International Journal of Computer Science Education in Schools, Jan 2018, Vol. 2, No. 1
ISSN 2513-8359

Abstraction in action: K-5 teachers' uses of levels of abstraction, particularly the design level, in teaching programming

Jane Lisa Waite[1 & 2]
Paul Curzon[1]
William Marsh[1]
Sue Sentance[2]
Alex Hadwen-Bennett[2]

[1]Queen Mary University of London
[2]King's College London

DOI: 10.21585/ijcses.v2i1.23

Abstract

Research indicates that understanding levels of abstraction (LOA) and being able to move between the levels is essential to programming success. For K-5 contexts LOA levels have been named: problem, design, code and running the code. In a qualitative exploratory study, five K-5 teachers were interviewed on their uses of LOA, particularly the design level, in teaching programming and other subjects. Using PCK elements to analyse responses, the teachers interviewed used design as an instructional strategy and for assessment. The teachers used design as an aide memoire and the expert teachers used design: as a contract for pair-programming; to work out what they needed to teach; for learners to annotate with code snippets (to transition across LOA); for learners to self-assess and to assess 'do-ability'. The teachers used planning in teaching writing to scaffold learning and promote self-regulation revealing their insight in student understanding. One issue was of the teachers' knowledge of terms including algorithm and code; a concept of 'emergent algorithms' is proposed. Findings from the study suggest design helps learners learn to program in the same way that planning helps learners learn to write and that LOA, particularly the design level, may provide an accessible exemplar of abstraction in action. Further work is needed to verify whether the study's results are generalisable more widely.

Keywords: design, abstraction, levels of abstraction, computational thinking, programming, K-5, algorithm

1 Introduction

Despite a lack of consensus on exactly what computational thinking is, proponents of computational thinking, surveys of computational thinking, and emerging curriculum frameworks all propose that *abstraction* forms a core component (Barr & Stephenson 2011; Seehorn et al. 2016; Kramer 2007; Selby et al. 2014; Wing & Barr 2011). Wing suggested "*The abstraction process, deciding what details we need to highlight and what details we can ignore, underlies computational thinking*" (Wing 2008, p.3718). Others have suggested an alternative focus with abstraction being related to reducing unnecessary detail (Csizmadia et al. 2015) or simplifying to create a generalisation (Barr & Stephenson 2011). New curricular and guidance incorporate abstraction, sometimes in computing and programming, sometimes embedded in other subjects, (Berry et al. 2015; Bienkowski et al. 2015) with ambitious potential and possibly exaggerated claims (Tedre & Denning 2016).

At a first glance at the English computing curriculum, one might think abstraction is not of significance until pupils turn eleven years old and start secondary school. It is only at this point a curriculum attainment target specifically states pupils should be able to "design and evaluate computational abstractions" (DfE 2013b, p.2).

The attainment targets for younger pupils do not specifically mention the term 'abstraction'. Instead, terms such as algorithm, decomposition, design and programming are used (DfE 2013a).

In University studies a practical use of abstraction, the levels of abstraction (LOA) hierarchy has been suggested to support novices learning to program, with levels of problem, object, program and execution (Perrenet et al. 2005; Perrenet & Kaasenbrood 2006). To situate the hierarchy in K-5 settings two of the levels have been renamed to support teacher and pupil understanding and a hierarchy of problem, design, code and running the code (Waite et al. 2016) is suggested. In this renamed hierarchy, the design includes the algorithms which are then implemented as code.

Whether K-5 teachers understand an algorithm is an abstraction, or that to create a design one needs to abstract, or that a program lies at a different LOA to an algorithm is not clear. Teachers may informally deal with abstraction without realizing it. They may model 'ignoring detail', articulating their thought processes of why they have included some features and not others. This may occur when: designing and writing a program; planning a story; creating maps in geography, and so on. Alternatively, teachers may introduce specific activities to help children practise moving between different LOA such as when adding sub-headings, summarising learning, creating visualizations, and using check-lists to help consolidate understanding.

In topics across the curriculum, teachers may notice where they already use and teach abstraction and then start to assess progress, set pupils' targets and plan specific learning experiences related to aspects of abstraction. However, is it not clear whether there is a widespread appreciation of abstraction and its role in learning, nor whether understanding and implementation of LOA is happening in schools overtly, inadvertently or not at all. Nor is it clear whether this is important or useful.

The contribution of this paper is to add to the body of knowledge in the teaching of abstraction to K-5 learners. This is done by investigating a specific aspect of abstraction, the LOA hierarchy, particularly the design level, as used by K-5 teachers in teaching programming and other subjects. Contributions include a renamed LOA hierarchy for use in K-5 programming contexts; a set of uses of LOA grouped by pedagogical content knowledge (PCK) elements (Magnusson et al. 1999); the concept of *emergent algorithms* to support teachers understanding; and recommendation that the LOA hierarchy may be a useful tool for improving and reviewing teaching and learning in programming. However, findings, due to the small number of participants, require further investigation to assess the extent to which they are generalisable.

2 Related Work

2.1 *University Computing*

Past studies have considered how to teach abstraction to learners in programming contexts, with most attention focusing on university students (Aharoni 2000; Cook et al. 2012; Cutts et al. 2012; Fuller et al. 2007; Hazzan 2002; Hazzan & Hadar 2005; Perrenet et al. 2005; Perrenet & Kaasenbrood 2006). Perrenet et al. (2005) developed a level of abstraction (LOA) model focused on explaining university students' thinking about algorithms with levels of problem, object, program and execution. The LOA model was proposed to help understand how novice programmers approached programming tasks and support them making progress in programming. Hazzan & Hadar (2005) developed the links between abstraction and learning mathematics, and abstraction and learning computer science, referring to a theme of reducing abstraction. In their work on Computer Science and Software Engineering, Hazzan & Kramer (2007) propose "question patterns" that teach abstraction through any subject. Questions included: comparing two representations of the system at different levels and explaining which is more abstract and why; ranking representations by levels of abstraction; and grouping representations by level. Cutts et al. (2012) looked at introductory programming courses for university students. They analysed peer instruction questions to reveal the 'talk' associated with programming, developing an Abstraction Transition (AT) taxonomy. It had three main levels: code; CS speak; and English. Nine transitions across the three levels were identified, and for each transition a how and why goal defined. Cutts et al. claim these 18 goals will develop students' programming. An example transition goal given by the study was "Given a technical description (CS Speak) of how to achieve a goal, choose code that will accomplish that goal" (Cutts et al. 2012, p.68). The authors plan future work with high school teachers and written English.

Building on Bloom's learning taxonomy (Bloom 1956), Fuller et al. suggest a computer science specific taxonomy particularly suited to the learning of programming, called the Matrix Taxonomy. The authors state it "better deals with modularity and increasing levels of abstraction" (Fuller et al. 2007, p.163) than other taxonomies. They incorporate abstraction by suggesting five activities: "design, model, refactor, debug and present" that "may easily be seen to involve extensive consideration of abstraction" (Fuller et al. 2007, p.166). It is an open question as to whether approaches based on experience of teaching higher education students such as abstraction themed question patterns (Hazzan & Kramer 2007), the LOA hierarchy (Perrenet et al. 2005; Perrenet & Kaasenbrood 2006) teaching abstraction guidelines (Armoni 2013), the AT taxonomy (Cutts et al. 2012) and the Matrix Taxonomy (Fuller et al. 2007) are useful for teaching *all* novice programmers. They may be best suited to older students who have elected and been selected to specialize in computer science. This is an important question for our work, concerned as it is with the teaching of younger, and a broader group of, students.

2.2 *High School Computing*

Taub et al. (2014) looked at how abstraction may be taught through the design of physics simulations with talented Israeli high school students. A psychology study in the US compared how Grade 7 pupils showed their understanding of complex systems (aquariums) compared to experts (Hmelo-Silver & Pfeffer 2004). Neither computational thinking nor abstraction is specifically mentioned, but the layers of understanding (abstractions) are finely studied. These studies indicate that the generic nature of abstraction is such that useful results may be found across disciplines.

Armoni (2013) stated that abstraction is hard to teach and that there are no validated methods to assess abstraction ability. However, she suggested a framework for teaching abstraction which forms a basis for further exploration of this area. Suggested criteria include being persistent and precise; using different vocabulary according to what level you are at; starting with the problem and moving towards execution; and making a clear distinction between each level. These guidelines were based on the level of abstraction hierarchy proposed by Perrenet et al. (2005) and Perrenet & Kaasenbrood (2006) which Armoni called the PGK hierarchy (from the authors names: Perrenet, Groote and Kassenbrood). Armoni renamed the object level the algorithm level to support understanding, explaining:

> *"we rename level 3 of the PGK-hierarchy as the algorithm level (instead of the object level). This might seem awkward, since this hierarchy is about the concept of an algorithm, and hence all its levels deal with an algorithm. However, the object level grasps what is commonly and popularly referred to as an "algorithm" – a solution which is not presented in a specific programming language." (Armoni 2013, p.273).*

The context for the creation of the guidelines and renaming of the PGK hierarchy was in studying and developing material for middle school (Grades 7-9) learners (ages 13 to 15) working with Scratch. This renaming is significant as it assumes that educators and learners at this level of learning are familiar with the term algorithm. Whether teachers of K-5 pupils are similarly familiar is yet to be identified. Statter & Armoni (2016) taught the PGK hierarchy using Armoni's framework to 119 Grade 7 pupils when programming in Scratch: namely the execution level, program level, algorithm and problem level. The experimental group attended more to the algorithm level, using more written and verbal descriptions, than the control group. They also reported that girls, as well as scoring better on CS knowledge tests and on the abstraction assessment, also tended to write better algorithm descriptions and use them more frequently than boys, in both the control and experimental group.

2.3 *Piaget and research in K-5 education*

An often-asked question is at what age can children learn from, or about, abstraction? Piaget's work is sometimes alluded to or directly cited, suggesting that he indicated that children cannot learn about abstraction until they reach an age and stage of development, formal operational, around the age of twelve. This related ceiling for abstraction is referenced by authors (Armoni 2012; Barendsen et al. 2015; Kramer 2007; Lister 2011; Mannila et al. 2014) with calls for a review of this assertion in the light of the current educational landscape (National Research Council 2011).

In Piaget's later work (Piaget & Campell 2001) on reflecting abstraction, he writes that children utilise the process of abstraction from as young as 18 months and that abstraction is used continuously when learning, "without end and especially without an absolute beginning" (Piaget & Campell 2001, p.306). This sentiment exhibits similarities with the SOLO taxonomy of spiral learning (Biggs & Collis 1982; Sheard et al. 2008). Piaget's later work is rarely cited and was translated only recently, some 25 years after its original writing. It seems this work is in opposition to earlier work and counters 'the age-related abstraction ceiling'.

Looking at the specific research of using abstraction in K-5 education, Syslo & Kwiatkowska (2014) directly addressed the assertion that young learners cannot learn from, or about, abstraction. The authors introduced children aged 5 to 12 to problems such as using graphs and the Konigsberg Bridge problem, classical maths themes such as the Tower of Hanoi and binary numbers. The researchers concluded that children were introduced "to some concepts in computing which are certainly abstract and demonstrate that they are capable to work with abstraction and to apply algorithmic (computational) thinking" (Syslo & Kwiatkowska 2014, p.104). Similarly, Gibson (2012; 2008) also used puzzles and problem based learning to introduce children to abstraction rich computer science and mathematical material that is normally associated with older pupils. He introduced formal mathematics and again graph theory, using physical objects, diagrams and verbal descriptions to further explain and scaffold learning. What is clear from these writers is there is an appetite to challenge an *age-dependent ceiling of abstraction*, and opportunities to build upon Piaget's later work on abstraction. This challenge is picked up and a practical classroom application of learning from, and about abstraction with K-5 learners, this time related to programming and the use of LOA, and particularly the design level is investigated in this study.

3 The Study

An explorative qualitative study of five K-5 teachers using semi-structured interviews (Cohen et al. 2011; Drever 1995) augmented with unplugged activities (Nind et al. 2016; Punch 2002; Seeratan & Mislevy 2009) was undertaken. A thematic qualitative data analysis approach was used to analyse the transcriptions (Kuckartz 2014) and PCK elements (Magnusson et al. 1999) were used to group the findings.

3.1 *Aims*

The study focused on one scenario for the teaching of abstraction, that of using LOA. The aim was to better understand the opportunities for use of the LOA hierarchy, particularly the design level, with K-5 teachers.

The research questions were:

RQ1: What are K-5 teachers' uses of LOA, particularly design, in programming and other subjects?

An additional research sub-question was:

RQ2: What common vocabulary do teachers use to describe LOA in programming?

3.2 *The levels of abstraction hierarchy at K-5*

Levels of abstraction has been interpreted as a hierarchy to enable teachers and learners to describe which level they are working at, rather than as a methodology for programming projects. For example, learners might be learning how to trace a program, and therefore be working at the code and running the code level. If teachers employ the Use, Modify, Create (Lee et al. 2011) approach to teach programming, during the Use phase when learners are running pre-prepared programs to find out what is possible, learners might focus on the task, code and running the code levels, but as they move to change the code in the Modify phase they might be shown the design for the code and be encouraged to modify this in line with their code changes. Alternatively, students might be learning how to use the agile methodology and so be working at the problem and design levels briefly before moving to the code and running the code levels, then rapidly returning to the problem and design levels. Evidence indicates that learners who are aware of the level they are working at and are given opportunity to become more competent at moving between those levels may become more effective programmers (Cutts et al. 2012; Statter & Armoni 2017; Statter & Armoni 2016).

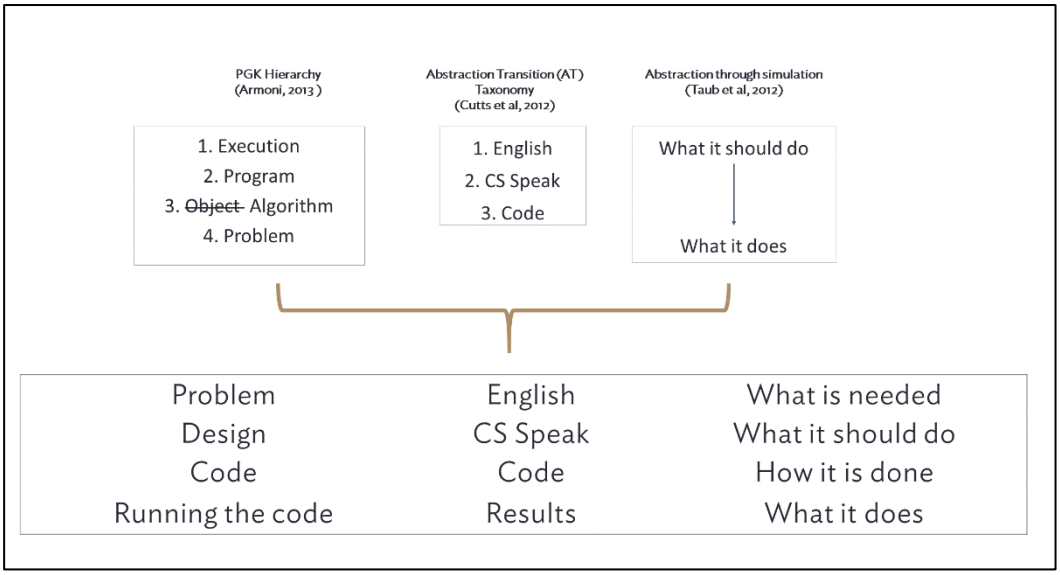

Figure 1 Aligned LOA hierarchy for K-5 programming

In previous work, to situate LOA in K-5 classroom activities, the PGK hierarchy was aligned to work by other authors and the object level renamed as the design level (Waite et al. 2016). This was to support understanding by K-5 teachers and their pupils, as shown in **Figure 1**.

Levels were named: problem, design, code and running the code. In doing this the term design is used as a noun, to depict the artefact and understanding of a problem at that level, rather than as the verb and to *'do design'*.

In K-5 programming projects design includes thinking about the user interface design as well as the algorithm and data structure design. K-5 pupils or teachers are not likely to distinguish between such components in the same way that specialist computer science educators teaching older pupils might. The boundary between the original hierarchy's problem level and the object level compared to the boundary between the K-5 task and design level, is much simplified. For example, requirements analysis might sit as part of problem level definition for older learners, however, for younger pupils, a simple task definition is suggested with further analysis and design being encompassed in the design artefacts. These caveats need to be further explored in ongoing research.

K-5 appropriate definitions for each level are provided as:

> **Problem:** The problem is a summary, high-level, written or verbal description of a project. This might be expected to be a short paragraph of one or two sentences defining what the requirements of the project are, e.g. "Create a quiz program for year 3 pupils to test them about their volcano topic".
>
> **Design:** The design is a thought, written, verbal or drawn depiction of the project. This is more detailed than the problem, but does not refer to the code that might be used to implement it. The level of detail is not prescribed, so it might be a sequence of simple steps depicted through a storyboard, a detailed flowchart or a verbal description of rules, e.g. "Keep a score and add 1 whenever they get an answer correct".
>
> **Code:** The code level is the code itself or any verbal or written reference to the programming language constructs and commands that might be used to implement the design. Terms used would include the commands themselves rather than a summary in English of what the code does. Here vocabulary is programming language specific, e.g. "Make a score variable, set score to 0 using an orange data block".
>
> **Running the code:** This is either the code running, or any reference to the output of the program, perhaps including reference to debugging the code and seeing what the outcome is, e.g. "When I ran the code, the variable score went from 0 to 1".

Table 1 Teachers participating in the research

Teacher	Gender	Class in pupil study	Number of years teaching	Year Group taught	School	Age range of pupils in school	Teaches other teachers computing	Experience teaching pupils computing
A	Female	C1	8	6	I	K-5 (5 - 11 years)	✓	extensive
B	Male	C2	10	5	I	K-5 (5 - 11 years)		limited
C	Male	C3, C4, C5	6	5	II	K-5 (5 - 11 years)	✓	extensive teaches computing across year group
D	Male	C6	2	2	III	K-1 (5 to 7 years)		limited
E	Female	C7	4	2	III	K-1 (5 to 7 years)		limited mostly route-based activities

3.3 *Why find out about common vocabulary?*

Common vocabulary is investigated, as according to Bloom (1956), the first stage of learning is to remember about *'a thing'*. With this in mind, K-5 teachers spend much time introducing new vocabulary. Sapir (1921) linked the development of a specific vocabulary with understanding concepts and emphasized the importance of ownership of vocabulary for understanding stating:

> *"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…Not until we own the symbol do we feel that we hold a key to the immediate knowledge or understanding of the concept." (Sapir 1921, para.15).*

Carlisle et al. (2000) suggested topical word learning, "contributed significantly to the improvement that students made on the applied problems" (Carlisle et al. 2000, p.207). Topical word learning is where reading about the subject matter is not the main source of initial exposure. The authors suggested learners with partial knowledge of words before a unit of learning were more likely to make more progress than those that had no knowledge, and that "depth of topical word knowledge" (Carlisle et al. 2000, p.184) had a significant impact on progress. Linking this to recommendations for precision (Armoni 2013) and focus on talk (Cutts et al. 2012; Grover & Pea 2013), ensuring vocabulary has the correct meaning, or at least a common shared understanding of meaning, is therefore important. Whether there is a shared vocabulary already being used for the terms used in the LOA hierarchy is therefore investigated.

4 Methodology

This study is one part of a longer research piece investigating abstraction, design and programming in K-5 teaching and learning. It involves one group of respondents in an early school based phase. Over 52 pupils and 5 teachers were interviewed on the use of design in programming and planning in other subjects. The teachers' responses are outlined in Section 5. The study was qualitative, using semi-structured interviews. Semi-structured interviews enable a deep exploration of teachers' experiences with freedom for the teachers to develop ideas and giving the interviewer option to adapt questions to explore emergent themes (Drever 1995). The interviews were augmented with unplugged activities to situate and engage discussion (Nind et al. 2016; Punch 2002; Seeratan & Mislevy 2009). A thematic qualitative data analysis approach was used to analyse the transcriptions (Kuckartz 2014) and PCK elements (Magnusson et al. 1999) were used to group the findings.

4.1 *Participants*

Five teachers from three schools participated in the study. Teachers volunteered to take part following direct emails to local primary schools; five schools were emailed, and three schools responded. The number of participants is small and the selection approach was limited so caution is needed for any generalised conclusion (Cohen et al. 2011). The 5 participants of the study had varying levels of experience of teaching computing to pupils and teachers. **Table 1** gives relevant information about the participants.

Table 2 Overview of projects

Project Name	Design Type	Genre
Lifecycle of a frog	Labelled Diagram	Animation
Maths Quiz	Labelled Diagram	Quiz
Handa's Surprise	Storyboard	Animation
Racing Game	Concept Map	Game
Traffic Lights	State Diagram	Simulation

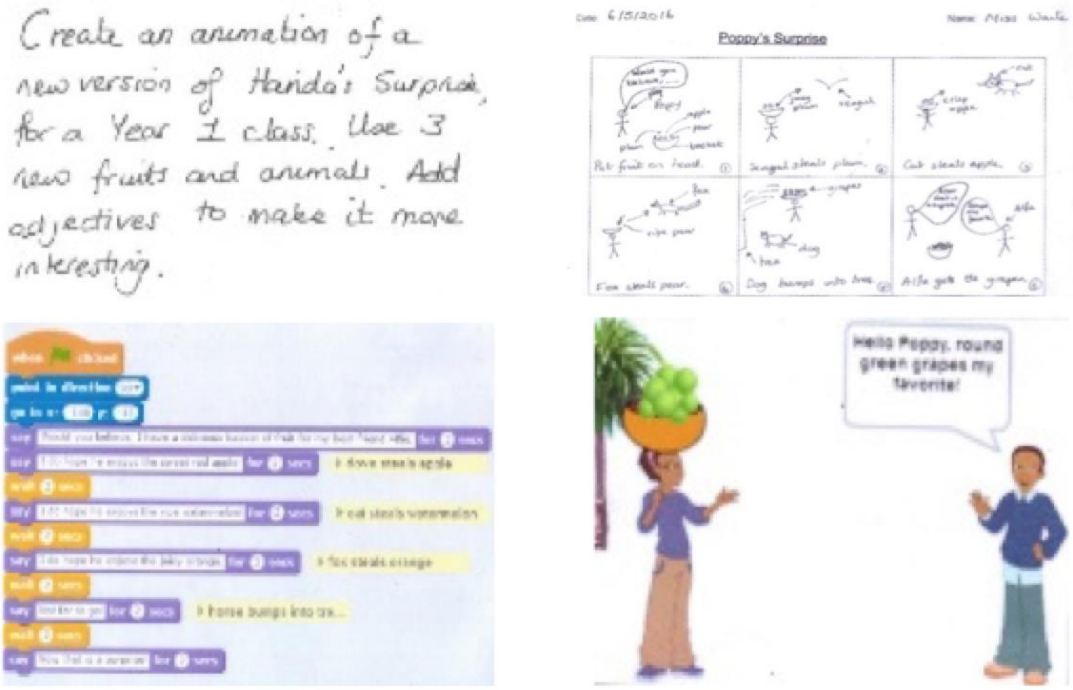

Figure 2 Example Project Cards

4.2 *Interviews*

During the activity interview, participants were asked to complete a set of six short unplugged interrelated tasks. The tasks were included to build rapport (Punch 2002), provide stimuli material to prompt discussion (Nind et al. 2016) and contextualize the interview questions (Seeratan & Mislevy 2009). The tasks required participants to group, order and label representations of the LOA in programming projects. These representations were physical pre-prepared laminated project cards depicting each LOA. **Figure 2** examples of cards for each level. By using images of the LOA, the name of the levels was not revealed, thereby eliciting participants use of vocabulary. The LOA terms were not introduced until the final labelling activity. **Figure 3** gives photographs of examples of an ordering and the labelling task. Participants used no online resources during the study. Five projects were used as shown in **Table 2**. Before being introduced to the project cards, participants completed a simple pre-test where they were asked how they would undertake a simple programming project. During the Teaching & Learning Interview, teachers were asked questions grouped into five areas: design, abstraction, general questions, feedback questions and pupil information. These sections were chosen to explore the main research question RQ1. Interviews were audio recorded and then transcribed. The activity interview took around 35 minutes and the teaching & learning interview around 25 minutes. All interviews across the schools were completed in the summer of 2016.

Figure 3 Example photographs of the ordering and labelling activity interview tasks

4.3 *Data Analysis*

A thematic qualitative data analysis (QDA) approach was used to analyse the transcribed interviews and outcomes of tasks based on the methodology detailed by Kuckartz (2014). NVivo was used to support the process of coding text segments. QDA focuses on creating a clear interpretation of the meaning of the text, providing an opportunity to summarise unstructured data into a set of themes that can be categorised by cases to provide generalized overviews (Kuckartz 2014). In this study, each case was a teacher. The first author is an experienced primary teacher and primary computing teacher as well as a computer science education researcher. Using knowledge of the field (Kuckartz 2014), this author, prior to analysing the data, first generated high-level categories deductively from the research questions. Using these initial categories, the first author coded Teacher A's interview, adding and amending categories and sub-categories inductively (Mayring 2000).

After this first pass of coding, the resultant categories were reviewed and revised to confirm they matched the data coded. This teacher interview provided approximately 1/5th of the overall transcripts in line with recommendations of 10 to 20% for the first pass (Kuckartz 2014). Following this, all interviews were coded. Emergent patterns were recognized, and new codes created to hierarchically group codes. This process was repeated across the categories creating, merging and splitting codes inductively (Kuckartz 2014; Mayring 2000).

Further analysis took place through the creation of case overviews, presented as thematic matrices. These enable comparison of codes and themes across cases within a category and provide an opportunity to give an indication of any patterns or broader characteristics and overarching themes (Kuckartz 2014). Finally, a second researcher, also a computing teacher and computer science research practitioner, then reviewed the coding. Discrepancies were discussed, and a consensus agreed, recoding as necessary.

5 Results

First the results of the vocabulary related research question “RQ2: What common vocabulary do teachers use to describe LOA in programming?” are reviewed. Twelve findings are then presented as the results of the main research question “RQ1: What are K-5 teachers' uses of LOA, particularly design, in programming and other subjects?”

5.1 *What common vocabulary is used?*

Whether the teachers used a common vocabulary to describe LOA in programming projects and other subjects is addressed before the main research question, as it sets the scene.

Throughout the interviews, teachers used a variety of conflicting terms for design, algorithm and code, and had a limited vocabulary to describe running the code. The common term used for the problem level was 'task'.

Teacher C, pointing at the project card depicting the problem for one project, said:

> *"With the traffic light, again we have the task instruction." (Teacher C)*

Teacher E referred to the problem, saying:

> *"That is like a title but is more of the task. That is the task you have to do." (Teacher E)*

Teachers used different terms for each design type. For the labelled diagram they used the terms 'labelled diagram', 'layout', 'plan', 'design part', 'picture', 'annotated diagram', 'lifecycle', 'mind map', 'drawing of what the screen might look like', and 'how it is going to look in the final piece'. For the storyboard the terms 'story S'[1], 'comic strip', 'story map', 'plan' and 'storyboard' were used. 'Mind map', 'concept map', 'plan' and 'brainstorm' were used for the concept map. 'Mind map', 'flow diagram', 'instructions', 'planning diagram', 'thing with bubbles', 'algorithm written out in bubbles' were terms used to name the state diagram.

All but one teacher used the term 'code' for the code level; there was no more popular term. Teacher E struggled to find a term for the code, at one time describing the image of the code for the traffic light simulation as:

> *"Then this bit, is what the instructions would look like on Scratch that have been made with all the steps and all the narrations." (Teacher E)*

Later looking at the code for the frog lifecycle she called it the algorithm saying:

> *"That is the algorithm [pointing at the code], the instructions you make for it to follow." (Teacher E)*

The term 'algorithm' was used interchangeably by several of the teachers (novice and expert) to mean both the coding LOA and the design LOA. Several teachers were very unsure what an algorithm might be for a programming project. One expert teacher used term 'Scratch algorithm' and 'algorithm on Scratch' alluding to an algorithm being implemented as code on Scratch and showing an understanding of transitioning across LOA.

Teacher A, an expert teacher described a process of problem solving including creating the algorithm, saying:

> *"Well, there is a lot, it is decomposition, breaking it down into parts, you are thinking about the logical effect you are going to take, you are writing an algorithm, what step by step you are going to need to do." (Teacher A)*

The same teacher described a concept map representing a design as having 'algorithms', saying:

> *"Last of all I have got an arrow based algorithm [pointing at a concept map], an algorithm with several arrow-based directional algorithms, with score and lives algorithm with variables which belong with the racing game." (Teacher A)*

But then describing the same project, the teacher used both the terms 'algorithm' and 'code' for the code, saying:

> *"The racing game with the bubble, is the plan, the mind map. We have a copy of the code, the algorithm [pointing at the code] to make the game work" (Teacher A)*

For a different project the teacher called the code the 'algorithm' again, saying

---

[1] A story S is a planning format used in primary school writing.

> *"I have got an algorithm here [pointing at code] for the traffic lights that uses broadcast blocks" (Teacher A)*

Describing the same code, but in a different activity, the teacher again called the code the 'algorithm' saying:

> *"I would expect the instructions first, so the children know what they are going to do. Then I would have the racing game planning next, and then the algorithm [pointing at the code] and the screen side by side." (Teacher A)*

For a different project, Teacher D, a less experienced teacher called the code the 'algorithm':

> *"First the frogspawn is laid by the female frog in the water, this a picture with a speech bubble, and I am going to match with I am going to tell you about the lifecycle of the frog and that is the algorithm [pointing at code], the Scratch there." (Teacher D)*

In the final activities using the project cards, Teacher E was shocked with the idea that the term 'algorithm' might describe something associated with the design.

When asked what was the same or different across the projects the conversation was:

> *Teacher E: This is the instructions the algorithm for it [pointing at code].*
>
> *Interviewer: The things, the purple bits?*
>
> *Teacher E: The blocks you get on scratch when clicked repeat.*
>
> *Teacher E: The same, they all had an overview title, a task [pointing at the problem], they all have a hand drawn plan [pointing at the design], they all had a picture of what it might look like [pointing at running the code] and they had an example of the algorithm, instructions [pointing to the code].*

Then when asked to label the cards with a set of pre-prepared terms of problem, design (including algorithm), code and running the code, the conversation was:

> *Interviewer: Here are some words, I'd like you to have a go at matching them to the different components.*
>
> *Teacher E: I think the initial task is the problem and you have to solve the problem. [long pause and shock].*
>
> *Interviewer: Has this [pointing to the card - design (including algorithm)] surprised you?*
>
> *Teacher E: Yes, I think they all go here. Because I think the word design goes with that one [pointing at design], but not the [word] algorithm.*

Teacher C an expert teacher, used the term 'Scratch algorithm' to describe code. This term may imply the idea that the algorithm is implemented on Scratch, a transition from design to code. The teacher said:

> *"then I have got Scratch algorithm [pointing at code] for something involving fruit and animals so that must be Poppy's Surprise." (Teacher C)*

Later for the same project, the teacher called the code the 'algorithm on Scratch', again implying a transition from design to code level saying:

> *"I would also expect the plan to be with them throughout the process and to in some ways to add notes to the plan or add comments to the algorithm on Scratch [pointing at code] to keep a record somewhere of what they are doing and changes they make as they go along. Because their original idea is probably likely to adapt as they go through the process." (Teacher C)*

Table 3 Thematic Matrix for pre-test terms

Teacher	Problem mentioned	Design mentioned	Code mentioned	Running Code mentioned
A		✓	✓	✓
B			✓	
C	✓	✓	✓	
D	✓	✓	✓	✓
E			✓	

At another point the same teacher described a different project's design as an 'algorithm', saying:

> *"I have a traffic light picture and there was a traffic light plan, there. I have got a planning diagram there, an algorithm, the thing with the bubbles [pointing at state diagram]." (Teacher C)*

When discussing the 'running the code' LOA the activity was flawed, as teachers did not interpret the screen-shots to be the code running, but sometimes a design of the background for the project. However, when pinpointing this level, there was no common term used.

5.2 *Initial pre-test response focused on coding*

It was of interest to see which LOA teachers might talk about before they started any activities or were asked any other questions. Participants were asked how they would tackle a simple programming project of animating the life-cycle of the butterfly. The results are summarized in **Table 3.**

The two most experienced teachers, A and C, and one of the novice teachers alluded to design, all teachers mentioned coding, with this being a focus for all.

The novice teachers all mentioned they would need to investigate Scratch more before embarking on such a project. The two Year 2 teachers were very uncertain of what to do. This is not surprising as they had limited experience using Scratch beyond teaching simple route-based activities.

The more experienced teachers confidently described their approach, with Teacher C saying:

> *"The first thing we would have to do is work out what the life-cycle of the butterfly is. So, we would have to take that entire process and break it down into its parts. In terms of programming it, we would need to think about changing costumes, probably the best way of doing it so that the different costumes. That would have to be programmed to switch across 4 different costumes. There would probably be movement involved as well as animating the movement across the screen. But also, how they control the life-cycle, whether they go through it with arrows keys or press the space-bar or whatever input makes sense." (Teacher C)*

And teacher A saying:

> "*We would look at what a background looks like, what a sprite looks like and what each object is. Then we would go over again the different areas that they can program and what they mean. Then I would get them to have a fiddle with it first and see if they can make it move, and then look at the different ways they made it move and then we would look at what the most time efficient way. So, looking at, decomposition, a while since I taught Scratch. Then we would look at the other ways that people had done and work out the best way to make that happen. Then I would introduce them again how to change the background, change the sprite and probably let them have a go at it for a little bit and again stop them and have another look at how they got to that stage, how they debugged the bugs when a problem comes and then talk through what they have asked the sprite to do at each stage to then work out where they went wrong. Yes, that would be the first lesson and see how they go. Then we would have to stop and see what they got up to and what is next." (Teacher A)*

Teacher B also answered confidently, but focused on finding out how to find out about Scratch.

> *"I would have to look at how to use Scratch first of all and look at which coding blocks to use to start off the process." (Teacher B)*

This is perhaps due to Teacher B being a novice user of Scratch and the fact he realised he did not know what the software was capable of doing, so had no pre-experience of whether the task could be completed or what commands he might use.

Similarly, Teacher D, another novice exemplified the distinction between planning and the need to work at the coding level to find out what could be done.

> *"If I was planning it I would have to find out what the lifecycle is first, and find out what it is going to be and then I would have a play around and see what I can and can't do and just trial and error it and say if something does not work take it out and start again, take the little blocks out and that is how I would start off and I would just be like constant trial and error why does it not work, as I have not been taught how to do it so I would have to learn." (Teacher D)*

Teacher E another novice, who has used Scratch for route-based activities, e.g. moving a sprite around an object was very uncertain and said,

> *"Gosh I don't know. What steps would I take?" (Teacher E)*

Notably, the novice teachers, once they started the activity and were prompted to talk about design, suggested a range of uses and an understanding of design, which was not revealed from the pre-test.

5.3 *Level of detail*

Teacher C, an experienced computing teacher and Teacher E, mentioned the level of detail included, or omitted by pupils in designs. Teacher E's comments related to planning in literacy, but there were clearly close parallels with Teacher C's comments. Both cited pupils who ignored too much in their plans, and other pupils who included more detail than was needed. When Teacher E was asked if she ever asked pupils to have less detail, said:

> *"I don't think I have ever said [there is] too much detail." (Teacher E)*

Teacher C explained he demonstrated to pupils how to include *'less detail'* in computing designs than might be expected in plans for other subjects:

> *"It is about learning to ignore, it's almost the opposite of what the other subjects expect. And so, it is a different way of thinking. I look at the labelled diagram of the maths quiz, if that was in a year 5 book, and it was science it would not be enough. It has to be overloaded with detail and description. But in computing we want the opposite to begin with. We want a simple sequence, that will do what we want it to do." (Teacher C)*

At another point this teacher described how he demonstrated including the right level of detail, saying:

> *"I make it very clear, I am going to draw a very basic leaf, to me that is a leaf, does it look like a leaf to you. To me it looks like a leaf, and that's fine as I'm the only one who is going to look at it and program that leaf. Yes, it is about learning to ignore, it's almost the opposite of what the other subjects expect. And, so it is a different way of thinking. But it is a life skill way of thinking. Often in other situations we have to abstract things." (Teacher C)*

Despite this expert computing teacher noting the need to get the right level of detail at the design LOA, he did not then take this step further to suggest moving from the code to the design level. However, he did mention

Table 4 Thematic Matrix of Design Types

Teacher	Labelled diagram	Storyboard	Concept Map	State diagram
A	✓	✓	✓	
B	✓	✓		
C	✓	✓	✓	✓
D		✓		✓
E				

using comments in code to record what students are doing, perhaps alluding to linking code back to the design and summarising code at the design level saying:

> *"I would also expect the plan to be with them throughout the process and to in some ways to add notes to the plan or add comments to the algorithm on Scratch [pointing at code] to keep a record somewhere of what they are doing and changes they make as they go along. Because their original idea is probably likely to adapt as they go through the process." (Teacher C)*

5.4 *Using different design types in computing*

During the interviews, teachers talked of their use of different design types in computing lessons, as shown in **Table 4**.

Teacher A mentioned the use of a concept map and labelled diagram for creating websites and using a storyboard for an animation and a quiz. Teacher B mentioned using a labelled diagram for a power point activity to link pages, using a storyboard for a sequencing task, and a state diagram for a physical computing activity. Teacher C mentioned using a labelled diagram for a quiz, storyboards for animations and a quiz, a concept map for games and a state diagram reminded him of work he had done with physical computing. Teacher D mentioned storyboards for an unplugged sequencing activity. Teacher E did not mention she had used any specific formats for computing activities.

When asked what the different design types might be good for, Teacher C, one of the most experienced computing teachers demonstrated his understanding of different design types:

> *"In programming projects, the storyboard is very good for animations, and to a lesser extent games where you move the character around a maze and move from one state to another. The mind map is good for games where the actions that take place are relatively similar throughout like a racing game. What is important is you understand the role of different parts of the game, so how the character is controlled, whether there are lives or scores and any other variables and how they change." (Teacher C)*

5.5 *Using different design types for different subjects*

All teachers said they used storyboards in English and labelled diagrams in Science. Concept maps were mentioned as being used in English, Maths, topic work, Science, Geography, History and Personal Social Health and Economic lessons (PSHE). Labelled diagrams were said to be used in English, Maths, topic work, Science, Design and Technology (DT) and History. Storyboards were mentioned for English, Maths, topic work, Science, Geography, History and Music. State diagrams were only mentioned for Science and DT.

The teachers in the study could explain why design types were suited for different subjects, they cited storyboards being good for sequencing, and concept maps being flexible to add new ideas. Teachers showed familiarity, confidence and a depth of understanding on design in other subjects that might be exploited for using design more effectively in programming.

Teacher B explained how concept maps could be used, saying:

> *"And then here brainstorming ideas and adding detail could be used in lots of different subjects, anything from English, Maths, you could use that anywhere to organise ideas and*

Table 5 Thematic matrix for aide memoire

Teacher	Outlines what to do	Records what done	Reminder what next	Helps them stick to design
A	✓	✓	✓	✓
B	✓		✓	
C	✓	✓	✓	✓
D	✓			✓
E	✓		✓	✓

> *categorise things. Here you have car, track but that could be anything from another subject where you could organise the code what's going to happen along this branch of the brainstorm or mind map" (Teacher B)*

Teacher E explained how different design types were used.

> *"The storyboard the sequence is clearer and if they have a mind map, it is good for getting down ideas, but they may be confused because there is no order. If what they are doing needs an order, a beginning a middle and an end." (Teacher E)*

5.6 *Using design when teaching novice writers and the significance of self-regulation*

Throughout the study, both novice and expert computing teachers, mentioned the importance of, and utility of, using planning, a form of design, to support novice writers as they learn to write.

One teacher said that when teaching writing, her pupils spent two thirds of the time planning and a third of the time converting their plans into written text. She also explained that gathering ideas on a plan was a different skill to then translating it into text, and that the design provided a tool to bridge between these two stages. She described why a plan was important:

> *"First and foremost [a plan is] a reminder of all the ideas that they had and because the demands of writing are so high they need the support for their ideas. If they have got their ideas there, they can remind themselves with the plan, they have got it there to refer to constantly as much as they want." (Teacher E)*

5.7 *Using design as an aide memoire for promoting cohesion, completeness, improved quality of work*

All teachers mentioned, in some way, the role of a design in managing the process of pupils progressing their programming project. Design served as an aide memoire of what was to be done, what had been done and what to do next. It thereby improved cohesion, completeness and quality of work. This theme is shown in **Table 5**.

Teacher A explored the idea that over time, having a plan was very important and that marking off what had been done helped pupils know what was next:

> *"You know what it is like. They have their ideas one week and actually they don't do it for another week, because you have not got the computer suite. So, if you have not got it on there or where you are up to. They are at stage one because they just have a picture and they have forgotten, whereas if they annotate as they go along, you can see, it will prompt them, oh yea, I remember I found that problem and I had to do this, and that will help me know what to do next time." (Teacher A)*

Teacher C drew attention to the use of design to help pupils finish work. The design helping pupils check coverage was mentioned by Teacher E. Teacher C explained designs help pupils stay 'on track'. Teacher B raised the idea that a design helped you outline the steps and what was to be achieved.

5.8 *Pair programming*

Teacher C raised the idea of using a design as a contract between pupils when working in pairs.

> *"If children are programming in partners, which is very good. You have got two different mind-sets, ideas. So, I wanted to add this, and he wanted to add that. But if you agree on something and get it down on paper. It's like a contract, this is what we are going to do. This is what we are going to stick to it avoids that oh he's off ill today, so I am going to put in that thing that I wanted to. They are always welcome to adapt designs, but by writing them down it secures them, makes the process much more set up. Scaffolding for themselves, so they know where they are going." (Teacher C)*

5.9 *Using design to know what to teach next*

Teacher C also pointed out design was useful for the teacher as well as pupils, saying teachers could look at the design and see what was needed to be taught to implement ideas, saying:

> *"Or equally for teachers to know what is coming up. So, the teachers can plan for it." (Teacher C)*

5.10 *Adding annotations*

When talking about the creation and use of designs, the two more experienced computing teachers, A and C mentioned adding notes to storyboards or labelled diagrams. These annotations provide ideas on how to implement, as code, certain features of the design (e.g. what code blocks might be useful). By adding notes on how to implement a design, teachers and pupils are crossing a LOA from design to code.

Both Teachers A and C raised the idea of adding annotations both before coding started and amending the notes once coding was taking place. When asked if they used designs after coding was completed both indicated they did not refer back to the designs as much as they would like to. Teacher A explained how pupils used their design and added notes.

> *"They have it (the design) in the suite with them so they can make notes on it about what has worked and what has not worked. So especially if there are things going wrong they can note about what changes they made there and what they thought would work and hadn't and how they debugged it." (Teacher A)*

A further comment was how annotations are used to distinguish between original ideas and changes made, linking to self-assessment and the idea of a growth mindset (Dweck 2015).

> *"We normally have a green pen that we edit with. So, that shows the difference between what was the first idea and what it ended up like." (Teacher A)*

5.11 *Design as means to promote self-assessment*

Both expert teachers required pupils to mark their design with a self-assessment of their confidence to implement its components before it came to them writing the code and as they made changes during the development process. Teacher C explained:

> *"Ragging is something we do a quite lot. Red, amber, green. Now you have planned it, what bits do you think you can do on your own. A plan is very useful when children are thinking ahead. I need to do something, and I don't know how to do it yet. I know how to make a character talk, I know how to change a background, so I can do that. But I am not quite sure how to make a butterfly look like it is flapping across the screen. So, children are doing pre-learning. Thinking ahead. I am going to need to know how to that. So, can I learn it for that time.*" *(Teacher C)*

5.12 *Using design to help evaluate 'do-ability' of what is planned*

The concept of 'do-ability' and supporting learners' reflection of what they can code, was raised by Teacher C several times.

> *"Part of the design process is not just knowing what you have to do next, but knowing what you have already achieved. AND looking back and evaluation of their design. What was doable and was there anything you could not achieve in Scratch. They are going to come across problems. It is likely they will have planned something that they thought would work and be very easy to program, but then, in reality, find it is quite hard to do." (Teacher C)*

> *"So, they are constantly reflecting on the design process and how it evolved from the beginning ideas to later on. Often what I will get is where they have either learned to do something new and decided to use it because it worked better or whether they were overly ambitious. Ok you were ambitious. Were you ambitious with your own ability or with the ability of the software?" (Teacher C)*

6 Discussion

To facilitate discussion, in this section the results outlined above are grouped by pedagogical content knowledge (PCK) elements: Goals and objectives; Student understanding; Instructional techniques and Assessment. (Magnusson et al. 1999). Across all elements further investigation is needed to explore whether opportunities raised by the interviewed teachers are being exploited, their effect and whether the study's findings are generalisable.

6.1 *Goal & Objectives*

Teachers knowledge of the goals and objectives to be taught incorporates what curricula is to be taught as well as the concepts to be learned. Under this heading, findings related to vocabulary for the LOA, the level of detail taught and design types used are included.

6.1.1 Use of Vocabulary and 'What is an algorithm?'

The five teachers interviewed used a variety of conflicting terms for design, algorithm and code, and had a limited vocabulary to describe running the code. A lack of understanding of the running the code level may impact on learners not developing an understanding of the notational machine (Du Boulay 1986).

The terms design (including algorithm) and code, used as labels for different LOA, were used interchangeably by the interviewed teachers. In discussion, the term code and algorithm were used interchangeably, or the teacher were very unsure of what an algorithm might be for a programming project.

Not having a common term for running the code, and not having a shared understanding of the term algorithm and code is significant. Progression in programming, as with any topic, is aligned to learners building understanding based on precise vocabulary (Armoni 2013; Carlisle et al. 2000; Sapir 1921) in a discourse rich environment (Cutts et al. 2012; Grover & Pea 2013). Therefore, progression in programming may be being compromised by a lack of teachers' shared understanding of programming vocabulary.

6.1.1.1 Learning about algorithms through route-based activities

Misconceptions related to the distinction between algorithm and code may be associated with how early algorithms are taught using programmable toys and route-based puzzles (either on a real map or on an on-screen map). This might entail working out the route for a toy to move from the Red-Riding-Hood's home to Grandma's house, avoiding the Big-Bad wolf squares. The algorithm might be rehearsed by the pupil, "*playing turtle*" (Papert 1980, p.58) the learner embodying (Barsalou et al. 2003) the device and moving around on a person-sized mat.

Alternatively, a physical manipulative representing the bot (Bruner 1963), such as a 'fakebot' (Berry et al. 2015) might be used to work out the route. The planned instructions might not be recorded, or they might be noted as a list of words, such as forwards, left, forward, forward. Or the instructions might be depicted as a series of drawings of each movement, using arrows, a symbolic or iconic representation (Bruner 1963). This drawn form of depiction is likely to be encouraged as pupils undertaking this kind of work are likely to be 4 to 6 years old, and some of them may not yet be confident writers. In using the drawn depiction of the moves, this resultant algorithm may map 1:1 with the programming language commands. In this instance, the learner and teacher may view the code as being the algorithm and vice versa. From the small cohort of participants in this study, the

terms ‘code’ and ‘algorithm’ were used interchangeably by some teachers, or they were very unsure of what an algorithm might be for a programming project. This may be as they developed this view of what an algorithm is from their work with programmable toys and route-based activities. Using different vocabulary at the algorithm level to the code level has been suggested (Armoni 2013; Cutts et al. 2012) which could counter this problem of muddling the two levels.

6.1.1.2 Moving on to other project types

As learners move onto block-based languages, there is opportunity to create animations, games, quizzes and simulations with many possible solutions for any task, as there is no longer a single solution. At this point, what does an algorithm look like? Is it the ordering of action on a storyboard? Is it the informal note of the rules for a score written on a labelled diagram? However, if no design is created is the algorithm *the thought process* which decides what is to be coded just as choices are made before the code is written? Or perhaps there is no awareness of the algorithm as the code has evolved from trial and error, bricolage, or coding experience (Turkle & Papert 1992).

If learners are planning, then this is their design. It is likely that there will be a whole range of attributes, included in the design. This might include ideas about the user interface such as the colour, size and shape of objects. Objects drawn might include backgrounds, characters, obstacles and scores. The processes and rules that are required to act on those objects might be indicated by arrows and notes.

Decisions about where a sprite might be first located (initializing it), working out the speed of movement of a sprite in an animation, or the rules for a game might only be made at the point of coding as learners discover what is possible, what is ‘doable’. When working with expert developers, such as in industry programmers, such choices may also be made at the point of coding, but here the programmer understands what is doable and chooses not to duplicate their choices in separate documentation. However, the design for a novice provides a scaffolding technique as they explore and learn what is possible, rather than a documenting technique (Schulte et al. 2017).

6.1.1.3 Formative unplugged algorithm activities

This confusion over what is an algorithm may be further compounded by unplugged computational thinking activities which draw on the wider and original, non-code-dependent meaning of the word algorithm which refer to instructions and rules governing the human actions as algorithms. Activities such as making a jam sandwich, creating a dance routine, performing card tricks, drawing artwork or making pupils reflect on the multiplication or long division algorithms they use themselves (Berry et al. 2015; CSTA 2011; Curzon & McOwan 2008), are used to develop algorithmic thinking. However, the algorithms created or discussed are not implemented as code. Through these activities, perhaps, there has been insufficient focus on developing teachers' and learners' understanding of the link between the design/algorithm level and the code level.

Combined with the fact that coding at this level may often be done by tinkering and bricolage rather than a design-based approach, this may result in novices seeing block-based coding as not being associated with the algorithm creation, as their experience of algorithms has either been when working with route-based programming tasks or when creating unplugged instructions for people. Therefore, the novice does not perceive there is an algorithm associated with solving a more open block-based programming task and they therefore ‘jump’ from task to coding. This preference of novices to work at the code or running the code level has been noted before (Armoni 2013; Cutts et al. 2012). Perhaps here the formative unplugged algorithms intended to teach computational thinking could be viewed as ‘emergent algorithms’ in the same way that early writing is termed emergent writing (Strickland & Morrow 1989).

6.1.1.4 Emergent algorithms (emergent algorithmic thinking)

Returning to the original PGK hierarchy (Armoni 2013; Perrenet et al. 2005; Perrenet & Kaasenbrood 2006) the place for emergent algorithms might be at the task level, as only precise object framed algorithms may exist at the object level. The object level requires a sophisticated understanding of the problem as objects, not processes (Sfard 1991). This may be difficult for K-5 teachers and learners to understand. For K-5 contexts, a design level, with emergent algorithms, like emergent writing, might be an acceptable ***bridge*** to facilitate progression.

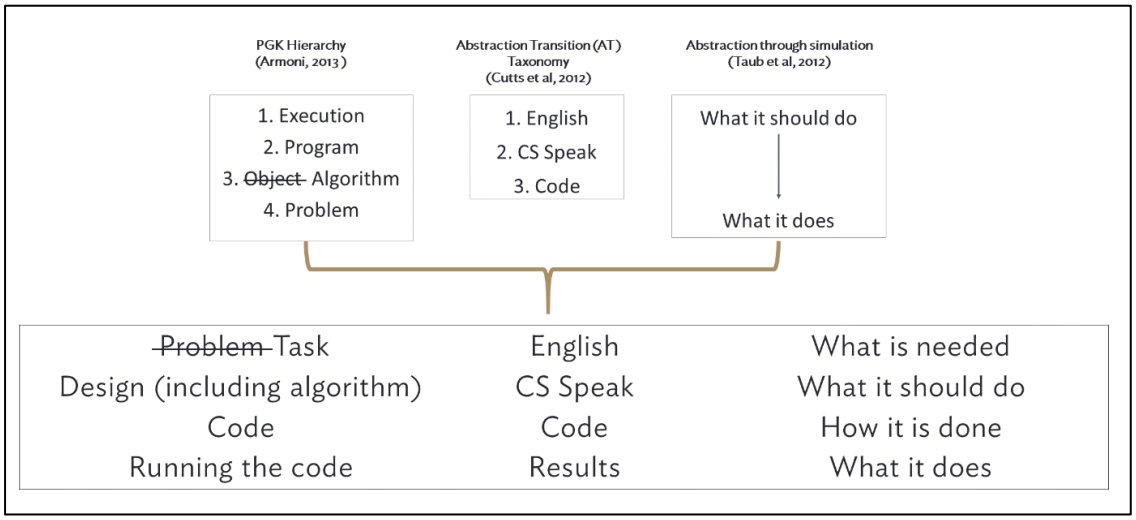

Figure 4 Revised aligned K-5 LOA hierarchy

Further research is required to investigate what an algorithm is for K-5 learners, as well as how algorithms might be depicted in design. Methods for teaching about algorithms and design should be investigated to compare the impact of different approaches, with attention made to progression in understanding and possible misconceptions from the development of emergent algorithms at a design level. In particular, a greater focus on making more explicit links from unplugged activities to coding is needed. Similarly, there is work to be done to explore preferences and attitude towards design as well as the impact of design on creativity.

As a result of the findings and discussion, the naming of the K-5 LOA hierarchy is: Task; Design (including algorithm); Code; Running the Code as shown in **Figure 4** . These terms will be used in next steps to investigate whether the results of the study are generalisable over a wider community of teachers.

6.1.2 Level of detail

Both novice and expert teachers mentioned the level of detail included, or omitted, by pupils at the design level in literacy and in programming. One expert teacher explained that he demonstrated to pupils how to include ‘less detail’ in computing design than might be expected in plans for other subjects. Teachers' understanding of the amount of detail needed for each level may impact on teaching and learning of LOA.

This understanding of the difference between wanting more detail in other subjects and needing less detail in design of programming projects is significant, particularly with respect to training and professional development for teachers. It could be that teachers do exemplify less detail in other subjects, but have not linked this to their teaching in computing. Examples may include: when they ask for summaries such as book reports, or overviews of what has been learned, or when suggesting the addition of sub-headings to sections in writing.

There may be opportunities to investigate annotating code and sub-goal modelling (Margulieux & Catrambone 2016; Su et al. 2014), peer review of code annotations (Hsiao & Brusilovsky 2011) and doodling to create designs from code (Lister et al. 2004) with K-5 pupils to support learning how to move from the code LOA to the design level. Creating designs from code may support learners to better understand the level of detail needed in computing designs.

6.1.3 Familiarity with and using different design types in computing and across other subjects

The interviewed teachers are using different design types in computing lessons, and expert teachers showed a depth of understanding in why certain design types are suited to specific types of programming activities. The teachers showed familiarity, confidence and a depth of understanding of design in other subjects that might be exploited for using design more effectively in programming. Whether curricula resources encourage teachers to use the most suitable design type in computing activities was not revealed in this study. However, this merits further investigation. How different design types exemplify the LOA at which one is working at, or are effective to support the transition across levels, requires further investigation.

6.2 *Student Understanding*

This component of PCK refers to teachers' understanding of how pupils learn, rather than what they should learn. Findings related to self-regulation as linked through the synergy between using planning to support teaching writing and using design to support teaching programming are included here.

6.2.1 Synergy with teaching writing & self-regulation

Novice and expert computing teachers in the study, mentioned the importance of, and utility of, using planning, a form of design, to support self-regulation for novice writers as they learn to write. Self-regulation, planning, revising and editing compositions is recommended as having a high impact on the improvement of writing (Graham et al. 2012; Higgins et al. 2013). Novice writers use their plan to focus on one part of it at a time, transition LOA as they write each part, revise and edit, and check back to the plan. Similarly, a design in programming becomes a personalised scaffolding map that supports traversing the LOA as learners decompose their problem, implement each component, debug each part as they run the code and check back to the problem level, before moving onto the next component of their design.

Lewis (2000) claimed that writing success is increased by using genre specific meta-structures, a form of design abstraction. Graham & Perin (2007), in their meta-analysis of writing instruction, recommended planning, revising and editing compositions as the most effective approach for improvement of the quality of writing. The Education Endowment Toolkit recommends self-regulation approaches that help pupils, plan, monitor and evaluate their own learning as a 'best bet' to improve attainment of disadvantaged pupils (Higgins et al. 2013). More recently, Dockrell et al. (2015) recommended 'Talk for Writing' as an approach for teaching K-5 pupils to become effective writers. Planning lies at the heart of this approach, including children planning what they want to say and writing this plan down in some form. Research on self-regulation strategies to support struggling writers have claimed advantages of improved attainment through pupils being able to understand the processes involved and focus on what is required next (De La Paz & Graham 1997; Glaser & Brunstein 2007; Graham et al. 2012; Mason et al. 2011; Santangelo & Olinghouse 2009; Schunk & Swartz 1993; Whitebread & Basilio 2012).

In teaching writing, planning is promoted as it both decomposes a complex process into manageable parts and divorces the stage of gathering ideas from the high demands of translating those ideas into a finished product. Planning in writing is a self-regulation tool that develops independence for novice and struggling writers. There are clear synergies between planning to write and designing to program. Therefore, it is suggested there are opportunities to use design as a self-regulation tool to develop independence for novice and struggling programmers.

6.3 *Instructional Strategies*

Instructional strategies relate to the specific techniques teachers use to facilitate learning of subject goals and objectives (content) within the context of how students learn. Using LOA, particularly the design is the strategy that was explored through the study. A range of teaching and learning opportunities were cited by the interviewed teachers as resulting from this strategy.

6.3.1 Aide memoire

All teachers mentioned the role of a design in managing the process of pupils progressing their programming project. Design served as an aide memoire of what was to be done, what had been done and what to do next.

Cognitive load theory claims that if a complex scenario is broken down into discrete units, then the complex task becomes easier to solve (Van Merrienboer & Sweller 2005). Designs based on artefacts such as storyboards do just this, breaking a large problem into bite-sized chunks. The design becomes a personalised scaffolding map that helps pupils create a cohesive and complete solution and therefore increases the quality of their work.

6.3.2 Completeness, coverage, cohesion

One teacher drew attention to the use of design to help pupils finish work, improving completeness, another to the design helping pupils check coverage, and another to how design kept pupils on track improving cohesion. Here the quality of the finished piece is improved by using the design level to manage a complex task. There are links again to cognitive load theory (Van Merrienboer & Sweller 2005).

6.3.3 Annotations

The expert teachers mentioned adding notes, such as code constructs, to designs, both before and after coding had started. Annotations not only transitioned the LOA, but were also used for differentiation and to provide a record of the original and changed ideas supporting a growth mindset (Dweck 2015). Using annotations also gives teachers the opportunity to differentiate, as they can add annotations for those pupils who cannot yet remember what code might be needed to implement an aspect of the design: thereby scaffolding pupils' learning until they can memorise the coding pattern needed to implement a specific design aspect. There are links to pupils' memorising of phrases that are pertinent to specific genres of writing (Dockrell et al. 2015) and teaching pupils standard mathematical procedures for types of maths word problems.

6.3.4 Pair programming

One expert teacher raised the idea of using a design as a contract between pupils when working in pairs. There is an opportunity to use this to build upon Werner et al.'s (2013) rich seam of work on pair programming, and their pair effectiveness compatibility measure that includes dimensions of friendship, prior programming experience and confidence. There are links also with research related to professional programmers' use of design and artefacts to support pair programming (Plonka et al. 2011).

6.3.5 What next?

A teacher raised the idea that having a design enabled him to know what to teach next and raised the question of what not having a design means. Not having a design, may restrict teachers' options for teaching programming. Without some notice of what is coming next teachers must 'pre-teach' everything, or react 'just-in-time' to teach new skills as they are needed, or constrain pupils' projects to a predefined design. Just-in time teaching may be seen as an ideal approach in some contexts; however, novice teachers may not have the confidence nor expertise to teach in this manner. The impact of introducing design on novice teacher confidence merits further investigation.

6.4 *Assessment*

The final PCK element is assessment; two findings fall into this category the theme of do-ability and self-assessment.

6.4.1 Do-ability

The expert teachers required pupils to consider 'do-ability'. 'Do-ability' is understanding whether a pupil, at their current and anticipated level of experience within the time frame of a project, can implement their design within the constraints of the programming language being used. How novice teachers who have little experience of design in programming projects, limited understanding of progression and a basic understanding of programming languages are assessing 'do-ability' is not clear, nor what impact this has on their perception of their own competency (Hattie & Yates 2014).

6.4.2 Self-assessment:

Both the expert teachers required their pupils to mark their designs with an indication of their confidence to implement parts of it. The design then showed the whole problem, with pupils able to see what they may not yet be able to do, as well as being able to look back and see what they may have learned to do along the way. Annotating a design with notes of changes and showing the development of understanding aligns with ideas of a growth mind set (Dweck 2015).

7 Conclusion

Using PCK as a lens to review the interviewed teachers' uses of LOA, particularly the design level, has highlighted potential impacts on teaching and learning. Three specific aspects of the interviewed teachers' PCK of goals and objectives in teaching programming were revealed. Firstly, their knowledge of the vocabulary to describe the LOA was observed. Secondly, teachers' knowledge of the need to teach the level of detail required for a design in programming lessons was seen. Thirdly, teachers understanding of different design types was noted with expert teachers being able to explain how certain design types were particularly suited to a certain genre of a project.

The issue of teacher knowledge of vocabulary may be particularly important for teaching and learning in programming. Some of the teachers, in the study, interchangeably used the term algorithm and code to describe the code samples they were shown, or they were very unsure of what an algorithm was in a programming project. One teacher used the term 'Scratch algorithm' and 'algorithm on Scratch' to describe code. There appears to be some confusion as to what vocabulary to use for code and what constitutes an algorithm for the K-5 teachers. This may be due to the different ways that algorithms are taught in primary classrooms. Route-based activities with a 1:1 mapping between the design/algorithm and the code may lead to confusion between the levels. Similarly, unplugged cross-curricula activities where an 'human-level' algorithm (not intended for coding) is created to teach computational thinking may insufficiently reinforce the link between the algorithm and code. Not knowing what is 'doable' at the point of design, and using trial and error to develop code may result in learners creating' emergent algorithms' or no algorithm at all. What, within a design, can be usefully called an algorithm, or whether the notion of 'emergent algorithmic thinking' is useful, requires further research. Also, the interviewed teachers did not have a common term for running the code, which may compromise a developing understanding of the notational machine.

Regarding student learning, when teaching writing, the interviewed teachers used planning, a form of design, to scaffold pupil learning and promote pupil self-regulation, the similarities between this approach and using design in planning was explored, with promising synergies revealed. The expert teachers in the study suggested interesting and novel opportunities for improvement of teaching and learning by using LOA, particularly the design level, as an instructional strategy. These suggestions included using the design as a contract between learners when pair-programming, using the design to work out what they need to teach next and asking learners to annotate plans with useful code snippets to transition between LOA. The interviewed teachers used design as an aide memoire to promote cohesion, completeness and quality of work. For assessment, the expert teachers in the study used the design level to help learners assess whether their design is 'do-able' and expected learners to use designs to self-assess their confidence to implement ideas.

Despite a limited population of participants, findings of the study suggest that the LOA hierarchy may be a useful tool for improving and reviewing teaching and learning in programming. However, these findings, due to the small number of participants, require further investigation to assess the extent to which they are generalisable.

Based on the findings of the study, it is suggested there is merit in investigating the use of design as a self-regulation tool to develop independence for novice and struggling programmers in the same way that planning is used to support novice and struggling writers.

8 Further Work

The immediate next step is to explore the relationship between abstraction, design, algorithms and LOA. To discover if the results of the study presented are generalisable, a wider population of teachers has already been surveyed on their use of design in programming and planning in teaching writing; analysis of data is ongoing. A review of resources will also be made to discover the extent to which design and other LOA are incorporated in curricula material available to K-5 teachers. Working with focus groups of teachers, a start has been made to verify in more detail the results, validate the LOA hierarchy and create guidance on the practical application of abstraction, through design, in K-5 programming teaching.

There is much work to be done investigating the synergies between the LOA hierarchy and other developing models and approaches for teaching programming. The computer science education research community should explore these synergies and consider successful pedagogies used in other subjects, as they investigate how best to support teachers as they teach novice and struggling programmers.

9 References

Aharoni, D., 2000. Cogito, Ergo sum! cognitive processes of students dealing with data structures. ACM SIGCSE Bulletin, 32(1), pp.26–30. Available at: https://doi.org/10.1145/331795.331804.

Armoni, M., 2013. On Teaching Abstraction in Computer Science to Novices. *Journal of Computers in Mathematics and Science Teaching*, 32(3), pp.265–284.

Armoni, M., 2012. Teaching CS in Kindergarten: How Early Can the Pipeline Begin? *ACM Inroads*, 3(4), pp.18–19.

Barendsen, E. et al., 2015. Concepts in K-9 Computer Science Education. In *Proceedings of the 2015 ITiCSE on Working Group Reports*. ITICSE-WGR '15. Vilnius, Lithuania: ACM, pp. 85–116. Available at: http://doi.acm.org/10.1145/2858796.2858800.

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), pp.48–54. Available at: http://doi.acm.org/10.1145/1929887.1929905.

Barsalou, L.W. et al., 2003. Social embodiment. *Psychology of learning and motivation*, 43, pp.43–92. Available at: https://doi.org/10.1016/S0079-7421(03)01011-9.

Berry, M. et al., 2015. Barefoot computing resources. Available at: http://barefootcas.org.uk/.

Bienkowski, M. et al., 2015. Assessment Design Patterns for Computational Thinking Practices in Secondary Computer Science: A First Look, Menlo Park, CA: http://pact.sri.com/resources.html: SRI International.

Biggs, J. & Collis, K., 1982. Origin and description of the SOLO taxonomy. *Evaluating the quality of learning: The SOLO Taxonomy. New York: Academic Press Inc*, pp.17–30. Available at: https://doi.org/10.1016/B978-0-12-097552-5.50007-7.

Bloom, B.S., 1956. Taxonomy of Educational Objectives, Handbook I: The Cognitive Domain, New York: David McKay Co Inc.

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

Bruner, J.S., 1963. Needed: A theory of instruction. *Educational Leadership*, 20(8), pp.523–532.

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

Cohen, L., Manion, L. & Morrison, K., 2011. *Research methods in education*, Routledge.

Cook, C.T. et al., 2012. A systematic approach to teaching abstraction and mathematical modeling. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*. ACM, pp. 357–362. Available at: https://doi.org/10.1145/2325296.2325378.

Csizmadia, A. et al., 2015. Computational Thinking a Guide for Teachers. Available at: http://community.computingatschool.org.uk/files/6695/original.pdf.

CSTA, 2011. Computational Thinking Teacher Resources 2nd Edition. Available at: https://www.iste.org/explore/articleDetail?articleid=152&category=Solutions&article=Computational-thinking-for-all.

Curzon, P. & McOwan, P.W., 2008. Engaging with Computer Science Through Magic Shows. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*. ITiCSE '08. Madrid, Spain: ACM, pp. 179–183. Available at: https://doi.org/10.1145/1384271.1384320.

Cutts et al., 2012. The abstraction transition taxonomy: developing desired learning outcomes through the lens of situated cognition. In *Proceedings of the ninth annual international conference on International computing education research*. ACM, pp. 63–70. Available at: https://doi.org/10.1145/2361276.2361290.

DfE, 2013a. *Computing programmes of study key stages 1 and 2 National Curriculum in England*, Department of Education. Available at: https://www.gov.uk/government/publications/national-curriculum-in-england-computing-programmes-of-study.

DfE, 2013b. *Computing programmes of study: key stages 3 and 4 National curriculum in England*, Department for Education. Available at: https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/239067/SECONDARY_national_curriculum_-_Computing.pdf.

Dockrell, J., Marshall, C. & Wyse, D., 2015. Education Endowment Fund Talk for Writing Evaluation report and Executive Summary. Available at: https://www.google.co.uk/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0ahUKEwiZ1vDQs43SAhUHkRQKHUlgBLwQFggcMAA&url=https%3A%2F%2Fv1.educationendowmentfoundation.org.uk%2Fuploads%2Fpdf%2FTalk_for_Writing.pdf&usg=AFQjCNFlLAFouweXrL_CMwMe1slCiRCPvA.

Drever, 1995. Using semi-structured interview in small-scale research A teacher's guide., The Scottish Council for Research in Education.

Dweck, C., 2015. Carol Dweck Revisits the'Growth Mindset. *Education Week*, 35(5), pp.20–4.

Fuller, U. et al., 2007. Developing a computer science-specific learning taxonomy. In *ACM SIGCSE Bulletin*. ACM, pp. 152–170. Available at: https://doi.org/10.1145/1345443.1345438.

Gibson, J.P., 2008. Formal Methods: Never Too Young to Start. *Formal Methods in Computer Science Education (FORMED 2008)*, pp.151–160.

Gibson, J.P., 2012. Teaching graph algorithms to children of all ages. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*. ACM, pp. 34–39. Available at: https://doi.org/10.1145/2325296.2325308.

Glaser, C. & Brunstein, J.C., 2007. Improving fourth-grade students' composition skills: Effects of strategy instruction and self-regulation procedures. *Journal of educational psychology*, 99(2), p.297. Available at: https://doi.org/10.1037/0022-0663.99.2.297.

Graham, S. et al., 2012. Teaching elementary school students to be effective writers. *What Works Clearinghouse, US Department of Education*.

Graham, S. & Perin, D., 2007. A meta-analysis of writing instruction for adolescent students. *Journal of educational psychology*, 99(3), p.445. Available at: https://doi.org/10.1037/0022-0663.99.3.445.

Grover, S. & Pea, R., 2013. Using a discourse-intensive pedagogy and android's app inventor for introducing computational concepts to middle school students. In *Proceeding of the 44th ACM technical symposium on Computer science education*. ACM, pp. 723–728. Available at: https://doi.org/10.1145/2445196.2445404.

Hattie, J. & Yates, G., 2014. Visible learning and the science of how we learn, Rout.

Hazzan, B. & Hadar, I., 2005. Reducing abstraction when learning graph theory. *Journal of Computers in Mathematics and Science Teaching*, 24(3), pp.255–272.

Hazzan, O., 2002. Reducing abstraction level when learning computability theory concepts. In *ACM SIGCSE Bulletin*. ACM, pp. 156–160. Available at: https://doi.org/10.1145/544414.544461.

Hazzan, O. & Kramer, J., 2007. Abstraction in computer science & software engineering: A pedagogical perspective. *Frontier Journal*, 4(1), pp.6–14.

Higgins, S. et al., 2013. The Sutton Trust-Education Endowment Foundation Teaching and Learning Toolkit: Technical Appendices. Education Endowment Foundation, London, available at: http://educationendowmentfoundation. org. uk/uploads/pdf/Technical_Appendices_ (June_2013). pdf.

Hmelo-Silver, C.E. & Pfeffer, M.G., 2004. Comparing expert and novice understanding of a complex system from the perspective of structures, behaviors, and functions. *Cognitive Science*, 28(1), pp.127–138. Available at: https://doi.org/10.1207/s15516709cog2801_7.

Hsiao, I.-H. & Brusilovsky, P., 2011. The role of community feedback in the student example authoring process: An evaluation of annotex. *British Journal of Educational Technology*, 42(3), pp.482–499. Available at: https://doi.org/10.1111/j.1467-8535.2009.01030.x.

Kramer, J., 2007. Is abstraction the key to computing? *Communications of the ACM*, 50(4), pp.36–42.

Kuckartz, U., 2014. *Qualitative text analysis: A guide to methods, practice and using software*, Sage. Available at: https://doi.org/10.4135/9781446288719.

De La Paz, S. & Graham, S., 1997. Effects of dictation and advanced planning instruction on the composing of students with writing and learning problems. *Journal of Educational Psychology*, 89(2), p.203.

Lee, I. et al., 2011. Computational thinking for youth in practice. *ACM Inroads*, 2(1), pp.32–37. Available at: https://doi.org/10.1145/1929887.1929902.

Lewis, M., 2000. Extending literacy: pupils' interactions with texts, with particular emphasis on the use of non-fiction texts. Available at: http://hdl.handle.net/10026.1/518.

Lister, R. et al., 2004. A multi-national study of reading and tracing skills in novice programmers. In *ACM SIGCSE Bulletin*. ACM, pp. 119–150. Available at: https://doi.org/10.1145/1044550.1041673.

Lister, R., 2011. Concrete and other neo-Piagetian forms of reasoning in the novice programmer. In *Proceedings of the Thirteenth Australasian Computing Education Conference-Volume 114*. Australian Computer Society, Inc., pp. 9–18.

Magnusson, S., Krajcik, J. & Borko, H., 1999. Nature, sources, and development of pedagogical content knowledge for science teaching. In *Examining pedagogical content knowledge*. Springer, pp. 95–132.

Mannila, L. et al., 2014. Computational thinking in k-9 education. In *Proceedings of the Working Group Reports of the 2014 on Innovation & Technology in Computer Science Education Conference*. ACM, pp. 1–29. Available at: https://doi.org/10.1145/2713609.2713610.

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

Mason, L.H., Harris, K.R. & Graham, S., 2011. Self-regulated strategy development for students with writing difficulties. *Theory into practice*, 50(1), pp.20–27.

Mayring, P., 2000. Forum: Qualitative Social Research Sozialforschung, 2. History of Content Analysis. In *Forum: Qualitative Social Research. Sozialforschung*.

Van Merrienboer, J.J. & Sweller, J., 2005. Cognitive load theory and complex learning: Recent developments and future directions. *Educational psychology review*, 17(2), pp.147–177. Available at: https://doi.org/10.1007/s10648-005-3951-0.

National Research Council, 2011. Committee for the workshops on computational thinking: Report of a workshop of pedagogical aspects of computational thinking., National Research Council.

Nind, M., Curtin, A. & Hall, K., 2016. *Research methods for pedagogy*, Bloomsbury Publishing.

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

Perrenet, J., Groote, J.F. & Kaasenbrood, E., 2005. Exploring students' understanding of the concept of algorithm: levels of abstraction. *ACM SIGCSE Bulletin*, 37(3), pp.64–68. Available at: https://doi.org/10.1145/1151954.1067467.

Perrenet, J. & Kaasenbrood, E., 2006. Levels of abstraction in students' understanding of the concept of algorithm: the qualitative perspective. *ACM SIGCSE Bulletin*, 38(3), pp.270–274. Available at: https://doi.org/10.1145/1140123.1140196.

Piaget, J. & Campell, R.L., 2001. *Studies in reflecting abstraction*, Psychology Press.

Plonka, L. et al., 2011. Collaboration in pair programming: driving and switching. In *International Conference on Agile Software Development*. Springer, pp. 43–59. Available at: https://doi.org/10.1007/978-3-642-20677-1_4.

Punch, S., 2002. Research with Children: The same or different from research with adults? *Childhood*, 9(3), pp.321–341. Available at: https://doi.org/10.1177/0907568202009003005.

Santangelo, T. & Olinghouse, N.G., 2009. Effective writing instruction for students who have writing difficulties. *Focus on exceptional children*, 42(4), p.1.

Sapir, E., 1921. An introduction to the study of speech. *Language*.

Schulte, C. et al., 2017. The design and exploration cycle as research and development framework in computing education. In *Global Engineering Education Conference (EDUCON), 2017 IEEE*. IEEE, pp. 867–876. Available at: https://doi.org/10.1109/EDUCON.2017.7942950.

Schunk, D.H. & Swartz, C.W., 1993. Goals and progress feedback: Effects on self-efficacy and writing achievement. *Contemporary Educational Psychology*, 18(3), pp.337–354. Available at: https://doi.org/10.1006/ceps.1993.1024.

Seehorn, D. et al., 2016. Interim CSTA K-12 Computer Science Standards. Available at: https://c.ymcdn.com/sites/www.csteachers.org/resource/resmgr/Docs/Standards/2016StandardsRevision/INTERIM_StandardsFINAL_07222.pdf.

Seeratan, K. & Mislevy, R., 2009. Design patterns for assessing internal knowledge representations. *Retrieved October*, 26, p.2011.

Selby, C., Dorling, M. & Woollard, J., 2014. Evidence of assessing computational thinking. , pp.1–11. Available at: http://eprints.soton.ac.uk/372409/1/372409EvidAssessCT.pdf.

Sfard, A., 1991. On the dual nature of mathematical conceptions: Reflections on processes and objects as different sides of the same coin. *Educational studies in mathematics*, 22(1), pp.1–36. Available at: https://doi.org/10.1007/BF00302715.

Sheard, J. et al., 2008. Going SOLO to assess novice programmers. In *ACM SIGCSE Bulletin*. ACM, pp. 209–213. Available at: https://doi.org/10.1145/1384271.1384328.

Statter, D. & Armoni, M., 2017. Learning Abstraction in Computer Science: A Gender Perspective. In *Proceedings of the 12th Workshop on Primary and Secondary Computing Education*. WiPSCE '17. Nijmegen, Netherlands: ACM, pp. 5–14. Available at: http://doi.acm.org/10.1145/3137065.3137081.

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

Strickland, D.S. & Morrow, L.M., 1989. Emerging literacy: Young children learn to read and write., ERIC.

Su, A. et al., 2014. Investigating the role of computer-supported annotation in problem-solving-based teaching: An empirical study of a Scratch programming pedagogy. *British Journal of Educational Technology*, 45(4), pp.647–665. Available at: https://doi.org/10.1111/bjet.12058.

Syslo, M.M. & Kwiatkowska, A.B., 2014. Playing with computing at a children's university. In *Proceedings of the 9th Workshop in Primary and Secondary Computing Education*. ACM, pp. 104–107.

Taub, R., Armoni, M. & Ben-Ari, M.M., 2014. Abstraction as a bridging concept between computer science and physics. In *Proceedings of the 9th Workshop in Primary and Secondary Computing Education*. ACM, pp. 16–19. Available at: https://doi.org/10.1145/2670757.2670777.

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

Turkle, S. & Papert, S., 1992. Epistemological pluralism and the revaluation of the concrete. *Journal of Mathematical Behavior*, 11(1), pp.3–33.

Waite, J. et al., 2016. Abstraction and common classroom activities. In *Proceedings of the 11th Workshop in Primary and Secondary Computing Education*. ACM, pp. 112–113. Available at: https://doi.org/10.1145/2978249.2978272.

Werner, L. et al., 2013. Pair programming for middle school students: does friendship influence academic outcomes? In *Proceeding of the 44th ACM technical symposium on Computer science education*. ACM, pp. 421–426. Available at: https://doi.org/10.1145/2445196.2445322.

Whitebread, D. & Basilio, M., 2012. The emergence and early development of self-regulation in young children. *Profesorado, Revista de Currículum y Formación del Profesorado*, 16(1), pp.15–34.

Wing, J.M., 2008. Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 366(1881), pp.3717–3725. Available at: https://doi.org/10.1098/rsta.2008.0118.

Wing, J.M. & Barr, V., 2011. Jeannette M. Wing @ PCAST; Barbara Liskov Keynote. *Commun. ACM*, 54(9), pp.10–11. Available at: http://doi.acm.org/10.1145/1995376.1995380.