, J.-C., & Jiménez-Fernández, C. (2017). Which cognitive abilities underlie computational thinking? Criterion validity of the

Computational Thinking Test. *Computers in Human Behavior, 72*, 678–691. **https://doi.org/10.1016/j.chb.2016.08.047**

Romeike, R., & Götte, T. (2012). Agile projects in high school computing education: emphasizing a learners' perspective. *Proceedings of the 7th Workshop in Primary and Secondary Computing Education (WiPSCE '12)*, pp.48–57. **https://doi.org/10.1145/2481449.2481461**

Rose, J. (2009). *Independent review of the primary curriculum*. Department for Children, Schools and Families. Available at **https://dera.ioe.ac.uk//30098/** (accessed 5 October 2021)

Rowe, M. B. (1986). Wait time: slowing down may be a way of speeding up! *Journal of Teacher Education, 37*(1), 43–50. **https://doi.org/10.1177/002248718603700110**

The Royal Society (2017). *After the reboot: computing education in UK schools*. The Royal Society. Available at **https://royalsociety.org/~/media/policy/projects/computing-education/computing-education-report.pdf** (accessed 5 October 2021)

Rubin, M. J. (2013). The effectiveness of live-coding to teach introductory programming. *Proceedings of the 44th ACM technical symposium on computer science education*, pp.651–656. **https://doi.org/10.1145/2445196.2445388**

Ruvalcaba, O., Werner, L., & Denner, J. (2016). Observations of Pair Programming: Variations in Collaboration Across Demographic Groups. *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pp.90–95. **https://doi.org/10.1145/2839509.2844558**

Salleh, N., Mendes, E., & Grundy, J. (2011). Empirical studies of pair programming for CS/SE teaching in higher education: A systematic literature review. *IEEE Transactions on Software Engineering, 37*(4), 509–525. **https://doi.org/10.1109/TSE.2010.59**

Sapir, E. (1921). *Language: An introduction to the study of speech*. Harcourt, Brace & World Inc.

Savery, J., & Duffy, T. (1995). Problem Based Learning: An Instructional Model and its Constructivist Framework. *Educational Technology, 35*(5), 31–38. **https://www.jstor.org/stable/44428296**

Schulte, C. (2008). Block Model: An Educational Model of Program Comprehension As a Tool for a Scholarly Approach to Teaching. *Proceedings of the Fourth International Workshop on Computing Education Research (ICER '08)*, pp.149–160. **https://doi.org/10.1145/1404520.1404535**

Schulte, C., Magenheim, J., Müller, K., & Budde, L. (2017). The design and exploration cycle as research and development framework in computing education. *2017 IEEE Global Engineering Education Conference (EDUCON)*, pp.867–876. **https://doi.org/10.1109/EDUCON.2017.7942950**

Seiter, L., & Foreman, B. (2013). Modeling the learning progressions of computational thinking of primary grade students.

Proceedings of the ninth annual international ACM conference on international computing education research, pp.59–66. **https://doi.org/10.1145/2493394.2493403**

Sentance, S., & Csizmadia, A. (2017). Computing in the curriculum: Challenges and strategies from a teacher's perspective. *Education and Information Technologies, 22*, 469–495. **https://doi.org/10.1007/s10639-016-9482-0**

Sentance, S., & Waite, J. (2017). PRIMM: Exploring pedagogical approaches for teaching text-based programming in school. *Proceedings of the 12th Workshop on Primary and Secondary Computing Education (WiPSCE '17)*, pp.113–114. **https://doi.org/10.1145/3137065.3137084**

Sentance, S., & Waite, J. (2021). Teachers' Perspectives on Talk in the Programming Classroom: Language as a Mediator. *Proceedings of the 17th ACM Conference on International Computing Education Research (ICER 2021)*, pp.266–280. **https://doi.org/10.1145/3446871.3469751**

Sentance, S., Waite, J., & Kallia, M. (2019). Teaching computer programming with PRIMM: a sociocultural perspective. *Computer Science Education, 29*(2–3), 136–176. **DOI: 10.1080/08993408.2019.1608781**

Šestáková, J. (2016). Case Study of Using Peer Instruction at Upper Secondary School. *Scientia in Educatione, 7*(2), 111–127. **https://doi.org/10.14712/18047106.298**

Shah, N., & Lewis, C. M. (2019). Amplifying and attenuating inequity in collaborative learning: Toward an analytical framework. *Cognition and Instruction, 37*(4), 423–452. **https://doi.org/10.1080/07370008.2019.1631825**

Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM, 29*, 850–858. **https://doi.org/10.1145/6592.6594**

Sorva, J. (2012). *Visual program simulation in introductory programming education*. PhD Thesis, Aalto University. Available at **https://aaltodoc.aalto.fi/handle/123456789/3534** (accessed 5 October 2021)

Sorva, J. (2018). Misconceptions and the Beginner Programmer. In S. Sentance, E. Barendsen, C. Schulte (Eds.), *Computer Science Education Perspectives Teaching Learning School*, p.171. Bloomsbury Publishing.

Statter, D., & Armoni, M. (2016). Teaching Abstract Thinking in Introduction to Computer Science for 7th Graders. *Proceedings of the 11th Workshop in Primary and Secondary Computing Education*, pp.80–83. **https://doi.org/10.1145/2978249.2978261**

Statter, D., & Armoni, M. (2017). Learning Abstraction in Computer Science: A Gender Perspective. *Proceedings of the 12th Workshop on Primary and Secondary Computing Education (WiPSCE'17)*, pp.5–14. **https://doi.org/10.1145/3137065.3137081**

Swartz, S. L., Klein, A. F., & Shook, R. E. (2001). *Interactive writing & interactive* *editing: Making connections between writing and reading*. Dominie Press, Inc.

Taylor, C., Spacco, J., Bunde, D. P., Petersen, A., Liao, S. N., & Porter, L. (2018). A multi-institution exploration of peer instruction in practice. *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE 2018)*, pp.308–313. **https://doi.org/10.1145/3197091.3197144**

Teague, D., & Lister, R. (2014a). Programming: reading, writing and reversing. *Proceedings of the 2014 conference on innovation & technology in computer science education*, pp.285–290. **https://doi.org/10.1145/2591708.2591712**

Teague, D., & Lister, R. (2014b). Longitudinal think aloud study of a novice programmer. *Proceedings of the Sixteenth Australasian Computing Education Conference*, pp.41–50.

Tedre, M., & Denning, P. J. (2016). The long quest for computational thinking. *Proceedings of the 16th Koli Calling Conference on Computing Education Research*, pp.24–27. **https://doi.org/10.1145/2999541.2999542**

Thies, R., & Vahrenhold, J. (2016). Back to school: computer science unplugged in the wild. *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, pp.118–123. **https://doi.org/10.1145/2899415.2899442**

Thomas, J. W. (2000). *A review of research on project-based learning.* Autodesk Foundation. Available at **https://www.asec.purdue.edu/lct/HBCU/documents/AReviewofResearchofProject-BasedLearning.pdf** (accessed 5 October 2021)

Toh, L. P. E., Causo, A., Tzuo, P., Chen, I., & Yeo, S. H. (2016). A Review on the Use of Robots in Education and Young Children. *Journal of Educational Technology & Society, 19*(2), 148–163. **http://www.jstor.org/stable/jeductechsoci.19.2.148**

Ubiquity staff (2007). An Interview with Peter Denning on the great principles of computing. *Ubiquity, 2007*(June), 1. **https://doi.org/10.1145/1276162.1276163**

Umapathy, K., & Ritzhaupt, A. D. (2017). A meta-analysis of pair-programming in computer programming courses: Implications for educational practice. *ACM Transactions on Computing Education, 17*(4), 16. **https://doi.org/10.1145/2996201**

Veerasamy, A. K., D'Souza, D., & Laakso, M.-J. (2016). Identifying Novice Student Programming Misconceptions and Errors From Summative Assessments. *Journal of Educational Technology Systems, 45*(1), 50–73. **https://doi.org/10.1177/0047239515627263**

Venables, A., Tan, G., & Lister, R. (2009). A closer look at tracing, explaining and code writing skills in the novice programmer. *Proceedings of the fifth international workshop on computing education research*, pp.117–128. **https://doi.org/10.1145/1584322.1584336**

Waite, J. (2017). *Pedagogy in Teaching Computer Science in schools: A Literature Review (After The Reboot: Computing Education in UK Schools)*. Available at **https://royalsociety.org/-/media/policy/projects/computing-education/literature-review-pedagogy-in-teaching.pdf** (accessed 5 October 2021)

Waite, J., Curzon, P., Marsh, D., Sentance, S., & Hawden-Bennett, A. (2018). Abstraction in action: K-5 teachers' uses of levels of abstraction, particularly the design level, in teaching programming. *International Journal of Computer Science Education in Schools, 2*(1), 14–40. **https://doi.org/10.21585/ijcses.v2i1.23**

Waite, J., Curzon, P., Marsh, W., & Sentance, S. (2020). Difficulties with design: The challenges of teaching design in K-5 programming. *Computers and Education, 150*, 103838. **https://doi.org/10.1016/j.compedu.2020.103838**

Waite, J., & Liebe, C. (2021). Computer Science Student-Centered Instructional Continuum. *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE '21)*, p.1246. **https://doi.org/10.1145/3408877.3439591**

Waite, J., Maton, K., Curzon, P., & Tuttiett, L. (2019). Unplugged computing and semantic waves: Analysing crazy characters. *Proceedings of the UK and Ireland Computing Education Research Conference (UKICER)*, pp.1–7. **https://doi.org/10.1145/3351287.3351291**

Webb, D. C., Repenning, A., & Koh, K. H. (2012). Toward an emergent theory of broadening participation in computer science education. *Proceedings of the 43rd ACM technical symposium on computer science education*, pp.173–178. **https://doi.org/10.1145/2157136.2157191**

Weinman, N., Fox, A., & Hearst, M. (2020). Exploring Challenging Variations of Parsons Problems. *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20)*, p.1349. **https://doi.org/10.1145/3328778.3372639**

Weinman, N., Fox, A., & Hearst, M. A. (2021). Improving Instruction of Programming Patterns with Faded Parsons Problems. *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems, 53*. **https://doi.org/10.1145/3411764.3445228**

Weintrop, D., Killen, H., Munzar, T., & Franke, B. (2019). Block-based Comprehension: Exploring and Explaining Student Outcomes from a Read-only Block-based Exam. *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*, pp.1218–1224. **https://doi.org/10.1145/3287324.3287348**

Weintrop, D., & Wilensky, U. (2017). Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms. *ACM Transactions on Computing Education, 18*(1), 3. **https://doi.org/10.1145/3089799**

Werner, L., Denner, J., Campe, S., Ortiz, E., DeLay, D., Hartl, A. C., & Laursen, B. (2013). Pair programming for middle school students: does friendship influence academic outcomes? *Proceedings of the 44th ACM technical symposium on computer science education*, pp.421–426. **https://doi.org/10.1145/2445196.2445322**

Wing, J. (2008). Computational thinking and thinking about computing. *Philosophical Transactions of The Royal Society A, 366*, 3717–3725. **https://doi.org/10.1098/rsta.2008.0118**

Zakaria, Z., Vandenberg, J., Tsan, J., Boulden, D. C., Lynch, C. F., Boyer, K. E., & Wiebe, E. N. (2021). Two-Computer Pair Programming: Exploring a Feedback Intervention to Improve Collaborative Talk in Elementary Students. *Computer Science Education*, 1–28. **https://www.tandfonline.com/doi/abs/10.1080/08993408.2021.1877987**

Zingaro, D. (2014). Peer instruction contributes to self-efficacy in CS1. *Proceedings of the 45th ACM technical symposium on computer science education*, pp.373–378. **https://doi.org/10.1145/2538862.2538878**

Zingaro, D., & Porter, L. (2015). Tracking student learning from class to exam using isomorphic questions. *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pp.356–361. **https://doi.org/10.1145/2676723.2677239**