Programming Unplugged: Insights from Theoretical Models and Teacher Experiences

---

A thesis
submitted in fulfilment
of the requirements for the Degree
of
Doctor of Philosophy
in the
University of Canterbury
by
Bhagya Munasinghe

University of Canterbury
2023

To my country Sri Lanka, the *Serendib stolen ...*

Abstract

Unplugged approaches to teaching Computational Thinking (CT), which are based on activities that do not require the use of a digital device or programming, are widely used in computing education. Evidence from the literature and practice indicates that this approach can be used successfully, although views on the value of Unplugged computing have been varied. Recently it was found that rather than comparing Unplugged with other approaches, *combining* Unplugged with teaching programming enabled students to achieve the same level of programming competence, but with higher self-efficacy, and a larger vocabulary in the programming language compared to a similar time span spent on programming alone. Despite this improved understanding of *how* to use Unplugged activities, there is little understanding of *why* they are effective and *what* ways they can be combined with plugged-in exercises effectively in a programming classroom and for teachers' professional development (PD). In this thesis we use practical observations viewed through the lenses of theories of learning to understand why the Unplugged approach is effective.

Computational Thinking in school curricula is about teaching students to understand how to use computation to solve problems, to create, and to discover new questions that can fruitfully be explored in other disciplines and professions as well as Computer Science. Teachers need to be able to effectively communicate the ideas of Computational Thinking to students and apply these within the context of their classroom. Our initial studies with teachers indicated that understanding the nature of the commonly identified difficulties and confusion caused by computer jargon among teachers is important for finding ways for effective classroom delivery. We found that the concerns from teachers finding computer jargon difficult can be because the computational context in which they are applied makes them difficult for teachers to understand, rather than not knowing their meanings in the first place, and appropriate support can enable teachers to learn the techniques and skills that the terminology refers to. Using Unplugged material in teachers' professional development, we tried to understand how they perceive the utility of Unplugged, particularly in introductory programming and understanding the jargon. Findings indicate that alternating Unplugged content in introductory programming does not hinder the teachers' teaching efficacy and self-efficacy towards computer programming, yet teachers can be equipped with more content within the same time frame as a conventional teaching approach.

Another lens that we use to understand how Unplugged and programming relate is the Notional Machine (NM), an abstract model of a computer created by teachers to facilitate learners' understanding. It represents something they can (mentally) interact with to draw learners' attention to hidden aspects of computing, is implicit in all programming teaching methods, and is a key to successful programming. We explore

how Unplugged activities seem to have a close connection with Notional Machine, and therefore use the lens of Notional Machine to understand the relationship between Unplugged and programming. Reviewing the existing Unplugged activities through this lens, we can understand where Unplugged has been successful in teaching programming and why. We also identify the possible gaps in Unplugged activities that need addressing for it to be further successful as a programming education tool. Accordingly, in our professional development experimental studies we developed and trialled new Unplugged activities focusing on modeling basic programming concepts, and studied their usefulness in alternating with conventional programming teaching practices.

The usefulness of Unplugged activities in introductory programming was then considered through the lens of Semantic Waves, a concept that describes an ideal learning journey of a novice learner over a course of learning while shifting between expert and novice understanding, abstract and concrete context, and technical and simple meanings. Studying the behavior of the Semantic Waves of Unplugged activities we saw how, heuristically, the Zone of Proximal Development (ZPD) can be seen as a differentiation of a semantic profile of an Unplugged activity, essentially shifting learners back and forth between existing and new knowledge, while learning a programming concept. The Semantic Waves of Unplugged activities used to model programming concepts were analysed and compared with a plugged-in only lessons that taught the same concepts to show how alternating Unplugged activities with plugged-in experience successfully covers a wider semantic range, indicating the possibility of avoiding both learner anxiety as well as boredom, and enabling teachers to find better teaching strategies that suit their classrooms. Semantic profiles show the balance between what learners know and what they should know about what is actually happening, and the use of Unplugged activities supports the flow needed for creating effective semantic profiles, particularly in programming classrooms.

Acknowledgments

I am immensely and blissfully thankful to the following people for so many reasons than what is mentioned here.

My parents, for *my life*. Amma and Thaththa, words fail to express how gratefully happy I am for being your daughter.

My supervisor Professor Tim Bell, for being the best supervisor and the best role-model I could learn from, follow, copy (wish I could do this better) and brag. Sir, I consider myself lucky to be your student and I will always be grateful for your guidance and support.

My co-supervisor Professor Anthony Robins, for being my mentor and a wonderful friend in the shape of a supervisor. Thank you Anthony, for your talks which often led me to many rabbit holes where I enjoyed crawling around, digging deep. It was fun!

My home University, Wayamba University of Sri Lanka, for releasing me to get this experience, and AHEAD project, for financing it.

The CSERG team, for the greatest time in Room 344, Jack Erskine. Tracy Henderson, Joanne Roberts, Melissa Jones, and Jack Morgan, thanks for your contributions in my study. You and all others at Department of Fun Stuff were the best thing that happened to me in UC. Only we know how fun the stuff really is!

All the people in CSSE Department, for all the support and care. Thanks all, for the comfort and safety I felt during this long and lonely stay away from home during COVID19 pandemic.

Associate Professor Ben K. Daniel, Dr Nadheera Ranabahu, and Dr Daniel Gerhard, for their valuable advises in research methods and data analysis, and Ceri de Boo for her support in the workshops.

All the teachers who participated in my surveys, for choosing to be a part of my research.

All my friends in New Zealand, for taking care of me. Especially Celine Aunty and Rexi Uncle, thank you for being my NZ parents; Himali, Samitha, David, and Leaf, thanks, those house parties were really fun; Chami, thanks for your calls; Manthika, thanks for the invaluable support during a very difficult time; Haipeng and Chen, thanks for the hikes, Chinese food lessons, and arguments about the sunset.

My friends, for continuously checking on me, knowing that I might have been feeling lonely. Telani, just *thanks!* Clemi and Manori, thanks for the oddly well-timed calls.

My sister Saumya, for taking care of my parents and everything else while I was away. Akka, I do not (and probably will not) speak out loud how valuable you are to me, because I know you know!

My husband Chaminda, for *being you*. Let's continue to fight over love and friendship, for those are the most we share with each other. What else do I want!

Table of Contents

List of Figures

List of Tables

Chapter I

Introduction

This chapter lays the foundation for research carried out from 2019 to 2022 that aimed to understand the nature and application of unplugged style teaching and learning in Computational Thinking (CT) and introductory programming, with a particular focus on teachers' professional development. It includes key elements of this research such as motivation, research questions addressed and the overall methodology followed. The chapter ends with a description of the remaining chapter structure and of terminology used in this thesis.

1.1 Background

'Computing' in the modern day is so broad a subject discipline that it is defined and understood in many different ways and contexts, and trying to find a singular definition is not achievable. Teaching 'Computing' at grade school[1] level today seems to have three common strands: 1) creating, evaluating and using digital artefacts (e.g. creating useful software application or digital output using software), 2) knowing how to blend computers with everyday life and work (i.e. using weather apps in agriculture and farming), and 3) learning the essence of Computer Science (CS) (i.e. programming and theories of computing). Viewed this way, 'computing' is not only about the computer as a device, but rather about a complete thinking process related to computing and computation. Computers are an important tool to learn computing, just like paint brushes are to art. Learners have to understand them and learn how to use them effectively, but the concepts of the subject go beyond that [34].

Attempts at interpreting and/or representing the true nature of what's happening inside a computer has always been open for discussion in computing education. On one hand, these attempts focus on communicating the general computational concepts to learners effectively, for general academic knowledge and effective application of computing concepts into other subject disciplines. On the other hand, particularly in computer programming education, their intention is to support learners developing correct mental models. Using some form of spatial representation (e.g. an abstract machine or a computational agent) to support technical or pedagogical explanations is common in computing. The representational and abstract nature of such pedagogic tools indicates mental comprehensions needed by the learners that can be related to their everyday knowledge, in order to understand the true nature of computing.

At a time when the first physical computers were just introduced to the world, Alan Turing foresaw how and what the thinking pattern for that newly emerging computation should be, and

[1] By grade school we refer to the elementary, middle and high school levels, also known as K-12, or primary and secondary school.

his explanation is remarkably similar to what has been described as Computational Thinking later. In his famous 1950 paper about the 'Imitation Game' for measuring Artificial Intelligence, interestingly, Turing also discussed the difference between digital computers and human computers:

> *"The idea behind digital computers may be explained by saying that these machines are intended to carry out any operations which could be done by a human computer. The human computer is supposed to be following fixed rules; he has no authority to deviate from them in any detail".* [235]

While the main part of his 'Imitation Game' discussion was about computers thinking like humans, he also talked about humans thinking like computers. How he foresaw the mechanisation of the role of a *Computer* (which was only beginning to transition from 'humans' to an 'electronic device' at that time) shows some of the earliest indications to a subtly tangible nature of CT and the possibility of representing or explaining computation with physical means. What Turing was probably trying to articulate as computing at a time when the concepts of computing were not tightly coupled to the physical device, has not been changed much to this day, where detaching the physical device from computation has become literally unthinkable. The computer is still often seen as an indispensable necessity in computing education.

Computing educators have been using physical metaphors and physical representations since the early days of computing learning. In the 1990s Tim Bell and his colleagues introduced a set of physical activities that can be used to explain CS concepts without using computers [22]; commonly known as CS Unplugged (CSU), a central repository of these activities is at `https://www.csunplugged.org`. Bell and others' unplugged computing was largely popularised in primary school education, and created a global movement around it.

One of their objectives was to provide a pedagogical interpretation to how computing happens inside of a computer, using various physical activities that do not require using any digital devices. In other words, they attempted to engage students with the 'thinking process' behind computation using physical activities, by giving a 'physical sense' to a rather intangible and unseen process within a computer. What Turing [235] was suggesting before the age of computers, that humans do not need to do the computing because a digital computer can do it, in the digital age Bell and others turned around, with the unplugged approach, to show that a human can do what a digital computer can do (for pedagogical purposes). This approach of 'unplugged computing' also resulted in bringing together many avenues of computing education such as algorithmic thinking and programming, to make meaningful connections through constructivist pedagogical approaches [20, 167].

In 2009, Jeannette Wing brought the term 'Computational Thinking' to the fore, that essentially talked about the thought process behind formulating computational problems and solutions to them. Her definition included how this 'solution' should be formulated — "in a form that can be effectively carried out by an information-processing agent [Computational Agent]" [249]. This CT definition also employs a representational element (the Computational Agent) in its elaboration, much like the other representational models that had been already used elsewhere in computing such as the Notional Machine (NM) in programming [75], Turing Machine in conceptualisation [48] or Intelligent Agents in Artificial Intelligence [182]. Wing's definition also situated the Computational Agent in a position that it can either be human or machine, bringing Turing's

Imitation Game and Bell et al.'s CS Unplugged into a range of ways students might engage with computing: virtuality represented by physicality, and vice-versa. It intertwined Turing's early efforts of articulating 'a machine's ability to think' with Bell et al.'s later attempts of expressing 'computing concepts using physical means'. Wing explicitly mentioned CS Unplugged in the same article that introduced her "information processing agent" [249].

Wing's definition of CT and her advocacy for it became widely popular, and has since been highly influential in incorporating CS in school curricula around the world since it appeared in 2006. CS Unplugged has become widely used to support CT teaching at all levels of grade school, and has also been found to work well to support students learning programming, while providing a range of curriculum integration opportunities [21]. It has also become useful for introducing teachers, who are often sceptical about the new subject, to CT. Although a rationale for the success of unplugged computing in computing education is yet to be articulated well, the above discussion indicates a strong connection for teaching CT to unplugged computing's ability as a pedagogical approach to provide explanations for computing concepts using everyday knowledge, and relating virtuality to physicality.

1.2 Motivation

Unplugged approaches to teaching CT are widely used in computing education. It has the benefit of removing barriers before students can work with the concept, such as having access to a computer, or having to learn how to program. However, it also raises the concern that it might prevent students from properly learning to use digital devices, and especially that it might become a substitute for learning to program. Some initial evaluations of using CS Unplugged activities exactly as they are written (without using a computer) found, not surprisingly, that students find difficulty in seeing the connection with computing, or how the skills related to CS [87, 227]. Stager [218] assumed that CS Unplugged activities were intended as a *substitute* for learning to program, and strongly advocated that they should not be used. Other research found that unplugged was effective as a supporting approach in CS classrooms (e.g. [118, 194, 231]). Despite strongly believing that computing should be tightly coupled to computers, Denning [62] appreciates unplugged as a "genius" way to expose how many CS concepts actually do not need computers to be understood. In short, some see unplugged computing as a wonderful way of teaching computing while some others (rarely) have suggested that it should not have happened, and some others have mixed views about it.

A turning point in the understanding of unplugged activities was the work of Hermans and Aivaloglou [115], who found that by combining unplugged activities with teaching programming, compared with a class spending the same total amount of time on programming but without unplugged activities, enabled students to achieve the same level of programming competence, but higher self-efficacy, and a larger vocabulary in the programming language. From this point of view, it appears that unplugged learning works as a *catalyst* for teaching programming, and it squarely established that the debate should not be "either/or", but how best to combine them. Despite this improved understanding of *how* to use unplugged activities, there is little understanding of *why* they are effective and *what* ways they can be combined with plugging-in exercises effectively in a programming classroom.

Moreover, little has been explored about how unplugged computing can be helpful in teachers'

professional development (PD), particularly with teachers who are new to computing and/or to programming, with notable exception being Curzon [57]. With an interest in the challenges faced by teachers who are teaching computing and programming, and developing resources to help them, and improving their self-efficacy in teaching these topics, the obvious success of unplugged computing with students (i.e. in learning) needs to be further investigated for its applicability to teachers (i.e. in teaching).

1.3 Research Questions

This thesis attempts to understand the success of unplugged computing in computing education by looking at it through the lens of some important educational theories, and with experimental studies using unplugged activities in introductory programming, that investigate their usage in teachers' professional development. In the process, viewing CT through the lens of relevant educational theories is used to understand the complementary relationship between CT and unplugged computing. The research questions addressed in these studies are:

RQ 1: How does CT relate to the mental models that students are forming? The definitions of CT and its applications to learning computing indicates psychological relationships to a learner's learning as well as cognition. In learning a computing concept, learners form different kinds of mental models and make connections, on their own and/or with the support of their teacher. This question intends to explore this and thereby provide a rationale for teachers for supporting students more effectively in their learning process.

RQ 1.1: How can models for student learning inform our understanding of learning CT?

A few well-known educational, psychological and computing theoretical frameworks are explored critically to draw out connections in understanding learning about computing, particularly unplugged computing and programming. We propose a theoretical connection between Computational Agents in CT and Notional Machines that could be useful in learning to program (explored in Chapters 4 and 7).

RQ 1.2: How does the view of CT vary with the stage that teachers/learners are working at?

This question attempts to find answers by analysing definitions and various applications of CT based on theoretical explanations to show how an incremental understanding of CT is applied in learning, particularly in programming (explored in Section 7.6, Chapter 7).

RQ 2: How does the use of unplugged as a PD tool impact teachers' confidence, expectations, and knowledge?

Unplugged style teaching has proven to increase the confidence levels of young students, suggesting that it can be used for professional development of teachers, particularly for those who are new to the subject. This question intends to explore what impacts unplugged had on teachers when using it as a teaching pedagogy in their professional development, as well as using it as a teaching aid in their classrooms.

RQ 2.1: What are the impacts of alternating or combining unplugged teaching methods with conventional "plugged-in" (computer based) methods when providing PD for teachers?

This question explores the use of unplugged as a pedagogic device, measuring the impact in terms of their self-efficacy towards programming, and self-efficacy and motivation towards teaching CT topics (explored in Chapters 9 and 10).

RQ 2.2: What are teachers' expectations for students who experience unplugged in a CT curriculum?

Following professional development experiences on CT and programming, this question explores teachers' expectations around having to teach CT in their classroom. How the PD experiences would improve/enhance/negatively impact their perspective on the subject is explored by surveying teachers' expectations after they have completed a workshop on teaching CT concepts (explored in Chapters 5, 8 and 9).

RQ 2.3: What is the nature of teachers' understanding of computational terms (jargon) related to CT concepts, and how can a relevant professional development intervention help to resolve issues related to them?

Having identified that the computing jargon used in learning and other material used in school education has an impact on teachers' taking of the subject, this question further explores how professional development would/can make an impact, by measuring teachers' perception of jargon before and after an intervention (explored in Chapter 6).

RQ 3: Why is unplugged useful when combined with programming?

The proven success of using unplugged combined with plugging-in is further explored with this question, with a particular focus on teachers, both as a professional development approach as well as a pedagogic tool. The six sub-questions approach the main research question in different angles to find answers.

RQ 3.1: What is the role of unplugged style activities in the context of the computing concepts they address?

The unplugged style activities usually address concepts like algorithms or sequence in programming and so on, which can be implemented on a computer. For this question we analyse existing unplugged activities to understand their role in representing or explaining computing concepts they intend to deliver (explored in Chapter 3 and elaborate further in Chapter 8).

RQ 3.2: Is there a need for a separate, programming focused set of unplugged activities that connects the concepts through "plugging it in"?

Most of the unplugged activities published do not purposely explain programming concepts. The ones that do tend tend to deal with sequence mostly. Are they sufficient, or would there be an effect of having unplugged activities to deal with more programming concepts like variables, iteration, etc.? (explored in Chapter 8 and 9)

RQ 3.3: What are the features of unplugged that are useful when combined with programming?

Research has shown that combining unplugged activities with plugged-in is a successful approach in introductory programming. Here we explore why this is the case, analysing introductory programming experiences with novice to beginner teachers and using Semantic Waves and theories of Zone of Proximal Development (explored in Chapter 10).

RQ 3.4: Does unplugged help teachers build a useful Notional Machine?

Understanding the Notional Machine (a conceptual computer in learning to program) accurately underpins successful performance in CT. How unplugged activities contribute to novice and beginner teachers in comprehending the concepts of Notional Machine is explored here, looking at it from several different perspectives (explored in Chapters 7, 9 and 10).

RQ 3.5: How can a more programming-focused set of unplugged activities help teachers build useful Notional Machines?

Unplugged activities are effective and efficient pedagogic tools, particularly with young learners. Understanding the relationship between programming-focused unplugged activities and their usefulness in helping teachers building robust Notional Machines, therefore, could open many new avenues in programming education such as teachers' professional development, pedagogic tools and learning material development, classroom delivery, etc. We explore this relationship by attempting to understand the connection between Computational Thinking and Notional Machines, and how unplugged activities particularly could be useful in this regard (explored in Chapter 7).

RQ 3.6: What impacts do programming-focused unplugged activities have for teachers and learners?

Due to their peculiar focus, programming-focused unplugged activities may differ from a usual unplugged computing experience. This question explores the impact that such focused unplugged activities could have on teachers and learners. We explore this in both theoretical and practical contexts. Due to the project limitations, the views of teachers who are new to computing and/or new to programming participating in professional development courses in introductory programming were considered to gain insight into the learners' perspective (explored in Chapters 4, 7 and 9).

1.4 Methodology

This thesis looks at unplugged computing through three different lenses: 1) teachers' reactions to unplugged, with a special focus on introductory programming, 2) Notional Machine, which is implicit in all teaching of programming, to explain the position of unplugged activities, and 3) the Semantic Waves concept that describes an ideal learning journey of a novice learner over a

course of learning, to study unplugged's ability to use their existing knowledge to introduce new knowledge in programming lessons.

The overall research methodology followed throughout this research study is mostly a pragmatist approach [225]. The studies conducted throughout this research are of two kinds, 1) theoretical analyses using educational theories and other literature to understand the interplay among CT, Notional Machines and the unplugged approach as a pedagogic approach to teaching programming, and 2) experimental studies with New Zealand teachers on using unplugged as a tool to teach programming.

The theoretical analyses of this research are based on empirical studies on observed and measured phenomena and derives knowledge from actual experience, supported by educational theories and literature. The empirical studies were mainly used to understand the nature and applicability of unplugged computing as a pedagogy, which is the main focus of this research.

The experimental studies, that include the pilot study and two main studies, were conducted alongside some introductory programming courses conducted by the Computer Science Education Research Group of the University of Canterbury (CSERG) in their PD programmes conducted during the period of this research (from 2019 to 2021). These experimental studies employed a combination of descriptive quantitative, randomised-experimental quantitative, and phenomenological qualitative research methods, and are reported in separate chapters. The teachers' self-efficacy in CT was studied from three perspectives (in teaching CT content, in computer programming, and motivation to teach Digital Technology subjects) before and after introductory programming workshops, followed by opinion surveys and interviews.

A mixed methods approach was used for data collection in the experimental studies conducted during the project. Mixed methods research draws on the potential strengths of both qualitative and quantitative methods, allowing the exploration of diverse perspectives, and uncovering relationships that exist between the intricate layers of the multifaceted research questions [52]. It also assists in the triangulation of results based on multiple data sources.

A thematic analysis approach was used when analysing the qualitative data collected in each study, using the NVivo qualitative data analysis software[2]. Thematic analysis is a widely used method for analysing qualitative data and is used to identify patterns and meanings across a data set related to the research questions being investigated. The transcript texts of interviews were the main focus of these analyses. The qualitative data were analysed using statistical analysis methods that suited each individual study. Non-parametric data analysis was used for the studies with small sizes and multi-level modelling was used to analyse the experiment data with complex designs where both within and between-subject shared variances needed to be considered. The qualitative approaches and statistical models used for data analysis are discussed in detail in the respective studies.

1.5 Thesis Contributions

This study contributes to the ongoing discussion of unplugged computing and its usefulness in developing CT, and in teaching and learning computer programming. It builds on the existing

[2] NVivo Software: https://www.qsrinternational.com/nvivo-qualitative-data-analysis-software/home

literature about CT, and educational theories on learning and programming education, by providing theoretical perspectives on learning to program with unplugged, as well as by providing empirical research about the use of unplugged computing activities as pedagogic tools. This thesis show that the CT definition's inclusion of Computational Agents (CA) makes the CA a relevant learning tool that makes abstract ideas concrete, and thereby becomes a simplified variant of a Notional Machine that provides both observational and operational perspectives for learners to support them in forming robust mental models. The thesis proposes that teaching programming should make use of this relationship between Computational Agents and Notional Machines at different stages of learning, as a link that connects a learner's mental model to a full Notional Machine.

This new understanding of the relationship between Notional Machines and CT is then extended to review unplugged activities, and develop new activities that can objectively model programming concepts. This study looks at unplugged computing through the lens of Semantic Waves to rationalise and further discuss how the interplay between learners' concrete knowledge and the conceptual knowledge introduced during an unplugged computing activity contributes to effective delivery of computing in classrooms, as well as cementing learners' understanding of the subject, thus provide a rationale for the complementary relationship between CT and unplugged computing. It provides insights into how unplugged computing resonates with the idea of the Zone of Proximal Development (ZPD) of a learner, by engaging them in rich social interactions with knowledgeable others and peers while maintaining their focus objectively on computing concepts.

Understanding why unplugged strategies succeed in teaching CT (and learning programming when they are combined with plugged-in lessons) is important to effectively incorporate them into teaching and learning CT topics, particularly programming, in an effective manner. With the aid of the Semantic Waves concept, this thesis rationalises why unplugged activities have been successful in the past, and why alternating or combining them in programming lessons can improve the effectiveness of teaching introductory programming. For this analysis we also adapt Dual Process Theory from psychology, to show how alternating unplugged activities with plugged-in exercises achieve the ultimate expectation of moving the programming concepts to learners' long term memory, so that the conceptual knowledge become part of their crystallised intelligence (acquired knowledge), and they can use it automatically when they program.

This study also provides insights to the applicability of unplugged computing in teachers' professional development in introductory programming, particularly with teachers who are new to computing and/or new to programming, and identifies effective ways unplugged can be used in learning to program, and in teachers' professional development.

1.6 Limitations and Threats to Validity

There were some limitations and threats to validity for the methodology used throughout this project that need to be noted. The main limiting factor throughout all of the studies reported in this thesis that involved school teachers is that the needs of participating teachers took priority over our research. On top of this, the consequences of the COVID19 pandemic (which started at the early stages of this project) caused some drastic changes in the education systems all over the world, such as rapid shifts to use an online mode for long periods and cancellation of the

in-person PD events. This resulted in many challenging changes in teachers' usual work practices and classroom discourses. These practical difficulties faced by the main target group (i.e. school teachers) frequently interfered with the main goals of this research as well as smooth conduct and data collection of the individual studies. This was an ongoing challenge faced throughout this project.

The use of mixed methods and pragmatic approaches is time consuming and requires complex research designs. Changing circumstances meant changes to the original research design as well as deviations from intended short-term objectives. The practical issues created due to the unprecedented circumstances of the pandemic moved the main focus of the research towards more theoretical explorations. The reactive nature of pragmatic research also greatly decreases its repeatability. However, despite these disadvantages and challenges, the flexible nature of the mixed methods and pragmatic research methodologies suited this research well.

The main threat to validity of this research is the low numbers of participants, which affected the quantitative data analyses. For this reason, a proper scale development and validation process could not be followed for the survey instruments. As an attempt to mitigate any effects, validated and published scales were adapted to accommodate the research specific requirements. However, scale evaluation after adaptation was limited to tests of reliability. The effect of the small sample size was mitigated by using non-paramatric methods for analysis where applicable. Nevertheless, some anticipated investigations such as the influences-based demographic data could not be usefully incorporated in the research for this reason.

The participants of the experimental studies were New Zealand school teachers. The New Zealand curriculum is quite flexible in nature, and has been incorporating Digital Technologies (i.e. CT and CS) for a reasonable length of time. Therefore, it was observed that teachers' general uptake and/or reaction to CT and learning to programming was positive and proactive. Moreover, the participation in the workshops as well as in the surveys was voluntary. This may indicate a degree of bias in representation. The potential influences of this were not investigated.

This research involved interviews that were conducted by the researcher herself. The participants were informed about the nature of the research and were participating in the interviews voluntarily and they were aware of the identity of the researcher and the nature of her research within the CSERG of UC, which has been a popular and trusted source of information and resources for DT to teachers throughout New Zealand for a considerable period of time. These could have caused several threats to the validity of the studies, of which the first is acquiescence bias; participants may feel obliged to please the researcher, and so give overly proactive and/or positive responses, compared to the what they may have given otherwise. This risk could have been reduced by having a neutral third party conduct the interviews, yet having a third party unfamiliar with the research would likely limits the depth of information gathered.

Care was taken to avoid leading questions that could bias teachers' answers. However, the reactions of the interviewer may have introduced bias to their subsequent answers. The researcher has attempted to remain impartial while interviewing, but it was observed that, in some cases she has given positive and/or supportive reactions to the interviewees' statements. Moreover, only the researcher herself analysed the interviews, and no secondary thematic coder were involved in the analysis. The impact of this situation was attempted to be mitigated by regular discussions with the supervisors and members of CSERG, which essentially introduced a level of peer review and increased supervision.

1.7 Thesis Structure

This thesis is separated into 11 chapters. Chapter 2 discusses directions in computing education and how CT has become an integral part of computing education by reviewing the literature for various views on CT, and critiquing its role in grade school[3] curricula in depth. Chapter 3 then explores unplugged approaches in computing education in depth, distinguishing the unique position of unplugged computing as a pedagogic approach, and highlighting its role in programming education in particular. The chapter attempts to dig deep into unplugged activities to analyse their applicability in different perspectives in learning, using examples. Chapter 3 extends the discussion of CT in Chapter 2 to discuss its complementary relationship with unplugged computing.

Chapter 4 is a literature review of related educational and psychology theories we used in the process of understanding the evidently successful performance of unplugged computing in computing education. Here, the unplugged approach's use of existing knowledge to introduce new knowledge is considered as the baseline for theoretical investigations. This chapter also looks at the literature related to teaching and teachers, to understand their concerns and considerations in teaching computing. Chapters 2, 3 and 4 provide the theoretical foundation and underpinning literature for the core research of the thesis.

Chapter 5 covers a pilot study conducted on school teachers who participated in a series of introductory programming workshops, which aimed to investigate the future directions for studies using unplugged computing within the scope of this project. Although the pilot study largely contributed to fine-tuning the experimental approach and instruments used, it also provided great insights to the direction of the main research as well as provided partial answers to the research questions. This chapter also covers the implications this had for the following work. Chapter 6 discusses the findings from an empirical study based on the data gathered alongside the pilot study about the teachers' understanding of jargon in a CT curriculum.

Chapter 7 covers a critical analysis and discussion on the relationship between Computational Agents in CT and Notional Machines, and attempts to provide insights on how it may or may not be helpful in introductory programming education. In this chapter, we argue that the Computational Agents can be seen as an abstract (and constantly developing), simplified variant of a Notional Machine that provides an observational (external) perspective as well as operational (internal) perspective to the learner to support them to form robust mental models of Notional Machines more efficiently and effectively. We propose that teaching programming should make use of the idea of a Computational Agents at different stages of learning, as a link that connects a learner's mental model to a full Notional Machines.

With the new understanding about the connection between the Notional Machines and Computational Thinking, designing unplugged activities that model basic programming concepts was part of this project. A detailed discussion on the usability of the existing unplugged activities in programming is given, and an extensive survey of the availability of programming-focused unplugged activities was carried out prior to designing new activities. Chapter 8 includes these discussions in detail. The evaluation of the programming focused unplugged activities developed were done in the main experimental studies detailed in Chapter 9.

[3] By grade school we refer to the elementary, middle and high school levels, also known as K-12, or primary and secondary school.

The two main experimental studies are detailed in Chapter 9, while Chapter 10 extends this discussion further, blending the experimental observations with theoretical analysis. In these two studies, carried out as intervention studies, the activities designed (as discussed in Chapter 8) were combined with plugged-in programming in teachers' PD programmes, in different combinations, to study their effectiveness. In Chapter 10, the mixing of unplugged with plugged-in exercises in introductory programming is looked at through the lens of Semantic Waves, where we propose a possible rationale for its recorded success.

Chapter 11 gives a summary of the thesis, the contributions it makes, and potential directions for future research.

1.8 Terminology

The following terms are frequently used throughout this thesis and their meanings are discussed when relevant. For clarity, summarised definitions are provided here:

Computing: A broad field that encompasses the processes of using computers and computer technology, studying about computers and computational systems, and performing calculations and information processing using computers. This includes any activity that uses computers to manage, process, and communicate information.

Computer Science: The study of computers and computing, including its conceptual body of knowledge, theoretical foundations and practical applications, and of the application of computing on data and information of other disciplines.

Computational Thinking: Considering all its different definitions, Computational Thinking can be very broadly described as the thought processes in rational thinking with a goal of producing an algorithm or a computer program/computing system. This is directly related to the thinking skills and practices used by Computer Scientists. The many definitions of this term is explored in detail in Chapter 2, and a theoretical exploration is discussed in Chapter 7.

Unplugged Computing (Unplugged for short): Systematic pedagogic approaches of teaching and learning CS concepts *without* using computers. The term "unplugged" is often used to refer to CT activities that do not require the use of a device, or at least do not require programming.

CS Unplugged: The unplugged style activities developed by Bell et al. [22], found as a central repository at `https://www.csunplugged.org`. These activities have been a frequently referred to source of unplugged activities throughout this research and often has been referred to as CS Unplugged, by the acronym CSU or by the web link `csunplugged.org`. More details about these unplugged computing resources are discussed in Section 3.2.

Unplugged Programming/Programming Unplugged: Learning programming *without* using computers or any other digital devices. Here, the teaching and learning involves programming *concepts* (such as variables or loops), since the term in its literal sense seem contradictory. The metaphoric use of the term "unplugged" indicates detachment from digital devices during the *learning process*.

Plugged-in/Plugging-in Programming: Learning programming using computers (i.e. digital programming devices and/or applications). The metaphorical "plug-in" refers to the direct use of digital device as the tool to learn computer programming, in contrast to "unplugged" activities. This has been the conventional approach for learning to program.

Non-programming Unplugged Activities: Unplugged activities that are designed to model (one or more) Computer Science concepts (such as sorting algorithms or error correction) but do not purposely focus on a particular programming concept. The concepts that these activities attempt to model are loosely coupled with programming. The context for this term is explored in Chapter 3.

Programming Unplugged Activities: Unplugged activities designed explicitly to model concepts (such as using flip-books to model variables) relevant to learning programming, and therefore tightly coupled with programming. The context for this term is explored in Chapter 3.

Chapter II

Computing Education in Schools

Computing education is a field with many facets and directions. From earlier focus on learning Information and Communication Technology (ICT) and digital literacy, the discussions have shifted to Computational Thinking (CT), a skill for all to learn [105, 249]. However, there is much debate around the meaning of CT, which has attracted the attention of both Education and Computer Science researchers alike. Despite the ongoing discussions, CT has been accepted by many and has been added as a core learning area to school curricula around the world. However, unlike long-standing school subjects, such as Mathematics or Science, this is a new area which is still in need of clarification and better explanations. Teaching CT is not a straightforward task and teachers need to be up-skilled.

As a foundation for this thesis, this chapter first discusses the many facets of computing education and looks at how its many directions have been narrowed down mostly to CT and its related definitions (e.g New Zealand curriculum combining CT with Design and Development of Digital Outcomes), which has become the representational term for many facets of CS in many grade school[1] curricula. It then looks at the many descriptions and definitions of CT and the challenges of settling on one overarching definition, as well as concerns and considerations in incorporating it in computing education. Finally, a focused discussion on learning to program is given, as a stepping stone to the rest of the discussions in this thesis.

2.1 Computing Education in Grade Schools: from Where to Where?

The importance of introducing computing into grade-school level has been under active discussion from the beginning of the 21st century. Countries worldwide have used CT (detailed in Section 2.3) as a key to incorporate Computer Science (CS) concepts and programming in their school curricula, introducing it to learners as young as five years old [13, 165, 230]. Significantly, these initiatives require learners' digital literacy to go beyond being users and consumers of digital technologies, and focus on building their skills to become innovative creators of digital solutions. This shift has naturally resulted in using the computer and computing related technologies for educational purposes. While CT is at the foreground, CS is tightly coupled with it in the background, and the CS concepts and their supportive technologies have been alien content to many, especially in grade-school curricula. The introduction of CT, however, requires teachers to be reasonably proficient with the CS content, supported through pedagogy and technology.

A serious challenge for CS education in school level is that it is typically confused with

[1] By grade school we refer to the elementary, middle and high school levels, also known as K-12, or primary and secondary school.

Information and Communication Technology (ICT) or digital literacy, and learners are taught how to use the computer as a tool aiming to improve their skills in using computer applications rather than focusing on educating them about what happens 'inside the box' or to 'control the box', to create digital outputs using it. Teachers' lack of formal computing education and/or the difficulties they face in building enthusiasm among learners to move from mere use of the computer to understanding CT, and thereby CS, add to the problem.

Introducing programming in school levels has continually been a struggle, with such introductory courses risking intimidating learners rather than enlightening them. As a result, learners could feel that programming is difficult and mysterious, or frustrating and boring. These challenges could also adversely affect their understanding and enthusiasm for CS. Moreover, as learning CS is much more than learning 'how to program' or 'how to code', these challenges cause severe adverse effects to the diversity of CS's existence as a knowledge domain in learning environments. Noone and Mooney [169] suggest that introducing learners to CS, and more specifically programming, should be done at an early age in order to best pique and maintain their interest. These observations have been supported by many researchers in recent years [107, 147, 170], which confirms the necessity of systematic introduction of CS to learners from an early stage. However, teachers need to be equipped to do this effectively.

Finding good ways to teach CT and thereby deliver CS content effectively has been in discussion over the years [10, 51, 104, 251], with arguments vacillating between everyone having to learn and understand programming concepts, and expressing algorithms as being the most important [33]. Further, the small number of teachers having a computing background has always been an issue [232]. The NZ 2018 Education Review Office (ERO) Report [120] states that the capability of teachers as the most common barrier to fully implementing the curriculum, and reports that teachers find the language used in the subject related material available for them (i.e. progress outcomes) as dense and challenging to come to terms with.

Computing teachers need pedagogical and content knowledge (PCK), which includes awareness of common misconceptions, methods to diagnose them, and ways to intervene to help learners to develop more robust conceptions. However, the technological, pedagogical and content knowledge (TPACK) [127] model widely adopted by teachers and education researchers across subject areas for understanding and designing purposeful classroom technology integration has had little research for supporting CS educators [237]. Despite many resources made available for teachers supporting them with the delivery of CS in school classrooms, systematic research and approaches for teachers' professional development (PD) in CS limited.

2.2 Problem Solving and Understanding Algorithms

The discussions on how CS should be taught has been on going since as early as 1960s, arguments vacillating between that it should be primarily about learning to program or about understanding the concepts and expressing algorithms [33, 104, 125]. Peter Naur suggested the term Datalogy or Data Science, rather than Computer Science, stating that a person's understanding and formulation of problems are relative to the tools they have at their disposal and thus their problem solving involves an understanding of those tools and the relationship between tools, problems, and person as a unified whole [33, 223]. Knuth described CS concepts as a set of general-purpose mental tools capable of developing deeper understandings in other subject areas, thus learning

them to cause "educational side-effects" [125]. He pointed out that computers and algorithms go beyond computing only with numbers, but also "with information of any kind, once it is represented in a precise way". Seymour Papert broadened this argument by stating that "certain uses of very powerful computational technology and computational ideas can provide children with new possibilities for learning, thinking, and growing emotionally as well as cognitively" [174].

Papert's idea that computing should be available for children was brought into the limelight again in early 2000s by Jeanette Wing [248], who combined the idea with the term *Computational Thinking* and suggested that education systems should add it "to every child's analytical ability". This sparked a resurgence of interest in "thinking like a computer scientist" and of the idea that this kind of problem-solving could be for everyone. Many of the more recent school computing and/or digital technologies curricula have adopted the term CT and incorporated problem-solving, algorithmic thinking, decomposition, etc. as well as programming under the same umbrella.

Algorithmic thinking, which specifically talks about the ability to understand and execute computational procedures step-by-step, evaluate procedures for both correctness and efficiency, and developing solutions algorithmically [11, 134, 220, 252], is considered an integral part of CT. It requires distinct cognitive abilities such as decomposition (breaking problems into sub-problems) and abstraction (making general statements regarding underlying concepts, procedures, relationships, and models) [220]. The concept is also tightly coupled with problem solving. Computational problem solving involves understanding problems and formulating solutions to them that can be solved by using a computer as a tool. A step-by-step process is essential when instructing to a computer, thereby algorithmic thinking becomes a necessity for computational problem solving.

Implementing an algorithm as a computer program is integral to programming [252]. This close knitted connection among problem solving, algorithmic thinking and computer programming is the essence of CT, and therefore the objectives of grade-school computing education revolve around effectively developing CT among learners, and then programming skills; although some authors may disagree with a prominence given to CT over programming, suggesting that programming should be tightly coupled with teaching and learning computation (e.g. [64]). Learners with a good understanding of algorithmic thinking can effectively generate solutions to computational problems, and can possibly transfer them to computer programs efficiently.

2.3 Computational Thinking

While Knuth [125] and Papert [174] were the earliest to recognise the significance of the thinking process behind computing in computing education, it was Wing's definition that brought CT into the limelight in computing education context in the recent past. She brought *Computational Thinking* to the fore in her 2006 article [248], and later clarified as:

> *"the thought processes involved in formulating problems and their solutions, so that the solutions are represented in a form that can be effectively carried out by an information-processing agent"* [249].

Wing's definition was not precise, and a number of characterisations of CT have appeared that cover the idea, but from a different angle. CT is the term often used to denote the

conceptual core of CS [49] and an approach to problem solving that consolidates logic skills with core CS concepts [185]. Denning and Tedre [64] define CT as the mental skills and practices used for two purposes, 1) designing computations that get computers to do what is asked, and 2) explaining and interpreting the world as a complex of information processes.

The CT skills set Wing has suggested has later been presented as some combination of decomposition, abstraction of data and functionality, generalisation, algorithmic design, and evaluation & improvement [56]. Lu and Fletcher highlight four key points of CT namely, 1) "a way of solving problems and designing systems that draws on concepts fundamental to CS", 2) "a means of creating and making use of different levels of abstraction, to understand and solve problems", 3) "a means of thinking algorithmically and with the ability to apply mathematical concepts to develop more efficient and secure solutions", and 4) "a means of understanding the consequences of scale" [143]. An important point is that CT is not about getting humans to think like computers [19], rather about developing the full set of mental tools necessary to effectively use computing to solve complex human problems. CT does not propose that problems need to be solved in the same way a computer tackles them, but rather it encourages the use of critical thinking using CS concepts [132].

Hemmendinger [114] suggests that the uniqueness of CT from other disciplines is its contribution to algorithmic thinking in making various but precise notions of complexity, something that had not been a part of the mathematical study of algorithms, where the complexity is measured by space or time required or by power dissipated. They propose that, despite this complexity, CT encourages paying attention to scalability and feasibility due to the fact that CT requires users to be resource aware. Selby and Woollard [200] also propose a number of core CT concepts, including logical thinking, algorithmic thinking, decomposition, generalisation and pattern recognition, modelling, abstraction and evaluation.

Moursund [160] explains that the word 'computational' as an important descriptor that emphasises the applicability of CT in other disciplines such as Mathematics or Physics. The author highlights the important role of CT in Mathematics, showing how computer modelers took about 30 more years to model Einstein's general theory of relativity, and emphasise that CT needs to be integrated into mathematical thinking, suggesting it as "an important component of math maturity". At times they refer to it as 'procedural thinking', emphasising the need of sequencial execution of instructions in a solution, and show that CT largely overlaps with other disciplines for having to use computers in their problem solving.

Several different views to Wing's definition can also be found in literature (e.g. [63], [64], [143], [65], [229]). Wing's definition of CT mainly differs from these other definitions by its perspective on the breadth of applicability and the nature of the computational

agents. This definition has been criticised in latter discussions, particularly regarding the information processing agent and thereby the involvement of computer programming. Denning [63] questions Wing's claim, "CT can benefit everyone and not just computer scientists", arguing that this claim has not been substantiated with empirical research. He points out that 1) CT is not an essential skill for many professions, 2) users of computational tools do not necessarily develop CT, 3) "a skilled performer of actions that could be computational does not necessarily make that person a computational thinker and vice versa", and 4) "CT enhances general cognitive skills that will transfer to other domains where they will manifest as superior problem-solving skills". He suggests that "CT primarily benefits people who design computations and that the claims of benefit to non-designers are not substantiated".

Curzon et al. [56] survey many definitions and discussions about CT to find the common themes and they suggest that CT is about developing systems that involve information processing, and it is the focus on algorithmic solutions based on computation that differentiates it from other problem-solving approaches. It has been touted as a fundamental skill for everyone, not just computer scientists, and there are good arguments for this based on students understanding the digital world that they live in; some authors take this further and suggest that CT skills can be useful in everyday situations, such as decomposing large problems into small ones [15, 119, 211], although this risks making the concepts too broad to be meaningful [63, 135, 200].

Contrasting to Wing's understanding, Selby and Woollard [200] have not included problem solving in their definition claiming that, despite the CT definition appearing to contain problem solving, it is not sufficiently specified to include as part of the definition. David Barr, John Harrison, and Leslie Conery. Barr, Harrison and Conery [15] and Bers [25] indicate disagreements with Wing's CT definition claiming that it is not enough to solve problems with CT but the solution must be implemented using a computational device. Denning and Tedre [64] point out that CT for a beginner differs from that for a professional: the former is a simple, practical understanding of computing concepts and the latter is critical, complex and technical.

Many introductory computing courses tend to be about programming, but usually programming is a relatively small part of the curriculum (e.g. [122]), and by having this a a major part of the first course, it can give the wrong impression about what the subject of Computer Science is about. For example, Denning and Tedre have noted the misconception that "CS is about programming" [64].

These discussions indicate that the CT reaches beyond the context of mere coding (i.e. the translating ideas from natural language into machine commands through an intermediary coding language) and relates more towards programming (i.e. developing a well functioning software solution systematically). Despite these many views about its operational applicability in curriculum contexts, we believe that CT definition holds a

much deeper meaning and connection to learning to program, with its special focus on 'Computational Agent' and its unique positioning in both physical and virtual realms. We believe that this representational element plays a similar yet simpler role to a concept referred to as a Notional Machines (NM) in programming education, and thereby can be used with a purpose similar to NM, particularly in early programming education. This proposition is discussed in detail and its applicability is elaborated in Chapters 4 and 7.

2.3.1 Computational Thinking in Computing Education

In order to obtain a ubiquitous understanding of computing across teaching and learning, it is essential to find ways to teach every student the theory of computation as well as the practice of programming in a way that makes sense to them. Guzdial [108] suggests that one way to achieve this is through making CT compulsory in education. It can be seen that terms such as "Computer Science", "Computing", or "Digital Technologies" in curricula are used to cover the context of CT, and CT is used as a term to represent this content outside computing curricula as well. However, the diverse views of what CT is and the diversity of the ways it can be adapted in teaching and learning has caused the contexts of CT to appear complicated and messy to those who are new to computing, especially to novice teachers. CT teaching efforts needs to be focused to transform it to become part of students' four possible primary skills (critical thinking, collaboration, creativity and communication [35]). It is useful for the educator to realise this and that CT teaching is not to exclusively teach them how to code, because their objective should be to equip young students with potential for CS, programming or the application of CT in other disciplines and/or be good digital citizens [230]. Programming is an important part of CT, but hardly covers the entirety of it. Teachers having limited computing background makes it difficult for them to realise these distinctions. This section looks at some of the different views, concerns and suggestions made by various authors about CT in computing education.

Lu and Fletcher [143] suggest that foundations of CT should be laid long before students experience their first programming language. This claim has been supported by many other researchers [77, 108, 185]. Webb et al. [244] suggest that, "as their expertise in CT develops, students would be expected to use their skills and build their understanding of applications of CT via a range of examples across different subjects". They list CT as a core component of CS and an important 21st century skill. However, they also view CT as difficult to implement in schools due to its complexity. Sherrell and Qualls [185] suggest that "the idea of CT is to integrate computational techniques and approaches into all disciplines requiring problem-solving skills" and suggest promoting CT through the means of a CS curriculum with an aim to make it commonplace.

In her earliest discussion, Wing expressed the need for teaching CT in every school and

to every student, arguing that it can be regarded as several thinking patterns or habits of mind [248]. She described CT as a set of "mental tools" that is important for everyone and should be a part of basic education of everybody and not limited to computer scientists. In K–12 education, some curricula have embraced CT by interpreting Wing's viewpoint of its definition - the "information processing agent" need not necessarily be "computational agent" at some levels, and CT as part of the general education of every child (e.g. [230], [165]).

Many experiments involving teachers (as the educators of CT to school students) show that they have common misunderstandings about CT both before and after they are trained on the subject. These misconceptions range from thinking 'delivery of CT requires knowledge and usage of computer or technology', 'teaching CT is about only problem-solving', to 'it is related to mathematical thinking' or 'a scientific skill'([160, 253]). According to Sherrell and Qualls [185], the underlying misconception that equates CS with programming is the first step in paving a way for bringing the importance of CT into the discussion.

According to Corradini et al. [49], the most widespread methodologies for teaching CT are: 1) teaching to program, often with a programming language that suits the learner's age and environment, 2) unplugged activities, in which CS concepts are taught without using computers, and 3) use of educational robots that must be programmed. Moreover, empowering teachers with valid information about CT is claimed to be a key element for inspiring students to explore computing as well as its various applications across different disciplines.

2.4 Learning to Program

Learning programming has been a central part of CS education over the years. Papert has proposed that programming is the most powerful medium for developing the sophisticated and rigorous thinking needed for other subjects and that it should be a key part of the intellectual development of students growing up [174]. This provides an attractive rationale to situate computing in cross-curricula contexts [4]. Effective teaching and learning of programming has also been the interest of many research projects and discussions. The diversity of the ages of students exposed to computer programming is vast, ranging from as young as 5 years [77] to adult students. The teaching approaches are also equally varied, ranging from traditional programming teaching methods that directly moves into coding (Instructionism) to more modern teaching techniques that use other teaching approaches such as Unplugged computing or adopting the PRIMM (Predict-Run-Investigate-Modify-Make) model [206] (Constructivism) [90, 127, 240].

Many researchers have highlighted that teaching methodologies are often more important than the actual programming languages of choice [168]. The effectiveness of tradi-

tional, mostly instructionist, methods of teaching programming has been questioned over the years, for it has been observed that students lack understanding concepts, which was also seen as the cause for the decrease in undergraduate level students' interests to pursue further exploration and self-experimentation [190, 213, 226]. Milne and Rowe in [156] argue that the reason behind students' inability or difficulty in understanding deeper programming concepts is that they are incapable of creating a clear mental model of program execution. In [226] and [213], the authors have observed that the use of practical situations has been very helpful to learn programming more efficiently whereas learning through lectures can decrease their interest level. They further observe that students found designing a program to solve a certain task as the most difficult for them [213].

These observations mainly indicate two things: 1) understanding programming is easier if it is related to students' knowledge from their environment (prior knowledge), and 2) students lack skills in visualizing the problem and solution spaces. They can be explained as caused by the students' lack of CT skills when they engage in problem solving and attempt programming to implement solutions. There is literature reporting that negative perception, motivation and attitude are some of the greatest barriers in programming education, and their influence is much greater compared to the other factors [172, 180, 226]. Korkmaz [129] suggests that "in [the] Scratch environment, the programming structures are more physical (visual) and the findings [results] can be monitored physically in a thematic environment" and that it contributes positively in students' algorithmic thinking.

Considering all the above discussions, learning to program seems a demanding task that a person may require the skill of CT to describe a problem and propose a solution, possibly followed by the need to design and program in order to convert the solution into the syntax of a programming language. Establishing a good CT foundation from early ages of computing education may possibly dissolve these barriers and motivate students to see the area as more than just coding and make learning to program a much more enjoyable learning experience.

Chapter III

The Unplugged Approach to Computing Education

If computer programming is the main focus in teaching and learning Computational Thinking (CT) and Computer Science (CS), because the nature of programming is to have formal and technical details, it could possibly hinder the understanding of other important aspects of computing such as problem solving and algorithmic thinking. In fact, algorithmic thinking, which is essentially a part of CT, could be considered an individual cognitive skill closely related to a person's ability to formulate abstraction, and can happen without the use of a computer. When teaching and learning computing, it is natural to use a computer or a device that is similar to a computer, but this natural connection between computing and the 'computer' can overshadow the potential avenues of developing the cognitive skills outside of the use of a computer. A pedagogy that focuses on dialogue and encourages questioning therefore could be more relevant in computing education earlier on, in order to achieve a degree of balance between these two aspects later. 'Unplugged computing' facilitates this by paving ways to connect existing concrete knowledge to new conceptual knowledge.

This chapter discusses 'unplugged computing' in detail, providing contexts to both understanding its various uses in computing education, and facilitating the 'unplugged' discussion in the rest of this thesis. It reviews the many different uses of the term 'unplugged' in education, and the distinction between unplugged activities intended to teach CS in general, compared with a focus on programming skills. A variety of sources of activities are surveyed, and we narrow the focus of unplugged activities that will be considered in this thesis.

3.1 Unplugged Style Teaching and Learning: Various Views

A wide range of teaching and learning tools and approaches that avoid electronic or digital devices, such as the use of metaphors and analogies, physical activities like hand clapping or dancing patterns, and the use of tools like a flowcharts, puzzles, etc., is often used in education. They are not constrained by formal structures and details, yet successfully facilitate dialog and questioning to draw out ideas from the learner and connect them with new knowledge. Many of these alternatives, that leave devices behind completely or do not use them directly, are frequently associated with terms like 'unplugged' or

'offline'. Therefore, it is worthwhile to look at the various uses and views associated with the term 'unplugged' in teaching and learning in order to formally distinguish them from what we will refer to as 'unplugged computing'.

Teachers may opt to incorporate offline activities (e.g. discussions, journals, debates, etc.) into their teaching for either pedagogical reasons, or for reasons such as to purposely minimise cognitive load, to ensure concentration, because of negative perceptions around "screen time", or due to resource limitations (e.g. computers, network, power). In English language teaching, the term 'Teaching Unplugged' is used for a teaching method and philosophy that encompasses teaching through conversation, taking out external input and letting the lesson content be driven by the students rather than being pre-planned by the teacher. This method, based on the 'Dogme ELT' [153], is similar to some unplugged approaches in computing classrooms as well (e.g. reading and tracing code, brainstorming, discussions).

Sometimes in computing, the term 'unplugged' is also used to indicate the physical disconnection from a computer or to indicate the use of wireless devices and/or communication. In computing areas related to network communication, the term is sometimes used to distinguish wireless communication (e.g. WiFi) from Ethernet, and also disconnected or loosely connected physical Ethernet cables. However, the unplugged computing in this thesis does not include these contexts in its discussion.

Unplugged computing can also refer to doing things away from a *conventional* computer. In such situations, the distinction between the device usage and unplugged learning can become vague. A good example is a 'Beebot' [1], a simple computing device that is not physically connected to a computer. Using a Beebot is often seen as an unplugged activity; a lesson away from a computer. Moreover, there are many emulator applications available online that simulate a Beebot on a computer. Consider two teaching scenarios with a Beebot, 1) use as a programming device to teach the basics of programming (especially sequences), and 2) use as pedagogic device to teach basics of directions (left, right and forward) or arithmetic (e.g. 5 forward and 3 back is equivalent to 2 forward) with no reference to programming. Given that the physical Beebot is a simple computing device, it is not clear which of these scenarios can be considered unplugged computing. And, the use of a Beebot emulator seems equivalent but happens to be on a computer. Similarly, is a teacher using the whiteboard to draw a figure to show how a 'stack' works in a programming classroom unplugged computing? Moreover, the inevitable use of metaphors, analogies, puzzles, games and the like in academic explanations also falls under the same 'unplugged' umbrella and have been integral pedagogic tools in all academic disciplines including computing. Another distinction of the use of unplugged in is using

[1] Bee-Bot Programmable Floor Robot: https://www.tts-group.co.uk/bee-bot-programmable-floor-robot/1015268.html#

unplugged demonstrations where a small number of people actively take part, versus CS Unplugged [54] style (that were originally intended for outreach) activities that involve large groups or a whole class.

The collective answer would be that, different interpretations and views of the term 'unplugged' inevitably exist and, there is no black-and-white distinction between 'unplugged' and 'not unplugged'. For the benefit of this thesis, the learning experiences that purposely withdraw from the context of computing (i.e. completely away from computers) and use learners' existing knowledge to introduce a new computing concept are collectively referred to as 'unplugged computing' (with a lower case 'u'), and the teaching and learning approaches or techniques that literally 'do not use a computer or any digital device' are referred to as 'offline'. Accordingly, unplugged computing is a subset of offline activities, with a very specific focus and purpose about computing.

For example, a 'debate' would be an offline activity. One may consider a debate on whether "using Artificial Intelligence to detect people's internet search patterns is ethical or not" as an 'unplugged computing' activity. Yet, such a debate may have more social context than computing context (and next to none of computing concepts), and does not use users' existing knowledge to introduce a new computing concept. Such learning experiences would be considered as 'offline activities', but not 'unplugged computing'. On the other hand, using knitting to introduce sequences (with the teacher objectively guiding the learner's existing knowledge to a introduce a computing concept) would be 'unplugged computing'.

3.2 Unplugged Computing

Different senses of the word *unplugged* can be seen even within the computing contexts. It is often associated with physical activities (e.g. knitting, hand clapping, dance patterns), regardless of them being carefully planned lessons or made available as systematically developed resources. As mentioned earlier, 'unplugged computing' goes beyond the pedagogic approaches and techniques of 'not using a device', and is about withdrawing from the context of computing to learn about computing. Unplugged computing does not need a computer at all and is about moving away from its usual place of learning, which is the computer, to learn about the concepts that create it.

Unplugged computing is (mostly) a constructivist pedagogic approach that uses learners' existing knowledge and familiar contexts to explore new computing concepts and potentially avoid having to learn a computer programming language before learners can start learning computing concepts, particularly for younger students [18]. Bell and Vahrenhold [22] have explained that the key principles of Unplugged approach as:

- avoiding using computers and programming,

- a sense of play or challenge,
- kinesthetic,
- follow constructivist approach,
- short and simple explanation, and
- a sense of story.

Therefore Unplugged is a more reachable way to introduce children to CT and CS [58, 231], and when children are introduced to computing concepts using unplugged before they are exposed to programming. This has turned out to increase their self-efficacy [115]. Waite [240] proposed that teaching computing without a computer, or the 'unplugged pedagogy', could be classified as an instructional technique.

Unplugged computing has been gradually becoming a systematic pedagogic approach in computing education over the years, with more and more carefully designed and tested learning activities that directly transfer CS concepts without using computers or similar devices. Unlike offline pedagogic tools of other academic disciplines, these activities have a direct relevance to computing and/or computers, mainly because computation, and therefore the 'computer', is at the heart of CT and not a mere learning tool, thus complete separation from a computer is fundamentally meaningless in computing. In computing, it is essential that both teachers and learners are capable of distinguishing the many roles a computer plays, and aptly equip themselves with knowledge to effectively handle the device according to their objectives and needs.

A number of authors have adapted CSU material or created unplugged activities to overcome the barriers in traditional introductions to computing topics that have been difficult to access for those with a weak background, particularly in programming or mathematics (e.g. [22, 58, 92, 231, 251]). Nevertheless, it should be acknowledged that if the purest form of unplugged activities are the only learning tools the learners use, it does not connect them with computing and therefore becomes unreliable [87, 227].

Most of the studies about unplugged computing in literature have used the popular CS Unplugged project (detailed in section 3.2.3) as their main resource material. This collection of activities covers a range of CS concepts introduced using simple kinaesthetic activities. Due to their popularity, systematic presentation and the wide range of CS concepts covered, these activities are often used as the basic resource repository for unplugged computing. For the benefit of this thesis, the CS Unplugged resources from the original authors as found on csunplugged.org are referred to as CSU activities or CSU.

Hylke et al. [118] report that "the unplugged aspect of the lesson materials seems to elicit positive reactions from both teachers and students" and suggest that CSU activities are "a valuable alternative to regular, online programming lessons". Curzon et al. [58]

suggest that "unplugged activities make for an inspiring and fun session for teachers that they also find useful, interesting and confidence building". The unplugged style of teaching is an effective and reassuring teaching approach for teachers with even limited or no prior computing background and provides a gentle introduction, avoiding the fear some teachers have around programming. Thies and Vahrenhold [231] suggest that unplugged activities are well suited for outreach as well as for introducing new topics in class. Moreover, it is also observed that it is an effective approach when a short time slot is available and/or there is a large audience [251].

Synthesising unplugged computing activities that might allow students to see analogies between various computing concepts or between CT and other sciences/real life experiences. This will lead them to see the breadth of computers' applications through analogy. It can also lead to meaningful curriculum integration and extended understanding between computing concepts and real world applications and vice-versa (e.g. understanding iteration through repeated actions in different subjects like music or crocheting, and realizing the possibility of implementing them in a computer program using 'loops'). Moreover, unplugged activities may be effective when introducing unfamiliar concepts [251]. Rodriguez et al. [194] confirms that priming activities that address naive or pre-existing ideas are very helpful in fostering understanding. They can be an effective teaching methodology when blended with other teaching methodologies to produce improved end results [115, 251].

The unplugged approach is also useful in explaining computing concepts that are either difficult or cannot be proven/implemented using a computer. A good example is the Halting Problem, which states that it is not possible to write a program for a Turing Machine (see Section 4.4.2) that can predict whether or not any other program halts after a finite number of steps. The only way to prove this claim is to prove by contradiction and most certainly by unplugged means. Alan Turing [236] proved that the Halting Problem is undecidable using a Turing Machine, using mathematical means. The fact of the matter is, there exist problems which are impossible or extremely difficult to solve computationally, yet can be explained more easily and clearly using unplugged methods. Thus, unplugged computing can be very useful in computing education, especially where programming examples may fail or mislead a learner in understanding the concepts behind what is learnt.

Unplugged teaching generates a high level of understanding about the concepts of algorithms, logical predictions and debugging among the students compared to other methods, while improving self-efficacy in programming [115]. However, not working with technology (machines) is a risk that can make learners feel like 'stepping backwards' [115, 251] and they may not even connect the activities with programming completely [87, 227]. Another potential risk is that teachers might completely avoid programming and use unplugged only, due to their lack of knowledge about programming, which can be unhelpful.

In unplugged computing, therefore, once the learners' understanding is secure, they are encouraged to connect their learning back to plugged-in computing using the computer. This is often facilitated via either introducing them to programming, or a teacher explaining/demonstrating the actual, applied computing context. This connecting back or relating back to combine with plugged-in computing essentially scaffolds and cements the learning while helping learners develop robust mental models.

Although they are most commonly used as teaching and learning exercises/tools for introducing CT or general CS concepts, unplugged activities are also often used in introductory level computer programming education, particularly in K-12 level and/or with novice audiences, due to their simplicity and their connection to physical world context. It is observed that where CS is introduced in school curricula, particularly to primary level, unplugged activities are very popular among teachers as a successful teaching as well as an outreach tool. Feedback from teachers often indicates that the physical and playful nature of unplugged activities is engaging.

Popular CSU activities have been critiqued by some authors [227, 231, 240], some recommending the need to adapt learning for specific class settings. Theis and Vahrenhold [231] stated that CSU activities do not cover two core levels in Bloom's taxonomy: 'Evaluating' and 'Creating' and do not extend to 'meta-cognitive knowledge'. Incorrectly assuming that unplugged activities were intended as a *substitute* for learning to program, Stager [218] strongly advocated that they should not be used. Apart from these pedagogic disadvantages, unplugged computing may also have the disadvantages of not holding up to people's stereotypical expectation of having to use a computer in learning computing, or appearing outdated and old fashioned. Moreover, a less experienced teacher can feel nervous about not knowing what they are going to teach in advance, not clearly understanding the computing concept(s) the activities intend to communicate, nor being able to make meaningful connections. In an unplugged lesson anything can come up, and if the teacher is unable to deal with the unplanned, the students could lose confidence in them.

The effectiveness of unplugged activities may vary or be highly dependent on both careful design and execution. For example, the use of the physical analogy of an "envelope" to explain the concept of a variable is essentially an unplugged style approach but, it may or may not develop the expected impression on the learner (such as that the value of a variable can change, yet it can have only one value at a time), depending on how the teaching activity/explanation is executed in the classroom. An envelope can hold more than one thing at a time, therefore a mere reference or not specifically mentioning of 'hold one thing at a time' can cause simple yet serious misconceptions. Nevertheless, an improved analogy can improve the unplugged activity. For example, the Box Variables

activity[2] that is discussed later in this chapter aims specifically to overcome that by theatrically shredding old things when something new is stored.

A mere reference to a physical analogy can be considered as unplugged, but may be poorly defined. Nishida et al. in [167] have provided specific guidelines on how CSU activities can be designed to minimise/avoid such issues. Moreover, in order for an unplugged activity to be completed, some form of assistance must be available for the learner to make/show the connection between the contextual understanding they gain from the activity and computational concepts. In the absence of such, they could easily become a simple physical/mathematical exercise. Taking into account the considerations presented thus far, we suggest that teachers can use unplugged activities to 1) develop CT through analogical reasoning; 2) highlight the links between computing concepts and thereby facilitate the creation of computing artifacts; 3) encourage students to see analogies between CT and other disciplines or real life; and 4) teach students that not all analogies might prove to be correct, or facilitate learning by demonstrating concepts that cannot be shown through programming.

3.2.1 Unplugged Computing and Computational Thinking

Based on the premise that cognitive processes are deeply related to bodily activities in the physical world, Sung et al. [222] show how "a greater degree of bodily engagement supports the perceptual experiences of learners by providing concrete experiences". Moursund [160] also implies a similar connection between CT and Papert's [174] idea of association between procedural thinking and kinesthetic activities. This is actualised well by the unplugged computing activities. The unplugged activities in CSUnplugged.org site specifically connects each activity to CT concepts (algorithmic thinking, abstraction, decomposition, generalising and patterns, evaluation, and logic). Aranda and Ferguson [8] state that unplugged computing is a key to valuing a multi-modal approach to CT, and suggest that the engagement of the body as part of a larger distributed system is important in a learning process. They propose that this aspect must be further investigated in the context of unplugged computing.

Rodriguez et al. attempted to map Bloom's taxonomy levels and CT using CSU based projects, and found positive results in knowledge retention and progression [195]. Looi et al. [141] propose that teachers can develop CT through analogical reasoning using unplugged activities and can highlight their links with computing concepts to facilitate creating computing artifacts. Rodriguez et al. in [194] confirmed that priming activities that address naive or pre-existing ideas are very helpful in fostering understanding. The results of the studies using Unplugged computing activities have shown that they help

[2] Box variables: https://teachinglondoncomputing.files.wordpress.com/2014/02/activity-boxvariables.pdf

learners in developing CT skills and learning CS concepts, and positively influenced their interests towards CS and improve their self-efficacy.

3.2.2 Unplugged Computing and Learning to Program

Research indicates that the unplugged approach have many advantages in teaching to program in particular. In [1], authors have reported that the students who used physical manipulatives performed better at rule construction, and students who then engaged with the programming environment had a better mental simulation of the rules and a better understanding of the concepts than those who did not use such physical manipulatives. Students engaged in "cyber-related" learning also have shown increased understanding of the material, following an unplugged project [92]. Hylke et al. [118] suggest CSU activities as a valuable alternative to regular programming lessons.

Even in teaching programming in particular, the unplugged activities are used with different twists. Using semi-deterministic devices (e.g. Bee-bots) or devices that have no physical connection to a computer (e.g. Micro-bit), and learning completely away from a conventional computer or any other device have all been referred to as 'unplugged'. Using unplugged computing activities that model general CS concepts and not a programming concept in particular prior to introducing programming is also sometimes referred to as 'unplugged programming'. It is observed that the unplugged activities that can be directly converted to a simple, straight forward programming exercise are limited; most of them require advanced programming skills if they are used as programming exercise examples. The CSUnplugged.org site provides such examples under 'CS Unplugged Plugging it in'[3], where Blockly, Python and Scratch are used to provide programming challenges with growing difficulty to implement an unplugged activity, for example Binary Numbers or Kidbots. Analogies to physical objects (e.g. "variable is like an envelope"), metaphors (e.g. "imagine the queue data structure as a checkout line in a store") or mere mentions of day-to-day activities as examples (e.g. a daily morning routine as an example to a sequence) in teaching computer programming are also sometimes referred to as unplugged programming.

Despite the advantage that an unplugged approach does not require learning programming before engaging with an algorithm, working with a computer program is needed for a complete experience of understanding the implementations and limitations of an algorithm, and in knowing the value of the ability to follow many instructions reliably. The computer's role in teaching programming has also been in the discussion over the years, with researchers being curious about whether programming should be taught with or without computers.

Dwyer et al. [79] used CSU activities to study elementary students' pre-instructional

[3] CS Unplugged Plugging it in: https://www.csunplugged.org/en/plugging-it-in/

ability to develop algorithms and step-by-step instructions. Their findings indicated that the learners were capable of analyzing the existing instructions for deficiencies, leading to incremental improvement, yet were unable to produce thorough, precise instructions themselves. The learners were able to solve smaller problems, but had difficulty describing a working algorithm when problems scaled. The authors found that the students were focusing on accelerating the mechanics of the operation (i.e. doing the prescribed unplugged activity or completing it faster) rather than the algorithm itself, when using the unplugged approach. This could be a potential concern when using unplugged as a classroom activity, or it could be used to have students think about the value of a fast computer compared with a fast algorithm.

Students can engage in interactive activities that demonstrate and explain CS algorithms and concepts, independent of actually implementing those concepts using programming [4, 40]. But, discussions about how unplugged activities could contribute to the development of mental models when learning to program (i.e. understanding a Notional Machine) are rare, compared to the considerable discussions available about the same using a computer (e.g. using visualization software) [141]. Given the flexibility Wing has provided in the definition of a Computational Agent (CA) in her CT definition (i.e. a CA can be either human or machine as long as it follows instructions precisely and blindly), and considering Moursund's [160] take on Papert, unplugged computing seems to have a good potential to provide a rather physical sense to the concept of Computational Agent and thereby Notional Machine when they are used to teach programming concepts in particular. This is discussed in detail in Chapter 7. However, longitudinal studies are usually needed to study the causal impacts of unplugged activities in these areas [207].

The engaging nature of unplugged makes it a good teaching tool to provide a gentle path into novice programming as well as in challenging misconceptions [168]. Studies have focused around the traditional straight to programming (plugged-in) approach versus mixing unplugged in introducing the programming concepts before programming. In a study of elementary school students in [115], the authors have concluded that the best approach is a combination of both plugged-in and unplugged, and found that despite no difference in mastering of programming concepts, the group exposed to unplugged computing prior to programming was more confident of their ability to understand the programming concepts.

While computer programming is primarily about using the computer as a device, which contrasts with the basic concepts of the unplugged approaches, the findings in [251] and [115] squarely sets the scene that the debate shouldn't be "either/or", but how best to combine unplugged with programming. Though many of the studies have demonstrated the success of unplugged in registering the CS concepts in students and improving their self-efficacy, the long-term effects of alternating or combining them with conventional

plugged-in teaching needs to be studied further.

3.2.3 Unplugged Computing Resources: CS Unplugged and Other Organizations

The unplugged approach of introducing CS concepts was popularised with the *Computer Science Unplugged* (CS Unplugged) project[4] in the 1990s. Its genesis goes back to the early 1990s where Mike Fellows, a mathematician working on mathematics outreach crossed paths with Tim Bell, a computer scientist working on popularising great ideas of CS, which led to the collaboration that became known as Computer Science Unplugged. The name *Unplugged* was influenced by the style of music in the 1980s where musicians were recording versions of their music using acoustic guitars instead of electric ones, avoiding music cluttered by technology, and particularly the award winning album of the same name by musician Eric Clapton in 1992. This idea of avoiding technology to appreciate the 'real thing' resonated with what Fellows and Bell were doing, trying to avoid technology distracting students from appreciating the real means of knowledge. The now widely used CS Unplugged (CSU) material includes the early work of the original authors and material created by others later [21]. CSU is a collection of activities and lessons that can be used to teach CS concepts without using computers or any other digital technology and with no prior knowledge in CS. These activities are often identified as 'Unplugged computing' with a capitalised 'U', and are found as central repository at the web link `csunplugged.org`

From its introduction in 1998, CSU material has since become one of the most popular unplugged toolkit in teaching CT and other CS concepts. Now they are released web based, under a Creative Commons Attribution-ShareAlike license[5]. First introduced and popularised with the objective of exposing big ideas of mathematics and CS to children or general public without having to learn programming, mostly targeting outreach earlier on [22, 55], Unplugged computing gained a boost in computing education after the introduction of Computational Thinking (CT) [248] or Computer Science in grade school[6] curricula. CSU material is also by far the most used Unplugged material in many research studies (e.g. [39, 115, 146, 194, 195]), as well as the most reused, modified and/or adapted material [41, 112, 202]. Research on unplugged computing mostly uses the original set of CSU activities, variations of them, or combinations of them with new activities [76]. They have been adapted by many to overcome the barriers in traditional introductions

[4] CS Unplugged: `https://classic.csunplugged.org` and later the updated version `https://www.csunplugged.org`

[5] Creative Commons Attribution-ShareAlike 4.0 International license: `https://creativecommons.org/licenses/by-sa/4.0/`

[6] By grade school we refer to the elementary, middle and high school levels, also known as K-12, or primary and secondary school.

to topics that have been difficult to access for those with a weak background, particularly in programming or mathematics [40, 79, 92].

A range of other organisations and people have also developed many different activities and lessons that follow the unplugged approach. Cutt's *CS Inside* [60] was an early project that used unplugged in a concerted way. Another popular collection is *Computer Science for Fun* (CS4FN)[7] that includes a collection of activities to make learning computing fun whether as a hobby, in a computing club or in lessons. CS4FN was created by Paul Curzon and Peter McOwan in 2005, under a public engagement project with an objective to enthuse and teach both students and others about inter-disciplinary CS research [55], and their resources are collected on the Teaching London Computing site[8]. Barefoot Computing[9] and Code.org[10] also contain collections of unplugged activities, mostly extended versions or adaptations from CSU activities as well as original material.

3.3 A Closer Look at Unplugged Computing Activities

CS Unplugged[11] provides 10 principles that their Unplugged activities (that extends from the original principles mentioned in [22]) are based on (i.e. no computers required, real CS concepts, learning by doing, fun and engaging, no specialised equipment needed, open for variation, adaptation or extension, for everyone, co-operative, stand-alone activities, and resilient). Future Learn [96] defines it as "tasks that take place away from a computer in order to model key concepts (e.g. selection, variables, algorithms) in different ways". Some authors refer to these wide range of unplugged activities as just 'Unplugged Programming' (e.g. [8]), which has the problem that it could narrow down the focus. For example, an activity like Parity Magic[12] does not relate directly to programming, and therefore that term would exclude it.

The different senses of uses commonly acknowledge the fact that unplugged computing avoids some of the problems with conventional computing teaching and learning (i.e. using computers as often) such as increased screen time or limited physical engagement. The 'unplugged method' always refer to some form of existing knowledge, a very traditional and successful technique in education: introducing new knowledge based on

[7] Computer Science for Fun: http://www.cs4fn.org/

[8] Teaching London Computing: https://teachinglondoncomputing.org/

[9] Barefoot Computing: https://www.barefootcomputing.org and https://www.stem.org.uk/resources/collection/4227/barefoot-computing

[10] Code.org: https://code.org

[11] Classic CS Unplugged Principles: https://classic.csunplugged.org/about/principles/#real-computer-science

[12] Parity Magic: https://www.csunplugged.org/en/topics/error-detection-and-correction/unit-plan/parity-magic/

existing knowledge. Accordingly, the following two distinctions of unplugged activities have been identified and are used, for the benefit of this thesis. A further discussion is given in Section 3.3.1.

Non-programming Unplugged activities: The unplugged activities that are *designed to model a computer science concept and are rather loosely coupled with programming* (e.g. Parity Magic[13]). The CS concepts of interest here go beyond programming. A student does not need to know programming to understand these concepts, nor would they need to use their learning to be applied into a computer program; thus it is loosely coupled to programming.

Programming Unplugged activities: The activities that are *designed explicitly to model concept relevant to learning programming, and therefore tightly coupled with programming* (e.g. Box Variables[14]). Here, by 'programming concepts' we largely mean the basic concepts that are most relevant to beginner programmers and are most commonly covered in a typical introductory programming text book (e.g. variables, expressions, selection, sequence, iteration, etc.). Although some of them are general computational terms that are relevant to human activity and not just programming activity, these concepts are coupled with programming rather tightly.

Almost all the unplugged computing activities can end up as computer programs; some (such as Box variables) are straight forward and simple, some (such as Graph Algorithms) are complex and may need advanced programming skills. Naturally, the Programming Unplugged activities therefore, are more straightforward for use in a plugging-in exercise (i.e. converting to a computer program), compared to Non-programming Unplugged activities.

3.3.1 Learning with Unplugged Activities in Perspective

Apart from their engaging nature and physical representation, unplugged computing activities are versatile in modeling computing concepts from two perspectives: the general computing context and a programming context. This section attempts to understand the unplugged activities in those different contexts, with the aid of a few examples. Table 3.1 contains a set of selected unplugged activities, with a summary of contexts in which they can be used in a computing classroom. The unplugged activities in the Table 3.1

[13] Parity Magic: https://www.csunplugged.org/en/topics/error-detection-and-correction/unit-plan/parity-magic/

[14] Box variables: https://teachinglondoncomputing.files.wordpress.com/2014/02/activity-boxvariables.pdf

are from popular unplugged computing websites CS Unplugged[15] and Teaching London Computing[16].

An introductory programming book would typically introduce variable declaration and data types by simply mentioning their ranges, without providing any clear and simpler explanation (e.g. [73, 101, 123]). For example, in [123] the function of int and float data types are given as " ... 16-bit ints, which lie between -32768 and +32767, are common, ..." and "A float number is typically a 32-bit quantity, with at least six significant digits ...", without any further explanations. A novice learner may find themselves at a loss with these mere references to various lengths of bits and the value ranges, without knowing why. The 'Binary Numbers' activity[17], which models the concepts of data representations using bits and bytes, can be used to explain the computing concepts behind these programming aspects.

It can be used to explain the rationale of variable declaration and long and short integers in a programming classroom (e.g. in a C programming lesson). Understanding the fractional numbers is closely coupled with programming, because it explains why one should use integer types instead of floating types. But, if one is asked to write a program to convert binary to decimal, it may not teach any programming concept but merely be a rather boring programming exercise. The conversion process is loosely coupled with programming, because it is not a particular programming skill but rather a programming context that can be used in a programming exercise. The Binary Numbers activity is an example where exactly the same activity is at times closely coupled with programming and not so closely at some other times, depending on the perspective the activity is looked at, and the time it is used in a programming lesson. However, a programming exercise preceded by the Binary Numbers unplugged activity could be a more meaningful way to cement the learners' understanding than conventional, straight-to-code lesson.

The Muddy City activity[18], which introduces an optimisation algorithm in the context of graphs, also can be used to teach programming concepts, although it is not designed for programming; writing a program for it is possible, but may need advanced programming skills and it is therefore probably not suitable for children or novices. It can be used to introduce minimum spanning trees and Kruskal's algorithm to a mature programming audience, as a story that provides an authentic context whilst introducing the concepts. Similarly, The Poor Cartographer (Graph Colouring) activity[19] is used to

[15] CS Unplugged: https://classic.csunplugged.org/

[16] Teaching London Computing: https://teachinglondoncomputing.org/

[17] Binary Numbers: https://classic.csunplugged.org/activities/binary-numbers

[18] Muddy City: https://classic.csunplugged.org/activities/minimal-spanning-trees

[19] The Poor Cartographer: https://classic.csunplugged.org/activities/graph-colouring/#the-poor-cartographer

introduce intractability; that there are some problems that do not have good solutions. One can still write a program for the Poor Cartographer scenario to realise the issues and concerns behind exponential times in a program, but it may not be useful as a programming exercise to draw attention to programming concepts. It can also be used to discuss the programming concepts around graph data structures. However, these underlying programming concepts do not come under the topics of an introductory programming text book, but rather fall under advanced programming topics.

A lot of programming languages these days reduce a programmer's workload with various tools such as in-built functions (e.g. `sort()` or `max()` function) or packages (e.g. `Math` package in Python or Java). This could prevent learners from gaining a deeper understanding of the basic computing concepts underlying in their implementation mechanisms. The Lightest and Heaviest activity[20], which is ideally used to model the efficiency in computing (sorting), is a good exercise that can be used to explain the underlying programming concepts in sorting to a learner. A lot of programming languages have `sort()` functions in them, but the unplugged activity puts the mechanism into perspective that, though using a `sort()` function is just one command in the program code, it is not a one unit of time in execution. The same activity can also be used to introduce 'if statements' and 'repetition'. The Lightest and Heaviest activity is therefore tightly coupled with CS concepts and with sorting in programming, but loosely coupled with introductory programming concepts 'selection' or 'iteration'.

The example activities discussed here so far are not originally designed to model a particular introductory programming concept, and are good examples of Non-programming Unplugged activities. The concepts they model are not directly connected to programming skills, yet are closely related and can be useful in communicating a programming concept. They are complete without the need of any programming exercise, but can also be used as programming exercises (probably to a learner group with strong programming skills). In fact, any unplugged activity that models algorithms (e.g. Sorting, Searching, Minimum Spanning Tree) can naturally lead to a programming exercise, despite being used to model a general computing concept. Yet a novice teacher may not be able to see all the connections a Computing Unplugged activity can have to programming and may feel at a loss in supporting the learners with making such connections or scaffolding. Moreover, making such connections available for a beginner teacher can possibly be overwhelming. Maintaining simplicity is a key consideration in unplugged; an expert or a more knowledgeable teacher can extend a Computing Unplugged activity to communicate programming concepts, others may not.

On the other hand, an unplugged activity like 'Box Variables'[21], which is used to

[20] Lightest and Heaviest: https://classic.csunplugged.org/activities/sorting-algorithms/

[21] Box variables: https://teachinglondoncomputing.files.wordpress.com/2014/02/

explain the concepts of variables, differs from the activities that have been discussed so far, because the concept it models is directly related to programming. It is a good example of a Programming Unplugged activity (see Section 3.3), since it is absolutely natural to use in a way that is tightly coupled to programming and is quite unnatural to use it in any other way. Moreover, although the activity is completely unplugged, it is rather incomplete without looking at how the variables work in actual program code. The underlying concept, which is applicable to any programming language, becomes completely meaningful only when it is realised in a programming exercise. The strict focus that a Programming Unplugged activity has to programming makes it easier for even a teacher new to programming to use them to teach programming and help their students to make meaningful connections.

Unplugged computing activities can have different perspectives and can be used to explain more than one aspect of computing. Despite being originally designed to model a general CS concept, some activities can be used in a programming lesson in a more tightly coupled way than others. Nevertheless, they are complete without the need of any programming exercise. However, when an unplugged activity is designed to model a programming concept, it is rather incomplete and less useful until the learner gets the learning experience completed by trying the concept they learnt in a coding exercise. Even a seemingly simple unplugged computing activity such as Binary Numbers can be quite challenging if it is used as a programming exercise. Unplugged activities designed particularly to model programming concepts (i.e. Programming Unplugged activities), therefore, are better if they are simply modelling as few computing concepts as possible in one activity, so that they can be extended to simple programming exercises that can be used by a larger learner audience.

`activity-boxvariables.pdf`

Table 3.1: Some selected unplugged computing activities and their different perspectives in computing

Activity	Computing Concept(s) Modeled	Link to Programming	Comments
Binary Numbers [22]	• Data representation: Bits and Bytes	• Variable declaration: float, long/short int, etc. (tight) • binary conversions (loose)	• Not designed to model a programming concept; complete without the need of a programming exercise. • Example of an activity that is both tightly and loosely coupled with a programming concept.
Muddy City [23]	• Algorithms • Optimisation • Complexity • Tractability	• Graph data structures (loose) • Minimal spanning trees (loose)	• Not designed to model a programming concept; complete without the need of a programming exercise. • Can be used as a programming exercise; needs advance programming skills. • Example of loosely coupled connections to several programming concepts.
The Poor Cartographer [24]	• Algorithms • Complexity • Tractability	• Graph data structures (loose) • Exponential times in programs (loose)	• Not designed to model a programming concept; complete without the need of a programming exercise. • Can be used as a programming exercise; needs advance programming skills. • Example of an unplugged activity loosely coupled to programming concepts.

Continued on next page

[22] https://classic.csunplugged.org/activities/binary-numbers/

[23] https://classic.csunplugged.org/activities/minimal-spanning-trees/

[24] https://classic.csunplugged.org/activities/graph-colouring/

Table 3.1 – *Continued from previous page*

Activity	Computing Concept(s) Modeled	Link to Programming	Comments
Lightest and Heaviest [25]	• Algorithms • Complexity • Tractability	• Sorting (tight)	• Activity is not designed to model a programming concept and is complete without the need of a programming exercise. • Can be used as a programming exercise; needs advance programming skills. • Example of an activity that is both tightly and loosely coupled with a programming concept.
Box Variables [26]	• Variables • Sequence	• Variables; declaration, initialisation, assignment (tight) • Sequence	• Strongly relates to computing; fully meaningful only when a programming exercise is followed. • Example of an activity tightly coupled to programming concepts.

[25] https://classic.csunplugged.org/activities/sorting-algorithms/

[26] https://teachinglondoncomputing.files.wordpress.com/2014/02/activity-boxvariables.pdf

Chapter IV

Supporting Theories and Literature

This chapter provides an eclectic discussion of a range of educational theories and related academic discussions that provide a theoretical underpinning for this study. The thesis draws on these theoretical underpinnings and research to develop an understanding of the usage of Unplugged computing as a pedagogic approach and/or tool. Most CS education literature focuses on teaching and learning programming, particularly earlier research. Robins points out that this explores one or more of three general topics: programming knowledge; the strategies needed and used to deploy that knowledge successfully; and the mental models that programmers need to construct [191]. Unplugged approaches in computing education, and in introductory programming learning in particular 1) makes use of learners' existing knowledge to either establish a good understanding of new conceptual knowledge that they are exposed to, and/or 2) supports learners to build efficient mental models to design and develop effective computer programs. In this, unplugged computing uses the age old strategy in education: using existing knowledge to introduce new knowledge.

On one hand, educational theories like the Zone of Proximal Development (ZPD), Legitimation Code Theory (LCT) and Dual Process Theory (DPT) in psychology discuss the existence of the duality of existing and new knowledge in learners' cognition, and how those two extremes can be used effectively to complement and enhance the other. On the other hand, important conceptual models in computing educational theory like Notional Machines (NM) and Computational Agents (CA) stipulate the need for a spacial (that is, physical) and tangible representation to the intangible and invisible processes within a computer for better understanding. A closer look at these models gives an insight into why unplugged computing has shown positive results in programming classrooms when they are combined with plugged-in exercises. These discussions unfold perspectives that can provide explanations of the popularity, effectiveness and efficient application of unplugged in computing education, and support the rest of the discussions in this thesis.

Self-efficacy has been used as the prominent measure used in the experimental studies in this research that supports the main thematic discussion in this thesis. Accordingly, a discussion on self-efficacy, and its implications are also included as the last part of this chapter, describing why and how it is important in computing education.

Part I: Supporting Educational and Other Theories

A notable aspect about unplugged approaches to learning is their use of existing knowledge to introduce new knowledge. However, in computing education, and particularly in introductory programming, the chances of a novice possessing any relevant conceptual knowledge is very low, despite their knowledge of using computers and other digital devices possibly being quite high. The reasons for this could vary from people's everyday knowledge in using computers as a tool for everyday information processing purposes, but overlooking what is behind its real computational power, to learners' ignorance or inability to relate their everyday knowledge to computational contexts. Unlike many other subject disciplines like Mathematics or Sciences, in which some basic theoretical and/or conceptual aspects have already well blended to become almost common knowledge through traditional education, computing and programming concepts are yet to become such common knowledge. Unplugged computing activities attempt to extract computing contexts from everyday knowledge and present them back to learners by meaningfully connecting them with conceptual knowledge. Therefore, looking at educational theories that discuss this aspect can be useful in understanding unplugged computing's success as well as helping us to find ways for better application of unplugged methods in teaching computing. The following sections look at three such relevant educational theories that support the theoretical conclusions made in this thesis.

4.1 Zone of Proximal Development

Unplugged computing is about playful engagement in physical activities that can be used to explain computing concepts in a simple manner. A teacher or an expert helps learners to make connections between the learners' existing knowledge and computing concepts in a constructivist approach by drawing out ideas during the activity or by explaining them in simpler terms. The Zone of Proxmial Development (ZPD) describes a knowledge development zone in which a learner operates with assistance, bringing them along a learning process that utilises their level of competence and/or skill to learn challenging knowledge and/or difficult content. The term was introduced by Lev Semyonovich Vygotsky in 1931, which he originally defined as:

> *"the distance between the actual development level as determined by independent problem solving and the [expectedly higher] level of potential development as determined through problem solving under guidance or in collaboration with more capable peers."* [238]

Vygotsky believed that this is most likely to emerge during play [45], where he believed that a child gets involved in discussion with adults or capable peers (whom he explicitly referred to as more knowledgeable others (MKO) [189]) through social interaction. Children develop their problem solving ability and/or skills to do tasks by 'making sense'

during such discussions. Engaging with a MKO, learners actively construct the knowledge into their own way of independent thinking [171]. ZPD is one the foundational elements of constructivism theory.

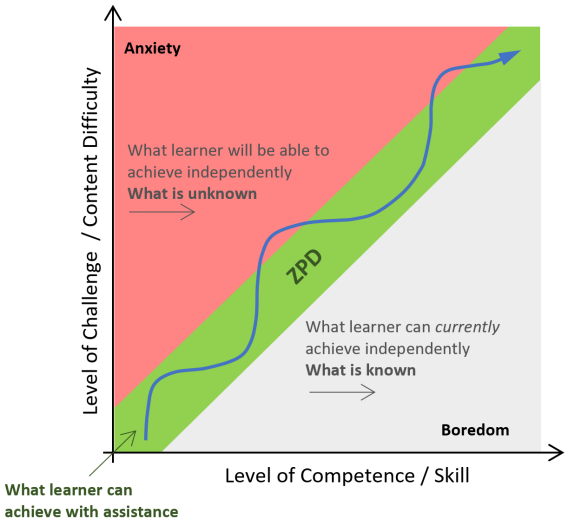

Figure 4.1: The Zone of Proximal Development, based on [163, 188] and [208]

There are many graphical representations available attempting to visualise the ZPD. Figure 4.1 adapts several graphical presentations of ZPD based on [163, 188] and [208], where the blue arrow indicates the student's trajectory through time in the space of learning. Tasks which fall below the ZPD show a low level of challenge and a high level of competency for the learner, and are therefore likely to result in them becoming bored and/or disengaged. Tasks above the ZPD show high level of challenge and low level of competency, meaning that learners are likely to become anxious and/or confused [171]. The authors have attempted to show that the progression through the trajectory is not linear. Limitations of assisting a learner (e.g. resources) determines the "effective ZPD" which is defined by "the difficulty of tasks possible if the student is given the available help" [163].

The ZPD theory has been generalised beyond its original developmental framing to encompass many facets of human learning [85], and is often used in educational contexts to understand the socio-cultural involvement in a learning process. It describes a zone of learning that should bring the person to a knowledge level slightly more complex than what they currently have, highlighting the role and value of social interaction among learners, and between learners and adults (i.e. teachers and/or experts), in the process.

The concept of ZPD has been adapted to many disciplines and studies (e.g. [113, 189, 196, 197]). Vygotsky suggests that adults and more knowledgeable peers guide and mediate learning for the learner [189]. Collaboration and assistance in achieving a specific

goal and establishing understanding is the basis of ZPD [85]. Despite some contrasting criticisms of its socio-cultural perspective, such as the ability of a person also expanding from interaction with the use of computers, technological objects or resources [189], and not all learning being achieved with social collaboration and/or by linking with everyday knowledge [197], the ZPD has been popular in various learning and educational discourse studies.

In the ZPD, the participants involved (e.g. learners and teachers) should objectively engage in a collaborative activity, working towards a specific goal. Operating within the zone, instructions should focus on learners developing knowledge using what is already known to them to understand the unknown, with the appropriate support from a MKO. The presence of another person with more knowledge and skills (i.e. an educator), social interactions with them for the learner to observe and practice their skills, and scaffolding provided by the means of scaffolded activities to support the learner are important components of a learning process in order to assist a learner to move forward in the ZPD [151]. Ideally, the learners should be able to use the 'new' knowledge independently after the scaffolding is removed.

4.1.1 Unplugged Computing in the Proximal Development Zone

Engaging in an unplugged computing activity involves a dual participation in an activity that has design and execution linking to everyday experiences and knowledge, with the interventions of a MKO (e.g. teacher). Social interaction and learners' active engagement is an integral part of unplugged learning, and thereby engaging in an unplugged computing activity necessarily attempts to situate the learners within the ZPD during the learning process.

The learning experience during an unplugged computing activity is very similar to the learners' development described by authors who have discussed the ZPD. Although originally the ZPD was described as the distance between individual performance and assisted performance, different interpretations have emerged subsequently [85, 113, 196]. Hedegaard [113] has interpreted ZPD as the distance between "understood and active knowledge", where understood knowledge is acquired via formal instruction, and active knowledge is gained by informal interactions of the individual with the world. Unplugged computing necessarily involves learning technical concepts through everyday concepts, and accordingly is closely coupled with the ZPD model in more than one aspect.

During an unplugged computing activity, learners, most often novice or beginner, use their existing knowledge to learn or understand a computational concept. They link their abstract knowledge into a technical understanding, with the guidance of a teacher (and help of knowledgeable peers). Vygotsky distinguishes scientific concepts, which originate in formal schooling, from experiential or 'spontaneous' concepts, which arise in one's

day-to-day experiences [239]. A blending between these two concepts is essential for a substantive development to occur [242].

4.2 Dual Processes Theory

Learners acquire knowledge in many ways and educators have been using various approaches, strategies and techniques to support this process. Educators often make use of general knowledge, communication abilities such as written and verbal skills, vocabulary, etc. to establish more formal knowledge such as problem solving, rule learning, reasoning, etc. Behind its playful and kinesthetic nature, constructivist approach and seemingly simple explanations, unplugged computing is about drawing out learners' existing, established knowledge and awareness to introduce new and conceptual knowledge. Due to this duality of a learner's knowledge that is subtly used in unplugged computing, it is worthy to look at it using Dual Process Theory (DPT) from psychology, in order to obtain a broader understanding of its involvement in cognitive aspects such as learner's memory, cognitive load and intelligence.

Knowledge comprehension is a multi-faceted area of study both in education as well as psychology. The Dual Process Theory, a theoretical framework in psychology, puts forward the existence of two kinds of cognitive systems:

System 1 (S1) - The associations arising from long-term memory. S1 is fast and intuitive, associated with cognitive factors such as crystallized intelligence (knowledge that comes from prior learning and past experiences), long-term memory and associative learning, and

System 2 (S2) - The involvement of conscious reasoning with explicit rules in working memory. S2 is slow and reflective, associated with cognitive factors such as fluid intelligence (the ability to to think abstractly, reason quickly and problem solve independent of any previously acquired knowledge), working memory, and rule learning [192].

The Dual Process Theory is descended from a number of independent theories that discuss dualities in different contexts such as intelligence, memory, attention, learning, etc. DPT spans different domains where two qualitatively different cognitive systems are involved. In [192], Robins discusses a variety of characteristics of S1 and S2, as shown in Table 4.1, which summarises the various descriptors used for the two systems in DPT, the distinct cognitive factors that seem to be involved, and the associated brain structures.

The extent to which each system can be characterised as a single unit and the degree these properties differ between systems is an open question. The two systems interact

	System 1	System 2
Descriptors		
	Automatic / autonomous	Controlled / directed
	Implicit / intuitive	Explicit / reflective
	Heuristic / associative	Analytic / rule based
	Fast	Slow
	High capacity	Low capacity
	Unconscious	Conscious
	Evolutionarily old (animal)	Evolutionarily new (human)
Cognitive factors		
Intelligence	Crystallized intelligence	Fluid intelligence
Memory	Long term memory	Working memory
Attention	Automatic / passive	Controlled / active
Learning	Associative learning	Rule learning
Cognitive load	Schemas, automaticity	Intrinsic and germane load
Brain structures		
	Ventral medial prefrontal cortex	Right prefrontal cortex
	Posterior parietal regions	Prefrontal cortex
	Default Mode Network (DMN)	Global Neuronal Workspace (GNW)

Table 4.1: Summary of Dual Process Theories adapted from [192]

with and are interdependent of one another, suggesting a Dual Process Cycle that models the interactions between two systems over a short and long timescale (Figure 4.2).

4.2.1 Dual Processes in Unplugged Computing

Expertise develops over time and it is a consequence of a person's knowledge and understandings becoming concrete, the ability to recognise and reuse this knowledge increasing, and the skill to apply it for further learning. In other words, a novice programmer gradually becomes an expert as their programming knowledge starts to move steadily between their cognitive S1 and S2. Understanding how unplugged computing operates in S1 and S2 may provide deep insights into why it is popular and known to be effective in computer educational contexts, with outcomes such as improved self-efficacy, motivation and handling misconceptions.

Unplugged uses S1 and S2 together, but that balance includes much more S1 than plugged-in approaches (because unplugged tries to ground activities in the familiar). Therefore, unless it is objectively connected to a computing context such as programming, doing an unplugged activity alone is mostly S1, because by doing an unplugged activity like a simple game or a puzzle only relates to a learner's everyday experience or things

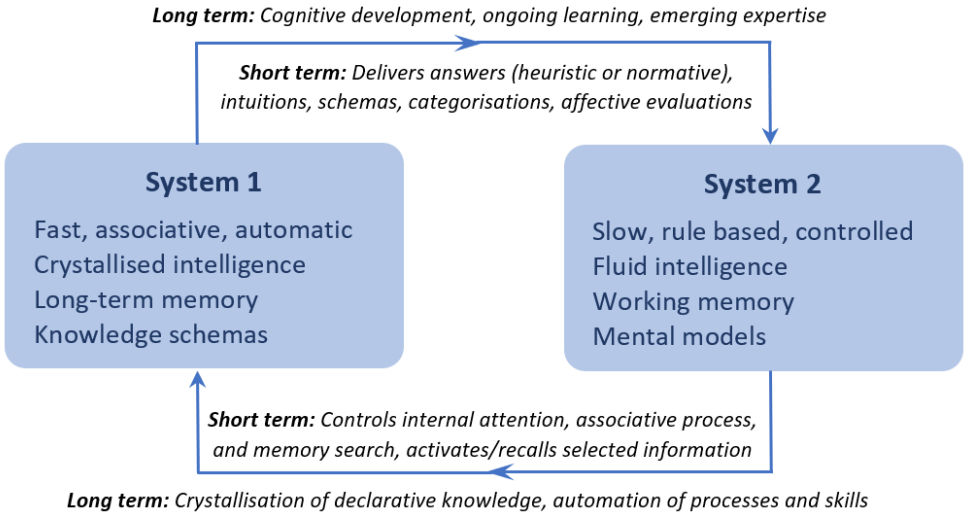

Figure 4.2: Dual Process Cycle adapted from [192]

they already know. However, when they are combined with plugged-in exercises, the learner starts thinking about rules and conscious manipulations in a more controlled and directed manner, and their cognitive system moving from a more S1 system and becomes more balanced in S1 and S2.

As an example, let us consider the Computing unplugged activity, Parity Magic[1], which does not necessarily involve a Notional Machine, yet still supports the learner in building a mental model. Reasoning about the Parity Magic trick is very strongly S2 because a magic trick subverts learners' expectations of their real world knowledge, while doing it with cards (unplugged) brings in more S1 than getting students to write programs to count bits (plugged-in). It deals with crystallized intelligence, but the learning that happens alongside fluid intelligence. Humans very seldom operate purely with one system or the other, and cognition naturally always involve both systems. S2 absolutely has to be fed by inputs, associations, ideas, suggestions, hands on strategies, crystallized intelligence, etc. and cannot operate without them. Some unplugged activities such as Parity Magic or Binary Numbers provide more grounding on S2 for the learners.

With Programming Unplugged activities like Kidbots[2], the more programming-like elements that are introduced, the more they extract elements from S2. Therefore, with Programming Unplugged activities, the process is more transitional and obvious, creating a Dual Process Cycle or a wave as the learners move between tasks that have a varying

[1] Parity Magic: https://www.csunplugged.org/en/topics/error-detection-and-correction/unit-plan/parity-magic/

[2] Kidbots: https://www.csunplugged.org/en/topics/kidbots/unit-plan/rescue-mission

mixture of elements from S1 and S2.

What sort of learning happens in a novice when they learn a programming concept for the first time? Although educators frequently use analogies and context to support learners, conventional programming education would often focus on S2 – fluid intelligence for novel reasoning and problem solving, rule learning, analysis that mostly involve working memory and intrinsic load. It is worth considering the value of using the strengths of S1 in conventional programming, or taking account of where the S1 is helping or hindering the process; for example, misconceptions from S1 may hinder the process by delivering poor heuristics or ideas.

In an unplugged and plugged-in combined approach, the technical and conceptual context of the learning (e.g. learning about the NM, use of syntax and semantics of a programming language) is gradually exposed to the learner as the lesson progresses and therefore is probably S2; it is rule based, in explicit working memory, still fluid and germane. But the more often they use their learning and apply it to real world tasks (i.e. combining with unplugged activities), the more it will get consolidated. When they form a schema, cognitively, it is more like a germane load that is applied. However, when they integrate what they learnt with unplugged experiences, they automate it so that the learning becomes crystallized, and becomes something that they can then access intuitively and automatically, without having to 'think about it' or without having to use working memory.

Unplugged computing, and particularly unplugged programming, attempts to keep a process of crystallization continuing, while taking the fluid elements from S2 and getting good crystallized intelligence built out of it. Thereby, the learning process completes the feedback loop of a Dual Process Cycle, so that the learner is able to automate their knowledge, form schema that characterize expert behavior and therefore reduce the load on working memory and S2.

4.3 Legitimation Code Theory and Semantic Waves

Legitimation Code Theory (LCT) is a theoretical framework that describes the dimension of Semantics and how semantic waves, which map the movement between degrees of "semantic density" (SD) and "semantic gravity" (SG) over time, can play a part in building legitimate knowledge [149, 150]. LCT is a sociological approach to understanding and shaping social practice [148]. SG refers to the context dependence of knowledge (the context of meanings and how much meaning depends on everyday context) and the SD refers to the degree of complexity of knowledge (the complexity of meanings rather than their context) [150, 241]. For example, writing numbers on a scoreboard to record the current score in a game is high SG knowledge (everyday knowledge), whereas knowing how variables work in a program is high SD knowledge (knowledge specific to

programming/computing).

The distinction is compatible with the two types of cognitive process described in the psychological framework of Dual Process Theory [192], where SG can be seen as a feature of System 1 (based on associations arising from long-term memory) and SD is a feature of System 2 (involving conscious reasoning with explicit rules in working memory). The two theoretical frameworks reach essentially the same conclusions, that a person's learning and understanding new knowledge shows a duality that involves existing knowledge providing context to develop new knowledge or concepts.

Maton [150] suggests that semantic waves can model the transitions of knowledge between contextualised and simpler understandings (i.e. abstraction) and more integrated, deeper meanings (i.e. technicality), and model how meanings may be transformed through semantically braiding together different forms of knowledge. SD and SG are independent of one another and are realised on a continuum between weaker and stronger. The variation of strength and weakness of SD and SG (independent to one another), generate 'semantic codes' [116, 148] as follows:

- rhizomatic codes (SG-, SD+), context-independent and complex stances
- prosaic codes (SG+, SD-), context-dependent and simple stances
- rarefied codes (SG-, SD-), context-independent stances with fewer meanings
- worldly codes (SG+, SD+), context-dependent and manifold meanings

Martin et al. [148] suggest that the variation of these semantic codes can be seen as a movement on a topological space called a 'semantic plane', as shown in Figure 4.3.

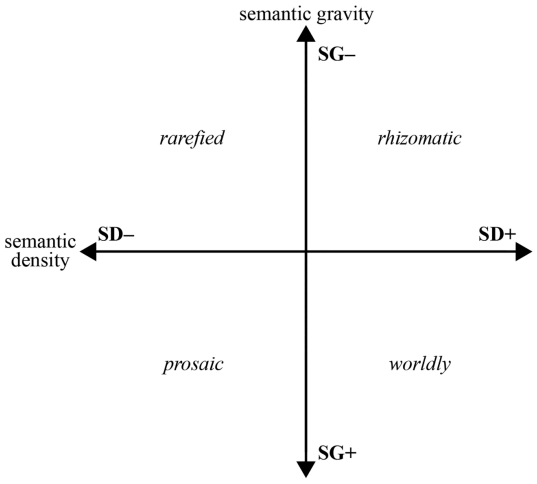

Figure 4.3: The Semantic Plane, from [148]

Figure 4.4 shows illustrations of three different semantic profiles that depict the association of semantic density and semantic gravity between their highest and lowest strengths (i.e. semantic range). According to Maton [149], the semantic profiles A and B in Figure 4.4 are high and low semantic density flattening curves respectively, where the semantic range is low in both cases. In a high flattening curve (A), the learning

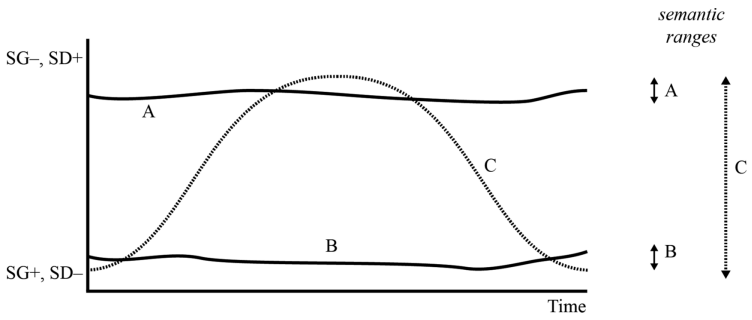

Figure 4.4: Different semantic profiles, from [149]

experience is based highly on abstract concepts and technicality with deep meanings, and making minimal effort to relate them into a simpler understanding explained in everyday language. Conversely, the low flattening (i.e. curve B) limits the explanations to simpler and concrete knowledge using everyday language, with minimal attempts to link them back to abstract, technical knowledge. Curve C, which indicates traversing between the high and low SG/SD continuum, has a higher semantic range than both A and B. Moving up and down the SG/SD continuum generating semantic waves by up-shifting and down-shifting through academic content and language is considered important for cumulative knowledge building [149].

The down-shifting (also referred to as 'unpacking' in [241]), which most likely involves the teacher making an explicit link between learning outcomes and the activity, means moving from complex meanings to simpler, more grounded meanings. The up-shifting (also referred to as 'repacking' in [241]) involves linking the abstract, grounded knowledge and understanding to technical or conceptual contexts. Repacking may involve both teacher and learner, with the teacher giving an overview of the main points (i.e. summarising, answering questions, briefly explaining, linking metaphors back to technical concepts, etc.) and the learner expressing their own learning experience (e.g. asking simple questions, summarising, etc.) [59]. Curzon et al. [59] suggest that explanations that can be easily understood by a novice learner are stronger in semantic gravity and weaker in semantic density: they involve context-dependant ideas expressed with the aid of non-technical knowledge using everyday language.

Many thinkers have distinguished everyday or commonsense knowledge from educational or "uncommonsense knowledge" [150], and pedagogy often makes use of these

distinctions in communicating ideas to learners. unplugged computing essentially makes good use of everyday knowledge (commonsense) in explaining technical or conceptual knowledge (uncommonsense) and uses everyday language in classroom discourse. Consequently, the semantic profile of an unplugged computing activity is largely determined by the way it is executed. A detailed discussion about the semantic profiles of unplugged activities and an analysis of their applicability and usage using an experimental example is discussed in Chapter 10.

4.4 Dual Process Theory's Implications on Legitimation Code Theory in an Unplugged Computing Learning Context

The distinctions described between SG and Semantic Density can be seen as being compatible with the System 1 (S1) and System 2 (S2) duality identified in the Dual Process Theory. It can be seen that the concepts and the elaboration of concepts (SD) are consciously listed at the System 2 level, where as in SG the relationship to existing experience is very much System 1. With respect to the learner's cognition, SD seem to cover a larger cognitive range than SG because SD involves fluid intelligence, operating with working memory, learning rules, etc., where as SG involves crystallised intelligence, operating with long term memory and associative learning.

Doing an unplugged computing activity, and thereby the ripples that are being created in the semantic profile of a lesson, creates cycles between the dual processes. It can also be seen as a trade-off between the two systems. At the top and the bottom of the Semantic Wave, it is trading off a lot of one system to the other. With a constructivist approach, the trade off starts at the bottom with high SG, and with an instructivist approach, it is vice versa.

The objective of a learning process is to bring the subject matter to a learner's System 1 as effectively as possible (e.g crystallising, transfer to long term memory, develop the ability to associate with other learning, rather automatically), so that it becomes *natural* to them. However, as mentioned earlier, most conceptual knowledge of high SD leans more to System 2. Moreover, physiologically, one cannot build System 2 cognition all by itself, but it must be anchored with as much System 1 as possible. Unplugged activities, operating at high SG level (and thereby System 1), essentially lets the learners connect the SD to their SG and thereby crystallising them. The fact that the teacher is using an Unplugged activity, a metaphor, an analogy or any similar reference to learners' concrete knowledge means they are introducing a semantic range hoping to pull conceptual ideas into the ZPD and thereby creating returning cycles between a learner's System 1 and System 2 cognition.

In some unplugged activities like those focused on problem solving, the learner's System 2 is provoked even staying at high semantic gravity level. In an alternating

Unplugged programming lesson, during the Unplugged activity, the knowledge that the learner uses to *do* the activity belongs to System 1, practical knowledge and crystallised intelligence. By alternating with plugging-it in, System 2 is brought into the picture, along with conceptual knowledge and fluid intelligence.

The ultimate expectation of a programming lesson is to enable the learners to crystallise this fluid intelligence as much as possible and push the conceptual knowledge they gain from their working memory to their long term memory, so that the learners can use this conceptual knowledge about programming automatically when they program; the conceptual knowledge should just flow into their CT. Alternating Unplugged with plugged-in exercises essentially enable these movements between System 1 and System 2, completing several Dual Process Cycles as discussed in Section 4.2, Figure 4.2.

Part II: Conceptualisation and Representational Models in Computer Programming

Education regularly uses simplified models (e.g. the Rutherford-Bohr model of the atom), analogies and representations to help the learner to develop good mental models, and this is inevitably necessary when teaching computing as well. The use of some form of an abstract or conceptual machine (e.g. Notional Machine, Turing Machine, Finite State Machine) or an intangible agent (e.g. Computational Agent, Intelligent Agent, Autonomous Agent) in scientific or pedagogic explanations is common in computing. To rationalise this and to understand the nature of 'machines' and 'agents' used in computing, this sections looks into the relevant philosophical backgrounds and definitions.

The Merriam-Webster dictionary defines a machine as "an assemblage of parts that transmit forces, motion, and energy one to another in a predetermined manner" or "one that resembles such", and an agent as "one that acts or exerts power" or "a means or instrument by which a guiding intelligence achieves a result" [68]. These definitions and their various adaptations in computing indicate that computer scientists have attempted to situate the nature of their main tool, the computer, as having both a physical nature through the term 'machine' as well as an organic nature by calling them 'agents', for the purpose of conceptualisation or supporting understanding.

4.4 Machines

Using a mechanical analogy as a model for understanding how ideas function is common, and has a long tradition. According to Aristotelian era writings, the reasoning behind machines, known as mechanics, is the art of solving the perplexity of "doing something contrary to nature" [250]. Put simply, mechanics is making something happen that would not occur naturally. In doing so, historically, the *how* was understood through math-

ematics, and the *what* through physics [250]. In modern day computational problem solving however, the solutions may have a formal explanation (i.e. an algorithm) to describe the *how*, as well as a physical (tangible) explanation to describe the *what*, although the physical explanation can be intangible if it is not tied to a particular machine.

Aristotle's writings about "automata" and his later discussions about "automation" describe an "engine of repetition", which is a machine that, with its internal complexity, "transforms one input into a motion of a different kind" [24], which could be seen as an output in a computing context. This contrasts with something like projectile motion, where the trajectory is determined by natural forces once the projectile is launched. Ancient descriptions did not have the luxury of examples from advanced technology, and thus relied on trying to provide mechanical analogies to living organisms (animals) and creations by divine characters, or hypothesis. As an example of a divine creation, according to the poet Hesiod (around 700 B.C.), Hephaestus, the Greek god of invention and blacksmithing was said to have built a giant bronze man called Talos, which corresponds to modern day robots. Berryman [24] points out that the term "mechanistic" can be taken to refer the method of investigating the natural world using the terms and principles of mechanics and "mechanistic conception" or "mechanical explanation," which is their application to understanding the natural world. In the absence of a natural explanation, a mechanistic conception to shaping ideas about the natural world was a plausible alternative.

These discussions show how, even in the pre-computer era, thinkers and researchers attempted to relate the conceptualization of ideas into some form of mechanical explanation for understanding. Accordingly, conceptual models in computing could sometimes be presented as machines, and in the next section we consider a commonly used machine that characterises computing.

4.4.1 Notional Machines

The *Notional Machine* is an influential concept in CS education literature, and programming in particular, which describes "an idealized abstraction of computer hardware and other aspects of the runtime environment of programs" [215]. It is an ideal that is objectively defined, and independent of any particular learner's understanding [192]. Du Boulay et al. [75] articulated this concept of an idealised model of a computer as:

> *"an idealized, conceptual computer whose properties are implied by the constructs in the programming language employed".*

The rationale for the NM is that the mastery of every detail of how a computer works is not essential in order to understand the dynamics of the programs at run-time. NMs essentially help learners to understand the invisible transactions and side effects of usually invisible processes that the instructions of their program cause inside the computer.

Many definitions of NMs, including its earliest definition, are usually tied to a programming language or a physical computing device. Robins et al. [191] further describing du Boulay et al.'s view, present the NM as a "model of a computer as it relates to executing programs". According to them, a NM must be able to explain all observable behaviour of the real machine (i.e. registers, fetch cycles, memory, etc.), and reasoning about it must allow accurate predictions to be made about behaviour of the real machine [23]. Du Boulay et al. highlight that experienced programmers may find special languages designed for pedagogical purposes with simple NM as "distasteful", yet languages intended for teaching take into account different human factors compared to languages being used by professional programmers.

The purpose of NMs is mostly pedagogic, to draw the learner's attention to hidden aspects of programming or computing [75, 89, 191, 215]. Some authors have tried to bring the NM towards a more conceptual view, where they have attempted to relate it to a person's mental modelling about computing, and not limiting it to a programming language or a physical device. Fincher et al. [89], discussing the relationship between conceptual models, NMs and mental models, suggest that teachers create NMs as pedagogical devices to help learners understand a (particular) conceptual model (i.e. to form their own mental model). They describe a NM that has a generic purpose (i.e. pedagogic) and generic function (i.e. uncover hidden aspects of programming or computing), and a particular focus (i.e. specific aspect of a program and their behavior) and particular representation that highlights the focus. A learner's mental model (i.e. about computing or programming) is often incomplete, erroneous, lacks firm boundaries and is changing, and a NM provides a bridge that abstracts a detailed and precise conceptual model, often using analogies that provide scaffolding to help refine the learner's mental model. Thus this alternate definition describes a NM as a special kind of conceptual model that may not be as declarative as a conceptual model, yet is representing something that can be interacted with (often mentally) [89].

When learners begin to program, some form of a NM is present and it is implied by both the programming language and the paradigm used in learning, and several NMs at different levels of abstraction may describe the execution of a single program [215]. Krishnamurthi and Fisher's definition of a NM [130] suggests that it is a "human-friendly abstraction" explaining program behavior of a "given language or family of closely related languages". In [110], the authors recognise the need for a distinction between a general description of behavior, independent of a specific program, and the explication of the behavior of a specific program, suggesting that the learners may need to shift between the two. In that case, a NM can be seen as an evolving abstraction of an execution environment, for comprehension and learning purposes. However, it should also be sufficiently simple to learn, yet sufficiently comprehensive to solve the problem of interest [29, 75, 89], and should convey descriptions of semantic behaviour [67]. A NM in such a context

is loosely coupled with a programming language, does not necessarily relate to a physical device (i.e. computer) for explanations, and may waiver between a physical machine and conceptual machine in describing a computing concept. A NM may not effectively give an observational (external) perspective as it is not contextually 'available' like a Computational Agent as described by the CT definition (given in Section 4.5.2), especially when it is explained through programming constructs. The learner is expected to be 'aware' (at least partially) of the presence of a NM through those constructs, as opposed to a Computational Agent that is contextually 'available' for them either tangibly or intangibly. A NM essentially explains the operational (internal) perspective, in that it helps learners to comprehend a concept better by (mentally) interacting with it.

Notional Machines can exist even if the real machine does not. For example, it is a popular misconception, although believable that, Charles Babbage designed the analytical engine, and Ada Lovelace developed the program for it. Even with the absence of a proper programming language for the analytical engine, Lovelace developed her own version of what could possibly have been a NM for it that enabled her to predict what a machine that she never got to see would produce, using trace tables [117]. Similarly, is it possible that anyone who develops a new programming language is likely to have written programs in that language before they write the compiler? And in the days of punch cards and limited computing access, students frequently wrote programs days in advance of them being run, although if the Lovelace story was true, it would have been decades in advance! Likewise, a fault tolerant quantum computer may yet to be constructed, yet algorithms have been designed for it by people who have a NM for this machine that may not ever exist! Nevertheless, what could the NM for a quantum computer look like, given that it involves operations that cannot be described with non-quantum physics? Is it possible to have NM for quantum computing without an in-depth education in quantum physics? Can a useful approximation that is as formal as a NM exist that does not require understanding quantum physics?

A possible way to describe a NM for advanced computer systems like quantum computing or AI could be to define certain fundamental operations expected of such a system without explaining those operations with the same level of detail otherwise. For example, in a quantum computer 'find every possible divisor of a number' can be defined as a single, unitary operation, never having to describe it as a sequence of operations involving quantum physics. Similarly, in a NM for AI with such abstract basic operations, accepting that some of them will be probabilistic is also reasonable. Thus, defining NMs for advanced computer systems become possible by allowing fundamental operations to be at a far more abstract level than that of conventional computing. A teacher can decide the level of abstraction of a NM according to the pedagogic needs they intend to fulfill.

Figure 4.5 positions these different definitions/explanations in a spectrum with regard to their emphasis on relating the NM to the physical device/programming language.

In [78], Duran et al. provide a range of uses and implied definitions of NMs, which specifically supports this claim. Early definitions (i.e. [75, 191]) are strongly coupled to the physical end of the spectrum. More recent definitions such as Fincher et al. [89] lean towards conceptualisation of computation where the NM is seen as a conceptual device that connects mental and conceptual models about computing, and loosely relates to a physical device/programming language. Sorva [215] proposes different levels of abstraction, potentially spanning the spectrum. The dotted lines are indicators of the positioning of different views in this space, but are not firm lines in the continuum.

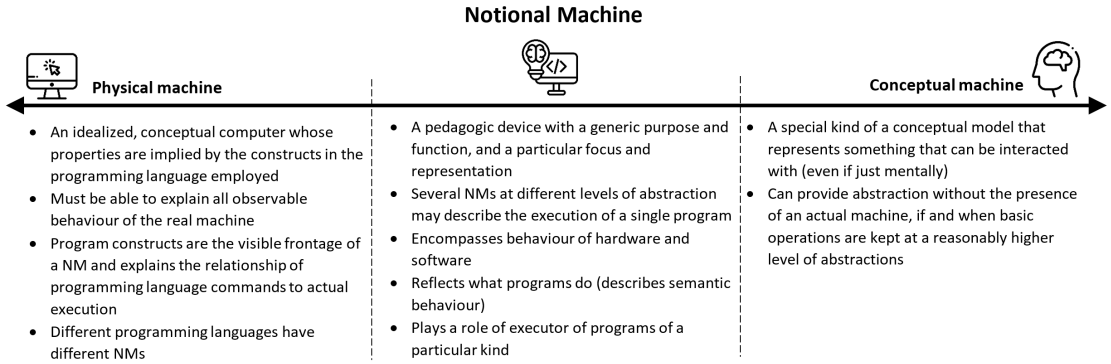

Figure 4.5: Variations of NM discussion in its connection to a physical device and to the conceptual landscape

Many discussions of NMs agree that there should exist a general NM that is a full computer, but no one has ever really defined it [110]. Work has focused on a NM for a specific programming language or context, because a general NM is too nebulous to the current understanding within the community [78]. Accordingly, for the benefit of the discussions regarding NM in this thesis, particularly later in Chapter 7, our general understanding of a NM would be that it has to correctly predict the outcome of a given program (and does not need to accurately represent what is happening with the physical computer). A NM should allow prediction of program outcome for any situation that will be seen within a given context. The term 'NM' is casually used to reflect this understanding. The following distinct definitions for NMs are also used at different points, when necessary:

NM$_{physical}$: NM that explains all observable behaviour of the real machine, and supports accurately predicting the behaviour of the real machine, as explained in [75] or [191] (far left of the continuum in Figure 4.5).

NM$_{conceptual}$: NM that attempts to relate to a person's mental modelling about computing, and not limiting itself to a programming language or a physical device, as explained in [67] or [89] (far right of the continuum in Figure 4.5).

NM$_{generic}$: NM that is implied by both the programming language and the paradigm used in learning, as explained in [130] and [215], or implied by [166] (mid region of the continuum in Figure 4.5).

All three of the above specific definitions also fit into the previously mentioned said general understanding, but are used distinctively at the points where we have to be specific.

Despite its many useful pedagogic implications and applications, the true mechanisms of a NM may not be obvious for a learner, or indeed to teachers who are new to computing. As explained by Fincher et al. [89], NMs are created by teachers as a pedagogic device that help learners understand a certain concept(s) that is intended to be communicated through a particular learning experience, mostly likely programming. NMs are implicitly defined by many visualizations of program execution [216], computationally or otherwise. Educators make use of a vast variety of teaching tools to convey the NM and to establish accurate mental models. These range from analogies and metaphors in the language they communicate, including physical artifacts, graphics and diagrams, to sophisticated software visualization tools.

Novice programmers develop different kinds of intuitive mental representations (mental models), often initially based on superficial language features and analogies [191, 215]. These mental models may not be as accurate or as efficient or effective as those held by an experienced practitioner, but the NM offered by the teacher helps these mental models to mature towards more complex, generally accepted technical conceptual understanding [89]. Learners build their mental models through the teaching and learning methods and materials used in programming courses. In many a case of learning programming, this is facilitated on a physical computer (e.g. a visualization tool, simulator, IDE or debugger), or in an unplugged style learning environment that purposely avoids the use of computers, where the instructions are executed by a fellow learner, or something in-between where a deterministic device such as a Beebot [3] executes programs based on only four instructions. Robins et al. [191] suggest that educators may use something similar to a NM but without explicitly calling it that, or not knowing that it may be a form of one. Du Boulay et al. [75] conclude that matching the visibility and simplicity of components of NMs to novice learners [mental model] at different stages of learning leads to improved educational outcomes.

4.4.2 The Turing Machine

A Turing Machine is a simple hypothetical machine that has been used to characterise what computation is, due to the fact that it can implement any computational algo-

[3] Bee-Bot Programmable Floor Robot: https://www.tts-group.co.uk/bee-bot-programmable-floor-robot/1015268.html#

rithm [48]. It is widely used in computing as a tool to reason about the limits of computation, rather than to perform practical computations. A Turing Machine supports a finite set of simple instructions and states, yet can compute anything that is *computable*. The Turing Machine highlights the transition of the term "machine" from a physical device to a concept that is only used as a thought experiment. The Turing Machine is usually described as a physical machine with an infinite tape and a read/write [erase] head, and physical approximations of Turing Machines exist only as curiosities, as they are normally the subject of thought experiments rather than implementations.

A Turing Machine is not practical for everyday computation, but it is a sufficiently strong concept that if a programming language can simulate a Turing Machine, this indicates that the programming language is computationally universal, and any language capable of doing this is *Turing-complete*. The structured program theorem relates Turing-completeness to conventional programming languages, establishing that only three control structures (sequence, selection and iteration) are needed for a language to be Turing-complete, and so establishes that most conventional languages are Turing-complete [26]. Many curricula explicitly cover the concepts in the structured program theorem, since in principle this means that students have been exposed to the full range of conventional computation. For example, the NZ CT curriculum explicitly mentions "inputs, outputs, sequence, selection, and iteration" [230], and the Australian curriculum mentions "sequences of steps, branching, and iteration (repetition)" [13]. However, not all educational languages are Turing-complete; some programming languages, tools and unplugged activities used by teachers to support learners developing effective mental models of a simpler NM are not (e.g. Scratch Junior, Kidbots, Lightbot, and Beebots), as they focus primarily on sequence, and may omit selection and/or repetition.

Given its nature as a representational "model of the computer as it relates to executing programs" [191] or acting like "an abstract computer responsible for executing programs of a particular kind" [215], a NM may often be used in teaching programming. NMs can be very useful in relating programming language commands to actual execution. Even if the programming language being taught is Turing-complete, a NM might eliminate some elements (e.g. loops), for abstraction and simple representation as required by a teacher or curriculum. For example, the first few chapters of most textbooks focus on a subset of a programming language, and if that is all that students are studying at the time, they are working with a NM that is not Turing-complete.

The goal in teaching programming is generally to get students to the point where they have mastered a model of computing that is Turing-complete, regardless of the programming language used in the learning process is Turing-complete or not. In either case, the underlying NM can be very simple; for a teaching device such as a BeeBot, the NM involves just a few instructions and is only a stepping stone to the full power of programming, but even a language like Scratch, which might be viewed as childish or limiting,

is Turing-complete and therefore understanding its NM implies an understanding of the core elements of computation.

The capability of a NM that is being used in a particular situation must include a subset of the capability of the device that is being programmed by a student (it may not be a proper subset, but it cannot be a superset). Thus, for conventional computers, the NM is quite possibly Turing-complete, but if not, it will be a subset of what a Turing Machine is capable of, in which case it cannot compute everything that is computable. The implication here is that students do not have access to the full power of computing; this is reasonable as they progress in their understanding of programming, but could be limiting if it becomes the end point of their learning.

Given that NMs are effectively decided by a teacher (possibly under constraints imposed by a curriculum or teaching resources), and vary in complexity, some will be Turing-complete and some may be a subset. Even if the language being taught is Turing-complete, the teacher might choose to focus on a subset that is not. This is particularly noticeable if we consider a textbook or online course; if students complete just a few chapters or modules, they may not learn about loops (for example) in the space of that course, and if this is a deliberate pedagogical choice, then the NM has been deliberately kept as a small subset of the full programming language to avoid overwhelming students.

4.5 Agents

Another common representational element used in computing and in CS educational literature are *agents*, used in attempts of providing a sense of animacy to computers in explaining and/or defining some of its behaviour. The discussion in this section looks into this aspect of the term 'agents' to understand its nature of application in the computing context, as an additional support for the discussions in Chapter 7.

When studying 'agents', the literature often turns to Aristotle's distinction between animacy and agency. Aristotle introduces two kinds of instruments: 1) animate instruments — instruments meant for action (e.g. slaves[4] and servants) and 2) inanimate instruments — instruments meant for production (e.g. things like a hammer or weaving shuttle) [9]. While inanimate instruments are fundamentally passive, animate instruments are initiators of their own actions (prescribed by "masters"). The movements of animate entities are best explained by the appeal to the goals that they are striving towards, and their intention. Inanimate entities appeal to mechanical causes for their movement, where repetitive work can be done without tiring or complaining.

The animate instruments (such as Aristotle's "slaves") must be present before inanimate instruments (things) can be used, because they are needed to operate the inanimate

[4] Here the term "slave" is used as it was used by Aristotle; in contemporary culture it would be more appropriate to think of these as agents, contractors or employees.

instruments [9]. A slave is concerned with the action they are told to do rather than deciding on the method of production: they do not take initiative in making things, but simply carry out actions that they are told to do. Aristotle goes on to point out that both slaves and machines alike carry out actions they are told to do, rather than initiating the action. There is no claim by Aristotle that animate instruments be completely deterministic. Moreover, neither the Aristotelian masters nor the computer's human users are understood to be instruments.

Based on the above, NMs and CAs (see Section 7.2) relate more to animate instruments than inanimate instruments, as they both focus on the execution of a program, regardless of their form being conceptual or actual, or their involvement being for tracing (predicting) or actually executing the instructions of a program. In the research literature around agents in computing, the term "agent" is often meant as a concept that the programmer uses as a basis for analyzing problems (much like a NM). Often times agents in such literature are autonomous to a greater degree in carrying out operations on behalf of a user or another program, and therefore referred to as "autonomous agents" (AA), although in many cases they are intelligent agents. Franklin and Graesser [95] defines AAs as "a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future". As such, an AA could be anything from a human to a thermostat, as long as it satisfies its objective regardless of how primitively (quite similar to Wing's perspective of a CA, yet different only in its capacity to act autonomously). This definition's relatedness to a possible CA is more applicable to the CAs discussed in Artificial Intelligence (AI) literature rather than the CT literature. Considering the degree of relatedness that can be seen in both of these applications, a further discussions about CAs in both AI and CT contexts are in Sections 4.5.1 and 4.5.2.

4.5.1 Computational Agents in Artificial Intelligence

The term "Computational Agent" is often used in computing to give a sense of agency to its main tool, the computer. In the Artificial Intelligence (AI) literature, a CA is commonly described as "an agent whose decisions about its actions can be explained in terms of computation . . . the decision [of a computational agent] can be broken down into primitive operations that can be implemented in a physical device" [182]. However, the agents in AI (as currently practised) need to be trained with an objective connection to the environment they operate in, having sensing abilities, actions and goals, and algorithms designed to support their selective actions. Coleman [44] uses the term CA to refer to "computers which are less rather than more dependent on other animate instruments (e.g. the human user) for their functioning", taking its autonomy into consideration yet leaning it towards an inanimate instrument. They suggest that CAs are independent

of other instruments for their functioning in spite of their dependency on the human users/programmers at some (early) stage.

According to Franklin and Graesser [95], a CA in AI is an Autonomous Agent (AA) that resides at a similar level as a biological or robotic agent in an agent taxonomy tree. But they state that an ordinary computer program is not an AA because, although it senses the world via an input and acts via an output, its output does not necessarily influence the process that it uses in the future. Their definition demands CAs in AI to have a certain degree of evolution (adapt to its environment) over time, but a simple computer program would not do so. Thus, all software agents are programs, but not all programs are agents.

CAs in AI are often considered as "black boxes". The reasons for this are 1) the complexity of the countless elements in an AI system and their interconnections makes it difficult, if not impossible, to determine precisely how decisions or predictions are being made, or 2) the fact that they process and optimize numerous variables at once by finding geometric patterns in a higher-dimensional, mathematically-defined space [16]. This prevents humans from visualising how they actually work, making them poor NMs.

4.5.2 Computational Agents in Computational Thinking

The "information processing" or "computational" agent is an interesting element in the CT definition, which the whole concept revolves around. By Wing's definition of CT as mentioned in Section 2.3, the thought process involves problem solving using logical thinking and formulating a solution that can be executed by a "computational agent". Although this early CT definition [248] does not specify the nature of the CA, it has later been suggested that a CA can be either human or machine, as long as it carries out only the instructions prescribed in the solution exactly as it is (precisely) and without judgment (blindly) [56]. As such, the CA provides some sort of limit on the algorithm the learner specifies as their CT solution: it must be executable by a computer-like mechanism. The sequence of instructions the system intends to execute is crucial for the CA to operate properly, thus the CA in CT is a "glass box" in which the underlying mechanism is completely understood.

Historically, the term "computer" meant "the one who computes" and was used to identify the people, often women, who did long, tedious mathematical calculations. Alan Turing described the human computers as "supposed to be following fixed rules; he has no authority to deviate from them in any detail" [235]. When electronic computers became available, those professional human computers were drafted as the first computer programmers, in a move that changed the professionals who had thus far been "thinking like computers" to actual "computer programmers" who literally *instructed* those physical computers how to do what they (human computers) had been doing [210]. The "comput-

ers" (people) developed their skill as computational thinkers to figure out how to work at a higher level in order to program the electronic computers that turned up.

Wing [248] is clear that the CT definition is about the computational process, regardless of whether the execution is done by a human or by a machine. However, this breadth of what an information processing agent might be has been questioned in later discussions, particularly regarding its relationship to computer programming, and highlighting a machine's capacity to do computational tasks better than human. Denning and Tedre [65] argue that allowing a human as a information processing agent undermines the distinction between doing small and large versions of a task. They point out that, while the view that the CA can be human applies to small tasks that can be completed in a short time it would be misleading, as a machine can complete large tasks that are completely beyond human capabilities. This is a point worthy of consideration.

According to Wing's CT definition in [249], the CA's role as the information processing agent in CT is tighter than an AA's. The CA's restricted nature in acting on instructions (i.e. precisely and blindly) highlights the need for precise, step-by-step instructions in arriving at a solution to a problem. The CA is not involved in any decision making. In other words, the problem solving must ensure that the solution is presented with a precise set of step-by-step instructions, so that a "[computational] agent" can carry them out to obtain the "result" successfully (in a mechanical fashion). The definition neither requires the CA to be necessarily a computer/device nor excludes it from being a human. This positions the CA in CT as more of an Aristotelian animate instrument than an inanimate instrument.

The nature of a CA in CT does not necessarily demand the presence of a computer, even in the context of a tool (i.e. inanimate instrument), to carry out instructions. Instead, throughout the problem solving process, the CA is a concept in the problem solver's mind that executes the instructions (therefore similar to a NM). This contrasts CT's CA vastly from an AI's CA, where the latter necessarily runs on a computer, thereby is tightly coupled to the device. This interesting nature of the CA in CT encourages a few key ideas when in applied problem solving:

1. algorithmic thinking and logical thinking in problem solving,
2. modelling or conceptualising (like abstraction, generalisation, decomposition) in forming the solution, and
3. attention to detail and precision in the (language used for) instructions in presenting the solution (to the CA).

The CT's CA should be simple enough to help "show" a teacher or learner what is going on in the problem solving and/or solution generation space. This enables the problem solver (i.e. human) to visualise how the "instructions to achieve the solution" (i.e. program)

work, much like the original definition of the NM authors [75]. In a learning environment, therefore, understanding the CA helps to build good NMs.

Accordingly, much like the CT or NM definitions (Figures 7.1 and 4.5 respectively), the CA in CT's position also can be seen as varying in a spectrum in its relatedness to a physical device and abstraction of concepts, with respect to its internal structure and properties. The closer the CA is to a physical device, the more explicit its structure and properties are; the more abstract the CA is, the less defined its structure and properties are, other than that it follows instructions blindly. The exact position of a given CA in the spectrum is determined by the level of knowledge of a learner/programmer or the pedagogic choices of a teacher/curriculum. The nature of the CA (i.e. computer or human) becomes less substantial in such contexts. Figure 4.6 shows this disposition of a CA in such a spectrum.

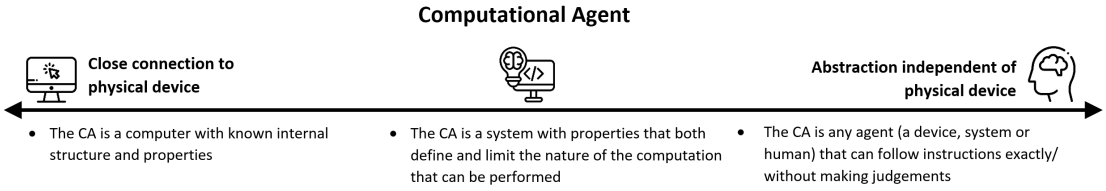

Figure 4.6: Variation of CA's dependency to a physical device

Upon generating the solution (i.e. program), in addition to the programmer predicting the outcome for themselves, the CA can also play the role of the 'executor' of the formulated instructions rather mechanically, thus naturally positioning itself as a good aid in debugging/tracing. The above three ideas are also well situated in computer programming. However, they are not limited to programming and certainly not limited to computing. In that way, "thinking like a computer scientist" could relate to other disciplines. For example, in science, conceptual development is considered as a combination of developing both epistemic and representational practices such as modeling [66, 100, 201]. Dickes et al. [66] provide evidence that when children (learners) are engaging in iterative modeling activities, the progressive refinement of their representations of some aspect of the world can contribute to a deeper understanding of a domain. Agent-based modeling is a type of modeling found to be particularly effective for science learning, where an "agent" is a computational object that is programmed with simple rules [66, 155]. Discussing the context of computer-based 2D graphical visualisation environments for learning, Shari et al. [155] suggest that as an agent interacts with its environment and/or other emergent agents, observable effects appear over time. This enables learners to gain a better understanding of complex concepts they are learning. Several authors have discussed the relationship between conceptual modeling, CT and ggent-based modeling in disciplines outside computing [66, 109, 155, 201]. Their explanations draw particular attention to

the CA in CT (the likely "agent" involved) as a simple yet highly suitable aid in developing successful conceptual models in learners that can facilitate deeper understanding of those disciplines.

4.5.3 Subtle distinctions between AI and CT agents

Interestingly, the AI and CT perspectives of the nature of a CA are as similar as they are different. Both perspectives agree that a CA acts upon instructions (i.e. their actions can be explained in terms of computation). Both CAs can be seen as machines. However, for a CA in CT a full understanding of the underlying mechanism is crucial, thus a "glass box"; in AI it is not, thus a "black box". The "glass box" nature of the CA in CT, as opposed to the "black box" nature of the CA in AI, makes it a fair candidate to become a useful NM. The relatable nature of a CA in CT, which made apparent through its "glass box" nature, provides a window for the learner to look at a NM that they never get to see otherwise. Teachers can use CAs as tools to connect a NM to a learner's mental model as well as diagnose misconceptions about the NM in them.

The CA in AI appears to have the ability to adapt in the sense that it makes decisions that cannot easily be predicted, because they are based on training. In contrast, the CA in CT is not so, because the behaviour in the 'glass box' is completely defined and predictable. Both run on Turing-complete devices – in principle one could understand what is happening inside the AI black box, but the complexity is beyond what is reasonable to understand. Both CAs lean towards an Aristotelian animate instrument: in AI, in its ability to adapt despite being implemented in a physical device, and in CT, in its nature of following instructions. Nevertheless, because of the absence of sensitivity to its environment and the inability to train itself accordingly, a CA in CT lacks adaptation abilities that a CA in AI has.

CAs in AI are defined to operate within a computer system, thereby implying they are Turing-complete, despite their 'black box' nature with the ability to train themselves according to their inputs, which include the effects of their outputs. With a human agent however, it may be hard to model a Turing-complete system accurately. It would be easy for a human to model a simple Turing-complete language (e.g. a human computer who follows an exact set of actions repeatedly), but one can argue that it is natural for human to interfere with their own action at any given time. The CA in CT is not required to be Turing-complete, but it should fulfill only a subset of Turing-completeness requirements to be eligible to become an "information processing agent". The only requirement to become a CA in CT would be the ability to follow instructions blindly and precisely, and therefore if appropriate rules are enforced, a human can act as a CA. However, when the CA in CT is presented as a human there are a number of reasons that it might not be suitable as a fully functional CA, particularly if the human cannot restrict themselves to

following rules literally.

These ideas are developed further in Chapter 7, where they are used to show the connection between ideas of Computational Thinking, particular to programming, and Notional Machines.

Part III: Theories and Literature Related to Implementing Constructivist Teaching Pedagogy

Prior to the introduction of Computational Thinking (CT), the role of computing in the grade school[5] curricula was largely about Information and Communication Technologies (ICT) and digital literacy. With the introduction of CT in school curricula, concepts of computer programming were also introduced to school level, and teachers have a key role in implementing a CT curriculum of which they still have limited knowledge and/or experience [120, 232]. Conventional programming education was largely an instructivist approach, but the tide has turned recently towards incorporating constructivism as well, mainly through the introduction of CT and pedagogic approaches like unplugged computing at primary school level.

Teaching a topic to learners has two forms: formal teaching, known as "pedagogy", where the learning is directed by the teacher (teaching to dependent personalities), and informal teaching, known as "andragogy" where the learning focuses on the learner (facilitating learners to be self-directed by methods such as group work, discussions). However, formal teaching (i.e. pedagogy) does not always focus on an individual learner, but often focus on teaching to all learners at the same time. Andragogical approaches, on the other hand, emphasise involving the learners' experiences and knowledge in the learning process whenever possible, and building upon what learners already know and what interests them.

For computing, a subject that was introduced to curricula only recently, the importance of teacher learning and development is particularly acute. Rarely in the past, and at present, were teachers recruited with the expectation that they would be teaching computing or with a recognised qualification in computing (e.g. a degree in computing). Therefore, teachers often have little existing subject knowledge to draw on [90]. In order to have successful delivery of computing content, particularly programming, in grade schools, investigating teachers' perceived knowledge gaps in terms of computer technology integration becomes important. Due to the many facets of integration of computing in curricula, the variety of possible applications of computer technology available in the field of education and teachers' lack of content knowledge in the subject area, teachers may find this a tedious or daunting exercise.

[5] By grade school we refer to the elementary, middle and high school levels, also known as K-12, or primary and secondary school.

Teachers need continuous professional development in order to be comfortable and updated with what they teach. The objective of teachers' professional development is to effect positive change in teachers; teachers must challenge their familiar practices and knowledge to learn something new. Computing has been embraced with less enthusiasm, and found to be alien and difficult territory for teachers to become involved in [90]. The following sections looks at some important theories and literature on teaching and teachers taking on teaching computing, that will contribute to the main discussion of this thesis.

4.6 Components of Teacher Knowledge

Modern day education requires incorporating the growing demand of technology usage in the classroom, while focusing on the subject content and how to teach it. As communicators of knowledge to learners, teachers need to integrate different kinds of knowledge to succeed in effective classroom delivery. Fincher et al. [90] suggest that classifying "teacherly knowledge" is hard and that a teacher's expertise is largely professional, embodied, situated, and contingent. Knowing many ways to teach a topic, a teacher may use only a few of them, based on their expertise and selecting what is most appropriate to the circumstances of the moment.

In [7], the authors observe that different characteristics of teachers' knowledge exist and that they combine them at various levels. They define four distinct areas: 1) technical knowledge, which is their academic knowledge, 2) local knowledge, which is their narratives of the local school or community setting including the domain knowledge of local culture, sub-cultures and politics, 3) craft knowledge, which is the repertoire of examples, images, understandings and actions they build up over time, and 4) personal knowledge.

The idea of Pedagogical Content Knowledge (PCK), first proposed by Lee Shulman in the 1980s, categorises the special combination of content and pedagogy that is unique to teachers' knowledge about what they teach. According to Shulman, the PCK is "that special amalgam of content and pedagogy that is uniquely the province of teachers – their own special form of professional understanding" [90], and an attribute that only teachers develop. The level of content knowledge (CK) and pedagogic knowledge (PK) and the degree to which they are combined in practice determines a successful teacher; a teacher with high CK who does not possess the PK will be a poor teacher, while another with high PK but low CK may mask their lack of content but still is a poor teacher. The concept of PCK has been at the center of teachers' professional development as it highlights that teachers have specific knowledge that domain experts who are not teachers lack.

4.6.1 Technological, Pedagogic and Content Knowledge: The TPACK model

In 2006, Mishra and Koehler [127] introduced a framework named Technological Pedagogical Content Knowledge (TPACK) that extends the ideas of PCK "to identify the nature of knowledge required by teachers for technology integration in their teaching, while addressing the complex, multifaceted and situated nature of teacher knowledge". The model discusses three primary forms of knowledge namely: Content Knowledge (CK), Pedagogical Knowledge (PK) and Technological Knowledge (TK) and their complex interactions in the teaching domain (Figure 4.7), and TPACK is introduced as the heart of their complex interplay. Koehler et. al [128] explains TPACK as:

> *"the basis of effective teaching with technology, requiring an understanding of the representation of concepts using technologies; pedagogical techniques that use technologies in constructive ways to teach content; knowledge of what makes concepts difficult or easy to learn and how technology can help redress some of the problems that students face; knowledge of students' prior knowledge and theories of epistemology; and knowledge of how technologies can be used to build on existing knowledge to develop new epistemologies or strengthen old ones".*

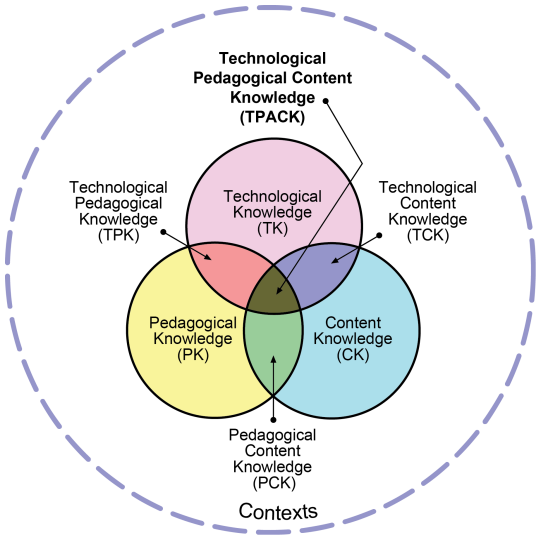

Figure 4.7: Technological, Pedagogical and Content Knowledge (TPACK)

Table 4.2 summarises the knowledge areas of teachers that develops the central concept of the TPACK model. As shown in Figure 4.7, a teacher's role in a classroom is an interplay among the three main knowledge areas they possess and manipulate that to effectively communicate the intended subject knowledge to their learners. The TPACK model provides a good framework to look at teaching from many perspectives, especially

to understand what areas computing teachers should pay more attention to than teaching a conventional subject in the curriculum. Although computer science teachers use technology for their teaching, the awareness of TPACK as a concept within the computing education research community is limited [90, 237].

Table 4.2: TPACK Explained

Technological Knowledge (TK)	**Knowledge of technology tools**: different ways digital technology (tools, applications, or software) can be used and the advantages and disadvantages of tools and software (technology-focused, not pedagogical considerations) [90]
Pedagogic Knowledge (PK)	**Knowledge of teaching methods**: how to teach, including professional beliefs and visions, theory, teaching, and learning strategies, knowledge about how learners learn best, reflection on practice, and the advantages and disadvantages of pedagogical approaches [90]
Content knowledge (CK)	**Knowledge of subject matter**: central topics, concepts, and areas of the subject matter that can be and are taught to students, as well as curriculum knowledge [90]
Technological Content Knowledge (TCK)	**Knowledge of subject matter representation with technology**: understanding of how technology and content influence and constrain one another. Teachers need to master more than one subject matter and have a deep understanding of they can be changed by the application of particular technologies; need to understand which the technologies best suit for addressing subject-matter learnt in their domains and how the content dictates or changes the technology or vice versa [128]
Technological Pedagogical Knowledge (TPK)	**Knowledge of using technology to implement teaching methods**: understanding of how teaching and learning can change when particular technologies are used in particular ways; knowing the pedagogical affordances and constraints of a range of technological tools as they relate to disciplinarily and developmentally appropriate pedagogical designs and strategies [128]
Pedagogical Content Knowledge (PCK)	**Knowledge of teaching methods with respect to subject matter content**: consistent with and similar to Shulman's idea of knowledge of pedagogy that is applicable to the teaching of specific content. Central to Shulman's conceptualization of the transformation of the subject matter for teaching, which occurs as the teacher interprets the subject matter, finds multiple ways to represent it, and adapts and tailors the instructional materials to alternative conceptions and students' prior knowledge. PCK covers the core business of teaching, learning, curriculum, assessment and reporting, such as the conditions that promote learning and the links among curriculum, assessment, and pedagogy [128]
Technical Pedagogical Content Knowledge (TPACK)	**Knowledge of using technology to implement teaching methods for different types of subject matter content**: knowledge and understanding of the interplay between CK, PK and TK when using technology for teaching and learning, including an understanding of the complexity of relationships between students, teachers, content, practices and technologies [128]

4.6.2 TPACK model in Computing Education

Denning's [63] quote of Donald Knuth's statement from 1974 that "expressing an algorithm is a form of teaching [to a dumb machine] that leads to a deep understanding of a problem" has hinted at the double-sided difficulties a teacher may face when teaching computing and CT: teaching a person to think (or solve a problem) in a way that the idea (or the solution) can be executed by a computing device (i.e. a machine) to achieve the end-result.

The impression the teachers develop about the subject in students becomes crucial in converting students from consumers to creators of digital technologies. Building teacher capacity in subject areas involves making strong links between the curriculum, student learning and teaching approaches, and that teachers require guidance on the use of resources effectively, including goals, assessments and examples of how other adults approach pedagogical strategies [237]. The CSTA/ISTE Standards for Computer Science Educators [12] suggest that the teachers may act as the lead learner, demonstrating the importance of life-long learning and a positive attitude in the face of new material or new challenges. The importance of recognizing the influential power teachers have on students' CS identities and trajectories is also highlighted. Moreover, it suggests that teachers should demonstrate comfort in problem solving and perseverance when encountering new or challenging content.

The TPACK framework could serve as a useful model for integrating CT and computing education, where the related ideas are connected within the subject matter and through pedagogical approaches teachers will use in their classrooms. However, the concept has limited awareness in the Computer Science teaching community [91, 237]. Fincher et al. [91] point out that it is probable that Computer Science teachers have qualitatively different TPACK than many other subjects due to their domain knowledge. For example, in CS the Technological Knowledge (TK) would be the use of technology like a compiler/interpreter and/or and Integrated Development Environment (IDE), in addition to what would be used in other subjects, such as PowerPoint, a projector, computer games, etc. Similarly, other subjects may have their own specific uses of technology (e.g. Google Earth in Geography, Sibelius in music) that use both CK and TK in their respective context. Therefore, it is easy to confuse CK with TK in CS. Nonetheless, the situation could be quite complicated for teachers with a limited background in Computer Science. An ambiguous understanding can result in teachers being unable to distinguish the existence of computers and the related technologies in teaching and learning from the actual subject matter, and may lead to confusion in realising the PCK of CS.

This situation is particularly evident especially among school teachers who teach CT in primary grades. Mouza et al. [161] report that the ability to weave knowledge of CT concepts, computing tools and practices with content and pedagogy varies. Some

teachers effectively design and enact lessons by seamlessly integrating content, pedagogy and computing tools to foster students' CT knowledge and skills, while others focus on uses of computing tools and problem-solving descriptions. They suppose that the participants in their research either did not indicate their knowledge of combining content, instructional strategies and computing tools to foster students' CT competencies, or they were unable to recognise and elaborate on the TPACK connections. They also report that even among pre-service teachers, the focus was primarily on CT concepts related more to automation, problem decomposition, data, algorithmic thinking, and to a lesser extent to abstraction and none related to simulation and parallelisation.

Apart from the popular CT competencies identified by many sources (i.e. algorithmic thinking, abstraction, decomposition, generalizing and patterns, evaluation, and logic [56]), CSTA/ISTE resources [12] suggest that the CT teaching can be "enhanced by a number of dispositions or attitudes that are important dimensions of CT including: (a) confidence in dealing with complexity, (b) persistence in working with difficult problems, (c) tolerance for ambiguity, (d) ability to deal with open-ended problems, and (e) ability to communicate and work with others". Another noteworthy point Mouza et al. [161] make is that, despite the teachers being introduced to a programming environment, none of the pre-service teachers chose to utilise programming environments to design their lesson, but relied on promoting CT through pre-designed software applications. Giannakos et al. [98] found that secondary teachers in Greece have indicated high levels of TPACK but expressed the need of knowledge in how to incorporate educational software in the teaching of computer science and how to improve the overall ability to transform and apply knowledge of algorithms with technology and pedagogy for effective teaching. This further indicates the vagueness of teachers' ability to blend their PCK when teaching CS subjects.

Figure 4.8 maps how different knowledge components can affect a teachers' confidence in teaching CT, and their relatedness to TPACK. Since it requires teachers to understand the manner in which technology and content influence and constrain one another, the essential curriculum knowledge in CT is mapped to TCK in the diagram (e.g. using an IDE to teach computer programming). CT teachers should also be confident in delivering the CT content, which require knowledge of different teaching strategies to use in their classroom and deciding when to use them, therefore is considered as PK. The knowledge in use of both hardware (e.g. computer, digital camera, 3D printer) and various types of software is also essential knowledge for CT teachers and therefore require teachers to have confidence in using them. Teachers usually possess the knowledge of this nature, if they are used to everyday use of devices like a mobile phone, but this kind of technology usage is different from using a programming IDE or program visualisation software; it is identified as TK in the TPACK model. As in any other subject, CT and CS teachers requires sound subject knowledge in the CT/CS content, in lesson

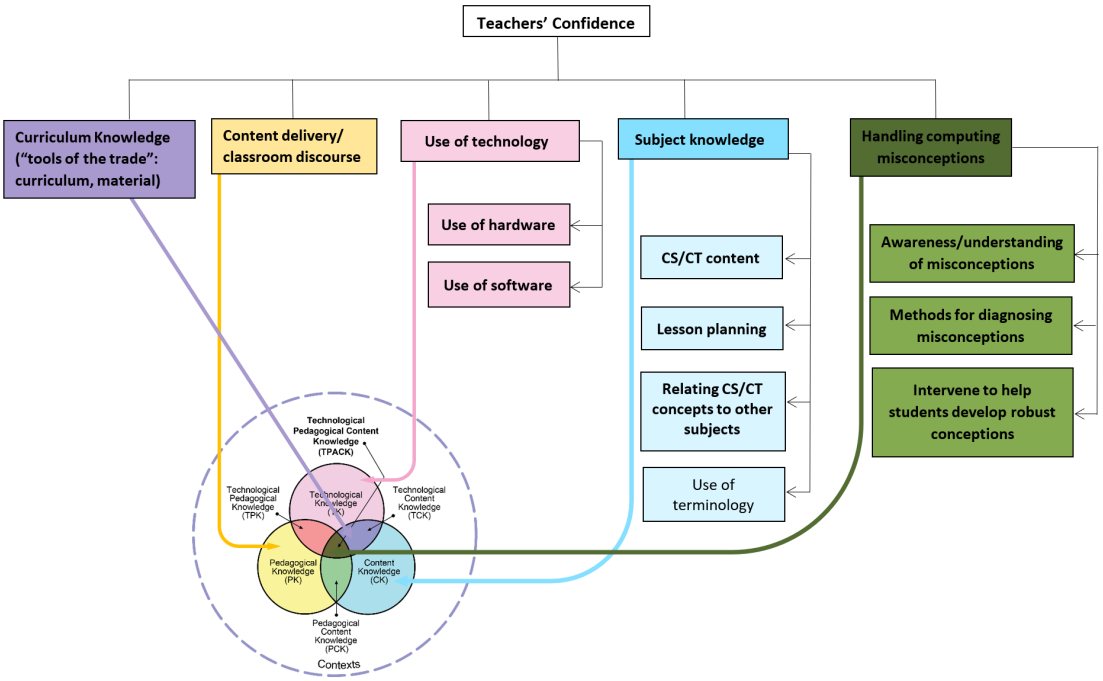

Figure 4.8: Teachers' Confidence in teaching CT and its relatedness to TPACK

planning, in using the appropriate terminology in delivery and in relating the CT/CS concepts to other disciplines and is mapped to CK. CT/CS teachers may be faced with various misconceptions regarding CT and CS. We break the knowledge they needed in this regard into three components: 1) awareness/understanding of misconceptions, 2) methods to diagnose misconceptions and 3) interventions to help students to develop robust conceptions. Handling computing misconceptions in such a manner essentially require all three kinds of knowledge blended together, and therefore is identified as the heart of a computing teacher's knowledge - the TPACK.

4.6.3 TPACK model and Unplugged Computing

Although research on the applicability of the TPACK model in CT and CS is limited, there are studies indicating that practising teachers do not fully exploit the affordances of ICT tools for constructivist teaching [138, 219, 243]. This indicates that, even in the context of ICT or digital literacy, a constructivist-oriented TPACK is particularly challenging for teachers. This also hints at an even more complicated situation for CT and CS teachers. Nevertheless, teachers' perceived knowledge gaps in this area are not well understood, as published studies have only examined teachers' TPACK perceptions with respect to conventional subjects such as science education or mathematics, or in an ICT context such as e-learning facilitation.

A possible reasoning behind these limitations could be the difficulty in distinguishing the kind of CK, PK and TK interplay needed to define a TPACK specific to teaching CT/CS. As discussed in the previous section, distinguishing a computing teacher's TPACK from their TK or TCK may not be as straightforward as in other subject disciplines, and less often it is seen as a knowledge needed to handle misconceptions. Moreover, the instructivist-bias seen in conventional programming education as well as many early (and to a certain degree, recently too) ICT teaching and learning may force teachers to assume that they should be restricted to either CK, PK or TK, whereas particularly for CT and CS, finding strict bounderies between the many elements of TPACK is practically impossible.

Unplugged activities in their purest form belongs to PCK, because it is an application of content knowledge for pedagogical purposes, outside a technological context. However, a predominant element of unplugged activities in teaching and learning CT is that teachers can use them to direct learners to *reflect* on the ways technology can be used in their problem solving, in a constructivist manner (e.g. using the Parity Card activity to teach how errors are handled using a computer); in other words, unplugged allows them to operate within the TCK region. Unplugged computing also helps teachers to use a pedagogy to communicate how to use technology to construct different forms of knowledge representations (e.g. use the Kidbots activity to teach how to "program" a robot), which thereby incorporates TPK.

The key pedagogical focus of unplugged computing is constructivism, a teaching and learning theory that describes learning being an active, contextualized process of constructing knowledge rather than acquiring it [103]. Despite constructivist approaches of teaching and learning being available as a learning theory for decades, implementing constructivist instructions in a computing classroom has not been sufficiently explored. Unplugged computing can be useful in a teaching and learning process as a tool to discuss a constructivist-oriented TPACK for teachers, in teaching a subject that is well grounded on technology, yet *without* using technology. It also is a useful tool to understand the different boundaries within the TPACK model, and can contribute to resolving teachers' confusions in the process of distinguishing teaching CT/CS from teaching ICT or digital literacy and using computer technology in teaching other subject disciplines. Moreover, unplugged computing's ability to support teachers in handling misconceptions in computing, particularly in programming, needs to be researched further in order to properly understand the gravity of its contribution in teachers' TPACK.

4.7 Self-Efficacy in Teaching and Learning Computing

Studies have often indicated that the high attrition rates in computer related studies (e.g. learning programming, obtaining a computing related educational qualification complet-

ing a degree related to computing, etc.) or in joining a profession in a computing related industry is not limited to the ability and/or knowledge in the subject, but caused by various other reasons [139, 193]. Various research has attributed these varying perspective of computing to a person's self-efficacy towards their knowledge and skill in the subject, and self-efficacy has been identified as a strong contributing factor for the improving interest in computing as a subject, in both the general student population as well as various other groups (e.g. lower represented groups, teachers, novice, etc.). Hence, researching self-efficacy has become a promising means of clarifying many questions in computing education.

Self-efficacy, originally highlighted by A. Bandura, is an individual's belief or personal judgment of their innate ability to execute courses of action required to successfully produce a desired outcome [14]. This belief can influence the choices individuals make and the courses of action they pursue. The discriminant validity and the convergent validity of self-efficacy in predicting common motivational outcomes, such as students' activity choices, effort, persistence, and emotional reactions, has been successfully researched in Education Psychology [14, 255]. Self-efficacy beliefs have been found to be sensitive to subtle changes in students' performance context, to interact with self-regulated learning processes, and to mediate students' academic achievement and confidence. This has become one of the most important motivational variables that helps explain the relationship between past performance and future results [140]. Zimmerman [255] suggests that when studied as a mediating variable in training studies, self-efficacy has proven to correspond to improvements in students' methods of learning, and predictive of achievement outcomes. Thus, self-efficacy research has made notable contributions to the understanding of self-regulatory practices and academic motivation.

Self-efficacy is more like how a person 'projects' their confidence to the outside as opposed to how they really feel confident about it for their inner self. This is an important aspect in both teaching and learning. As for a teacher, it helps them to emit the confidence that's needed to address/approach their learning audience. For a learner, it helps building a lasting impression over what they learn, even if the learning itself is not successful at times, leading to proper, more successful learning later on. For this reason, we focus on self-efficacy throughout the studies in this project as a meaningful measure of the different aspects we are interested in studying among the computing and programming teaching and learning focus groups.

According to Pajares [173], self-efficacy research in academic settings has focused primarily on two major areas: 1) exploring the link between efficacy beliefs and college major and career choices and 2) investigating the relationships among efficacy beliefs, related psychological constructs, and academic motivation and achievement. They identified, in the latter group of research, that the areas that contributed the most to self-efficacy were mental modeling, problem solving, teaching and teacher education and varied academic

performances, among others. Moreover, high self-efficacy has been demonstrated to influence students' academic persistence. Pintrich et al. [179] conclude that self-efficacy plays a mediational role in relation to cognitive engagement, and therefore improving self-efficacy might lead to increased use of cognitive strategies and thereby higher performance. Gibbs [99] suggests that "teachers' personal sense of control, and their beliefs in their capability to exercise control of their thinking during teaching, impacts on how they think, feel and feel and teach". He proposes that a central focus of teacher education should be more on developing teachers' self-efficacy and thought control over their actions.

In CS education it has also been found that students' self-efficacy is positively associated with course grades and previous experience (e.g. [115, 186, 246]). Lishinski et al. [140] examined the interaction of self-efficacy, intrinsic and extrinsic goal orientation, and metacognitive strategies and their impact on students' performance in a CS1 course, and found that self-efficacy and student performance has a feedback looping effect. They suggest that managing it throughout the learning process has great potential to improve student learning outcomes. Students' previous programming experience is strongly correlated with their extrinsic motivation, self-efficacy and CS career orientation.

Computer programming has been identified as being frequently used in teaching the skill of computational thinking to students. Since self-efficacy has been shown to have a strong relationship with skills attainment, evaluating the self-efficacy levels of students in any domain is considered important for making an interpretation of how successful they are or will be in that domain. Studies in students' computer programming self-efficacy, however, are comparatively less common than that of other subjects, and even lesser for younger students.

Hermans and Aivaloglou [115] in their research have observed that when primary students are introduced to computing with CS Unplugged techniques before they are exposed to the use of computers for programming, their self-efficacy has increased. They believe that the reason behind this is because children have increased confidence after using CS Unplugged techniques and they feel more confident exploring different Scratch blocks that they used to program using computers. Similarly, a systematic study of the influence of CS Unplugged in teachers' self-efficacy would give an insight into how CS Unplugged can improve their confidence in teaching computing to young students. Ramalingam and Wiedenbeck [187] report an early observation of a relationship between mathematical confidence and programming self-efficacy in computer programming learners.

Another aspect of self-efficacy in computing is *Computer Self-efficacy*. Compeau and Higgins [46] defined computer self-efficacy as "judgment of one's capability to use a computer", in other words, applying or representing an already formed sense of self-efficacy in the fields of use and mastery of computers. They suggest that attitudes

towards information technology are linked to computer self-efficacy as they are considered to be a significant factor in which individuals use computers. Albion [5] suggests that positive attitude towards computers and a strong sense of computer self-efficacy are basic preconditions for positive self-efficacy in computer-aided teaching.

This research shows that "teachers' attitudes towards modern technologies considerably influence the effective use of these technologies for school learning. Individual factors related to the teacher's personality, such as computer self-efficacy, motivation, needs, etc., seem to affect the integration and development of modern technologies in educational practice", as suggested by Paraskeva et al. in [175]. Teachers' self-efficacy might contribute to a considerable degree in the possibilities of the development of such technologies as important educational tools. Therefore, self-efficacy of teachers in various contexts such as teaching CT concepts, their programming ability, and their motivation to teach computing subjects are a key measure in the studies in this thesis.

Chapter V

Teachers Learning to Program: Details of the Experimental Studies and the Pilot Study

This thesis is supported by three experimental studies, all of which were conducted alongside teachers' professional development (PD) workshops on introductory programming events conducted by the Computer Science Education Research Group (CSERG) of the University of Canterbury in 2020 and 2021. The Digital Technologies learning area of the New Zealand curriculum was the key focus of the PD programmes conducted by CSERG, particularly the "Computational Thinking for Digital Technologies" Technological Area (CTDT). The first part of this chapter provides the context of the three experimental studies, detailing the resource material used in the introductory programming workshops, data collection strategies used and the data analysis methods followed in the experimental studies.

The second part of this chapter discusses the Pilot Study that was conducted in early 2020, which is the first of the three studies. Although the original planning intended to include several experimental studies to be conducted during the course of this project, the COVID19 pandemic and subsequent situations that built up globally interrupted those plans and adversely influenced the Pilot Study as well as the other two experiments that could be conducted. Nevertheless, the findings of these studies contributed to obtaining deep insights towards the theme of this thesis. Findings of the Pilot Study provided partial answers to RQ 2 (Chapter 1, Section 1.3) and its sub-questions.

Part I: Details of the Experimental Studies

5.1 Overview of Studies

Unplugged activities have long been identified as a useful pedagogical tool for teaching Computational Thinking (e.g. [62, 118, 194, 231]). The findings of Hermans and Aivaloglou [115], indicating that combining Unplugged activities with teaching programming can increase students' self-efficacy while achieving the same level of programming competency as using the traditional plugged-in approach, was a turning point for investigating its usefulness in teaching and learning programming at grade-school level. Accordingly, in order to study the impact of Unplugged activities for learning to program, and particularly how they can help teachers build confidence for teaching the CTDT topic,

experimental studies were planned alongside the introductory programming workshops conducted by CSERG for teachers' professional development.

The main curriculum focus of the professional development programmes was CT in the New Zealand curriculum [230]. The CT concepts that were covered during the workshops were on 'Data Representation', 'Algorithms', and 'Programming' topics from CTDT in the revised Technology learning area of the New Zealand curriculum.

The New Zealand Technology Curriculum

In the New Zealand school curriculum, the CS topics are listed under the 'Technology' learning area, which covers them under "Digital Technologies". The aim of DT curriculum content is to ensure that "all learners have the opportunity to become digitally capable individuals" [230]. It is separated into two sub areas:

Computational Thinking for Digital Technologies (CTDT) focuses on developing an understanding of the CS principles that underlie all digital technologies; this includes learning core programming concepts to enable learners to become creators of digital technology, and not mere users. CTDT consists of eight Progress Outcomes (PO). POs 1-5 focus on primary and junior high school students and cover the gradual introduction of CT concepts and introductory level programming. POs 6-8 cover the learning expected for students engaging in more intensive and specialised DT programmes.

Designing and Developing Digital Outcomes (DDDO) focuses on developing skills in designing quality, fit-for-purpose digital solutions. DDDO consists of six POs that describe the significant learning steps that students take as they develop their expertise in designing and developing digital outcomes.
The DDDO component of the NZ curriculum was not specifically covered during the PD workshops of interest in this study. However, the instructor often highlighted informally the possible connections to the DDDO content whenever it was applicable.

The flexibility in the NZ Curriculum allows the teachers to use curriculum integration, and thereby to blend the DT learning area with other learning areas. The context of DT content introduced to students at their respective year level depends highly on the teacher's knowledge and willingness to accommodate DT in their lessons. Therefore, factors like the teacher's confidence, self-efficacy and motivation inevitably becomes a barrier when their computing knowledge is low.

The teaching and learning context of the PD workshops of interest in this research cover the scope of the first five POs of CTDT in the NZ curriculum. The programming topics focused on in these studies were the three programming constructs: 'sequence', 'selection', and 'iteration', and it also covers 'variables' and 'input/output' (further discussion about this is in Chapter 8).

5.2 Resources Used in the Experimental Studies

5.2.1 Computer Science Unplugged Material

Several CSU activities and materials were used in the PD workshops to teach Algorithms, Data representation, and some of the Programming concepts. The following Non-programming Unplugged activities were used to discuss the CT topics in the workshops.

- **Binary Numbers**[1]
  Students use a set of 4 or more 'binary cards', (usually white with dots on one side and blank space overleaf, but variations can be used) to learn how base 10 numbers are represented in binary, as well as to count in binary and represent characters. A card is '1' when the side with dots are visible, and '0' otherwise, and collectively they create representational bit patterns. They observe the incremental pattern of the number of dots on each card (i.e. each card having twice as many dots as the previous) and learn how adding another binary card (or bit) allows representing twice as many numbers. The learning can be extended by using different things to represent 0 and 1, for example 'on' and 'off' or up and down arrows.
- **Parity Magic Trick**[2]
  Perform a 'magic trick' using a set of 'parity cards' (i.e. square cards with a different colour on each side: e.g.black and white). The performer uses an error detection and correction algorithm to trick the audience to always 'know' which card has been flipped over without looking.
- **Barcode Magic Trick**[3]
  The person performing the trick 'magically' says what the last digit of a product code is, by knowing only the other digits. They are actually using an error detection algorithm, as the last digit on the product code is a check digit.
- **QR Code Activity**[4]
  A QR code of a website is coloured to fill its white gaps one by one to see when the scanner would fail to detect to the correct information in the QR code, and what its behaviour is when it becomes unreadable.
- **Sorting Networks**[5]
  Students are given different numbers and asked to follow the arrows of a 'Sorting Network',

[1] Binary Numbers: https://classic.csunplugged.org/activities/binary-numbers/

[2] Parity Magic: https://www.csunplugged.org/en/topics/error-detection-and-correction/unit-plan/parity-magic/

[3] Barcode Magic Trick: https://www.csfieldguide.org.nz/en/chapters/coding-error-control/check-digits-on-barcodes/

[4] QR Code Activity: https://www.csfieldguide.org.nz/en/chapters/coding-error-control/qr-codes/

[5] Sorting Networks: https://classic.csunplugged.org/activities/sorting-networks/

which looks like a flowchart drawn on the ground. They compare each others' numbers as they go along, and without intentionally doing so find themselves in a sorted order when they arrive at the other end of the sorting network. This activity can be done with anything that can be put into an order (e.g. letters, words). The activity introduces the idea that a sorting algorithm can be constructed using simple comparisons between two values.

In addition, the following Programming Unplugged activities were used to introduce programming concepts. The first two activities are already developed and released, and their full descriptions can be found in the CS Unplugged website[6] and Classic CS Unplugged website[7]. The Kidbots activity in particular, can be used as an activity to explain general computing contexts as well as listed as a Programming Unplugged activity. However, in the experimental studies, the main focus of using this activity was to model the programming concept 'sequence', and it therefore listed under the Programming Unplugged activities. The Fitness Unplugged activity also had been refined to give more conceptual knowledge about programming. A further two activities were newly developed as part of this research, and a detailed discussion about the activity development process of these activities is given in Chapter 8.

- **KidBots**[8]
  Three students take the role of being a 'programmer', 'tester', and a 'bot' respectively. The programmer has to write a program using only a few specific instructions, as those are the only ones that the bot can understand, to get the robot reach a target placed on a grid. Usually the only three instructions are 'move forward', 'turn left', and 'turn right'.
- **Fitness Unplugged**[9]
  Students use instruction cards (i.e. a card to indicate the start, a few cards with simple fitness exercises, and a card that depicts a finishing pose) to write their own fitness program. They extend their programs to include 'loops' in their program, using a hula hoop and a few whiteboards for notation. For example, instructions on cards placed inside a hula hoop is repeated the number of times written on a whiteboard.
- **Variables Dice Battle (Using Flip Cards)**[10]
  Two players roll dice to battle against each other; whoever rolls the higher value gets a point. After 10 rolls, the highest scored player wins. One (or two) scorer(s) keep the scores of each player using two flip-card 'score cards' with values marked from 0 to 10. A time-keeper keeps track of the number of rolls using another flip-card 'counter' also marked from 0 to 10. Using flip cards for 'score cards' and 'counter', students learn several key

[6] CS Unplugged: https://csunplugged.org/

[7] Classis CS Unplugged: https://classic.csunplugged.org/

[8] KidBots: https://www.csunplugged.org/en/topics/kidbots/

[9] Fitness Unplugged: https://www.csunplugged.org/en/topics/kidbots/unit-plan/fitness-unplugged/

[10] Details of this activity can be found in the working document of Programming Unplugged activities at https://bit.ly/3NrueI5

concepts about variables such as they store only one value, and only one action can change their value at a time.

- **Conditional Dice Games**[11]
  Students in small groups play several dice games according to the conditions given to them as game rules. The game rules include Boolean conditions of increasing complexity, and student learn the nature of Boolean expressions and how 'if-else' conditionals work in a program. The games rules are designed to reflect some key concepts about selection, thereby avoiding misconceptions from forming.

5.2.2 Programming Resources

In the Pilot Study, reported in Section 5.2.3, teachers were provided with a basic introduction to programming using either the Scratch or Python programming language. In the other two studies participants used only the Scratch programming language. In all the studies, the instructor conducted the class while several helpers were available to support the learners if and when needed.

Except in the Pilot Study, the programming exercises were presented as Parson's problems[12] in the Scratch programming language [74, 83, 176]. Based on the feedback from participants, using Parson's problems was observed to be very effective, especially with novice programmers, because it allowed the learners to train themselves to use the visual programming language (i.e. Scratch) and the interactive interface as they go, with the instructor scaffolding the basic programming concepts at the same time. The learners were also provided with access to online resources developed by CSERG; recommendations to several online resources were also provided.

5.2.3 Data Collection and Analysis Methodology

A mixed methods approach was used for data collection in the experimental studies conducted during the project. Mixed methods research draws on the potential strengths of both qualitative and quantitative methods, allowing the researcher to explore diverse perspectives and uncover relationships that exist between the intricate layers of multi-faceted research questions [52]. It also assists in the triangulation of results based on multiple data sources.

All measuring instruments and processes used in this research have been approved by the Educational Research Human Ethics Committee of the University of Canterbury

[11] Details of this activity can be found in the working document of Programming Unplugged activities at `https://bit.ly/3NrueI5`

[12] Parson's problems provide learners with instructions of a program in a jumbled order, where they have to rearrange in an order that would make the program work. The puzzle-like format allows learners to practice basic programming in an entertaining manner.

(Appendix A). The three survey instruments used in the studies (described below) and sample questions asked during the telephone interview are given in (Appendix B).

Data Collection: Surveys

In the experimental studies, three survey instruments were used to measure 1) teachers' CT teaching self-efficacy, 2) their programming self-efficacy and 3) their motivation to teach CT topics. With no established tools available to measure these in relation to CT, three different validated and highly cited surveys that measure general teaching self-efficacy, programming self-efficacy, and motivation were adapted to suit the audience and context of this research. These adapted surveys were tested for reliability using Cronbach's alpha, a commonly used statistic to show that the instruments used in research projects are fit for purpose (a measure of reliability) [228].

Teaching Self-Efficacy Scale - This scale was developed by adapting the measuring instrument developed by Schwarzer et al. ([198] and [199]). The original instrument measures the teacher's self-efficacy in teaching by identifying four major job skills in the teaching profession namely: (a) job accomplishment, (b) skill development on the job, (c) social interaction with students, parents, and colleagues, and (d) coping with job stress. The measuring instrument consisted of 10 items each using a 4 point Likert-type scale.

The instrument was adapted to suit this study mainly by narrowing down the context focus of the items to teaching CT rather than the general teaching context the original version intended to address. The modified version used in this study carries 11 items and a 4 point Likert-type scale ("Not at all true", "Slightly true", "Mostly true" and "Exactly true"). The four Likert responses are similar to the original scale. The data obtained in the pre-survey of the Pilot Study revealed a Cronbach's alpha coefficient of 0.88, which lies within the acceptable range of $\alpha \geq 0.7$, indicating that the internal consistency of the items is strong.

Computer Programming Self-Efficacy Scale - This scale is an adaptation of computer programming self-efficacy for the C++ programming language designed by Ramalingam and Wiedenbeck [187], and a Computer Programming Self Efficacy Scale (CPSES) by Kukul et al. [131] for secondary school students for programming using Scratch and SmallBasic. As the main focus of this study is on primary school teachers with a variety of programming backgrounds, and the programming courses are conducted in Scratch, the context of the scale developed by Kukul et al. was more suitable in this study. It had 31 items and a 5-point Likert scale.

The adapted scale that was used in the studies contained only 24 items from the original version and a 4 point Likert scale instead of 5. Based on the expert opinion and feedback from the expert teachers in CSERG, 12 items that either were

targeted to measure complex programming tasks or were not directly relevant to the audience were eliminated, and 5 new items that reflect the role of a teacher in producing/debugging/correcting a computer program developed by others were added. A 4 point Likert scale was used to maintain consistency throughout the surveys.

Cronbach's alpha obtained for this scale from the preliminary data is 0.95. Though it lies within the acceptable range $\alpha \geq 0.7$ for reliability indicating strong internal consistency, some literature suggests that $\alpha \geq 0.9$ can be an indicator of redundancy or duplication among the items, or lengthiness of the scale [228]. Considering that the Cronbach's alpha value of both original scales developed by Ramalingam and Wiedenbeck [187] and Kukul et al. [131] were indicated as 0.95, we believe that the scale holds an acceptable level of internal consistency despite its high alpha value.

Motivation To Teaching DT - The scale to measure teachers' motivation towards teaching DT was developed by considering the theoretical framework of self-determination theory (SDT) proposed by Deci and Ryan [61], which explains that a teacher's motivation towards conducting various tasks related to teaching is mainly determined by five constructs namely, 1) intrinsic motivation, 2) identified regulation, 3) introjected regulation, 4) external regulations, and 5) amotivation. A more advanced scale developed in this regard, namely the Work Tasks Motivation Scale for Teachers (WTMST) [88], was also studied in developing the scale items. The WTMST discusses teachers' motivation based on six main tasks that teachers would be involved in when teaching in a classroom, namely, 1) class preparation, 2) teaching, 3) evaluation of students 4) class management 5) administrative tasks, and 6) complementary tasks. The scale considers the nature of these tasks within the five constructs suggested in SDT. They suggest a 15 item scale that should be measured individually with the 5 constructs suggested in SDT, resulting in a 90-item scale capturing teachers' motivation from a multidimensional and multitask perspective.

After careful consideration, since the focus of our study is only about a teacher's motivation towards teaching DT, motivation for other work tasks suggested in WTMST were not included in the adapted scale. A 4-point Likert-type scale of five items, each item formulated to reflect the five constructs of SDT towards teaching DT, was used instead. Cronbach's alpha for the pilot experiment data set is 0.41, and is lower than the acceptable standard range. As mentioned above, the literature suggests several reasons for a low alpha value (e.g. low number of questions, poor inter-relatedness between items, heterogeneous constructs). The literature also suggests that to increase alpha, more related items testing the same concept should be added to the test [228]. Having adapted only part of it, the

survey we used had fewr items than the validated original survey, which could have understandably resulted in a low Cronbach's alpha value.

All the survey data of the Pilot Study were collected online. Due to the low number of data points, the surveys used could not be statistically validated particularly for this study, and thus relied on scales inspired by validated scales instead.

Data Collection: Interviews and Observations

The qualitative data were collected mainly through interviews and observations. All the interviews were conducted and recorded through Zoom[13] and transcribed by the researcher herself. All the interviews were voluntary, where the volunteers initially shared their contact information for further communication, and the interviews were recorded with the consent of the interviewee. However, due to the unprecedented circumstances of COVID19 mentioned earlier, this approach further reduced the number of volunteers that participated in the interviews in all the experimental studies.

Interviews for the Pilot Study were between approximately 30 to 45 minutes, depending on how much the interviewee wished to discuss. The interviews of the other two studies were shorter, between 20 to 30 minutes. These were semi-structured interview and so a script of specific questions was not used, but a few interview guide questions were used (see Appendix B), based on the general interest of the respective study. The topics discussed in the interviews are discussed under each individual study. However, the following topics were discussed in general, mostly as a starter for the interview.

- The participant's background in teaching, and how they volunteered to participate in the PD programmes.
- The general culture of their teaching environment, and the nature of the student group they are involved in teaching (e.g. the year levels they teach, age and number of students in a class, their nature, etc.)
- The impacts that being a part of the programme had on them personally, their feelings about the programme, and challenges they faced.

In addition, observational notes of discussions during teaching sessions were kept mainly by the researcher herself, and in the main studies one other observer from the CSERG also took notes. All observational notes were made independently, and where applicable, the researcher and the other observer discussed them together prior to analysis.

[13] Zoom Communication Platform: https://zoom.us/

Analysis: Qualitative data

A thematic analysis approach [30, 31] was used when analysing the qualitative data collected in each study, using NVivo qualitative data analysis software[14]. Thematic analysis is a widely used method for analysing qualitative data and is used to identify patterns and meanings across a data set related to the research questions being investigated. The transcript text of the interviews and interviewer's notes were considered in this analysis. Since the interview questions were straightforward and the participants were familiar with the context of the interview, only a very few observations from the video recordings were used in the analysis, mostly to verify/clarify the interviewee's feelings through facial expressions when needed. But these occasions were very rare.

In patterns and meanings identification, the process used was 'data coding' (not the 'coding' in the context of programming), which tags sections of text with a code (generally a key-word or potential theme). In this iterative process the researcher goes through phases of becoming familiar with the data set, coding them, revising and validating the codes used, which involves reviewing and (when necessary) re-coding the data. Finally the researcher consolidates and describe the themes found, using the codes and coded data. No structural methodology or framework was used to identify themes, but an inductive approach was used instead. This approach allowed deriving more general concepts through interpretation of textual data. However, with this approach, there is a possibility that the analysis can be impacted by the researcher's preconceived theories about the data.

Analysis: Quantitative data

The quantitative data were analysed using statistical methods that suited each individual study. Non-parametric methods were chosen to be more suitable to analyse the Pilot Study (discussed in Section 5.2.3) data, due to the reasons that 1) the sample size was small, 2) samples are dependent/matched samples (as they are the views of a same person at two different times), and 3) normality of the distribution is unclear as the sample size is too small to guarantee that sample means are normally distributed.

The experimental design of the other two studies (discussed in Sections 9.5 and 9.6) had different hierarchical levels, given that the observations included multiple cases of data per participant. Therefore, multi-level modelling (MLM) [144] has been used in the analyses. It was assumed that different sources of variations were present: variability between experimental units (inter-individual) such as between groups and treatment types, and variability between observations within the experimental units (intra-individual) such as one participant's response over the two weeks. The experimental design also consisted

[14] https://www.qsrinternational.com/nvivo-qualitative-data-analysis-software/home

of several categorical variables (i.e. different self-efficacy scales) as well as hierarchical (e.g. pre and post survey of the same variable) variables. MLM accounts for shared variance within subjects while modelling between-subject differences.

A linear mixed effect (LME) [97] model that uses both fixed and random effects was fitted with participants as the upper level, and group, survey type and treatment as the lower level. A LME model separates the two sources of variability: 1) variability between individuals and 2) variability between multiple observations of the same individual. The statistical models used are discussed in detail in the respective chapters.

Part II: The Pilot Study

5.3 Overview

The Pilot Study was conducted in early 2020, with a group of New Zealand school teachers who voluntarily participated in a series of introductory programming workshops for a teachers' PD course conducted by the CSERG of the University of Canterbury. The participants consisted of a mixture of teachers who had experience in teaching computing (i.e. Digital Technologies in the NZ Curriculum) in their classrooms as well as those who are completely new to the topics of CS and programming. Most participants were beginner level programmers, including some who had no prior experience in computer programming. In their demographic survey, the vast majority of the participants opted not to reveal their gender information (which was given as a response option in the demographic survey).

Unfortunately, with the COVID19 pandemic outbreak in early 2020 and the subsequent lock-down of the country (in March-April, 2020), travel restrictions and many other uncertainties that followed, only one part of the planned experiment could be completed as planned, and the remaining workshops had to be cancelled. The original plan was to experiment with two teaching strategies to introduce introductory programming (i.e. plugged-in only and Unplugged mixed approach) during different sessions. However, only one strategy (i.e. plugged-in only) could be used during the sessions that were conducted. For this reason, some key aims of the Pilot Study could not be achieved, due to the absence of sufficient data for comparative analysis. Accordingly, the findings reported here are based on the limited data and observations available.

The main limitation and threat to validity of this Pilot Study was the very low number of participation in the surveys and interviews. Although 41 teachers were enrolled, only 12 teachers participated in surveys. Only 8 participants had responded to both surveys. The other four participants' data were incomplete and are not used in the data analysis. Six (6) teachers participated in the feedback interviews, which were conducted during a COVID19 lockdown period. All the respondents participated in plugged-in only

programming sessions, yet they have had experience with some CS Unplugged activities.

5.4 Aims of the Study

The main aim of this Pilot Study was to evaluate and refine the research instruments to be used in later experiments, observe the two teaching approaches for any future refinements, and to gain an overall understanding of the research interest group (i.e. teachers). We wanted to understand the nature of primary school teachers in teaching computing in their classroom such as their level of understanding, expectations and needs.

This study also aimed to observe and investigate the impact of professional development interventions using two different introductory programming approaches: 1) traditional plugged-in only approach, and 2) alternating Unplugged activities with plugged-in exercises. The comparative analysis of the impact between approaches was also intended to be studied. The impact was measured only in their self-efficacy in teaching CT topics, self-efficacy in programming, and in motivation toward teaching DT. Their programming competency was not measured, considering the short period of the PD program as well as the voluntary nature of the participants (i.e. most were programming novices, possibly being their first time ever to program, thus a programming test of any nature could have intimidated the participants).

Another aim was to investigate the impact of using Unplugged activities for introducing CT topics to teachers, particularly for those who are new to computer programming, using the comparative analysis between the two teaching approaches. The teachers' perspective on Unplugged activities as a useful teaching and learning tool in teaching computing, despite its known success as a method for teaching CT and introducing programming concepts to young students, was also aimed to be studied. However, this was not able to be achieved due to the reasons mentioned above. The findings were intended to support answering research question RQ 2 (How does the use of CS Unplugged as a PD tool impact teachers' confidence, expectations, and knowledge?) and sub-question RQ 1.1 (What models are at work when students are learning CT?) listed in Section 1.3.

Teachers new to computing who are not familiar with technical "jargon" can feel like they have landed in a foreign world, making them reluctant to take on the subject, and potentially leading to misconceptions and misunderstandings in the classroom. The support provided for teachers with regard to computational terminology is limited, and often do not go beyond the form of a glossary. This Pilot Study also aimed to investigate any impact of Unplugged style activities in helping teachers to overcome the issues related to the use of natural language versus technical language used in computing. However, the intended investigation could not be completed due to the cancellation of workshops which intended to used Unplugged activities. Nevertheless, an empirical study was conducted based on the limited findings of this study regarding teachers' understanding of technical

terms in a CT curriculum, aiming to answer RQ 2.3 in Section 1.3, and is discussed in detail in Chapter 6.

Another aim of the study was to investigate the nature of the teachers' general acceptance and views on DT in the curriculum and teaching DT in their classroom, as a partial answer to RQ 2.2, which is reported in Section 1.3. Areas such as the teachers' understanding of CT, awareness of the subtle difference between CT and computer programming, and the pedagogical approaches they would expect to use to teach CT in their classrooms were investigated.

5.5 Method

A small team lead by Prof Tim Bell from the University of Canterbury's Department of Computer Science Education Research Group conducted the PD courses. Prof Bell, who is an expert educator and a computer scientist, was the instructor for all workshops. Three other helpers, all expert teachers in DT, were available to support the teachers if and when needed.

Two workshops were announced initially on two separate dates and the participants selected the date of their preference. Each of the two strategies were intended to be used in each individual workshop. However, only one workshop could be conducted prior to COVID19 lockdown and the other could not be conducted in person. The duration of a workshop was three days, with a two hour lecture and a two hour practical session on each day. Demographic data of the participating teachers such as age, gender, length of time teaching DT, etc. were collected during these experiments to study whether there are any correlations.

Participants were invited to fill out an online pre-questionnaire before they arrived at the course, a short term-understanding survey at the beginning and end of the course ,and an online post-questionnaire after the course conclusion. The questionnaires contained the survey instruments described in Section 5.2.3. Participating teachers were invited to a follow-up telephone interview after the course conclusion.

Data gathered in this Pilot Study was used to measure the reliability of the instruments, in order for them to be used in the remaining studies. The same survey instruments were used as pre- and post- surveys, as a tool to measure the impact of the introductory programming intervention on teachers.

5.6 Results

5.6.1 Surveys

Looking at the nature of the study and the small sample size of the data, a non-parametric (interval or not normally distributed) test was selected to be more suitable to analyse

the quantitative data of the experiment. The reasons this analysis was chosen are: 1) small sample size, 2) the samples are dependent/matched samples (as they are the views of same person at two different times) and 3) normality of the distribution is unclear as the sample size is too small to guarantee that sample means are normally distributed.

The Wilcoxon Signed Rank Test was used to analyze the pre and post survey results of the experiment. This test is a non-parametric statistical hypothesis test used to compare two related samples, matched samples, or repeated measurements on a single sample when the distribution of the differences between the two samples cannot assumed to be normally distributed [247] (e.g. pre and post data of the same group).

The test statistic for the Wilcoxon Signed Rank Test is T, defined as the smaller of T_+ and T_- which are the sums of the positive and negative ranks, respectively. A T distribution table is consulted for the critical value corresponding to $T_{\alpha,n}$, where $\alpha =$ the level of significance and $n =$ sample size. Considering a statistical significance of 5%, the critical value for this study is $T_{0.005,8} = 5$. The decision rule is to reject H_o if $T < T_{\alpha,n}$.

Teachers' Self-efficacy Towards Towards Teaching CT

With respect to the null hypothesis

H_o: PD intervention does not improve teachers' self-efficacy towards teaching CT,

the test obtained a test statistic of $T_-= 6$, and that $T_- > T_{0.005,8}$. Therefore, H_o is rejected. There is a statistically significant improvement in teachers' self-efficacy towards teaching CT after the PD intervention.

Teachers' Self-efficacy Towards Computer Programming

The null hypothesis considered here is

H_o: PD intervention does not improve teachers' Computer Programming self-efficacy.

The resultant critical value for this survey is $T_-= 0$, that is $T_- > T_{0.005,8}$. Therefore, the null hypothesis is not rejected. There is no statistically significant improvement in teachers' computer programming self-efficacy after the PD intervention.

Teachers' Motivation Towards Teaching Digital Technologies

Considering a null hypothesis of

H_o:PD intervention does not improve teachers' motivation to teach DT,

the results indicated $T_- =7$, which is greater than $T_{0.005,8} = 5$. Therefore, the null hypothesis is not rejected. There is no statistically significant evidence to say that the PD intervention has improved teachers' motivation to teach DT topics.

5.6.2 Interviews

None of the teachers had the opportunity to use the knowledge and experience they gained from the workshops back in their classrooms. Moreover, it was observed that, since the COVID19 lockdown and subsequent online schooling was a new experience to many teachers, the priority given to DT content by the teachers was minimal.

A deductive approach was used in analysing the transcripts of the interviews with teachers, using the following criteria for tagging teachers' comments:

- By using their answers to direct questions of the theme and/or,
- By analysing their explanations to capture any indirect connection to the theme outside the key question.

Understanding of Computational Thinking

In the interview, teachers were directly asked what they think of as 'Computational Thinking'. It was observed that they used the term 'Computational Thinking' in the discussions even outside of this question. They showed confidence in using the term, even if they did not have a clear understanding or capacity to explain what is means. A clear division about the understanding of CT can be seen through the teachers' responses, where in some cases the same person held more than one view about it. Accordingly, the following themes were identified about their understanding about CT.

CT is a thinking process: CT involves thinking and is part of a cognitive process that people go through when learning or in everyday life.
" ... the processes that you go through in your head to make something easier for an end user" , *"We do it on the daily [basis]"*, *"it's not about thinking like a computer, but it's about thinking like a computer scientists."*, *" ... about thinking logically, with a very, I would say, like a lot of logical steps"*

CT is problem solving: CT is about everyday problem solving and is somewhat relatable to mathematical problem solving.
" ..the reasoning that you go about how to solve the problem to make the problem easier", *"a lot of Maths is computational thinking"*, *"apply some of the fundamentals of Computing to any given problem"*, *"trying to solve problems using computer skills or, in a way that a computer can do to solve that problem"*

CT is creating program code: CT is computer programming; implementation using a programming language.
"You're going to have to use computational thinking in order to create a computer program. Process that goes on in the creation of the computer program.", *"using like*

if or else statements and using data that some has come from somewhere, processing it and making it into something useful.", "...trying to solve problems using computer skills or, in a way that a computer can do solve that problem ...They need the devices to know some basic aspects of it."

CT represents wishful thinking about computers: CT is the expectation of computer executing a task it is asked.
"I tell them it's like magic ... you've just got to believe that it's there and that it's happening. And you just got to know that somehow the rabbits come from somewhere"

CT is the ability to manipulate a computer: CT is knowing how to manipulate a computer to get something done.
"Computational thinking is just the way, you don't really need a computer to ... You need a computer program, but you don't really need it to computationally think, you know"

CT is difficult to explain: Unclear about what the concept means.
"on one side it has like the skills like differentiation, sequencing, algorithms, things like that there. And on the other side it has approaches to, you know, like depositions - I thought the right word would be .. like to be .. umm..[thinking] for people to be social, ethical ..you know, Problem Solvers, Creators, ", "I think Computational Thinking is quite a hard thing to get your head around. But it is a process of steps that you would take to make a job simpler"

Distinguishing Computational Thinking from Computer Programming

Teachers were also asked if they think that CT differs from computer programming (CP), and provide reasons for their view. Except for one interviewee, all five others agreed that they are two different concepts/contexts.

CT and CP are different but related: *"I think they are different but I think they are related.", "Guess they are probably two different things really, [but] they relate because you are doing a lot of reasoning.", "I think they could be kind of one in the same. Although you might actually do a bit more problem solving at CT if you're coming at it from that perspective. I think you might be initially coming up with more with the process, the design, the different elements and, which will also happen on a CP, but they could be a slight difference.", "I won't draw a clear distinct line between each".*

CT is the 'reasoning' behind CP: *"To me, computational thinking IS reasoning. Because you are going to use a lot of reasoning. If you were creating a computer program for an end user.", " Computational Thinking is like, thinking like a computer*

and kind of almost anticipating what a computer might do with data. And then computer programming is making that program on the computer at work.".

CT is a subset of CP or vice-versa: *"It's very hard to clearly distinguish them, there's an aspect in each .. Sometimes one, CT, serves as a subset", "program is something that involves elements of computational thinking."*

Views of How to Incorporate CT in the School Curriculum

The teachers were asked their view on how to incorporate CT in a grade school[15] curriculum. All the interviewees generally showed uncertainty (in their expressions) in answering this question. However, the following themes emerged from their explanations and the examples they used in explaining.

By establishing a good understanding of the key concepts - The New Zealand Curriculum has highlighted key concepts that need to be established in learners (Section 5.1). Some teachers showed good awareness of this context, and despite their low understanding, identified the importance of establishing a good understanding of the key concepts described.
"The most important thing is getting the key concepts out for them to understand"

By integrating with other subject areas - The teachers who did not provide a direct answer to this question tended to explain their view using an example of what they do in the classroom, which involved exercises that incorporate CT concepts in other subject areas.
"I might do that [CT] for my Maths programme", "we did it inside our literacy programme"

By creating Computational Outcome though programming - Most interviewees were primary school teachers, who have had limited opportunity to attempt programming with their students yet. However, they seem to believe that programming would be the way to incorporate CT in their lessons.
"You could get children to create something on Scratch that showed the understanding of something else.", "... we looked at TaleBlazer. There's a small amount of block coding within it, CT"

By encouraging engagement in computing - Increased engagement in computing (and with the computer, as the device) through gaming, puzzles, etc. was also suggested.

[15] By grade school we refer to the elementary, middle and high school levels, also known as K-12, or primary and secondary school.

"It's all about engagement. It doesn't have to be on a computer, [but] if you doing on a computer, it's more raw"

Through problem solving and/or design thinking - Problem solving was a popular topic among the interviewees when explaining their ideas about CT, which is an indicator of their confidence in using it as an effective strategy in incorporating CT in the curriculum.
"I think, problem solving, and little games can be much more engaging in the meantime to get to a point where they feel more confident doing that thing.", *" I tend to integrate more design thinking"*

By maintaining authenticity - Maintaining an authentic context and use examples, exercises, and activities that are relatable to the student audience (e.g. age, literacy level) and purpose of the lesson.
"I think it's really important that it's an authentic context for students.", *"... integrating into daily life. If that programming exercises with no authentic context]* is the only way you are going to do it, they are not going to progress as fast as if you explicitly teach it."

Pedagogical Strategies to Develop CT

The interviewees were asked about the pedagogical strategies they have used or they think are useful in helping students to develop CT.

Unplugged style - Unplugged style teaching and learning, that included examples from CS Unplugged material as well as examples of unplugged approaches that are not necessarily Unplugged Computing, was a popular choice among interviewees. This strategy was mentioned as one of the ways to bring authentic context to lessons.
"I do like the unplugged activities as a precursor to what we're going to do on the screen. I think that really helps make a bit more sense when we got onto the screen and we did apply those principles as well", *"I love physical resources. Things that are hands on and that you can manipulate ."*

Peer and self learning - Using peer learning (i.e. let more competent student in the class to help weaker students, or use the students to challenge themselves). Allowing students to understand by themselves (under teacher's guidance and observation) was also mentioned.
"Children can explain things to other children in child speak", *"form a system where you let those brighter kids share their knowledge and coach the others around and serve as a resource"*, *"There is a certain element of you want[ing] them to play and discover for themselves as well."*

Use both unplugged and hands-on device use - Changing between physical activity and using some device (e.g. computer, programmable robot, Beebot) was popular among interviewees; most of them being primary school teachers, they found this swapping between the styles can keep the young students engaged in their lessons more objectively.
"After the hands on practical activities, it's also useful to have the actual systems.", *"They make paper copy books and they also made a digital augmented book ... not so much of a book - like a trial you know - as a trial."*, *"Digital tech kits that are functional computers where you need to build them from scratch. Self instruction is to complete that task or to solve that problem. And at the end of the day when they have actually assembled or put all the bits together in they press on the on button and it's functional."*

Use games and puzzles - Strategically using physical board games, puzzles or computer logical games or puzzles was seen as a useful strategy to convey CT concepts.
"Well for me, it would be through games, like a logical game, board game." , *"We looked at TaleBlazer[16]."*.

Use Immersion Model of Learning - This model was mentioned as a good strategy to bring authentic context and engagement into lessons.
"that immersion model of languages, combined with that explicit teaching is probably the best way to teach it. And I think to be really consistent and to consolidate learning,"

Use Programming - Programming was mentioned at various point of the discussions, as a key component behind teaching and learning CT. However, being primary school teachers, predominantly at earlier year levels, most interviewees had limited chances for using them yet in classrooms.
"I love Scratch. I don't think you can go past Scratch for primary age kids", *"CodeWOF, oh yeah. So I found that was a VERY good supplemental tool"*, *"Parson's problems"*, *"Programming Microbits or Arduno."*.

Themes Emerging from the Interviews

The following themes emerged from the discussions.

- Mixed feelings about CT and/or DT - *"I do think that it's valuable that children understand that it is not all about what happens on the computer"*

[16] TaleBlazer: An augmented reality (AR) software platform developed by the MIT Scheller Teacher Education Program (STEP) lab that allows users to play and make their own location-based mobile games. http://taleblazer.org/

- Uncertainty or lack of confidence - *"well maybe don't have to be an expert, but you know because you need to know quite a lot, at least."*, *"Well, actually, to where they were a little bit more unstructured was that it's all about engagement"*, *"I'm quite happy that if anyone else [but me] wanna to do that, give that a go if you want to actually go on and do something ... "*

- Difficulties in connecting CS Unplugged activities to programming - *"I kind of get it, how to use it [CS Unplugged material] for CT. But, I am, you know, kind of find difficult to use them for programming."*

- Appreciation for the opportunity for peer learning - *"I realise I'm not the only one who had the same question", "One teacher gave me some great examples I can use in my own class"*

5.7 Discussion

The results of the quantitative data analysis indicate that the professional development course to introduce CT and basic programming has improved the teachers' self-efficacy for teaching CT topics in their classrooms. However, their programming self-efficacy and their motivation to teach DT had not improved significantly after the course.

The lack of improved self-efficacy towards computer programming could be due to several reasons: 1) that this programming experience had made them realise their own level of understanding/knowledge in programming, and/or 2) that the length of the course is not sufficient for them to develop a substantial skill in programming, yet worked more like an 'eye-opener' and/or 3) just that the sample size was too small to see an effect given the other points. The indication of their increased self-efficacy in teaching the content could be seen as related to this argument, although the statistical support is minimal. Moreover, the workshop did not use any assessment to test the participants' programming skill, and the survey results only reflects the participants' personal feeling about their level of competency in programming.

The qualitative analysis of the study revealed that even after a professional development experience that objectively attempted to explain CT concepts and programming, the teachers' understanding about CT is somewhat vague. However, despite this ambiguity in their ability to give a definition, their views showed a good awareness of the CTDT component of the curriculum.

In the part of the Pilot Study that we were able to conduct, the introduction of both CT and programming were restricted to only lectures and plugged-in exercises, and the use of Unplugged material was carefully avoided. However, the teachers were knowledgeable about CS Unplugged material from elsewhere and were appreciative about them as both a good strategy to teach CT content in classrooms, and as a successful teaching

and learning resource. However, their focus of using Unplugged activities was mainly on delivering CT concepts, and little consideration had been given to using them for delivering programming concepts. Teachers showed concerns on how to connect the existing CS Unplugged activities (which are not generally designed to model programming concepts, as discussed in Section 3.3) into a programming context. Further, some teachers seem to have a confusion between offline activities and Unplugged computing (e.g. considering a colour-coded grading system as an example of an Unplugged computing activity).

The teachers showed a good understanding of a breadth of pedagogical strategies and tools that can be used in communicating the CT content in their classrooms. This observation is reflective of the positive responses during surveys to their self-efficacy for teaching CT and their motivation to teach DT in classrooms. Moreover, they suggest that participating in PD programmes with the peers 1) an effective and efficient way to learn to learn programming (i.e. to obtain conceptual knowledge required to use in their classrooms), 2) helps them overcome fears of teaching DT, and builds confidence in teaching the content, and 3) are an efficient portal for sharing teaching resources and ideas.

5.8 Conclusion

The Pilot Study data were used to evaluate the reliability of the three survey instruments used (see Section 5.2.3). Judging by the Cronbach's alpha values of each survey obtained with the limited results, the three survey instruments are considered sufficiently reliable. Based on the feedback from the Pilot Study participants, very minor adjustments were made to further clarify some elements of the surveys.

Professional development support provided on introductory programming and CT increased the self-efficacy of teachers towards teaching CT content in their classrooms. Feedback indicated that the teachers appreciated the peer learning and the opportunity to concentrate on their own personal improvement during such courses, which seem to boost their self-efficacy.

Although introducing programming combined with unplugged activities could not be evaluated in the pilot study, the teachers' enthusiasm about Unplugged as a successful teaching strategy to deliver CT content supported similar observations in literature (e.g. [58, 118]). However, their concern over the difficulties in connecting the existing CS Unplugged activities in a programming classroom seem a worthy concern that resulted in further investigations and detailed analysis of the existing Unplugged activities, as well as designing more programming-focused Unplugged activities. Detail discussions about these investigations are can be found in Chapters 8, 9 and 10.

Chapter VI

The Role of Language and Teachers' Understanding of Jargon in a Computational Thinking Curriculum

Teachers new to computing who are not familiar with technical "jargon" can feel like they have landed in a foreign world, making them reluctant to take on the subject, and potentially leading to misconceptions and misunderstandings in the classroom. The diversity of technical words, metaphors, and phrases in different contexts can make their meanings confusing, ambiguous or misunderstood for the diverse stakeholders in computing education. Therefore, understanding the nature of the commonly claimed difficulties and confusion caused by computer jargon among teachers becomes important for finding ways to address this issue. This chapter presents an empirical study conducted in early 2020 along-side the Pilot Study and with the same participants, to understand the nature of teachers' understanding of computational terms (jargon) related to Computational Thinking concepts, and how a relevant professional development intervention can help to resolve issues related to them (i.e. in support of RQ2.3 as mentioned in Section 1.3). The results of this study were published in 2021, in [162].

6.1 Background

Irrespective of the subject or discipline, successful teaching and learning demands a certain level of fluency in specific use of language in the school classroom, including precise use of terminology and established (academic) language that enables communication about abstract concepts [69, 154]. Effective use of language plays a vital role in explaining something in simple terms. Furthermore, teachers new to a subject who are not familiar with the "jargon" can feel that they are being asked to work in a foreign world, and this may even prevent some from considering taking on the subject.

Teaching and learning computing and CS is closely connected to understanding, describing, explaining and expressing CT concepts. Wing's [248] definition of CT (Section 2.3) suggests that CT is a way that humans think about solving problems that incorporates the set of mental tools used in CS. Referring to this early definition, CT is explained in much simpler terms in the literature as "thinking like a computer scientist to solve problems" [51]. CT is the term often used to denote the conceptual core of CS [51], and an approach to problem solving that consolidates logic skills with core CS

concepts [185].

CT is about teaching students to understand how to use computation to solve their problems, to create, and to discover new questions that can fruitfully be explored in other disciplines and professions as well as CS [15]. Teachers need to apply this idea within the context of their classroom, and should be able to effectively communicate the ideas of CT to students. The different roles/contexts in which terminology is used in computing, and ambiguity and confusions in terminology can make it difficult for teachers to communicate the accurate meaning of the concepts. Such situations can cause confusion in a learning community, especially for those who are new to computing. Teachers have a key role in implementing a CT curriculum, and if they find the language used in the curriculum to be challenging then this can be a barrier to achieving the intention of the curriculum [120, 232].

6.2 Aims of the Study

The main objective of this study was to investigate the nature of CT terms and concepts teachers find difficult to understand in the context of curriculum descriptions and how a Professional Development intervention can impact their understanding. This study intends to find answers to the sub-question RQ 2.3 (see Section 1.3) by considering specific questions:

Q 1: What is the nature of the terms teachers find difficult to understand in computational writings?

Q 2: What is the impact a professional development intervention can have on their understanding of terms in computational writing?

When considering the languages used in computing classrooms, there are many different kinds of contexts that apply. This investigation explores the proposition that it is not so much the computational jargon itself that the teachers find difficult to understand, but the context in which the language is used that makes things difficult for them.

6.3 Participants

A group of 41 New Zealand teachers participated in a three day, in-person Introductory Programming Course for Professional Development (PD) in early 2020. This study was conducted as an empirical study at the same time and with the same participants as the Pilot Study discussed in Chapter 5.

The group varied from primary to high school teachers who are currently teaching the New Zealand Digital Technologies curriculum, or interested in teaching Digital Tech-

nologies in future. The details of the survey instrument is given in Section 5.2.3 and a description of the PD workshops is given in Section 5.2.3.

6.4 Method

As an entrée to the empirical investigation, a literature survey was conducted at first to investigate the different contexts and roles that language can have in a computing classroom.

The PD course used as the professional development intervention of this study was advertised as being for complete beginners, and introduced the basic concepts of programming, with guided practical experience in a programming language of the participants' choice from Scratch or Python. Both the course delivery and any material provided were not specifically designed or targeted to address any vocabulary related issues. The main data collected for this study was a pre- and post- opinion survey. The PD course was considered as the PD intervention in this study, where the pre- and post- surveys were conducted. The participants were given an identical survey before the beginning and after the end of the professional development intervention (that focused on programming skills) to reflect on their understanding about a section extracted from the of NZ Digital Technology curriculum [230].

Participants were provided with a copy of Progress Outcome 5 text (shown in Figure 6.1), and were requested to highlight any words they found difficult to understand (e.g. computer terminology, jargon words) used in them. Further, additional space was provided for them to include any comments or concerns they have about the text, but no specific questions were asked. The objectives of this survey were 1) to identify terms (technical or otherwise) that teachers found difficult to understand in the curriculum, and 2) to observe any improvement/change in their view about such difficult terms after completing the course. Participants were provided with a copy of their individual pre-intervention response as a reference in the post-intervention exercise.

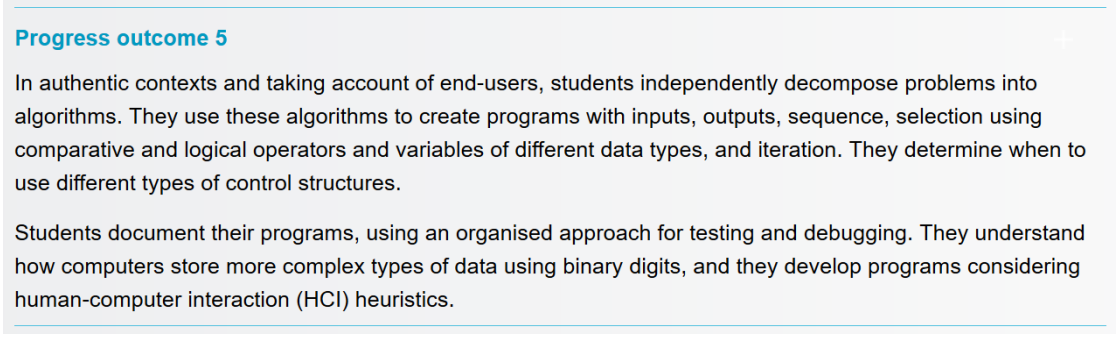

Figure 6.1: Progress Outcome 5 - CTDT from The New Zealand Curriculum [230]

The PD course was delivered as a collections of programming exercises explaining the basic concepts. During the course, the Instructor showed how the content covered relates to the Progress Outcomes of the curriculum.

An empirical evaluation was conducted based on the data collected from the opinion survey, to understand the teachers' understanding of computational language, in a pragmatic way. To evaluate how a teacher might find definitions of terms without the benefit of a PD session, a qualitative data analysis was carried out, considering several approaches that a teacher might use. All the words and phrases highlighted by the teachers were checked in: 1) a simple Google search, 2) a Merriam-Webster online dictionary [68] and 3) a Dictionary of Computer Science (Oxford Reference) [32]. The single words were also checked in the New General Service List (NGSL) [37] and the New Academic Word List (NAWL) [36]. Moreover, a qualitative analysis was carried out to investigate the nature of the comments made by the teachers to understand the nature of their difficulties, if any, and to investigate the impact of the professional development.

6.5 Language Use in Computing

Language use in computing education is multi contextual: it could be communicational (educational/instructional), formal (programming/modelling) or technical terminology (also referred to as jargon) [17, 70, 205]. English is a key natural language in computing; programming languages are often based on English, and words and phrases from the English language are often used as jargon in CT (e.g. *selection*, *iteration*, *heuristics*). The application of the meanings of such words or phrases in a CT context can be in their commonly known definition (e.g. *program*), a computational definition (e.g. *sequence*) or metaphoric (e.g. *cloud*). Computer jargon also includes words that are new to the English language, but many of these have entered common usage and are often based on existing words (e.g. *software*, *internet*, *GIF*, *phishing*, and *byte*). This situation can be even more confusing when similar words are found in different computing contexts; for example, an HCI heuristic and a heuristic algorithm. Moreover, formal languages, including high level programming languages, use words, phrases and metaphors in contexts that may or may not match their natural language usage meanings (e.g. *while* and *print*). Ko [126] argues that programming languages have socio-technical meanings due to their popularity and evolution in the modern world. As such, from an educator's point of view, distinguishing the meanings and explaining them or using them to explain computing concepts to a novice learner becomes even more challenging.

6.5.1 Computer Jargon

The term "computer jargon" is used in both computing and non-computing communities to refer to words and terms of computation, and can imply a pejorative intention.

Wikipedia defines computer jargon as "words to do with computers and surrounding topics" [81]. Many Computer Jargon listings available on the internet are either focused around a target audience (e.g. teachers [80] or beginners [145]), a specific area of the subject (e.g internet [38] or networking [47]), or can be very generic with a broad context (e.g. "A dictionary of jargon" [221], "The most common jargon list" [159]). Given this variety of target audiences, searching the internet for a technical term could be either helpful or more confusing to a novice user.

Two phenomena were observed from studying the many lists of computer jargon available online: 1) regular words or terms are given specific meanings or definitions when presented in computing related context (e.g. *selection*, *code*) and 2) the computational definition/meaning has become the more common usage of some words than their regular meaning (e.g. *program*, *bug*). Therefore, when a teacher says "the wording of the curriculum is challenging, daunting" [120], it is difficult to distinguish whether the difficulty lies in their understanding of the computing context, or in the unfamiliarity of the natural language meanings of the words. It may even be that they know the meaning of a word, but find the context overwhelming. Although the practitioners of computing may use computer jargon assuming that others in the computing community can understand them, this may not work for novice students and teachers who do not share the same knowledge or expertise of an experienced computer scientist. Computing education requires subject sensitive language awareness in classes, and educators need to be able to blend natural language and computer vocabulary in a manner that supports the learner to understand the subject effectively [71].

6.5.2 Formal languages and Programming Languages

In addition to jargon, disciplines such as CS and Mathematics often use formal languages that are precise. For example, in both Mathematics and Programming "*(a+b)*c)*" is incorrect (due to the missing bracket); and programming languages have keywords such as *for* and *while* that can only be used in very specific ways. In contrast to natural languages and their ambiguities, formal languages have a fully defined set of axioms, and all valid sentences of the language can be derived by the application of a set of rules over these axioms. This forces formal languages to be precise and complete. Due to their low redundancy, formal languages tend to contain sentences with every symbol contributing directly to the meaning of that sentence [60].

Precisely defined formal languages play a vital role in computing to ensure that the content can be accepted by a computer. The construction and interpretation of content from a formal language standpoint (e.g. program code) provides a structural understanding of the concept to the learner. Diethelm and Goschler [70] argue that in CS, the 'discipline-specific' or 'domain-specific' language is achieved with formal languages that

have been developed for special uses in particular CS specific application domains.

Programming languages, an important subset of formal languages in CS, are popularly expressed as "vocabulary and a set of grammatical rules for instructing a computer or computing device to perform specific tasks" [245] or as "a formal language, which comprises a set of instructions that produce various kinds of output" [82]. Programming languages' popularity and evolution in the modern world, and the resultant socio-technical meanings [126], could create confusion among the teaching community, especially among those who have limited exposure to computing. It is important that teachers are aware of the ambiguities that can be caused by the natural language they use in their classroom that could also lead to misconceptions, as well as the need for accuracy in expressing ideas in CS.

6.5.3 Use of Metaphors in CS

Computer jargon also uses a rich collection of metaphors (e.g. *bug*, *cloud*, *queue*, *garbage collection*, *handshake*, and *tree*) and there is a large body of literature about the use of metaphors in CS [42, 93, 121]. Metaphors help learners to form mental models (conceptual frameworks) leading to deeper understanding of emerging new knowledge in CS [42]. But poor metaphors can be misleading and students may draw false conclusions based on them [94]. Colburn and Shute [42] argue that CS metaphors deal with both pre-existing and emerging similarities between computational and traditional domains. They also show that metaphors play an explanatory role for creators and users of software, and provide context or terminology for both hardware and software creators in their creative exercises.

Importantly, when using metaphors in a context, the target (new idea) never matches the source (known idea) perfectly, thus the user can easily use the metaphor to infer incorrect conclusions about the target [94]. Therefore, a novice or non-expert in computing may find it extremely difficult to distinguish the meaning of a technical term if they transfer too much of the meaning of a metaphor from its original context, and the teacher may need to explicitly mention any limitations on the use of the metaphor.

6.5.4 Role of Language in Teaching and Learning Computational Thinking

Diethelm and Goschler [70, 71] highlight the importance of the human language used for learning and teaching CS, particularly CT, and the need for a language to communicate about CS that combines everyday language and discipline specific language, and sums up "all the terms of the discipline (the 'terminology' in a narrow sense), but it also contains the usage of these terms in disciplinary contexts". However, this observation brings only a linguistic explanation to the situation, but does not address the difficulties language use could create in a classroom context.

It has been reported that teachers have identified the need to encourage students to develop, embrace and articulate a common computing vocabulary so that they are aware of technical keywords and their correct use [204]. Soloway et al. [214] raise the concern that if the programming language does not "cognitively fit" with the problem-solving skills of a novice, it can become a barrier to their use of computers. They show that transferring the meaning of words in natural languages across to the same words used as tokens in formal languages, but with a different meaning, is challenging to novices. They suggest that the use of natural and formal languages in educational programmes should help students to face this challenge, since the usage and practice of language can influence the cognitive involvement in the process of programming. However, it is noteworthy that this observation can diverge when English is not the learners' first language or when a particular word or a phrase is not previously known to the learner, in which case the educator has the privilege of explaining or introducing the meaning of the word/phrase to match the subject being taught, thus making it easy for the meaning to "cognitively fit" the technical situation.

Good and Howland [102] observed that learners are able to use natural language quite accurately to describe a system requirement, but needed help in doing the same when using programming language and in coding. They suggest that natural language is well suited to the activities of comprehension, debugging and collaboration, but not for program generation. Lu and Fletcher [143] suggest that early age introduction to CT should emphasise: 1) students understanding the computational process rather than manifesting in a particular programming language, and 2) the skills of abstracting and representing information. They propose using a "computational-thinking language" that incorporates computing concepts in core content areas.

Experiments carried out with teachers as the educators of CT to school students show that professional development for teachers can address common misunderstandings about CT, although it does not seem to completely eliminate them [57, 71]. Such misunderstandings could cause teachers difficulties in eliciting appropriate responses from students, as students may be of a view that they "understood", albeit with a different meaning [71]. Curzon et al. [57] points out that there is a knowledge and a skill gap, especially in subject-based pedagogy, and highlights the importance of teachers having both a deep understanding of the knowledge, and ways to help students develop the skills. Accordingly, the teacher's ability to explain the meanings of words and phrases specific to computing becomes important. Empowering teachers with valid information about the proper usage of natural language in the CT teaching process provides an opportunity that could inspire students' understanding and application of the CT concepts [84].

Most K-12 educators understand a subset of computation related ideas used within the context of their classrooms [253], but it is important to ensure that teachers have a common understanding within their local teaching communities (e.g. school, peer groups,

etc.) about the commonly held jargon or acronyms used within them, otherwise they may either not understand, or have a different understanding of the meanings of the jargon used. Approaches for enabling appropriate use of language in teaching and learning CT and CS has not been researched by many. Diethelm et al. [71] suggest that computing jargon with very specific meanings within the computing community, and sometimes completely different meanings outside it, need to be taught and learned as completely new words, if the goal is to prevent learners from trying to understand them based on their knowledge from everyday use, which can lead to unhelpful misunderstandings or misconceptions.

6.6 Categories of Curriculum Language

As an attempt to understand the different ways natural language is used for CT education in a classroom, which can possibly create confusion among teachers without computing backgrounds, we have identified different categories of terminology, as presented in the diagram in Figure 6.2, based on the context in which they would be used. The language use in these categories could be in the form of a word (e.g. *control*), a phrase (e.g. *binary digit*, *control structure*), or a metaphor (e.g. *garbage collection*), etc.. Their usage and application in computational studies might be: 1) similar in meaning to everyday language (e.g. *create*, *binary*), 2) different from the everyday meaning (e.g. *bug*, *cloud*), 3) different when applied in computational context (e.g. *decompose*, *variable*) or 4) any combination of 1, 2 or 3, (e.g. *artificial intelligence*, *data types*, *heuristics*). Moreover, the distinction between these categories can become vague, depending on the expertise of the user and/or the depth of the context.

This research was conducted in New Zealand, where the Computational Thinking curriculum content is expressed as a series of "progress outcomes" (POs) [230]. There are eight POs that span primary and secondary school, and give the sequence of concepts that students are expected to work with. They are not tied directly to ages or year levels, although some guidance is provided, and the last three of the eight are closely tied to the last three years of schooling where students are taking national assessments for qualifications. Since this specifies "the curriculum", we have focused on the terminology used in the Progress Outcomes (especially PO5, which accumulates all previous POs for Computational Thinking, and is the last one that is expected for all students), and teachers' understanding of it.

Figure 6.2 demonstrates how pedagogic terminology in material like curricula may use terms with general meanings, as well as terms that are specific to the technical vocabulary of CS or CT. The terms in curricula and/or CS teaching related texts which are used in their generally known meaning are indicated in Set A. Set B indicates the terms with computational meaning, also known as computer jargon or technical terminology.

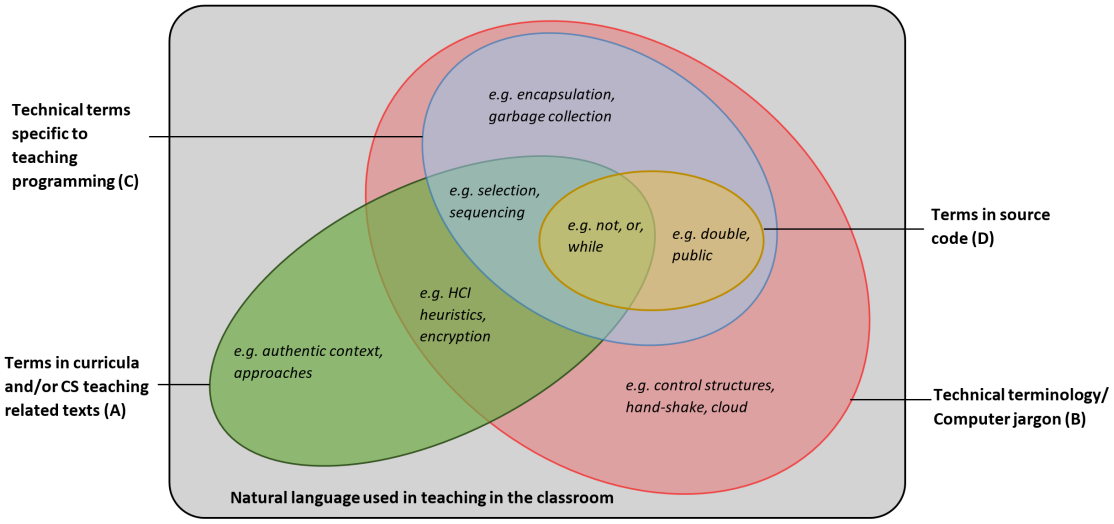

Figure 6.2: Different contexts in which natural language is used in CT education

Computer jargon with specific meanings when used in computer programming is indicated in Set C. Terms that have specific purposes when they are used in program source code are indicated in Set D.

For example, the Progress Outcome 1 of the New Zealand Digital Technologies curriculum contains terms like *authentic context* (Set A), *algorithmic thinking* (Set B) and *debugging* (Set C). Words like *and*, *or*, *not*, and *this* are found as keywords in program code in Python, C++ or Java, but they are also used elsewhere in the language of the curriculum – in fact, they may even be juxtaposed, such as the description of Boolean logic in the UK Key Stage 3 curriculum [165] ("AND, OR and NOT"). Set D represents the words that are used as part of the syntax and semantics of a programming language. Program source code can contain terms that are specific to the programming language in which the program source code is written (Set D), but is not necessarily specific to a programming vocabulary (e.g. the data type *int* is from the general technical word "integer"). However, such terms are known (at least partly) among the CT and CS community, and are thus considered in this categorisation as a subset of Set B.

The examples given in Figure 6.2 may not apply to all contexts; for example, many curricula will specify that "data types" should be taught without being specific about terms such as "int" and "float", but it is possible that a very narrow curriculum specifies a programming language and keywords from that language. The examples in Figure 6.2 are based on a more general curriculum. Also, although the diagram is shown with firm boundaries, there will be grey areas; for example, a term like *decompose* appears in curricula and is used in a technical sense, but is used for purposes other than just teaching programming (e.g. a computer system might be decomposed into subsystems).

The objective of this categorisation is to identify and understand the words and terms that are used in computing context that can unintentionally cause confusion among teachers who are new to computing. All these words, terms or metaphors are either direct or indirect descendants of natural language, which the teachers use to communicate in the class. This short categorisation helps us to realise how even a rather short textual description like Progress Outcomes could contain a large number of words and terms that may go unnoticed by the computing community as jargon, yet can either cause confusion or misunderstandings, possibly leading to failing to communicate its purpose to those who are new or outside to computing context.

It is important to note (as in Figure 6.2) with different contexts for natural language being used in CT education that intersections of these sets of terms can exist. The diversity of the words, metaphors, phrases, etc. in different contexts can cause their meanings to become misunderstood, confusing or ambiguous to the diverse user groups in CS. Therefore, understanding the nature of the commonly recognised difficulties and/or confusions caused by computer jargon becomes important for finding ways to address this issue.

6.7 Results Discussion: Exploring the Difficult Terms and the Impact of the Professional Development Intervention

The pre- and post- responses of the 41 participants in the study described in Section 6.2 – 6.4 are summarised in Table 6.1. The teachers had highlighted 31 distinct terms (16 single words and 15 phrases) as difficult to understand. The responses were grouped to three groups: 1) Marked in pre-survey only (28 terms), 2) marked in both surveys (11 terms), and 3) marked in post survey only (6 terms). The terms that most teachers found difficult to understand at some stage were control structures (57% of teachers), HCI heuristics (50%) and iteration (47%). The terms highlighted only in the pre-survey and not in the post survey indicated that the teachers have resolved many of their difficulties over the course of PD intervention (e.g. comparative and logical operators was mentioned by 39% in pre-survey and 0% in the post-survey). Some terms continued to be mentioned in the post survey indicated that the difficulty remained unresolved (e.g. HCI Heuristics had 27% mentions in both surveys; this is not surprising, since it was not covered in the PD intervention). Six terms were highlighted only in the post-survey, indicating that the teachers realised their lack of understanding of those terms over the course of PD intervention, or it may have been simply because they had not noticed them in the earlier reading (e.g. 5% had highlighted control structures only in the post-survey). With the exception of one case "Determine when to use different types of control structures", no terms were highlighted in the second survey that had not already been highlighted by at least one person in the first one, indicating that their difficulty of understanding revolved

around the same set of terms.

Comments made by the participants were recorded in three categories: 1) comments before the start of the PD intervention, 2) comments after the end of the PD intervention and 3) comments on individual words/phrases. An analysis of the comments is presented in Section 6.7.2.

Table 6.1: Summary of words and terms highlighted by the teachers

		Number of times mentioned		
		Pre-survey	In both	Post-survey
Words (Count = 16)	Algorithms	4	1	-
	Authentic	-	1	-
	Comparative	8	1	-
	Contexts	1	-	-
	Debugging	4	-	-
	Decompose	5	-	-
	End-user	3	-	-
	Heuristics	6	6	-
	Input	2	-	-
	Iteration	11	4	-
	Logical	2	1	-
	Operators	4	-	-
	Output	2	-	-
	Selection	6	-	-
	Sequence	6	-	-
	Variable	7	-	-
Phrases (Count = 15)	Authentic contexts	-	1	-
	Binary digits	4	5	1
	Comparative and logical operators	16	-	-
	Control structures	12	9	2
	Data types	1	-	-
	Decompose into algorithms	3	-	-
	Different data types	3	-	2
	Different types of control structures	1	-	-
	Determine when to use different types of control	-	-	2
	HCI heuritics	9	11	-
	Logical operators	4	4	2
	Organised approach	3	-	-
	Selection using comparative and logical operators	1	-	-
	testing and debugging	3	-	-
	Variables of different data types	2	-	1
Count of words and phrases		28	11	6

6.7.1 Exploring the Difficult Terms

Apart from the words *authentic* and *contexts* and the phrases *authentic contexts* and *organized approach*, all the terms highlighted by teachers had a specific computational context in the given description. The phrase *authentic contexts* appears in every PO, and is really a pedagogical directive rather than a technical concept. A teacher may understand what "authentic context" means in general, but because they may not have experience as a programmer, they may not know what an authentic context is when it is a novice student writing the program. In this sense, they know what the words mean, but they do not know how to teach it without guidance.

Simple Google Search

Considering its popularity as a web search engine, a simple Google search was carried out in an incognito browser with the intention of investigating the level of helpfulness of a web search if a teacher were to try to understand a term from the CTDT PO5 this way. The usefulness of a query result was measured using precision (the term used in Information Retrieval for the percent of results returned that are relevant to the query); this was done for the first page of results only, as this is where teachers are most likely to look. The results are shown in Figure 6.3. Overall, 74% of the terms were directed to a relevant computing or problem-solving related explanation (with a precision varying from 18% to 100%), indicating that help related to understanding the computational meaning of most of the terms can be obtained with a simple Google search. However, it should be noted that these searches may direct users to pages involving further reading of deeper technical content with even more jargon, as well as content irrelevant to the context (where the precision is less than 100%).

The single words that were directed to a computational/ problem-solving definition in the Google search included both very technical terms like *algorithms* (100% precision), *debugging* (100% precision) and *logical* (18% precision), and more general English words terms like *iteration* (100% precision), *operators* (80% precision) and *sequence* (40% precision). However, words like *selection*, *variable* and *decompose*, which are frequently used in computational contexts, did not receive any relevant matches in the first page of the Google search. The search for the word *heuristic* gives a Google definition as a noun that relates to "a heuristic process or method" and provides some first page results directing to pages describing computational/problem-solving processes and methods (20% precision), but does not specifically relate to HCI (which is the intention in the curriculum). Nevertheless, when searched as the phrase *HCI heuristics*, the first page results matched the intended context with 60% precision. Such ambiguities caused by common web searches could cause a novice to become even more confused in their understanding.

Figure 6.3 highlights three categories of terms: 1) those that are less common in gen-

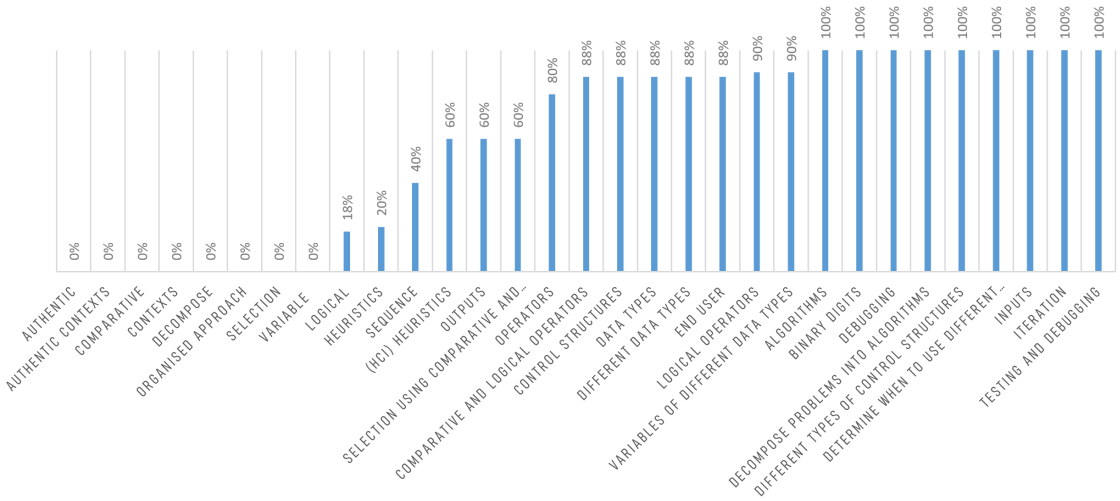

Figure 6.3: Google search first-page precision of curriculum terms

eral use (e.g. *algorithms*, *binary digits*, *debugging*), which have the highest precision; 2) those common in contexts other than computing but are less frequent in general use (e.g. *sequence*, *end user*, *control structure*), which have moderate precision; and 3) terms commonly in general use outside computation (e.g. *selection*, *variables*, *decompose*), which have zero precision. This indicates how the more specific the term is to computation and less it is used in general contexts, the easier related information can be found in a Google search.

When the term "computing" is added at the end of the search string, almost all the terms get computing related results in the first page (with over 75% precision in almost all the cases). Of course, this relies on someone new to the field knowing that this would help, and even though some of them directed to the correct and/or intended meaning of the term, some were directed to pages with resultant content that was related yet confusing to the given context. For example, when *selection* was searched with "computing", the results directed to links containing a range of explanations from the correct meaning for the given context (i.e. fundamental application of IF-ELSE) to selection sort, user interface selection, application of selection in algorithms, etc. When the word *authentic* was searched with "computing", first page results directed to research papers which discuss "authentic computing", as well as authenticity or authentication related to computer network access. As another example, searching for *variables* rather than *variable* produces better results, but this highlights how sensitive this form of information finding is to the users' ability to formulate a suitable query.

Merriam-Webster Dictionary Search

To hold another lens to the language, a dictionary search was conducted to investigate the extent to which a seemingly difficult term is well known outside computing. The inclusion of a word in a well-known and freely available dictionary is used as an indicator of the extent a given term has been adopted in general usage. Merriam-Webster's dictionary can provide multiple definitions for a search of either a single word or multiple of words. Figure 6.4 shows how well the dictionary definitions of terms highlighted by teachers related to computation.

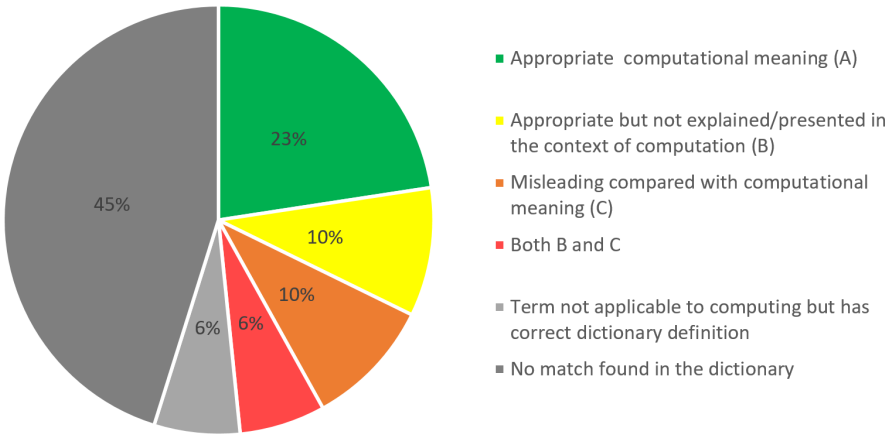

Figure 6.4: Merriam-Webster's Dictionary Definitions relatedness to computation

The longer phrases did not result in any dictionary matches (45% of the terms), but for 55% of the terms, at least one definition was found in the dictionary, with 23% providing a relevant computing definition (e.g. *algorithm*, *debugging*, *binary digits*). About 10% of the terms received a definition that is appropriate to a computational context, but was not presented in any relation to computation (e.g. *comparative*, *logical*, *selection*). A novice to the computational context may find it difficult or impossible to relate such a definition to its computational context. The search provided a definition that is misleading to the computational meaning for 10% of the terms (for example, none of the seven definitions for sequence were related to computing, but had definitions such as "a hymn in irregular meter between the gradual and Gospel in masses for special occasions" and "order of succession").

Most importantly, 6% of the terms (i.e. *heuristics* and *operators*) received definitions that are both appropriate to a computational context, but also presented non-computational definitions that would be misleading in a computational context. For example, the word *operator* receives the definitions 'one that operates a machine or device' and 'something and especially a symbol that denotes or performs a mathematical

or logical operation' in the Merriam-Webster dictionary online search. Such results could cause further confusion leading to misconception among the novice.

The words *authentic* and *contexts* showed the correct natural language definitions, and had a relevant natural language meaning in the given description, but no examples relating to computing. Therefore, the teachers' difficulty in understanding them can only be interpreted as not being able to relate their meaning to the given computational context due to their lack of pedagogical content knowledge in computing, as noted above. Also, although the phrase *testing and debugging* received no success as a singular dictionary search, when searched separately the word *debugging* received the computational definition ("to eliminate errors in or malfunctions of" with a reference to computer programs), whereas the word *testing* had a misleading definition ("requiring maximum effort or ability"). This is an example of a novice search attempt causing further confusion. Moreover, testing and debugging are also distinguishable as skills rather than concepts within the computational context, which may not be an apparent distinction to a novice.

A Dictionary of Computer Science — Oxford Reference Search

This dictionary [32] is a repository of over 6,500 entries related to CS, and is available both online and as a professional publication "revised by a team of specialists", so could be expected to be a reliable reference for computer jargon. This search had 58% success with the terms in providing a computational definition that was correct and related to the context of the given term. However, many of the successful definitions are incomplete in the freely available online version of this dictionary and readers must subscribe to access the full content.

A search for the phrase *testing and debugging* was not successful in this dictionary, however, when searched separately, both *testing* and *debugging* had accurate computational definitions. The phrase *comparative and logical operators* also received no definition when considered as a single term; the dictionary provides the correct computational meaning for both the phrase *logical operators* and the word *operators*, although no definition is available for *comparative operators* or the word *comparative*. This shows that even a dictionary in a computational context can be complicated for a novice user in forming a meaningful understanding of the idea, since slightly different words may be used for the same thing (in this case, the term comparative operators is used for the same meaning in other contexts).

Common Word Lists

The New General Service List (NGSL) [37] is a core of 2,802 high frequency vocabulary words for students of English. Therefore, appearing in NGSL is considered as an indicator that a word is commonly used and should be known to most teachers. About 44% of

the single words highlighted by the teachers as difficult are NGSL listed. For example, words like *input*, *output*, *sequence*, *selection*, and *variable* are listed here. However, it is worth mentioning that the words like *variable* are likely to have been included in this list due to their meaning in the context of maths, and *sequence* due to planning, but not specifically due to their computational meanings. Therefore, this indicates the possibility that the application of the word in a more specialized context is making things difficult to understand regardless of whether or not their meaning is known.

The New Academic Word List (NAWL) [36] is a list of 963 words which frequently appear in academic texts, but which are not contained in the NGSL. A listing in NAWL is considered as an indicator for a term being known among the academic community. About 19% of the words are NAWL listed. Words like *algorithm*, *comparative*, and *logical* are listed in NAWL and not in NGSL, indicating that such words are used in a more academic context than general. However, altogether 62.5% of the words are listed either in NGSL or NAWL, indicating that what would be commonly used words in educational contexts are seen as difficult by teachers in a computing context. Moreover, since NGSL does not define words, they may be there because they are commonly used with an irrelevant meaning.

6.7.2 Comments made by the Participants

Prior to the PD intervention, the participants had made comments like "*I either don't know the word or can't understand the meaning in the context*" or "*I understand the terms and what they mean but not how to do it*", which aligns with the observation that most of the words are in the NGSL or NAWL, indicating that they either do not know the computational meanings of the terms, do not understand the computational context they are applied, or do not possess the related skills. Comments like "*I know the word, but I am unsure how or what that may look like for learners*" indicate their inexperience with teaching the content (pedagogical knowledge) rather than purely an issue with their technical knowledge.

Analysis of the post intervention comments addresses Q 2, that the PD intervention has been successful in clarifying many of the difficulties (e.g. "*I could explain these concepts to someone else a lot more clearly now*", "*Makes much more sense now*", "*have a basic understanding of all the terms I circled earlier. Able to now know a little about each*"). However, comments like "*Some became clearer but as a result others became less clearer. Love to do another course*" indicates that the PD intervention has paved the way to new learning as well as new realisations for the participants about knowing what they don't know!

Some comments made on individual words and phrases clearly show the novices' confusion in understanding the computational meaning and/or the context, when their

prior knowledge overshadows the computational context they are referring to. A comment on the word decompose as "*Know as rotting flesh??*" and the phrase binary digits as "*0 & 1?*" are good examples of such situations. The comments made on the word algorithm as "*Kinda know but not really. Process?*" or the phrase "*decompose problem into algorithms*" as "*Methods?*" indicates confusions that could possibly lead to misconceptions in a classroom.

6.8 Conclusions

This study attempts to understand the nature of teachers' understanding of the computational terms related to CT concepts by studying their responses to a related computational text (a progress outcome description from a curriculum). The findings show that: 1) the teachers find the vocabulary used in computational contexts difficult to understand even if the meanings of the words and phrases used in them are known to them from other contexts; 2) over 80% of the terms that they did not understand had been resolved after a 3-day PD intervention related to CT/CS that was not targeted at vocabulary, so proper interventions can resolve language difficulties in computational contexts; and 3) a simple internet search usually provides directions to related computational/problem solving definitions, yet may cause further confusion in the absence of prior contextual knowledge.

The nature of the teachers' understanding of computational terms can be either: 1) the computational meaning is not known; 2) the term is unclear in a computational context; or 3) their applicability of the term in the classroom is unclear. In such situations, knowing the meaning of a term within the computational context alone may not be sufficient, and an operational definition is needed to put them into context. For this reason, rather than a dictionary or glossary, teachers are more likely to benefit from a book or other substantial explanation of the ideas in computing (and therefore the terminology) such as a glossary not of definitions but of explanations of computational terms in the curriculum. Computer jargon for which the computational meanings are more commonly adopted in general usage (e.g. *algorithm*, *binary digit*) raise less confusion than the terms where their computational meaning is less commonly known (e.g. *selection*, *decompose*). Inability to apply the prior known meanings of words when they are presented in a computational context, with sometimes specific computational meaning, can cause confusion and misunderstandings among teachers. Such confusion could possibly lead to misconceptions in classroom interactions.

The following suggestions may be used to address the issues encountered in this study, and further experiments could be conducted to determine how well they work.

1. Provide explicit guidance to teachers on how to find relevant information online

(including focusing on sites that are more relevant to the curriculum, and giving hints on how to craft searches that are more targeted)

2. Be explicit about mentioning common misconceptions about important words in computing
3. Deliberately use examples of terminology, linking the general idea that they may see in a curriculum (e.g. selection) with how they will encounter it in practice (e.g. if statement), making this connection multiple times to reinforce the general concept and the implementation
4. Encourage teachers to participate in a relevant community of practice (such as a teachers' association or online group) for support.

The claim that teachers find computer jargon difficult to understand would appear to be more of an issue of applying the jargon in a computational context. Therefore, appropriate support using the language in context may enable teachers to develop confidence in their understanding of the terminology. If one is learning a natural language, living in the culture can give much deeper understanding of the language, and the same is true for computing: memorising definitions is shallow, but using the terminology in the context of doing computing gives it depth.

Chapter VII

Computational Thinking and Notional Machine: The Missing Link

The Notional Machine (NM) is a conceptual computer created by teachers to facilitate learners' understanding of hidden aspects about computers and programs at run-time. With the evident success of unplugged activities in delivering Computational Thinking (CT) content as well as the success that has been indicated when combining them with teaching to program [115, 118, 194], the relationship between unplugged activities and teaching and learning CT and programming raises the possibility of a close connection with NMs. Given the physical nature of unplugged computing activities and their distance from the actual device, this relationship seems to lean towards developing a mental model among beginner learners, during their attempts to understand computing concepts. This chapter discusses the implications of two representational concepts that are commonly used in computing and programming education to support learners in developing mental models and conceptualisation, namely the Notional Machine (Section 4.4.1) and the Computational Agent in CT (Section 4.5.2), to explore how CT relates to mental models that students are forming.

We explore how unplugged activities seem to have a close connection with NMs, and the lens of NMs is used to give a closer look at the complementary relationship between unplugged and programming. Reviewing unplugged activities through this lens, the programming concepts where unplugged has been successful (or failed) for modelling, and the reasons behind them, can be explained. The discussion in this chapter provides a deep insight to the relationship between CT and NM in supporting learners' mental model development, a meaningful structure to the varying nature of the view of CT at different learning stages and thereby, a more practical perspective on using unplugged activities in learning to program (supporting research question RQ 1 which explores how CT relates to the mental models that students are forming, and its sub-questions as given in Section 1.3).

7.1 Background

In learning to program and understanding how a programming language controls a computer, learners develop both insights and misconceptions whilst their mental models are gradually refined. It is important that the learner is able to distinguish the different

elements and roles of a computer (e.g. compiler, interpreter, memory, etc.), which novice programmers may find difficult to comprehend. One of the potential sources of difficulty inextricably linked to mastering computing concepts and processes, and in learning computer programming, is forming accurate mental models. Some form of representation (e.g. an abstract machine or a computational agent) to support technical or pedagogic explanations is common in computing.

Two representational concepts that are commonly used in computing and programming education to support learners in developing mental models and conceptualisation are: 1) the Notional Machine, and 2) the Computational Agent in CT (detailed in Sections 4.4.1 and 4.5.2 in Chapter 4 respectively). On one hand, CT, a concept that reflects the need for problem solving by designing and analysing algorithmic processes for computational systems, often refers to an "information-processing agent" or a "Computational Agent" that performs the computation. On the other hand, Du Boulay et al.'s NM, which was first introduced in [75] as "an abstract computer responsible for executing programs of a particular kind" [215], is used as a pedagogical device to explain the hidden aspects of computing. Both these concepts attempt to support the learner with a representational and interactive perspective to support their conceptualisation and mental model development. Often in programming education, both these idealisations are facilitated either on a physical computer (e.g. visualisation software) in a plugged-in learning environment, or a deterministic device (e.g. a Beebot), or with the aid of fellow learners in unplugged learning.

Learners develop understandings of how the programming language can control a computer when they learn to program. When new knowledge or new understandings occur, some form of a mental model is inherently formed in a learner and is mentally interacted with, sometimes without any reference to a real machine. In the process, both insights and misconceptions can arise as the learner's mental model of how the system works is gradually refined. Education regularly uses simplified models (e.g. the Rutherford-Bohr model of the atom), analogies and representations to help the learner to develop good mental models, and this is inevitably necessary when teaching computing as well.

In teaching programming, the NM is not necessarily taught or introduced to learners explicitly. Rather, it often stays as an abstract model of a computer created by the teacher in the context of teaching. It is used as a pedagogical device that helps the learners to understand, which represents something that learners can interact with (often mentally), while teachers use analogies that provide scaffolding to help understanding [89]. In this context, when introducing students to computing via programming, research attention to NMs becomes essential, whether it is implicit or explicit in the teachers' or learners' thinking, especially when engaging them with the run-time dynamics of computer programs. Similarly, CT, which is a concept adopted by many school curricula around the

world, uses the CA to perform computation, which is a role that relates to a NM. This raises the question of the relationship between the CA and the NM that is worth a closer investigation.

Computational problem solving involves exercising CT to transfer ideas into the design and analysis of algorithmic processes for computational systems, and/or turn them into computer programs. These thought processes can start from early childhood, and mental models can form with or without the presence of an actual computer or the aid of representational tools like CAs or NMs. Ada Lovelace (building on the works of Charles Babbage) wrote some of the first "programs" (trace tables) for the Analytical Engine without ever seeing it built. Was she thinking computationally? What was the CA in her CT? What was her mental model like and how did she interact with it? Did Lovelace have a NM? Is anyone who develops a new programming language likely to have written programs in that language before they write the compiler? And in the days of punch cards and limited computing access, students frequently wrote programs days in advance of them being run, although if the Lovelace story was true, Babbage and Lovelace were decades in advance! Dijkstra advocated that the ability to prove the correctness of a program is more straightforward than testing it as an implementation (for a reasonable number of test cases) [72], which is an extreme example of developing a NM without having the CA physically available! Bower and Falkner [29] found that accurate NMs underpin successful performance in CT and suggest understanding NMs as a prerequisite for effective teaching of computing.

Considering the notable reference to CT in school curricula, understanding how CT relates to the mental models that students are forming can become helpful for teachers both in teaching CT concepts and in improving CT skills in students effectively, particularly in programming education. Investigating the relationship between CT concepts and NMs can give broad insights into how they relate to and complement each other in the teaching and learning of introductory programming. This is particularly the case at grade school[1], since the students' programming skills are more likely to be at an early stage. The inclusion of CT with a focus on its original definition (see Chapter 2, Section 2.3) can hide the importance of the NM in the curriculum context, partly because the CA is generally not well defined (particularly since there has been debate about what the agent can be [56]), and also because the link has rarely been made explicitly. Reflecting on how the two concepts CT and NM have been often referred to in various contexts of learning computing as well as programming, CAs and NMs can be seen to have been used by educators differently at different stages of learning, in changing roles and/or in evolving contexts.

[1] By grade school we refer to the elementary, middle and high school levels, also known as K-12, or primary and secondary school.

This chapter looks at the distinctions between CAs and NMs in computing, and provides insights to how those distinctions may and may not be helpful in the teaching and learning of introductory programming, while providing perspectives on answers to RQ1.1 and RQ1.2.

7.2 Computational Thinking Extended

Wing brought *Computational Thinking* to the fore in her 2006 article [248], and later gave a clarified the definition in [249] connecting a person's thought processes in problem solving with the need and possibility of the solution being carried out with the aid of an "information processing-agent", which was strongly implied to be the computer (also detailed in Section 2.3). She suggests that it is a way that humans can think about solving problems, incorporating the set of mental tools used in computer science. Referring to this early definition, CT is explained in much simpler terms in the literature as "thinking like a computer scientist to solve problems" by Wing, as well as others later [49, 248]. Denning and Tedre [64] define CT as the mental skills and practices used for designing computations that get computers to do things, and "for explaining and interpreting the world as a complex of information processes". This raises the questions: What is this special way a computer scientist thinks to solve a problem? How does their way of thinking differ from others? Curzon et al. [56] survey many definitions and discussions about CT, to find the common themes. They suggest that CT is about developing systems that involve information processing, and it is the focus on algorithmic solutions based on computation that differentiates it from other problem-solving approaches. Denning and Tedre [64] point out that CT for a beginner differs from that for a professional: the former is a simple, practical understanding of computing concepts and the latter is critical, complex and technical.

Figure 7.1 collects some common views of CT and its relationship to the actual physical machine (i.e. computer) [56]. The views towards the left of the spectrum indicate a stronger lean to the physical device in explaining the concept, whereas towards the right, the views relate less to the device but concentrate more towards mental comprehension of computation or relating computation to more generic concepts. In views where CT is seen as leaning extremely towards computation (extreme left of the spectrum), the CA is necessarily a computer and therefore precision of instructions is essential. In an extreme generic view of CT (extreme right), a human with a set of loose instructions acting based on their judgment can also be allowed as a CA (e.g. a cook following a cookbook recipe). However, the CT's applicability to computation in such a generic extreme is vague and doubtful, because computation (at present) is digital, and relies on numeric calculations and symbol manipulation. The dotted lines indicate the positioning of different views in this spectrum but are not firm lines in the continuum.

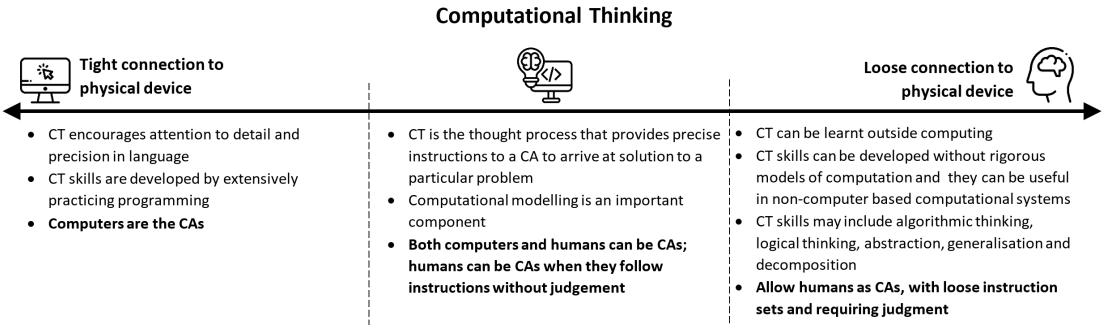

Figure 7.1: Variations of CT discussion in relation to the physical computer

Many situations that involve objective steps in which humans follow a sequential flow can be misunderstood as CT (e.g. a well organized routine daily work, tidying up a closet, stacking books by size, etc.). However, only the portion of thought processes in rational thinking with a goal of producing an algorithm or a computer program/computing system should be seen as 'Computational Thinking'. Such thought processes are often seen as closely related to 'problem solving'; they can be defined rather clearly when they are expressed in relation to information processing, and can be easily implemented using a computer if they are expressed in the form of a set of instructions to a 'computational agent' for processing. As such, CA is more strongly related to computation, programming and to the physical device. This thesis' interest in CT leans towards the this interpretation, where the typical goal is to produce an algorithm or a computer program, rather than the broader and generic perspective that is sometimes used.

Curzon et al. show that there is a common view to CT, as a way of thinking that is used to develop solutions in a form that ultimately allows 'information processing' or 'computational' agents to execute those solutions [56]. Views on CT can differ on what the CA can be, but there is a common ground that it is not simply solving a problem to provide a solution, but solving it in a way that directs an 'agent' to arrive at the solution by following instructions. This view implies that the problem solver need not be the executor of the solution themselves, but positions them to look at the problem scenario from an external perspective in which they can oversee how a CA (often times, the computer) should be instructed to solve it.

As shown in Figure 7.1, a CA may or may not be tightly coupled to a computer, depending on how they are used in a learning context (i.e. Computer CA, a semi-deterministic device or a human). A CA enables novice programming learners to primarily realise that the CA will merely be the device that does the job efficiently, and not think for itself or on the programmer's behalf. Denning and Tedre [64] point out that the efficiency is a key feature of computers that makes them the obvious CA to use: "The magic [of computers] is nothing more than a machine executing large numbers of very

simple computations very fast". Aho [2] points out that an important part of this process is finding appropriate models of computation with which to formulate the problem and derive its solutions. The capability of the CA is a key consideration when discussing Computational Thinking!

The moderate position for a CA that essentially encompasses all the definitions, different views and purposes of CT, as indicated in the survey done by Curzon et al. [56], is that it is an information processing agent that can follow instructions precisely and blindly without using its own judgement. By CA in the remaining discussions, we refer to this definition of a CA, since our focus on CA in CT is as an element that teachers can relate to in the representation of ideas, for pedagogical purposes.

7.3 Computational Agents: The Missing Link

Writing and reading programs requires mental representations of problem-solving strategies, familiarity with common programming patterns and the problem domain, and a model of the algorithm being implemented and the computer that executes the programs [50, 216]. The broader lens of CT is a good way to approach computer programming, as it can prevent a beginner programmer from perceiving learning to program merely as learning to write program text according to the syntactic rules of a particular programming language. It encourages the beginner programmer to develop an overarching view of problem solving, and to develop a step-wise solution that can later result in a computer program. Conversely, programming a computer enforces the limits of computation, so not only is teaching CT a good foundation for teaching programming, but learning to program is a key way to engage students with CT. Therefore, it is not surprising that teaching programming is a widespread methodology for teaching CT. Computer programming relies directly on skills to develop systems involving information processing, with the need to focus on algorithmic solutions, and for that reason it differentiates itself from other problem-solving approaches, and resonates with CT [56].

When learning to program, it is crucial to distinguish between the different roles a computer can play [75] (i.e. the machine that executes the program, the development environment that manages reading, storing and editing the program, and the programming language itself). This can be difficult for the novice to understand, because a language itself is implemented as a computer program. Furthermore, the computer's nature of strictly following instructions can be overshadowed by the expectation that it will do what is expected rather than what it is told [177]. The need to have a "model of the computer as it relates to executing programs" [28, 29] is equally important in both teaching and learning to program. This modeling — in other words understanding a NM — is one of the inextricably linked potential sources of difficulty when mastering computing concepts and processes, and in learning computer programming [28, 29].

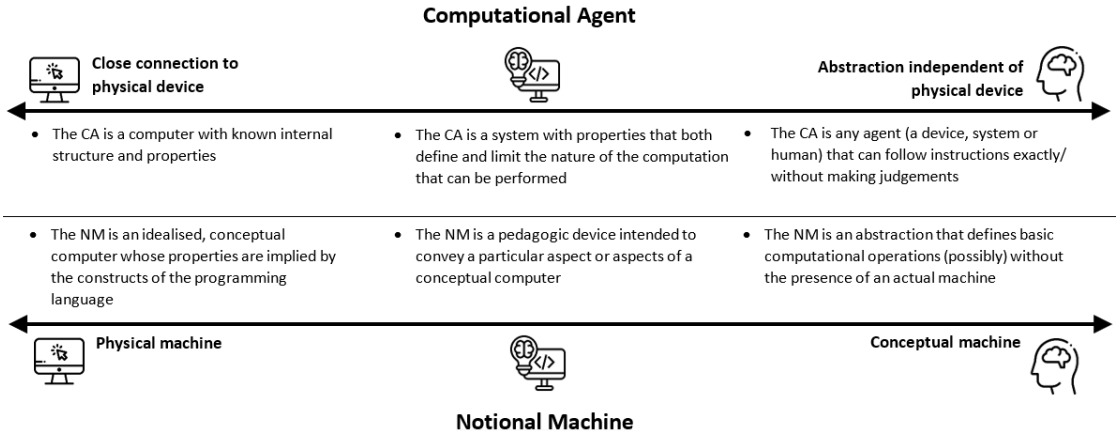

Figure 7.2: Wider range of abstraction in NMs and a CA's position as an aid in relating them to mental models

CT, by definition, helps learners to realise this distinction by introducing the CA (human or a machine) as an 'executor' that precisely and blindly follows instructions, thereby diminishing any particular creative capability given to the computer itself. The 'glass box' nature of the CA (human or computer), that is, that the precise set of instructions is available and known, can be useful in explaining abstract concepts of computing. This enables educators to offer clearer NMs, as well as enabling learners to form much closer mental models. However, a NM for early teaching may be grossly simplified and the full model hidden from the learner. This may cause some teachers to feel uneasy, as if they are teaching something that is incorrect. Nevertheless, incompleteness should not be confused with incorrectness [67]. A similar point is captured in Pratchett et al.'s "Lies-to-children" [183], who state that we need to start with simpler models to avoid overwhelming learners, and this incompleteness can appear to be incorrectness. Education regularly uses simplified models, such as the Rutherford-Bohr model of the atom (rather than quantum mechanics) or Newtonian physics (ignoring relativity), and this is inevitably necessary when teaching CT as well.

Figure 7.2 provides a comparison view of a CA and NM, and how they fall within a spectrum between the physical device and conceptualisation (mental modeling). As discussed in the previous section, CAs are the key element in CT definitions that link these two extremes. Various definitions of NMs on the other hand, range across a similar spectrum as explained in Section 4.4.1. NMs as pedagogical devices attempt to convey operational aspects of computing to the learner (and relate that to their mental model) in a quite similar manner that a CA helps them to design and execute 'sets of instructions' that should solve problems using computation. They both succeed in 'representation'; the NM is often less obvious (as well as not discussed explicitly) in teaching; the CA is

far more tangible and easy to use as a teaching aid/tool.

Reflecting on how Lovelace arranged her thoughts to predict the functioning of a machine she had never seen, or how the early 'Human Computers'[2] wrote instructions for a physical computer [210], it becomes clear that Computational Thinking happens even when the CA is imaginary. In the two examples of possible NMs for quantum computing and AI discussed earlier in Section 4.4.1, the CA is well-defined, a $NM_{conceptual}$ can be used to learn about it, and yet the learner may never get to see the CA in action. Consequently, a $NM_{conceptual}$ with a high level of abstraction may also be possible to describe using a non-quantum computing or a non-AI CA, thereby making it mentally comprehensible to a learner. The unifying thread here is that, in learning to program, the roles of the CA and NM change at different stages of development (of the learner and of the curriculum), yet are very closely related. This also rationalises their variation in positioning themselves in a spectrum as shown in Figure 7.2.

CT involves understanding/using algorithms. A typical definition of an algorithm is "a finite sequence of rigorous well-defined instructions, typically used to solve a class of specific problems or to perform a computation" [6], and most definitions of an algorithms specify a combination of ambiguous and executable steps (or instructions). To use a CA, one should understand the sort of computation it can do, which in other words is understanding its NM. The criterion of a "finite sequence of rigorous well-defined instructions" means that they are relative to the processing agent (i.e.the CA); the instructions that a child can execute are different from the instructions a computer can execute, and these are different from a quantum computer. Writing algorithms for a processor is writing them for a NM. Thus, every CA necessarily implies a NM. As such, a CA and NM are inseparably linked by definition.

Considering the range of definitions available for both CA and NM, a single and simple definition to explain the link between the two concepts is difficult. Exploring the continuum to understand the relationship between them, the following relationships between them will be used.

Relationship 1: A CA in CT can be seen as a constantly developing, simplified variant of a NM that provides an observational (external) perspective as well as operational (internal) perspective to the learner to support them, mostly at the initial stages of learning and problem solving, to form a robust connection between the NM and the learner's mental model more efficiently and effectively.

This relationship is explored in the discussions about the Kidbot unplugged activity and its Scratch implementation in Examples 1 and 2 in Section 7.4.

[2] The story 'Hidden Figures' by Margot Lee Shetterly explains how the women who were working as 'Human Computers' equipped to write programs for the new electronic computer.

Relationship 2: A CA and NM are thematically linked and can simultaneously exist and have different referents in the same learning context. A CA and a NM both play similar pedagogical roles but may have different properties at different stages of learning; a CA can substitute as a NM in the early stages of learning where an explicit NM is not needed.

Using PythonTutor as a CA and its connections to the $\text{NM}_{physical}$ will be used to discuss this relationship in Example 2 of Sections 7.4.

Relationship 3: The terms NM and CA can sometimes be used interchangeably, and in an extreme case they can be seen as synonymous. Nevertheless, in some situations, the terms NM and CA have the same referent but may suggest a different context of use.

The pedagogic role and limitations between a human CA and computer CA can be used to understand this relationship, which is explored using flowcharts in Example 3 of Section 7.4.

We recognise the delicate nature of both the NM and CA in supporting abstraction, through their range of definitions that coincide with one another more often. Accordingly, the three examples we have used in Sections 7.4 can be used to understand all three views of the relationships, and not limiting to the one they are specifically mentioned.

Revisiting the different definitions of a NM in Section 4.4.1, the general understanding in this discussion is that a NM can be used to correctly predict the outcome of a given program (and does not need to accurately represent what is happening with the physical computer). A NM that leans to the far left of the continuum in Figure 7.2 encapsulates all observable behaviour of the real machine, and supports accurately predicting the behaviour of the real machine ($\text{NM}_{physical}$). A CA of this end is a physical computer running a specific language (e.g. as can be seen using PythonTutor). A NM that attempts to relate to a person's mental modelling about computing, and not limiting itself to a specific programming language or a physical device leans to the far right of the continuum ($\text{NM}_{conceptual}$). Any CA whose computational ability is understood by or explained to the learner can become the CA that connects their mental model to a NM in this context, because at this stage, both the NM and CA are abstract. A NM that is implied by both the programming language and the paradigm used in learning resonates with the mid region of the continuum in Figure 7.2 ($\text{NM}_{generic}$). Use of a Kidbot unplugged activity (thereby using a human CA, the bot) and extending to implement using Scratch (thereby use the computer CA) to help learners build a better mental model of a NM essentially locates the concept of a CA in this mid region. A teacher may or may not use both CAs to convey the idea successfully, depending on the learner audience.

In early introductory programming, it is easier to establish the NM in a learner by trying to establish the fact that they program for a CA to execute the instructions given in their program. Wing's definition of CT (see Section 2.3 in page2.3) encourages the CA to be situated in a more abstract as well as simpler position than a well-defined NM, simple enough to be understood by the learner yet with sufficiently detailed instructions to draw the learner's attention to things that are crucial yet invisible in the program. However, the behavioral limitations of the CA in CT obscures this CA's relatedness to a NM. Had the CT literature adopted NMs, it would have had more clarity about the ultimate goal of CT, and added more richness and clarity to what computation is. Nevertheless, a CA as explained by Wing's CT definition facilitates the learning experience by positioning itself as a learning tool that learners can relate to that makes abstract ideas concrete for them by being more representative and reachable than a NM (Figure 7.2).

The CA provides a sense of 'tangibleness' to a learner's mental model that should comply with a NM, which is intended to be established cognitively, leading to a more relatable conceptual relationship between the two. According to Sorva's [215] view of several NMs existing at different levels of abstraction in programming, a suitable CA at different stages of learning that is capable of communicating the idea can provide a very useful link that meaningfully connects a learner's mental model and NM. CAs are simplified NMs, in that they are NMs made accessible to novice learners (as appropriate for teaching CT). Teaching programming should make use of the idea of a CA as a very useful link to connect a learner's mental model to the full NM.

7.4 Computational Thinking's Effect in a Learner's Mental Model Development

As a discipline that often times involves intangible products and an intangible design, and development processes that employ cognitive means in its life-cycle, studying how experts as well as learners conceptualise ideas and develop mental models becomes an important aspect in computing. Particularly, since CT is a key focus in computing education, it is worthwhile to look at what models are at work when students are learning CT. Further, computer programming is an integral part of the progression of CT over the life-cycle of a computing learner, and therefore understanding how CT can help learners developing good NMs could also provide useful insights into computing education.

Having discussed the close relationship between the NM and CA in CT in Section 7.3, to elucidate the idea that a CA could be the link that helps learners to connect their mental models to good NMs, this section explores the role of a CA and its relationship to the NM using three examples 1) an unplugged style programming activity that uses a human as a CA; 2) two different perspectives of a computer as a CA: using the Scratch programming environment and a PythonTutor simulation tool; and 3) a comparison

of two flowcharts from the perspective of a CA (human and computer). In all three examples, the CA of interest is seen in the perspective of the CT definition and discussed alongside a NM that the program(s) in each example intends to establish. The different distinct definitions of NMs are used at points where such a distinction is essential. These examples explore the relationship of the CA and NM, which often times can be seen as complementary alternatives at different stages of CT and learning to program.

7.4.1 Example 1: An unplugged Style Programming Activity that Uses a Human as a Computational Agent

Unplugged activities are popular with learners and teachers as a pedagogical approach to introductory computer programming, and used appropriately, they can help improve students' self-efficacy without adding to the learning time [115, 158, 203]. Designed to facilitate kinaesthetic learning, unplugged style activities (e.g. CS Unplugged [3], Code.org [4]) are non-computer based and have consistently been suggested as a successful method for teaching CT and introducing programming concepts. unplugged style learning can be seen as a useful teaching tool that provides a representation to inform a learner's mental model, which can later align with a NM, that is possible to be interacted with physically.

Kidbots [124] is an unplugged programming learning activity that involves three students: a 'developer', 'tester', and a 'bot' (see Figure 7.3. The developer has to write a program using only three instructions (move forward: F, turn left: L and turn right: R) to guide the bot from location A to B on a grid (typically objects with a story behind them, such as getting a soft toy to a desired destination). In the classroom setting, this grid can be drawn on the ground, and the bot physically moves from one square to another adjacent one.

The concept that the Kidbot activity focuses on is *sequence*. A key element is that the bot is not allowed to move until all the instructions are written in the program; this simulates writing a program and then testing it, and forces the programmer to reason about the instructions in advance. The intelligence of the student who is acting as the bot is not considered to be an essential part of the CA they represent, much like the understanding (or otherwise) of the human who follows instructions Searle's Chinese Room argument (and the subsequent debate) [43]. Once the program is written, the tester then reads out the instructions; having a tester prevents the programmer from making corrections on the fly while the program is executing. The bot has to follow the instructions read out by the tester precisely and blindly to move around the grid.

Figure 7.3 shows a Kidbot layout with students playing the roles of programmer, bot and tester, and a toy pineapple as the target. The bot at location A moves around

[3] CS Unplugged: https://csunplugged.org

[4] Code.org: https://code.org/

the grid with the aim of getting to the target at location B. Figure 7.4 shows a set of Kidbot instructions that could move the bot to reach the target. In the Kidbot unplugged activity, the bot is a human CA that implements a kind of a NM that explains certain behaviours of the real machine using a certain set of program constructs (i.e. F, L, R). The CA also helps stimulating the learner's mental model and connecting that mental model to the behaviours of an actual computer fairly easily. The CA thereby brings a tangible representation to a NM.

The programming language used to program the bot consists of just three simple instructions. The bot is always initially placed on location A on the grid facing in a particular direction, and so has a predictable initial state. The program instructions are written based on the bot's initial state. When the bot starts to blindly follow the instructions in the program, the CA's behaviour becomes visible to the learner, along with the awareness of the nature of how an actual computer would only be executing a program code (follow instructions) but not doing any thinking for the programmer.

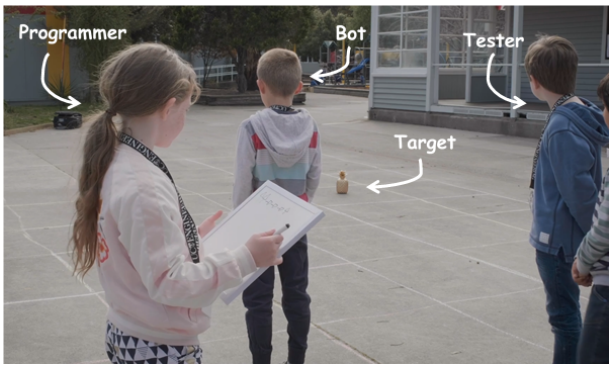

Figure 7.3: A Kidbot activity with kids

Figure 7.4: A set of instructions to Kidbot

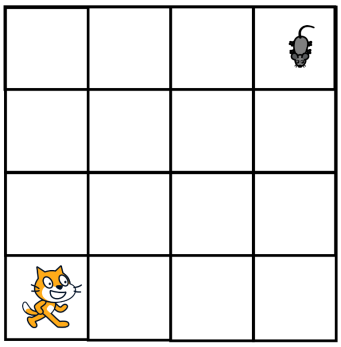

Figure 7.5: A Kidbot layout

The Kidbot NM is very small and can be fully understood; it has a very definite behaviour: a starting position and just three instructions that are executed in a sequence. The behaviour is typically taught by example: the bot (i.e. the CA) is asked to demonstrate the language by moving forward or turning. It is very typical for the first attempt to make the mistake of thinking that the "L" command moves the bot one square to the left instead of turning in the same square — for example, in Figure 7.5, the instruction to get the cat to the target might then be FFFLLL. This results in the bot spinning on the spot with the LLL; the bot has previously demonstrated what L does, so they are likely to have an accurate mental model of a NM (although a teacher may need to remind them), and the mistake becomes obvious to the programmer, which enables them to modify their own mental model.

Of course, an exact definition of the behaviours is not trivial; cases such as reaching the edge of the grid, and defining the starting state may be implied to avoid having to give a detailed specification. Students themselves may even be called upon to decide on what happens in these cases, which gives them a hand in specifying the behaviour of the CA. Nevertheless, with just a couple of sentences of instructions and some common sense, learners can fully understand the behaviour of this CA. Developing the (Kidbot) NM happens partly through the teacher's specifications about the CA, and partly through experience and having students make decisions about such details in a constructivist fashion, which supports them to fully understand the NM.

Having no direct connection to a physical computer, the Kidbot NM largely deviates from a $\text{NM}_{physical}$, but it also encompasses most of the attributes of one (such as states and rules). The conceptual understanding it represents also resonates with $\text{NM}_{conceptual}$. In the computer-based context, a NM is highly conceptual and difficult to comprehend for a learner, but this activity shows how a CA can be something a learner can relate to (i.e. a computer or human, in this case the bot), as directed by the teacher. Such understandings would be reflected either in the learner's program instructions (e.g. us-

ing turn instructions to adjust the bot's directions) or in their discussions (e.g. verbal instructions to the bot for its initial positioning). Either way, it encourages establishing a good mental model in the learner about the concept that the learning activity intends to communicate, as well as the behavior of the CA aligning well with a NM.

The system is simple enough that a teacher can diagnose misconceptions through simple tests (e.g. various sequences that use all three instructions). For example, the common initial misconception that "RRR" moves three squares to the right can be diagnosed by asking the learner where such a sequence would take the bot, without physically moving it. Moreover, the learner can also self-assess their mental model by simply becoming the bot to execute their own program, or imitating the bot's movement themselves while programming. The activity also provides opportunities for the students to use the language around programming (such as testing, bugs, and debugging), as well as understanding the nature of the task (testing and modifying programs to achieve a goal), the idea that there is more than one correct program to achieve a task, and (in an extended version) that it is possible to achieve the same outcomes with only the F and L commands, introducing them to the notion of a complete language.

The Kidbot programming language is not Turing-complete, yet with the aid of the limited instruction set, the CA (i.e. bot) enabling learners to encounter several basic programming concepts, thereby facilitating them to form clearer mental models, and develop language and attitudes that apply to programming a computer. With the aid of the CA they form a mental model, which may or may not have misconceptions in it depending on how well the teacher communicates the NM behind it (which is still abstract to a great degree, yet could be represented on an actual machine).

A NM in an actual programming exercise that attempts to implement the movement of the bot in Kidbots activity is much more complex than the simple 'F,L,R' instructions the activity employs (discussed in detail in Section 7.4.2). Although the programming language defined by the activity uses only three simple instructions, the effect of the next instruction is highly dependant on the previous state (position) of the bot. In a computer implementation of Kidbots, written using a particular programming language, the states could be defined more precisely — perhaps based on Cartesian coordinates, and mathematical definitions of the change of state (position) based on the current state and the command given, where the current state involves the direction the bot is facing and a location. The programming language employed may use different symbols than the simple 'F,L,R'. This newer version of the 'language' will reflect an improved version of a NM than the initial, very simple version. For example, moving forward involves adding one step to the previous location, and turning changes the current direction the bot is pointing. These in turn needs to include special cases for being at the edge of the grid. When doing the Kidbot activity however, there is no need for such detail to establish the simple Kidbot NM programming. Since the CA is physically visible, the learners become

aware of these simply through experience (using their common sense), improving their mental model along with it. Moreover, the physicality of the CA helps the teacher to visualise the learner's mental model as well as diagnosing misconceptions more easily.

Interestingly, if the developer and bot are both wrong in the same way, then the program is likely to work. For example, both the developer and the bot might assume that the R command moves one square to the right instead of turning. This issue is associated only with human bots, but still highlights the result of the programmer's mental model of the NM being the same as the actual NM of the CA. If a similar activity was done using a physical device such as a Beebot (which has the same commands), the problem of programmer's mental model not aligning with the bot's NM would not arise, as the learner is forced to use the $\text{NM}_{physical}$ that the physical device implements.

The bot in this discussion is a good example of a pedagogical representation of a NM that a teacher may use without having to explicitly mention it. The physical presence of the bot (or any CA in an unplugged style example) facilitates the teacher to provide conceptual explanations to and detect situations that could otherwise be left only for the mental comprehension by the learner. It shows how a CA can provide a sort of physical, tangible sense to a pedagogic NM that a teacher intends to establish in the learner, as well as how a learner could use it to build their own mental model. At the earliest stages of learning computing, where the learners are mostly expected to develop CT skills rather than learning to program, a physical CA such as a Kidbot can still provide sufficient conceptual representation for the learner to develop a good initial mental model, with or without the need for an explicit NM even in its simplest form. The NM is given in the explanation of the semantics of the instruction set (i.e. what F, L and R mean), but it is very simple, and does not need to be explicit, and can be defined by just a few quick and obvious examples that draw on real life experience.

7.4.2 Example 2: Different Perspectives of a Computer as a Computational Agent

Computers are used as an educational tool in computing education, particularly in programming education, in various ways. An integrated development environment (IDE) is a key tool to allow students to write programs, but there are also applications such as algorithm visualization tools [217] and program visualisation tools [209]. These technologies typically use automated visualisation as their main feature, and sometimes use familiar (and therefore intuitive) graphics and animation. According to Sorva [216, 217], visualisation software can facilitate effective learning by helping the learner with a visualization of computer memory, and not just by showing the steps of a program. The example in this section looks at the role of a computer as a CA and its implications for

mental model development and NMs, using: 1) Scratch [5] and 2) PythonTutor [6].

Scratch is a popular programming language in introductory programming courses, specially for children. A simple Scratch program can work in a similar way to the Kidbots example (reaching a target on a grid), however, it needs the programmer to pay attention to much more detail even in its simplest form. In Scratch programming, the CA is represented by the Sprite (the cat figure) moving in the visual area dedicated to displaying the program output on a computer (Figure 7.5). Actually, the Sprite is a visual representation of the computer as a CA. The programmer has to give instructions (i.e. a program) to the Sprite for it to move around to reach the target.

Although the Scratch programming language is Turing-complete, at this stage, for the learner this is not important, and the programming experience helps them develop mental models quite similar to the Kidbots example in Section 7.4.1. The programmer has to choose from many Scratch commands, to give precise instructions to the Sprite. Although this is similar to the objective of the Kidbots activity, moving the Sprite around the visual area needs a little more complicated thinking before programming. Figures 7.6, 7.7 and 7.8 show three Scratch programs that can achieve a similar result as the one shown in Figure 7.5. Moving the Sprite involves changing its position using move instructions with a number of steps, and turning instructions in an angle prescribed by the programmer.

In both the Kidbot activity and Scratch example, the direction of movements is tangible, but unlike the Kidbot example, the Scratch program requires a degree of awareness of the scale of a Sprite's movement (essentially measured in near-invisible pixels) on the visual area of a less tangible visual environment. This is a good indicator for the learner to realise the limitations of a computer, i.e. the fact that the computer only merely follows instructions precisely and blindly, and does not think for itself or on the programmer's behalf.

Figure 7.6 shows the simplest set of instructions to move the Sprite that corresponds closely to the Kidbot instructions in Figure 7.4. However, the Sprite movement is barely visible when running this program due to the computer's fast execution. Moreover, because the Scratch visual environment does not return to its initial state automatically, if the flag is clicked more than once the Sprite continues its next motions starting from previous execution's destination. In this case, repeated clicks on the flag in the program in Figure 7.6 causes the Sprite to move in a square, probably outside the anticipated grid area.

At this point, the teacher can improve or change the level of abstraction of the previous NM by introducing the concept of 'initial state', as shown in Figure 7.7, with two additional instructions added to the program, hence increasing the capacity of the

[5] Scratch: https://scratch.mit.edu/

[6] PythonTutor: https://pythontutor.com/

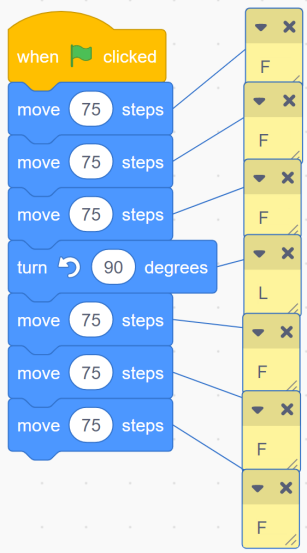

Figure 7.6: A KidBot Scratch program: initial position not defined

CA. However, repeated clicks of the flag with this program does not show any visible movement of the Sprite, due to the computer's fast execution that is not noticeable to human eye. The movement would not be visible in repeated executions, unless the Sprite is moved to its original position manually. Despite modelling the Sprite's movements and programming it correctly (i.e. establishing the NM efficiently), both of these behaviors of the Sprite would most probably be difficult for a novice programmer to understand. This may even confuse or frustrate the learner. Nevertheless, this situation itself may lead the programmer to soon realise the importance of the initial direction and position of the Sprite (i.e. the state of NM) in determining the next instruction parameters. This improved learning also helps learners to enhance their own initial mental model by adding an 'initial state'. Figure 7.8 shows a program with instructions for the Sprite to have a one second 'wait' after each movement. This new set of instructions slows the Sprite's movement, making it easier to observe.

Adding 'waits' between instructions essentially slows the next instruction being executed. This could be introduced for the learners to realise the actual potential/capabilities of their CA (i.e. the computer, and its ability to execute instructions very quickly). The teacher can take the learner beyond the obvious face of the CA (i.e. the Sprite) and thus educate them about the hidden run-time capabilities of a computer, further improving their mental model. The improved, more complex version of the program that helps students to understand the NM better is given in Figure 7.8 exposes hidden ideas like the transaction speed of a computer, an understanding that could not have been achieved with the corresponding Kidbot unplugged activity (i.e. with a human CA).

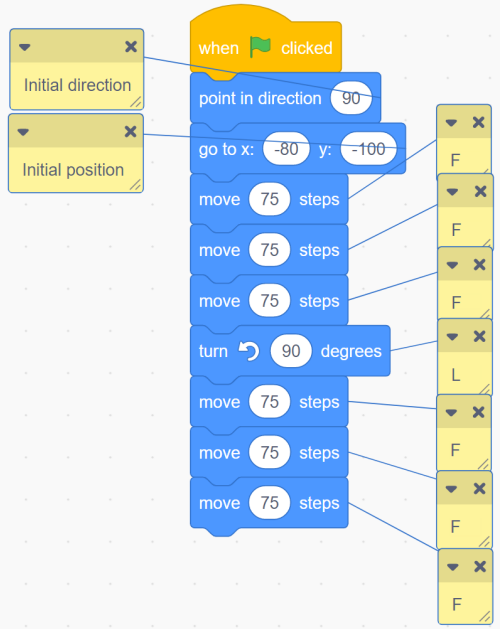

Figure 7.7: A KidBot Scratch program: initial position defined

To look at the computer's role as a CA in a more sophisticated visualisation tool setting, we now look at Python Tutor, a web-based program visualization tool for text based programming, primarily Python (although other languages are offered). Using PythonTutor, learners can visualise stepping forwards and backwards through the execution of their program, and view the run-time state of data structures. Python employs a well structured formal language for its instructions and programmers are expected to *recall* syntax, and unlike Scratch, which expects programmers to *recognise* syntax, learners are expected to have a degree of understanding about basic formal/mathematical instructions. However, what learners may not be aware of is the way the computer behaves, particularly how it allocates or uses its memory, when executing these instructions at run time. PythonTutor addresses this by showing explicitly what is happening, where Scratch primarily use sprite movement for visualisation. For example, Figure 7.9 shows a simple variable value assignment example in PythonTutor, which addresses a common misconception that assigning `a=x` means that `a` will be affected by future changes to `x`.

Through step by step execution in the PythonTutor interface, the learner gets to see graphically how memory is allocated for each variable after each instruction. Up front, the computer becomes a learning interface for the learner to visualise how each instruction of their code behaves — in other words, it acts as a CA — whilst the actual computation takes place behind the scenes (i.e. the compiler/interpreter, actual memory allocations, etc.). PythonTutor offers a more advanced view of the CA than a Scratch Sprite, where it

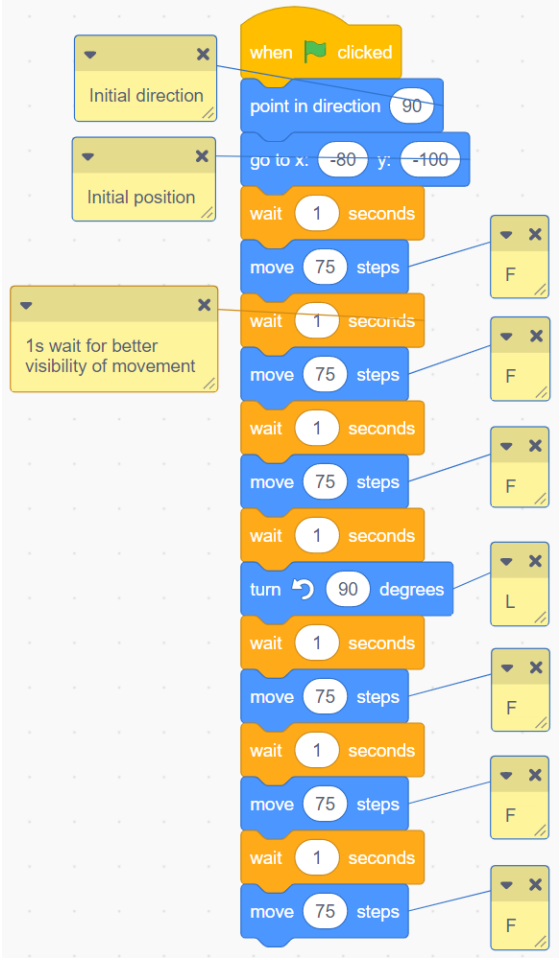

Figure 7.8: A KidBot Scratch program: 1second waits added for clearer visualization

shows some unseen aspects of the computer such as the memory locations and the current instruction. This perfectly demonstrates the two extremes of the NM discussion — that it should be coupled with the physical machine ($\text{NM}_{physical}$) as well as provide conceptual understanding ($\text{NM}_{conceptual}$). Despite being unaware of the actual back-end technical detail within the computer, learners get to develop a sound 'mental visualization' of their program, lead by the computer serving as the CA for their information processing. Meanwhile, the teacher can elaborate the program logic in line with the NM, as well as diagnose any misconceptions and/or errors in learner's mental models.

Visual representation of CAs are very useful in establishing NMs that develop mental models easily and fairly robustly in the early stages of learning to program. The Sprite in Scratch and variables in PythonTutor are good examples of visible CAs in CT, but neither completely reflects the exact properties of a real machine. PythonTutor relates

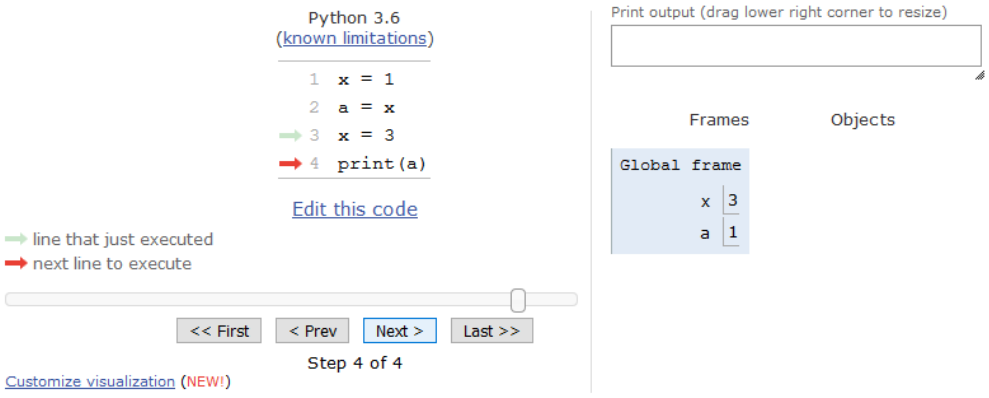

Figure 7.9: A PythonTutor example

more closely to du Bolay et al.'s original definition [75] of a NM that is tightly coupled to the physical device through the programming language used. They facilitate understanding of certain behavior of the real machine, with a focus that is independent to a certain degree on the programming language they are operated by (e.g. the computer's limitations in following instructions, and its efficiency), encompassing the definition of a NM. However, they also present a higher conceptual level by providing a metaphorical layer above the actual machine that is hopefully easier to comprehend by the learner, thereby serving as a NM. This shows how closely related the CA and NM can be, at introductory programming learning stages. Nevertheless, at this stage the learner should be both sufficiently mature in their programming learning process as well as skilled in CT to understand the relatedness of the CA to the underlying NM, regardless of whether the CA is completely understood and/or the NM is fully explained.

7.4.3 Example 3: A Comparison of Two Flowcharts in the Perspective of a Computational Agent

Flowcharts provide a simple graphical method of representing programming sequences and algorithms using a standard set of symbols that show program flow, work flow or processes [164]. They accommodate rather diverse programming language concepts in the same framework and are independent of the implementation and organization of physical computers [254]. Flowcharts are known to be Turing-complete [111, 137, 254], although they can be challenging to map onto commonly used programming languages, partly because the use of arrows corresponds mostly to the "goto" statement, which is generally avoided in modern programming languages. In this section, we look at two different flowchart examples to explore how the CT process is activated in a learner, the role of a CA (human and computer) in that learning process, and how they can contribute

to the NM discussion.

The two different flowcharts used here are: 1) a procedure to count to five (Figure 7.10) and 2) a simple emergency fire evacuation procedure (Figure 7.11, [181]). Both flowcharts use similar symbols and have initial states, Boolean decisions, intermediate process(es) and terminal states. We shall look at the two examples in the context of explaining a scenario to a learner to introduce the concept of *conditionals*. Employing a CA in both scenarios would be to use either a fellow learner to follow the sequence given in the flowchart, or use a computer simulation of the process.

The concept (thereby the abstract NM) that both examples intend to convey is fairly similar and straightforward: start with the initial state, check whether a Boolean condition is met, if not continue with the intermediate process(es), and if yes, [follow a simple sequence before] terminate. The instructions for the Boolean decision and intermediate process(es) in both flowcharts appear to be precise enough to be followed by a CA blindly. However, the contrasting difference in the two flowcharts is with the presentation of instructions: one uses a formal language, the other a natural language. If a teacher asks a student to act as a CA in the problem solving process, i.e. they intentionally try to establish the NM in the learner, using either of the examples, it would be straightforward for the student to understand and the teacher's intentions are likely to be achieved as well.

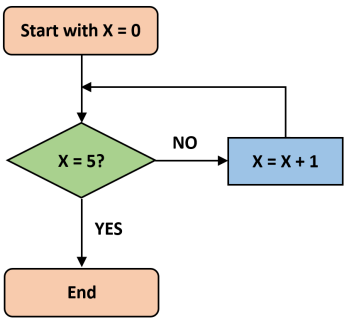

Figure 7.10: A flowchart example for counting to five

In the counting example (Figure 7.10), the instructions consist of mathematical operators that correspond directly to that of a programming language. Thus, prior knowledge in mathematics and/or a programming language supports developing a good mental model in the learner, including the learning having a robust NM, although instructions such as '`x = x+1`' may confuse some learners. However, due to the use of formal expressions as the instructions (i.e. the Boolean decision and other intermediate instructions), the learner may miss out some limitations of an actual computer, regardless of the nature of the CA employed. For example, with a human CA, it may involve a mental calculation

by the person involved (thereby the computation is overshadowed by the prior mathematical knowledge), and with a computer CA, it may involve an internal computational process that is not necessarily visible to the learner. Also they are interacting with an example that may not fully cover the capability of the NM. For example, Figure 7.10 shows that the comparison '`x = 5?`' is possible, but the learner might not infer from this that related comparisons are possible, such as '`x > 0?`', let alone '`x > 3 OR x = 0?`'. Therefore, the learner's mental model may be incomplete or inaccurate, and the teacher may need to diagnose this by asking questions.

However, converting individual instructions in this flowchart example (Figure 7.10) into a computer program written in a programming language with similar operators would be easy and rather straightforward (although converting the *structure* of the flow chart to a conventional language is a challenge because it most naturally maps onto "goto" instructions, which are rarely available in modern languages). In this sense, this example is less directly connected to programming for a learner.

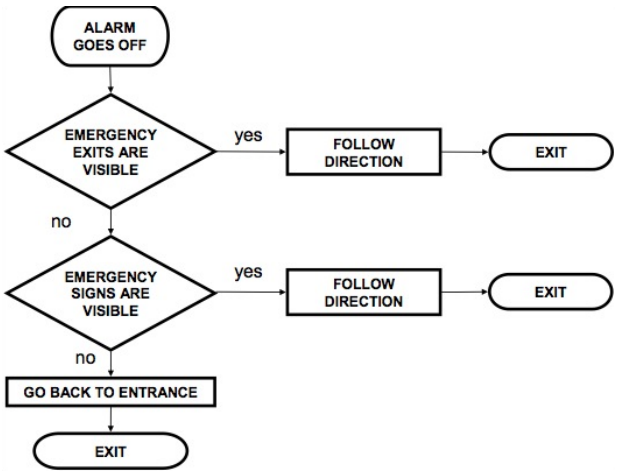

Figure 7.11: A flowchart example for emergency fire evacuation procedure from [181]

In contrast, distinguishing the limitations of a computer as a device can be made more apparent by employing a human CA with the fire evacuation example (Figure 7.11), mostly due to the way its instructions are presented. That is, despite being a Boolean decision, instructions like "*Emergency exits are visible?*" are descriptive rather than formal, and therefore would be straightforward for a human CA, but would not be a typical instruction in a programming language — computers cannot reliably follow natural language instructions. These kind of descriptive instructions cause ill-defined NMs, and may prevent a learner forming a sound computational mental model. Moreover, converting this model into a computer program may also not be straightforward, and may involve more cognitive load in converting it into workable program code. Nevertheless,

this depends on whether a formal semantics is provided. One could in fact define some simple meaning for them in a specific example using high level primitives to allow this. However, the reality is that doing this is difficult in general.

With the fire evacuation example (Figure 7.11), a teacher may be able to explain some *unseen* operations that take place within the computer more clearly by employing a CA: with a human CA, by explaining how a computer would not be able to comprehend the vague instructions, or with a computer CA by highlighting the need to implement additional, precise instructions. A learner on the other hand, may get to develop a broader mental model of the unseen limitations of a computer than with the counting example, even if they have less prior knowledge in formal language instructions — mathematical or otherwise. Converting the fire evacuation scenario into a computer program will not be as straightforward for a learner as with the Counter example, because the instructions are not computationally implementable, at least not directly. A learner may struggle relating their mental model to a NM, and may need support from a teacher in matching their mental model with the intended NM, since the model is poorly defined and the student is learning from examples.

The Flowchart examples highlight that the notation alone does not imply well defined instructions; as with text, the language that it has used is what makes it predictable and/or computational. For example, the Flowgorithm[7] language and our counting example (Figure 7.10) use expressions closer to mathematical expressions, whereas the language in an example like the fire evacuation procedure (Figure 7.11) could use any natural language instructions desired. In the latter, the instructions may include objective conditions like "are visible", which can be related to Boolean expressions fairly easily, as well as any subjective phrases like "is safe to use", which accept a vague, user-relative response rather than a Boolean response. As long as the learning objective does not require the production of an algorithm or a computer program, even a flowchart with ill-defined instructions can equally stimulate CT in a learner and support establishing (and gradually refining) good mental models closer to the NM.

Both flowchart examples provide an abstract graphical modeling of a problem scenario, facilitating the development of a good mental model in the learner. These models are independent of the possible programming language to be used to implement the solution, but instead present precise and step-by-step instructions to arrive at a solution, facilitating CT in learner. The counting example has an ordinary NM described in terms of operations of a simple computer, whereas the other has a far more abstract NM with primitive operations. The nature of the CA employed with these different examples, however, can be a decisive element in the evolution of the learner's mental model, in understanding what could have been initially a quite similar concept (i.e. *conditionals*). In

[7] Flowgorithm: http://www.flowgorithm.org/

the counting case, the small number of formal language-based instructions used are easier to transform to a computer program; in the evacuation chart, there is an unbounded set of instructions that can be written if we assume the language is the whole of english, even though the flowchart structure uses just a few constructs. In both cases, regardless of their nature, the presence of a CA in the problem solving process helps teachers to bridge the gap between the NM and the learner's mental model, as well as in diagnosing possible misconceptions or incomplete mental models.

If it is only to be used by a human CA, a flowchart can be useful in forming a good initial understanding of concepts that can be scaffolded to more robust mental models. Regardless of the nature of instructions (i.e. formal or natural language), they can be useful in stimulating the learner's CT based on their rational thinking. Modeling tools like flowcharts help learners as well as experienced programmers to organise their thought process. For a strong programmer or a mature learner with comprehensive and established CT skills, the CA is essentially a computer, their mental model is very detailed, and the NM is fully understood so that the conceptualisation is completely reflected in either of them. A highly abstract graphical model like a flowchart could only be a mere visualisation aid that helps to organise their rational thinking towards CT.

7.5 The CA as a Simplified Variant of a NM

The nature of a learner's mental model is, by definition, regularly evolving (and inevitably incorrect at times), being fine-tuned itself throughout the process of learning and thereby improving their understanding. The NM is a scaffolding that a teacher provides for the learner in the process of maturing their mental model to a level that sufficiently matches the particular conceptual model intended to be established from the learning component/experience. It is a concept that is often not considered explicitly, rather it is implicitly expected to be established and later diagnosed by the teacher. The CA as explained by Wing's CT definition at the start of Chapter 2, Section 2.3 facilitates the learning experience by positioning itself as a learning tool that learners can relate to that makes abstract ideas concrete for them by being more representative and understandable than a NM. The reason for its relatable nature is partly due to the way the CA is positioned — by definition it is something that can be either human or machine as long as it executes instructions precisely and blindly (an animate instrument), which makes it a concept more reachable or tangible for both learners as well as teachers, in comparison to the mechanistic yet intangible and complex behavior of an actual computer.

The PRIMM (Predict-Run-Investigate-Modify-Make) approach of teaching programming [206] highlights the importance and effectiveness of novice learners' ability to read, predict and modify simple programs before they actually write their own. It can be seen that in PRIMM, a novice programmer themselves becomes a CA to *predict* what is

happening in a simple program code. Initially their mental model as well as their NM related to the programming language may be quite weak. While *running* the program, the computer takes over the role of the CA. During the *investigate* and *modify* phases, the learner improves their CT, and refines their mental model. When they are capable of *making* a working program code on their own, their mental model is much more refined and they have also achieved a good NM. In that way, the PRIMM approach captures the close relationships and transitions among CAs (both human and computer) and a NM in the process of learning to program, as well as in learners improving their CT.

In teaching students the basic concepts of computing, a teacher should concentrate their efforts on communicating the computing content that transforms the student's view of the discipline whilst making the learning of the concepts easier [162]. For teachers new to computing in particular, a physical CA such as a Kidbot or Beebot provides a tangible perspective of the NM that can support discussion, especially when the teacher is not sufficiently familiar with the underlying NM of programming languages they are supposed to be teaching, or are supposed to be using to test the learning. However, it is important that both learners and teachers are aware of the need that the CA of their interest *must follow instructions precisely and blindly* as an essential characteristic of a CA.

If a human CA is restricted to follow only computationally implementable instructions (such as comparing two values, flipping a card, or following a sequence), then an unplugged activity (e.g. find the maximum of a set of weights) transfers comfortably to programming, and enables learners to see the connection. Students are able to experience the computation in a physical context, yet can transfer it directly into a computer program. That way, the CA becomes closely relatable to the NM that a program intends to establish. However, if a human CA is allowed to act loosely, their rational behaviour may interfere with the mechanical nature expected by a CA, and it may become more analogical. That disqualifies a human CA from being relatable to a NM. For example, a human CA is capable of changing the order of instructions, or interpreting ambiguous instructions based on their understanding of the goal, unless they are restricted. Changing the order of the first two instructions in the program code:

```
a = 3
a = 4
print(a)
```

may help a student to develop a better mental model by recognising the change in the outcome, without affecting the NM at all, regardless of the CA being a human or machine. In contrast, changing the order of the first two instructions of a cookie making recipe:

```
add sugar
add flour
mix well
```

may not affect the NM (or the outcome). In this way, the use of a human CA may bring in the risk that existing knowledge is applied to interpret or even change the instructions in a way that still brings about the intended outcome, whereas a machine CA will follow instructions strictly. This is not to say that the recipe example should not be used, but it requires some caution to avoid misconceptions.

However, regardless of the instructions being precise or not, in the absence of the strict restriction to follow the instructions literally, a human CA can bring in prior knowledge (e.g. a human CA might instinctively mix the two ingredients even if the instruction 'mix well' comes in a different order or was even removed), which may undo some of the essential natures of the NM. Another example could be getting students to act out a sorting algorithm. They can see where they would end up, and so might not use the algorithm, whereas the activity may work objectively better with blind students who cannot see their destination but essentially use the algorithm [133].

In the end, computation is a series of symbol manipulations. Only when the human CA has a similar NM to a computer (i.e. a machine CA), does the transfer of knowledge becomes more direct. Descriptive instructions written in natural language in a teaching example may work when a human is employed as a CA, but will require additional effort in transferring them to computation. In other words, such examples may be useful in developing an initial mental model in the learners, yet may not be reliable tools for maturing them to have a more robust and pedagogically efficient mental model of a NM in the long run.

An unplugged activity designed with a focus to teach computational concept(s) will inevitably use a human or a simple deterministic device as a CA that has a NM closer to that of a computer, providing strong evidence of the CA and NM existing simultaneously while linked thematically. Alternating a plugged-in (programming) experience that directly correlates with such an activity will be more effective in helping learners to develop mental models closer to the NM that a teacher intends to convey. The presence of a tangible CA facilitates a conceptual relationship between the learner's mental model and NM, that otherwise could have been only a tiresome mental exercise if done "on paper".

7.6 When Learning Evolves, so do the CT and NM

As pointed out by Denning and Tedre in [64], the CT of a beginner and experts are different; a beginners' perspective of CT evolves as the learning progresses (with more exposure to programming in particular) and becomes complex and more technical. Their mental models also get closer to a good NM. This implies that, in computing education, the view of CT should vary with the stage the teachers/learners are working at.

The roles of the CA and NM are different at different levels of development (of the

learner and of the curriculum). A panoramic perspective on the relationship between the two concepts in computing and learning to program can encompass several stages:

1. earliest stage of CT – the CA can be physical (e.g. Kidbot), and is so simple that a NM is not necessary since the whole language is very easily described, and may not be completely precise;

2. very early programming stage where CT is somewhat stronger – the CA is a system (either a computer, deterministic device or a human – such as a Sprite, a Beebot or a Kidbot that executes programs) and the NM reflects the capabilities of that system (i.e. what it can and can not do). At this stage the CA and the NM are essentially the same system to the learner (they are the properties of the thing that executes programs);

3. developing programming stage with higher level CT – the CA is a computer (still not explained in exact detail, yet the instructions for the computer are clearer) and NMs of different levels of abstraction that can help learning to understand the CA;

4. strong programming stage where CT is inherent – the CA is essentially a computer and a rich and detailed NM represents the CA specifically and accurately.

Another special stage that is highly abstract and conceptual can also be explained where a CA does not physically exist (e.g. Analytical Engine, Quantum Computer) yet the NM defines what a CA could be.

It can be seen that learning to exercise CT begins with a CA rather than a NM that a learner cannot relate to initially. As the learning improves and as CT scaffolds to incorporate programming into the learning process, the roles of the CA and NM also improve subsequently, where at first the CA and the NM are essentially the same: properties of a system possibly explained by a set of instructions. With more experience and awareness about the physical computer, the learner begins to fully understand the computer as their CA and the NM helps them to understand advanced and complex conceptualisations of the device. A higher level of abstraction can be useful in conceptualisation of advanced computation with the aid of highly abstract NMs, even with no known CA.

As shown in the example sections, it is evident that teaching programming should make use of the idea of a CA as a very useful link that can effectively connect a learner's mental model to a robust NM. The co-existence of CA and NM change over learning stages of the learner. Since the two concepts, CA and NM, are largely complementary and closely coupled to each other, teachers can define abstract NMs appropriately to communicate computing concepts to a learner's mental model and use meaningful CAs as a versatile link at different stages in the process.

7.7 Conclusion: How Unplugged Computing Fits in the Picture

Physical activities are a valuable learning tool that provides a different medium for students to experience computation, and reinforce that computational ideas can exist independently of computers. However, there is a risk that some analogous activities (such as following recipes or getting dressed) are sufficiently different from computation that they may be misleading. Nevertheless, physical activities are available that impose rules that focus on what is computationally reasonable, and these will be more effective at scaffolding students in building their own mental models that resonate with a good NM.

We hypothesise that Computational Agents pave a way to link computational concepts (including programming) and the establishment of a robust NM in teaching and learning computing. The nature and flexibility that allows the CA to be a human could be useful in teaching computational concepts, especially at early programming learning stages (e.g. grade school level). Developing a CA that becomes closer and closer to the NM can help students to develop more accurate mental models. The relationship of a CA to a NM helps teachers (particularly novices in computing) bridge the gap between a conceptual model of computing such as a NM, and the mental model created by the learner. Having a fellow student or a semi-deterministic device as a (tangible) CA enables learners to exercise their model instead of it being hidden in the machine (computer), giving them deeper insights into what a computer can achieve. As the NM becomes more sophisticated, having a CA that embodies the program execution is valuable, with the goal that students will eventually have an accurate understanding of the NM and be able to visualise for themselves how a program behaves.

In the process of teaching and learning, a CA facilitates teachers' consciousness of their own mental model against the NM they are expected to teach, especially if the teachers are computing novices. The semantics of a CA are easier for a teacher to understand, and therefore enforce, whereas in a programming language a novice teacher might struggle to know exactly what will happen. CT definitions have provided a degree of flexibility over the nature of its CA (i.e. either a human or a computer), that has potentially eliminated the need for novice teachers to be aware of NMs altogether, yet conveying a similar learning context; if the CA is an actual computer then the NM is very easy to recognise, if it is a human, then using appropriate unplugged activities can enforce the human to use a well-defined set of instructions (e.g. move forward/right; or only compare two values at a time) to establish a more abstract understanding of one of several NMs that describes computation. If the human CA does not have restrictions on their actions (as in free text instructions, or a natural language based flow chart of instructions) then we do not have a simple NM, and therefore a clear mental model would not be possible, and indeed many aspects of computation cannot be experienced

by the student. When an abstract NM is defined without reference to an actual machine, a CA presented with a set of precise yet less detailed instructions (i.e. using abstract operations rather than a certain programming language) can be useful for pedagogical explanations.

Models of computation are very simple and highly visible within the unplugged style learning activities; they provide simple rules (such as "compare two values at a time"), and are designed to scaffold students to understand genuine computational challenges (such as sorting algorithms, data representation and intractable problems). The tangible nature of the CA in unplugged activities possibly enables establishing robust mental models in learners and helps teachers to scaffold them to accurate NMs. We believe that moving from unplugged to "plugging in" enables teachers to establish increasingly better mental models in learners, as well as allowing learners to mature their mental models by avoiding/realising possible misconceptions when they build on their prior understanding, from the physical experience to understanding the expected NM. This also gives insight into why a combination of unplugged and programming experience can be effective, if the NM has commonalities between the two contexts.

In the end, the concept of a NM is vague, and the exploration of relationships in this chapter is intended to help the reader understand those relationships. But we need to be cautious about drawing black and white conclusions about them. Simply put, CA is a "closed box" that we know can follow instructions (without needing to know how it works). The NM is a "box we can open" to see (a conceptual abstraction of) how it works. Using these boxes meaningfully at different stages of learning allows learners to give more effective/efficient instructions (write better programs) because they can develop richer mental models and plans (especially in the case of complex instruction sets like a programming language!)

Chapter VIII

Unplugged Activities for Teaching Programming

With the renewed understanding of the relationship between Computational Agents and Notional Machines in the context of Computational Thinking, this chapter discusses the versatility of unplugged activities in learning to program, by reviewing the usefulness of some existing unplugged activities in teaching programming concepts, identifying possible gaps, and designing unplugged activities that can fill those gaps in modelling basic programming concepts. The concerns that need attention if unplugged activities are used as part of a programming lesson are discussed in detail. The study presented in this Chapter aggregates two contributions of this thesis: 1) findings from a thorough search and analysis of existing Programming Unplugged activities and 2) observations and results of a development/refinement process of a set of Programming Unplugged activities that can be used to model basic programming concepts in particular. The findings and results from this work (i.e. a refined set of Programming Unplugged activities and related discussions) are intended to be directly applicable to teachers' professional development and in programming lessons. This chapter provide partial answers to why unplugged is useful when combined with programming (RQ3), particularly RQs 3.1, 3.2, 3.3, 3.4 and 3.5 (see Section 1.3).

8.1 Overview

A key goal of teaching programming in grade schools[1] is to build understanding of fundamental programming concepts, to enable students to apply them in different contexts while improving their Computational Thinking skills (see for example [13], [165] and [230]). Using unplugged style activities to introduce the key concepts of basic programming elements to students even before they are introduced to digital programming environments is known to be effective for this [115]. Even the teachers with no programming experience appreciate the students' active engagement and reduced screen time when doing the unplugged activities [86], and it is among the top teaching strategies used by teachers to teach both programming and non-programming aspects of a computing curriculum [203].

[1] By grade school we refer to the elementary, middle and high school levels, also known as K-12

Basic programming concepts such as sequence, selection and iteration are programming language independent and therefore arguably better to learn separated from specific syntax, and a learner does not need to be familiar with syntax and semantics of a particular programming language to learn them. Unplugged activities are therefore useful in communicating basic concepts of programming, particularly to young learners. Unplugged learning relies on making use of existing knowledge and extending it into deeper understandings in an engaging, kinaesthetic approach, and often makes use of analogical references. For beginner programmers, using unplugged prior to a plugged-in experience can help establishing strong connections to subtleties of basic programming concepts [115]. Making use of existing knowledge to introduce newer knowledge in a contructivist manner enables learners to understand new concepts within the boundaries of their current understanding and then allow them to scaffold the new understanding more easily in an unfamiliar programming language environment. This can subsequently help learners to avoid forming potential misconceptions that could otherwise have lead to misunderstandings or further misconceptions later in learning.

Misconceptions can occur due to both a lack of proper understanding as well as misunderstanding. Many programming misconceptions indicate a lack of understanding of underlying concepts leading to incorrect or improper use of a programming language [216]. A carefully executed unplugged learning experience that precedes the actual programming exercise may help avoiding such misconceptions. For example, while using two numbered flip-card holders to represent variables A and B and following an instruction to do $A = B$ or $A \leftarrow B$ (i.e. "A becomes equal to B"), learners can physically see that it only changes A's value to that of B. Having seen that the action did not change the value of B, the learner can realise that an assignment statement like $A = B$ in a computer program only changes A while B remains unchanged, as well as that the equation sign has a different function from its use in mathematics. This seemingly simple realisation can avoid several misconceptions related to variables in computer programs later on, especially when learners move to use text based programming languages. The realisation as well as the understanding that follows can be quicker and easier than trying it out in a program code.

Visualization and animation have been often used in illustrating elementary programming and complicated algorithms by teachers during lessons [216]. When developed to gradually scaffold simpler yet important aspects of basic programming concepts, unplugged style activities can help teachers in a similar way to support learners in building mental models that can effectively convert to robust Notional Machines.

8.2 Aims of the Study

The main aim of this study was to explore what the nature of unplugged activities should be, if they are to be used in an introductory programming classroom effectively. As discussed in Section 3.3.1, multiple possible interpretations of an unplugged activity may sometimes be a distraction to a beginner learner as well as a novice teacher in a programming classroom, in their struggle to making meaningful connections between programming concepts and the program they code. Having this concern raised by teachers during the Pilot Study, the overarching research questions aimed to answer in this study were:

1. What is the role of existing style unplugged activities in the context of the computing concept they address?

   We need to consider which existing unplugged activities could be used for teaching programming, as opposed to those that are intended to teach broader CS concepts.

2. For those activities that relate to programming, can they be helpful in building a useful NM?

3. Can new unplugged activities be developed to address programming concepts not covered by currently known activities?

Based on the renewed understanding about the CA and NM in introductory programming discussed in Chapter 7, the focus of the NM in answering these questions is particularly on six basic programming concepts:***variables (including types and assignment)***, ***sequence***, ***selection***, ***iteration***, ***input*** and ***output***, and how unplugged activities would help developing mental models that can relate closely them. The rationale for this focus is discussed in detail in Section 8.4.1. These questions intend to provide partial support for RQ 3.1, RQ 3.2, and RQ 3.4 of the Research Questions described in Section 1.3 (and the first question above corresponds directly to RQ 3.1).

8.3 Method

The studies discussed in this chapter are conducted as a series of expert reviews on unplugged activities in teaching programming. They were conducted using a pragmatic approach as its core methodology, because the overarching theme of this research centres on how unplugged activities can be used in supporting CS and CT education, particularly in programming. This required flexibility in how the studies were conduced, to determine what information was needed, what data should be collected, and how.

These studies involved review by an expert panel. When developing Programming Unplugged activities, it is very difficult to get all the aspects correct because of the possibility of participants understanding aspects in different ways than intended or expected. Reacting and adjusting to shifting circumstances is crucial during an unplugged activity development process, since it is the nature of unplugged activity design to try and repeat with adjustments until the intended outcome is obtained [167]. The entire review exercise was conducted in five phases, and the first four phases were carried out over a period of seven months. The fifth phase was conducted alongside a series of introductory programming workshops as a planned experiment, and is discussed separately in Chapter 9.

Study Phases

This study was conducted in five phases.

Phase 1: *Investigative survey on existing unplugged computing activities that are used in teaching programming*

The objective of the search (by the researcher alone) was to find unplugged activities that are specifically used in teaching programming. Weekly discussions between the researcher and the academic expert were carried out during this phase.

Initially, the term "unplugged programming" was used as the query string. Often we observed that the term "CS Unplugged" is used interchangeably with "unplugged". Consequently the query string was adjusted to "unplugged programming OR CS Unplugged in programming". Another search string used separately was "programming without computers". It was observed that search results for this query were more or less the same as the other search queries. The searches was conducted in late 2020, using the Google search engine[2].

It was soon discovered that, when searching for unplugged activities used in programming lessons, two types of unplugged activities could be found: teaching and learning material published on the internet by various authors, and material found in literature developed for research purposes. Accordingly, the investigative search primarily took two directions:

- search for activities that are freely available on the internet, and are developed for educational or outreach purposes (e.g. CS Unplugged, Code.org, CS4FN-Teaching London Computing, etc.), and

[2] Google: https://www.google.com

- search for activities that are published in literature, and used as part of research experiments.

The activities of the former kind were published by their original authors and/or are found to be popular as teaching related resources. These activities are also frequently used in research studies and referred in teaching forum discussions (e.g. CS Unplugged). An extensive web search was carried out to find such collections or individual activities. These resources are often available with sufficient details to deliver them, with some giving systematically developed teaching material (i.e. CS Unplugged, CS4FN - Teaching London Computing). Moreover, incomplete/unclear activities (i.e. activities with no clear start, execution and end) and/or mere analogous references (e.g. "a variable is like a box") were not considered as unplugged activities.

A systematic literature survey was conducted to find activities of the latter kind. The search queries were used to search in databases including the ACM digital library, IEEE Xplore, and Scopus. It was observed that most of the research studies have often used identical or modified versions of popular unplugged activities that were already found in the former searching such as the CS Unplugged or CS4FN/Teaching London Computing websites. Very few freshly created activities could be found in literature, and often the nature of their actual execution was not clear due to the limited descriptions available.

Phase 2: *Determining the key programming concepts in introductory programming*

Prior to analysing the existing unplugged activities and their relatedness to programming, several text books about introductory computer programming, popular online resources in the same area and several school computing curricula were studied to identify the key elements/concepts of programming that are regularly introduced at a beginner level. Accordingly, the six fundamental programming elements commonly used in introductory programming identified were ***variables (including types and assignment)***, ***sequence***, ***selection***, ***iteration***, ***input*** and ***output***.

Phase 3: *Categorise the unplugged activities under programming concept(s) they model or can be used to model*

This phase of the study initially involved weekly discussions between the researcher and the academic expert, and were later joined by the entire group. The activities found in phase 1 was grouped by the expert panel into categories based on the programming concepts they can be used to model. The process involved careful

study of each individual activity, and detailed discussions with the academic expert about their applicability as unplugged material. The criteria for CS Unplugged activities as described by Bell and Vahrenhold [22] and the CS Unplugged Design Pattern by Nishida et al. [167] were the main references used for design attributes considered for qualifying unplugged activities and in analysing their usability.

It was observed that, despite their use in programming lessons, the majority of the existing unplugged activities used were Non-programming Unplugged activities (i.e. activities targeting to deliver a CT concept), rather than Programming Unplugged activities (i.e. activities targeting to deliver a specific programming concept). As discussed in Section 3.3, since almost all the unplugged activities follow some order of instructions, and some model computing concepts that follow a sequence, they can be used in programming lessons, particularly to explain *sequence*. However, amongst the other concepts such an activity objectively models, sequence could be only a small part of it. Although inevitably almost every activity has a sequence embedded in it, only a few of them concentrate on tackling 'sequence' as a programming concept. Therefore, a basis for primarily filtering the existing unplugged activities that can be useful in modelling/discussing a programming concept was needed. Accordingly, the concerns and considerations in deciding the usability of an unplugged activity in a programming lesson and baseline to distinguish the activities are detailed in Section 8.4.1. The findings of this phase are discussed in various places within this chapter (including how well they relate to programming), and the final filtered list that we decided linked well to programming has been given as a listed collection in Appendix C.

Phase 4: *Develop new activities to model a programming concept(s) or improve existing activities so that they model (or highlight) a programming concept(s)*

This resulted in a mixture of existing, modified and new activities. After the expert panel identified the limitations and taking their concerns and considerations into account, a set of Programming Unplugged activities were developed that model basic programming concept(s). The CS Unplugged Design Pattern by Nishida et al. [167] was followed in the activity design process. When designing these new activities, an effort was made to keep the focus of a single activity to limit to model as few basic programming elements as possible. A key consideration behind the design process was to develop a set of activities that model a programming concept that can also be referred to as simple and straightforward programming exercises. Therefore, care was taken for the algorithms behind those activities to be as simple as possible, enabling a direct correspondence when they are converted to programs. This will enable gradual scaffolding, when they are used in programming

lessons individually or as series of activities. Such a set of activities[3] would be also be easily usable by novice teachers.

Phase 5: *Evaluate the applicability of the new activities*

The new set of Programming Unplugged activities that has been developed in Phase 4 were used in a series of introductory programming Professional Development workshops for both in-service as well as pre-service teachers. The details of these experiments are presented separately in Chapter 9.

As mentioned earlier, the extensive searches for the study were conducted by the researcher herself. Two types of review discussions were conducted: 1) discussion between the researcher and the academic expert during Phases 1, 2 and 3, and 2) discussions among the entire group during Phases 3, 4 and 5. The new/refined activities were introduced to the group by the researcher by executing an unplugged activity, where the others participated actively, while objectively commenting how different types of learners would react. Limitations were determined based on the comments made by the expert group, considering their years of experience working with teachers, students, as well as using CS Unplugged activities in different audiences.

Expert Review Group

The expert group consisted of the researcher herself, an academic expert who is a computer scientist and was also the thesis supervisor, and a group of two expert primary/middle school teachers and one expert high school teacher. All the participants were members of the Computer Science Education Research Group (CSERG) at the University of Canterbury. The academic expert was one of the original authors of the CS Unplugged material, and the expert teachers were in-service teachers of Digital Technologies subject area in New Zealand who also work as research assistants in the CSERG. The group members (except for the researcher herself) had years of experience working with unplugged style computing education, and CS Unplugged material in particular.

8.4 Results Analysis: A Review of Programming Unplugged Activities

8.4.1 Programming Fundamentals, Notional Machines and Learners' Mental Modelling

As discussed in Section 3.3.1, an unplugged activity can be used to model more than a single computing concept. When used in programming lessons, it can sometimes be better if they model as few other computing concepts as possible, in order to avoid

[3] Details of these activities can be found in the working document of Programming Unplugged activities at https://bit.ly/3NrueI5

overwhelming the learners as well as to be able to convert the activity into a simple yet direct programming exercise. Considering that unplugged activities should model the concepts common to programming, but the NM's typical focus is on a particular programming language's behaviour [90], the key programming concepts identification in study phase 1 helped in gaining an insight to what the common focus should be when using unplugged activities for teaching introductory programming.

The six fundamental programming elements in introductory programming identified (i.e. variables (including types and assignment), sequence, selection, iteration, input and output) include three basic control structures identified by the structured program theorem [27] that relates to the Turing-completeness of a programming language (i.e. sequence, selection and iteration) (also see Section 4.4.2). They are also the beginner level programming content that is typically covered in grade school[4] level. The most complex programming constructs typically needed in a grade school content are *functions* and *procedures* (although some curricula go a lot further, e.g. Australia [13]). This implies that in early programming education, a NM should primarily model these concepts, as these constructs provide the basis for almost all computer programs.

The topics most commonly covered by introductory programming books are: Variables (including types and assignment), Loops, Conditionals, Input and Output, and Subroutines/Functions/Methods/Procedures. Considering the pedagogical purpose of a NM (i.e. to model a computer as it relates to executing programs), these individual programming elements in a program also can be seen as atomic NMs on their own, that help to simplify the actual, much broader purpose of a program, and give a better understanding.

It is essential for both the learners and (particularly novice) teachers to have a conceptual understanding about the subtle connections between these constructs and elements [29, 237], in order to develop a useful mental model for their program as well as to implement it using a programming language. For example, the programming construct *iteration* is often referred to by the term 'loops' in most grade school curricula, often associated with the term 'repetition', and also implemented through different types of loops in a programming language. Moreover, there are several ways that a loop can be implemented, depending on the programming language used (e.g. Scratch has a 'repeat until' block and Python uses a 'while' loop to do the same, but they have slightly different semantics even though they can be used for the same purpose).

The nature of a mental model is to build upon some previous knowledge, and be scaffolded from there. For example, a teacher can use an arrow symbol to show the value of variable B assigning to variable A as ($A \leftarrow B$) in their whiteboard explanation. They

[4] By grade school we refer to the elementary, middle and high school levels, also known as K-12, or primary and secondary school.

later use the same symbol (and the idea that was established using it) to scaffold to show how variables can be assigned with a result of an expression ($A \leftarrow B + C$). This emphasises the importance of robust initial mental modelling to develop robust NMs later in a programming learning process. Therefore, the following subsections discuss the use of unplugged activities as part of a programming lesson, drawing out examples from the activities that are already in use, discussing their usability, and suggesting improvements for better usability as well as applicability with new or improved versions of activities.

8.4.2 Concept: Input/Output

Input/output are fundamental components of a computer program, without which a program is meaningless. Therefore they can be identified and verbally explained by the teacher whenever an unplugged computing activity is used in a lesson, or taught explicitly through an activity such as the 'Create-a-Face Activity' by McOwen and Curzon [152]. However, in a plugged-in exercise, sometimes a beginner learner may not be clearly aware of what particular command(s) in their program are serving the purpose in their code (e.g. the 'ask' block in Scratch accepts an input into the 'answer' variable, `print( )` in Python prints out any number of parameters in one line). A learner may use the commands blindly before they actually realise the purpose of them or how they actually behave in their program. This situation was observed during the programming workshops we conducted (see Chapter 10).

8.4.3 Concept: Variables (including Types and Assignment)

While the nature of variables can vary between programming languages, for the programming languages that we have used in our studies (particularly Scratch and Python), the key ideas about a *variable* in computing that are often established in a learner at the earliest stages are: 1) a variable should have a reasonable, self explanatory name and (often) an expected, single data type, 2) it holds one value at a time, and 3) its value is replaced every time some change happens with no trace of the previous value. These three key ideas cover the most important characteristics of a variable in computing, and the correct understanding of them can avoid the most common misconceptions about variables in later stages of programming, especially with text based programming languages. Careful use of an unplugged activity can be very useful in establishing these concepts for a learner. However, if not carefully designed and/or executed, unplugged activities can also *cause* misconceptions. It is worthwhile to look at some different available examples to discuss how unplugged activities can possibly cause such confusion.

Table 8.1 contains a list of Programming Unplugged activities that are used to introduce variables identified using the process in Section 8.3. Often in unplugged activities/examples, variables are introduced using the aid of some analogy: a physical analogy

like an envelope, box, container, etc. (activity 1, 2, 5 and 7), a drawing of a shape (e.g. square or oval) on a whiteboard or on paper (activity 4 and 3), or a metaphoric reference (activity 2 and 6). Some activities (e.g. activity 1, 2, 3) focus only on communicating the concepts of variables, where as some activities (e.g. activity 4, 5, 6) scaffold the concepts of variables and extend it to data types and arrays.

Table 8.1: Some unplugged activities for *Variables*

Index No	Activity Name	Description	Author
1	Unplugged Variables [5]	Use boxes, containers, envelopes, labels, etc. to explain a variable and how setting and changing a variable works.	Code-it.co.uk
2	Everyday variables [6]	Discussion of understanding variables in everyday life. Role-play uses the relatable props, and finally create a programming project that uses the concept.	Code-it.co.uk
3	Variables Unplugged activity [7]	Learn about variables by keeping score of a simple game using whiteboard and pen.	Barefoot
4	Assignment Dry Run [8]	Desk check on paper a series of short fragments of code with value assignment to a variable.	CAS London & CS4FN
5	Box Variable [9]	Act out simple programs that involve variables and assignment, with fellow students holding box(es) for variable(s) in a program mimed as being boxes with integrated shredders and copiers, 'acting' as the computer.	CAS London & CS4FN
6	Emergency Room example	Explain the concepts of global and local variables using the functioning of an Emergency Room as a real life example. ER is explained as a 'function' where global variables are patients and local variables are tools in the ER	R. A. Alamer, W. A. Al-Doweesh, H. S. Al-Khalifa, and M. S. Al-Razgan [3]
7	Envelope Variables [10]	Design/draw a robot in groups; Introduce the variable concept using envelopes to hold various values of labeled placeholders to describe the robot (i.e. robot's name, purpose, height, etc.)	Code.org

Continued on next page

[5] http://code-it.co.uk/book/mq6/mq6

[6] http://code-it.co.uk/wp-content/uploads/2018/06/Variableasbox.pdf

[7] https://www.barefootcomputing.org/resources/variables-unplugged-activity#:~:text=This%20is%20an%20unplugged%20activity,computer%20program%20as%20it%20runs.

[8] https://teachinglondoncomputing.org/resources/inspiring-unplugged-classroom-activities/the-assignment-dry-run-activity/

[9] https://teachinglondoncomputing.org/resources/inspiring-computing-stories/box-variables-understanding-variables-and-assignment/

[10] https://studio.code.org/s/course4/lessons/4/levels/1

Table 8.1 – *Continued from previous page*

Index No	Activity Name	Description	Author
8	Story Variable Planner [11]	Show how variables work using a story with variables embedded as parts of the story; change the story by changing the values of the various variables.	Code-it.co.uk

What suits the best?

Physical analogies help learners to visualise some otherwise unseen behavior of a variable inside the computer. However, as explained earlier, a physical analogy can be confusing or misleading and cause misconceptions unless carefully put into perspective by a teacher. For example, using envelope as an analogy for a variable (e.g. activity 1 or 7 in table 8.1) provides a good support for the learners to understand the rationale of naming a variable. However, because an envelope can physically hold more than one thing at a time, there is a possibility of a learner forming the misconception that a variable can contain more than one value.

A common example is "a variable is like a box (or any form of physical container)". Without explicitly mentioning or showing how a value should be placed in the 'box', a novice learner might assume that a 'box' can hold more than one value, that it can 'make space for' a second value while holding another, or 'keep track of' the previous values. A teacher who understands this issue can enforce the single value, but the model does not naturally enforce it, and this could confuse the learner to a degree that creates misconceptions later in their learning like assigning different values repeatedly to the same variable may collect multiple values in it, or that variables can hold expressions (i.e. formula) rather than their resultant values (e.g. `int a = v/2` will store the expression 'v/2' in the variable `a`), or mixing that different types of data is allowed.

Such a simple reference to physical analogies may also not be sufficient to explain how a variable in computing is different from a variable in Mathematics, in the absence of a relevant explanation. A novice programming learner (who has experience in Mathematics) may find it difficult or even not notice such a distinction until they use the variable in actual program code. In such a situation, the mental model the learner forms may be false and/or weak for supporting further scaffolding of their learning.

The value of the engaging, kinaesthetic nature of unplugged activities is highlighted in [22] and [167]. Some activities like activity 4 and 5 use a printed copy of simple computer program code, which is closer to some of the knowledge used in coding. Activities 2, 6 and 8 lack active, physical engagement of the learner, other than in the context of participating in a discussion. Comparatively, activities 3, 5, 7 and also 8 fulfill some key

[11] http://code-it.co.uk/wp-content/uploads/2019/03/storyvariablesplannner.pdf

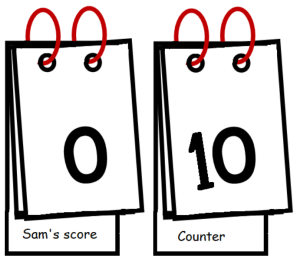

Figure 8.1: Flip-card Holders for Variables

principles of the unplugged approach (i.e. avoiding using computers and programming, a sense of play or challenge, kinesthetic, follow constructivist approach, short and simple explanation, and a sense of story [18]).

Better Alternatives?

After careful study of suitable physical artifact(s) that are suitable to use as alternatives, flip-card holders (Figure 8.1) or spin wheels (Figure 8.2) numbered with values were selected as a promising alternative for representing variables in unplugged activities. Using flip-card holders or spin wheels ensures that the variables can hold only one value at a time. The pre-marked values enable learners to realise the limitations of a variable (e.g. data type, value range) that can lead to scaffolding for further learning. Using empty whiteboards were also considered as a strong candidate, but the possibility of having additional writable space to write more, or not erasing the previous value properly may unnecessarily distract the learners' focus or lead to misconceptions. Spin-wheels and flip card holders can be easily prepared using everyday materials available in classrooms. However, they may not provide room for increasing the value range, and the teacher may need make an explanation or may even move to the actual plugged-in exercise at that point.

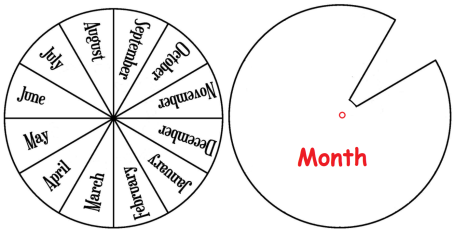

Figure 8.2: A Spin Wheel for the Variable 'Month'

The discussions within the expert group also revealed that beginning with an 'arrow'

(←) to indicate assignment, rather than an '=' sign, is better in unplugged explanations (e.g. in whiteboard explanations). The reasons for this are:

1. to distinguish the computational '*assignment of a value to a variable*' from the mathematical '*is equal to a value of a variable*' and
2. to avoid the learners' existing knowledge about using the equation symbol '=' leading to misconceptions later in program coding.

Figures 8.3, 8.4 and 8.5 show a set of instructions cards developed during the discussions, that can be used in an unplugged explanation of variables in a programming classroom. The learners are given two numbered flip-card holders (Figure 8.1) for variables A and B, where they are asked to turn the flip-cards to the value according to the instructions. Each instruction card is carefully designed with instructions to scaffold learners' knowledge to various aspects about variables.

An instruction set that 'initialises' two variables A and B with $A \leftarrow 0$ and $B \leftarrow 0$ is given in Figure 8.3 (a), and Figure 8.3 (b) continues to introduce 'assignment' by using $A \leftarrow 3$ and $B \leftarrow 4$. Instructions that follow a similar pattern, yet scaffold learning with an increasing cognitive load can be helpful for the learners in many ways. For example, with the instruction cards in Figure 8.3 followed by the instructions in Figure 8.4 learners can see how variables would behave in different value assignments.

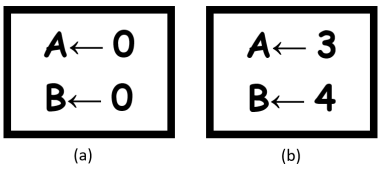

Figure 8.3: Activity Instruction Cards Set 1: (a) Initialising A and B, (b) Value assignment

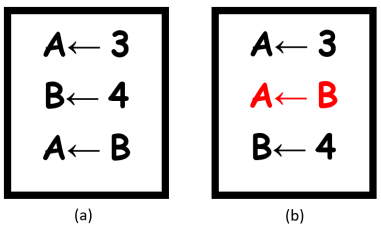

Figure 8.4: Activity Instruction Cards Set 2: (a) Assigning values to A and B, (b) Swapped instructions of card (a)

Flipping over the flip-cards according to the instructions in Figure 8.4 (a), learners realise that only the value of A changes and B remains unchanged after the third instruction ($A \leftarrow B$), leading them to realise that variables are merely acting like 'containers' of values and are only changed upon an instruction. This avoids the misconception that the value assignment flushes a value of B to A and empties B. The next instruction card (Figure 8.4 (b)) with a changed order of the previous instructions cements this understanding. It also scaffolds the learning to teach how variables react to instructions as well as the importance of sequence in instructions. It was observed in the initial trials with teachers that this is a big eye-opener moment for them, including realising the subtle difference between variables in Mathematics vs computing.

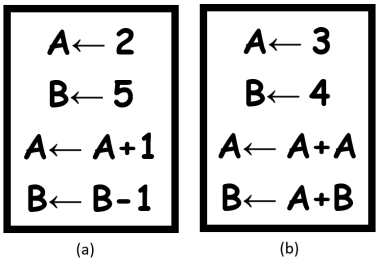

Figure 8.5: Activity Instruction Cards Set 2: (a) Expressions using a variable's own value, (b) Expressions using values of other variables

Figure 8.5 contains instructions that use expressions within expressions. By physically changing the flip-card values of A and B according to the instructions (a), the learners can see how the instruction uses the current value of a variable in the operation, and replaces the current value with the *result*; and how the variable has no traces of the previous value. This is an important and distinct feature of a variable that happens within a program, and is not normally visible to a learner.

For a beginner programmer tracing a program, the value change that happens with the instructions may not be obvious. A possible misconception of a novice is that, after execution of instructions in Figure 8.5 (b), A is 6 and B is 7 [212]. A novice can get confused when a program does not produce the (falsely) anticipated answer, and during a programming exercise, they may possibly not understand the reason. However, it is observed by the expert teachers later (during the main experiments, as discussed in Chapter 9) that, when a learner changes the flip-cards physically, they get to experience this differently and clearer (i.e. A ends up 6 and B ends up as 10). Something to note is that this activity simultaneously covers the concept of sequencing alongside the variable and assignments concepts, although the teacher may not purposely highlight it.

Unplugged experiences of this nature reinforce the learning and scaffold the understanding of what actually happens inside a computer. They also emphasises the unique

manner the variables respond to the sequence of instructions. This helps learners form mental models closer to the actual NM, without having to know a particular programming language. The instructions in Figure 8.5 continue to introduce sets of carefully selected instructions that require increasing cognitive depth regarding variables. However, it should be noted that the flip card approach may introduce a new misconception that all the values are there in the variable and one is just being selected rather than values being replaced.

A teacher can use a checkpoint question (such as asking "what is $A + B$?") after each instruction that can work in two ways: 1) to ensure learning by raising it after the students turn the flip-cards according to the instructions or 2) to assess the student's understanding of the variable NM by raising it before they flip the cards. The exact moments to raise the checkpoint questions depends on the class audience and is difficult to pre-determine. However, a checkpoint question that follows each set of instructions alone would enforce sufficient understanding. Moreover, observing the nature of the class, the teacher can decide when to move to actual programming exercise as well as provide more challenging exercises.

A Simple Game: Variable Dice Battle

Based on the observations made throughout the design and planning discussions, a simple dice game that can introduce some basic concepts and properties of variables was developed, which uses numbered flip-card holders as score cards and a counter. Two players roll dice to battle against each other in the game; whoever rolls the higher value gets a point. Using two flip-card holders assigned for each player as 'score cards' to record their scores, one (or two) others act as scorers(s) and another act as the time-keeper, with another flip-card holder used as a 'counter' that keeps track of the number of dice rolls. The flip-cards are numbered from 0 to 10. Dice are rolled 10 times and the time-keeper turns the flip-card 'counter' to track the number of rolls and the scorer(s) flips the pages of the 'score card' of the player with the highest dice value in each turn. After 10 rolls, the player with the highest value in their 'score card' wins.

This seemingly simple game embeds a few key points about variables. Numbered flip-cards ensure variables have one value at a time. Labeling the 'score card' to identify the player corresponds to variable naming. Setting the flip-card starting values corresponds to initialising a variable. The starting value of the 'counter' could be either 0 or 10; if 0, it counts up; if 10, it counts down. Learners realising that turning pages of the 'counter' could be either way can help them understand the importance of initialising a variable as well as that the initial value is determined by the purpose of the variable. It also can be used to highlight the importance of giving meaningful names; for example, looking at the value of the counter would be completely meaningful only if a counter that started

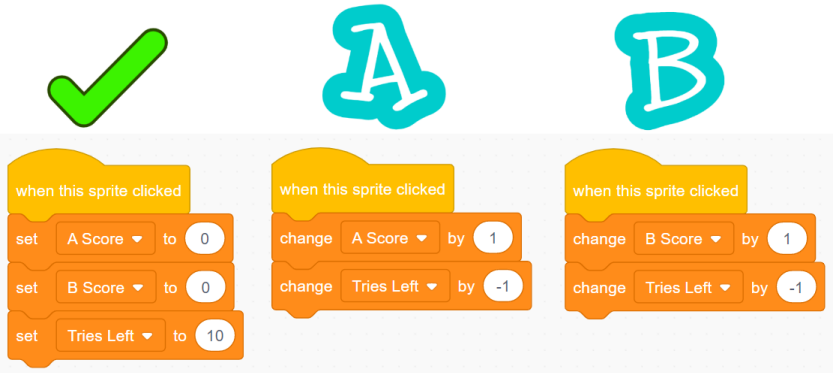

Figure 8.6: Example Scratch Program for Variable Dice Battle

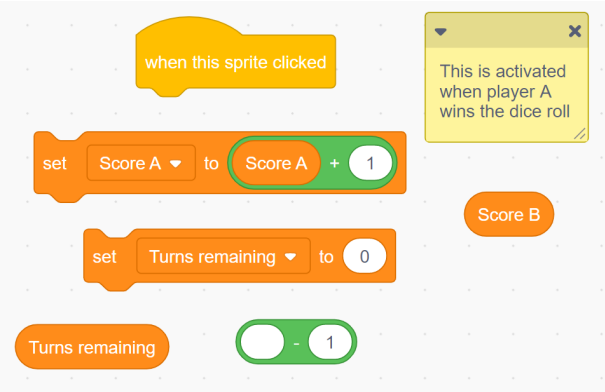

Figure 8.7: Scratch programming Parson's problem for Variable Dice Battle 'Player A' sprite

with 10 is named as something like 'rolls remaining' or one started with 0 named as 'roll number'. Choosing these names provokes a useful discussion on the literacy skills in naming a variable. Each flip shows the learner that the value of a variable is replaced every time some change happens, with no trace of the previous value.

Plugging-in

One concern raised in the literature about using unplugged material in learning computing was that it can draw students' attention towards the mechanics of the operation (e.g. winning the game or completing the task faster than others) rather than the learning objective [79]. A quick transfer from unplugged to plugged-in during a lesson would be an effective vary to overcome this issue. Accordingly, the next step to the learning process is to convert the unplugged learning into a meaningful plugged-in experience: convert the dice game into a simple program. Figure 8.6 shows a Scratch program that captures the

scenario. A simple programming exercise to implement the game enables the learners to establish a solid understanding of what they had experienced previously about variables. The intangible and invisible nature of how variables will behave in their program has already been demonstrated to a great extent with their prior unplugged experience. The dual mode learning provides a wider cognitive coverage for the learner, to build robust mental models and to cement the ideas. A teacher can make the programming exercise more approachable by introducing it as a Parson's problem[12] as in Figure 8.7.

8.4.4 Concept: Sequence

With algorithms at the heart of CT and CS, it is important to establish meaningful understanding about the order and precision of instructions when learning to program. The concept *sequence* can be explained using almost any unplugged activity that primarily models an algorithm. For example, there are many unplugged activities that: 1) embed patterns (e.g. knitting or origami), 2) have metaphorical references that have orderly execution (e.g. life-cycle puzzles, everyday sequences, recipes, map reading, etc.), or 3) include repetition (e.g. dance sequences or physical exercise routines), and these can be used to explain sequence including exploring the idea that changing the order in a sequence often changes the outcome. However, learners cannot directly see how the concept is applicable in a program unless they make meaningful connections to the computational context when an unplugged activity is executed, and so such activities may be best suited to a preliminary experience that is later scaffolded through a more directly connected activity that matches the programming concept directly.

Understanding the sequentiality of statements is known to be one of the main misconceptions novice learners face when understanding the control flow of a computer program [142, 216, 224]. It can be seen that this simple misconception leads to many other misconceptions novice programmers have, as listed by various authors (e.g. [216], [224], [106], [184]). Lowe [142] has reported that young children succeed with short sequences in their simple programs and fail when the sequences become longer, which Lowe attributed it to "cognitive overload". They conclude that the learners need a strong mental model of the domain they work, reporting that "spacial reasoning" (the reasoning by means of physical examples and/or physical means) as a more frequent cause of failure than the misconceptions of the NM of the programming language used. The physical nature of unplugged activities can successfully intervene to establish understandings of a "spacial" (physical) nature.

Table 8.2 lists some unplugged activities that has been used to teach sequence, with

[12] Parson's problems provide learners with instructions of a program in a jumbled order, where they have to rearrange in an order that would make the program work. The puzzle-like format allows learners to practice basic programming in an entertaining manner.

short descriptions of how they are used. Most unplugged activities attempt to model sequence either with repetitive patterns (e.g. activities 1, 2 and 3), by relating to real life scenario that needs step-by-step execution (e.g. activity 4) or by 'programming' a physical, non-computing 'device' to complete a task (e.g. activities 5 – 11).

A careful study indicated that the most noticeable drawback with these unplugged activities that are used to model sequence is that they may fail to establish the desired understanding of the programming context (i.e. they fail to connect the learners' mental model with the NM), unless they are strictly followed by the teacher purposely making a meaningful connection to the programming context verbally or by some plugged-in exercise. The main reason for this is that although the approaches most of these activities have adapted metaphoric references, analogies and so on to appropriately model a sequence, their connection to real world knowledge could distract learners from relating it to a programming context. With such activities, a novice programming learner may fail to see any meaningful connection to programming. Further, writing a computer program that relates to most of these unplugged activities may need a degree of advanced programming knowledge that may not be possessed by a beginner programmer yet.

Some activities may even lead to misconceptions, even when a teacher's support with making connections is available. For example, making cookies following a recipe may appear to model the concept effectively. However, if the rational actions of the learners interfere with the need to *strictly* following instructions (e.g. a human can decide to swap between adding sugar and flour) or they adapt to vague instructions (i.e. "add salt to taste" being understood by human but not a computer), this may cause serious misconceptions among learners, or give over confidence (e.g. "if you can follow a recipe, then you understand algorithms"). This points to the importance of either the unplugged activity naturally enforcing its own rules (e.g. a card can only show one side or another, and can't sit on its edge), or more reasonably, the teacher needs to enforce the rules by reminding students if they overlook them (e.g. reminding students to follow the instructions in the exact order they are written).

Discussion of Sequence Activities

The nature of the processes of learning and the resulting knowledge likely influence the learners' responses to misconceptions [136]. If the teacher provides careful and correct guidance to make robust connections, most of these unplugged activities in Table 8.2 can be useful in modeling sequence, particularly the activities that attempt to 'program' a 'device' to complete a task, such as activities 5 – 11 in Table 8.2. For example, the Kidbots[13] activity (activity 8 in Table 8.2) from CS Unplugged successfully models sequence while also allowing learners to understand other concepts like debugging and

[13] Kidbots: https://www.csunplugged.org/en/topics/kidbots/

algorithms.

Despite their ability to successfully model the concept of sequence, the usefulness of these activities also depends on various other attributes such as the resources needed (e.g. activity 5 and 11 need several props), preparation and setting up time (e.g. all grid activities need the grid layout set), simplicity of instructions (e.g. activity 8 has only 3 instructions whereas activity 5 has several) and of the props (activity 8 can be done on a grid drawn with chalk on ground whereas activity 11 use tailor-made props). Moreover, some activities may be more suitable for smaller groups, while others may require a larger group involved to do them. As explained by Bell and Vahrenhold [22] and Nishida [167], simplicity is a key element in unplugged computing, and this reason itself makes them more useful and usable in diverse contexts of learner groups. Conclusively, despite its successful modeling of the concept sequence, the Kidbot activity stands out from the rest of the activities for its simplicity, easy execution, need of little resources, and being equally suitable for small and large groups.

The Kidbots activity defines a set of simple instructions (move forward: F, turn left: L and turn right: R), thereby establishing its own NM, which directly relates to the nature of an actual programming language. As per the detailed discussion in Chapter 7, Section 7.4.1, this activity builds a comprehensive experience closely relatable to the behaviour of an actual computer during a programming execution. The activity has three clearly defined distinct roles: 'programmer' who programs, 'bot' who follows executes it and the 'tester' who tests the program written by the programmer. The rather passive role of the 'bot' helps learners to understand the relationship between an actual program and a computer while the 'tester' cements the idea of step-by-step execution whilst educating about debugging.

Plugging-in

An immediate transfer from unplugged to a plugged-in exercise helps to cement these ideas. The simplicity of the unplugged activity always contributes to quick and easy direct transfer to a programming exercise. For example, a simple Kidbot program as shown in Figure 7.4 in page 122 and be converted to a Scratch program similar to Figure 7.7. Not only does the switching from unplugged to plugged-in complete a programming lesson by moving to an actual programming environment and emphasising a more meaningful connection of the physical activity to the learning objective of the lesson, but it also cements the concept in a learner's understanding. The teacher can support the learners with a Parson's problem challenge as given in Figure 8.8.

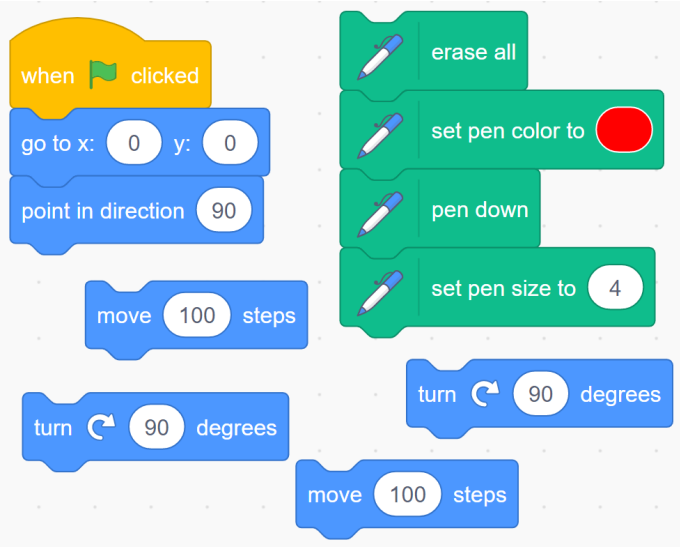

Figure 8.8: Scratch programming Parson's problem for Kidbots activity

Table 8.2: Some unplugged activities for *sequence*

Index No	Activity Name	Description	Author
1	Origami [14]	Follow the instructions provided precisely to create Origami artwork. Instructions are given in standard Origami symbols with special meanings.	CS4FN
2	Life-cycle Puzzles [15]	Cut out and place these pictures from the life-cycle of a frog into the loop provided. Start with the frogspawn and make sure you put them in the right sequence.	CS4FN
3	Dance Move Algorithms [16]	Display a sequence of 5 or 6 cards marked with body movements and ask the students to follow them to complete a dance routine. Ask simple questions like "what happens if you swap two cards around?". Explain that changing the order of the cards will change the routine. Let students create their own dance routines with a set of dance move cards and ask them to create dance sequences. Interchange each other's dance sequences for different dances.	Barefoot Computing
4	Everyday Sequences [17]	Class discussing sequences we encounter in our everyday life and relate them to programming	Code-it.co.uk
5	Human Crane [18]	One learner becomes a human crane arm that move blocks among three bowls set in a row, while another 'programs' the crane by lining up instructions given in 'instruction cards', to move some blocks from one container to another according to a given challenge. The instructions lined up are 'tested' by a third learner.	Code-it.co.uk

Continued on next page

[14] https://teachinglondoncomputing.org/origami-algorithms/

[15] https://teachinglondoncomputing.org/sequencing-and-looping-puzzles/

[16] https://www.barefootcomputing.org/resources/dance-move-algorithms

[17] http://code-it.co.uk/wp-content/uploads/2019/04/everydaycomputingconcepts.pdf

[18] http://code-it.co.uk/ks1/crane/humancrane

Table 8.2 – *Continued from previous page*

Index No	Activity Name	Description	Author
6	Move it, Move it [19]	Grid activity on a grid made of paper squares. One square is marked as the target and another as the start. Use physical movement (e.g. jump, shake, nod) defined as instructions to program a volunteer to move carefully on the grid until the target is reached.	Code.org
7	Happy Maps [20]	Grid activity to get the "flurb" to reach its fruit. Students practice writing precise instructions and then translate them into the symbolic instructions provided. If problems arise in the code, they should also work together to recognize bugs and build solutions.	Code.org
8	KidBots [21]	Involves three volunteers, 'developer', 'tester', and a 'bot'. The developer has to write a program using only three instructions (move forward: F, turn left: L and turn right: R) to guide the bot from location A to B on a grid. The tester reads the instructions to the bot once a program has been written	CS Unplugged
9	My Robotic Friends Jr [22]	Put two pairs of students in one group: one pair to develop their own program for a Robot using a given set of instructions, and the other pair to run it. Have each pair choose an idea from a given ideas collection that they would like their robots to do. Students should discuss how the instructions should line-up. Each pair take turns to become the "robot" for the other pair by following their instructions.	Code.org
10	Graph Paper Programming [23]	Use sheets of 4x4 graph paper. Starting at the upper left-hand corner, guide teammates with simple instructions. Those instructions include (Move One Square Right, Move One Square Left, Move One Square Up, Move One Square Dow, and Fill-In Square with color), with appropriate symbols.	Code.org

Continued on next page

[19] https://code.org/curriculum/course1/2/Teacher

[20] https://code.org/curriculum/course1/1/Teacher

[21] https://www.csunplugged.org/en/topics/kidbots/

[22] https://studio.code.org/s/coursec-2021/lessons/2

[23] https://code.org/curriculum/course2/1/Teacher

Table 8.2 – *Continued from previous page*

Index No	Activity Name	Description	Author
11	Crabs and Turtles - The Treasure Hunt	Strategy game between two groups (turtles and crabs). Players need to create effective sequences of commands (steps, turns and loops) to move their game piece (crab or turtle) towards treasures on a sequence board. They have to manipulate coloured game pieces to figure out the most efficient way to collect treasures. To approach the treasures, teams have to arrange their 'motion cards' on a 'sequence board', while considering the respective restrictions for crabs and turtles. Players have to consider other restrictions and conditions set by the game rules while creating their respective instruction sequence.	Katerina Tsarava, Korbinian Moeller, and Manuel Ninaus [234]
12	Treasure Hunter 4	The main objective is to move towards a treasure chest, walking on a grid squire from an assigned starting point, avoiding barriers (rocks) placed on the grid. Two players are paired up to complete each mission. Each mission has a different starting place, number of chests, position of the chests, a variety of bonus points, and different flow-block syntaxes with different commands. In order to move to the treasure chest in their chosen direction, the players have to connect flow-blocks provided, which include a set of ready-to-use syntaxes consisting of a start, an end, and the number of repetitions, as well as a set of shapes and individual syntax for players to connect by themselves.	A.Threekunprapa and P. Yasri [233]

8.4.5 Concept: Selection

'*Selection*' is the computing concept that decides the changes in the execution flow of a computer program based on a condition. The concept is quite important in programming since it is the primary basic way to change an otherwise top to bottom execution of a program. Selection is usually implemented through conditional statements that instruct the computer to make decisions based on conditions with a Boolean outcome, using constructs such as "if" or "if-else". The key concepts that we should establish in a learner about selection are: 1) at the most fundamental and atomic level, all conditions must evaluate to 'true' or 'false' (binary); conditions can be combined as long as the expression only produces one value (true/false), and 2) depending on the result of the condition, the flow takes one path or another. Selection is sometimes achieved using other constructs such as case statements, but the 'if-else' construct is the most common that beginners are likely to encounter. Learners understanding the 'binary nature' of a decision and the change to the control flow caused by such decisions in a program code at early stages enable them to develop robust program logic and improved trouble shooting abilities later in their learning.

The concept of selection is tightly coupled with logical thinking for the programmer (learner) where, they should be able to understand and logically reason about the necessary changes in the flow of their program, as well as what and how the alternative flows should do. In a programming classroom, this aspect could easily be overshadowed by having to learn different programming constructs of the programming language and its syntax and semantics to implement a condition which includes Boolean expressions and sub-procedures in the two paths. Using unplugged programming activities to establish the conceptual understanding prior to actual coding could therefore be a very effective alternative.

Table 8.3 contains some unplugged activities that can be used to model selection. As discussed earlier, mere metaphoric reference to everyday activities may not be effective and lead to unnecessary misconceptions if they are used as unplugged activities in a programming classroom. The real life context of an unplugged activity should not overshadow the computing context, or otherwise the learners may either feel that the computing relevance disconnected or completely abandoned. For example, activity 1 in Table 8.3 which discusses the daily routines to understand selection, which may only expose learners to logical reasoning and rational choice, unless a teacher specifies how they are applicable in a computing context. Even with correct connections to a computing context, for an early learner, such discussions may hide the 'binary-ness' (the need to reduce their condition to have a Boolean response) that a computer usually requires in a condition check.

Most activities that were used to teach selection found in the survey seem to model

the concept effectively, as well as adhere to the principles of unplugged computing. However, it was observed that the underlying algorithms of those activities may need some advanced programming skills if and when they are converted to a program (i.e. used as programming exercises). For example, the Sorting Network activity (activity 8 in Table 8.3) can be used to explain how a condition works and the need for a Boolean style decision and conveys that a combination of simple selection statements can have a sophisticated result. However, programming a Sorting Network is a challenging exercise for a beginner programmer since it is a parallel algorithm. Therefore, the Sorting Network activity may not be useful in a plugged-in exercise that immediately follows the activity.

Discussion of Selection Activities

Choosing an unplugged activity that suits best for the learning context of a lesson depends on many aspects. As explained earlier, mere or rather unspecific use of activities closely related to everyday life may risk misconceptions or not achieving the learning objective in a programming context (e.g. activity 1 using everyday examples may overshadow the programming context such as the need for Boolean-ness in the condition), and in this case, more responsibility rests on the teacher to ensure that the class remain within the intended restrictions of the activity. The complexity of instructions (e.g. activity 4, 5, 7) may cause difficulties in doing the activity in a classroom, that could cost time. Moreover, such complex instructions may not be very useful if the activities are to be used as programming exercises later on (e.g. activity 3 or 8 may not be suitable for a beginner programmer). The simplicity in both instructions as well as the resources used (e.g. activity 1 and 2) also a decisive aspect to consider when unplugged activities are used in classrooms.

We have already observed that simple games with simple condition-rules that are easily convertible to a computer program would be a good approach to blending unplugged into teaching selection in a programming classroom. Similar to activity 3 in Table 8.3, a series of games with rules that gradually increase in complexity could be used in a scaffolding manner to effectively introduce selection. A rule in a conditional game exercising a learner's prediction/assumption forces them to think logically and understand how a computer would act according with a binary decision of selection (choice). It helps them to understand the importance of decomposing a solution into binary decisions in computational problem solving that involves a selection, as well as reinforces their debugging skills. This could further cement the idea of the need for precision and the importance of the order of instructions in a program, as well as getting the logic correct. None of the activities in Table 8.3 addressed this well, so a new game was developed using the principles in Nishida et al. [167]. Figure 8.9 shows a few game cards from a series of Conditional Dice Games developed, that contain game rules with conditions of increasing

difficulty.

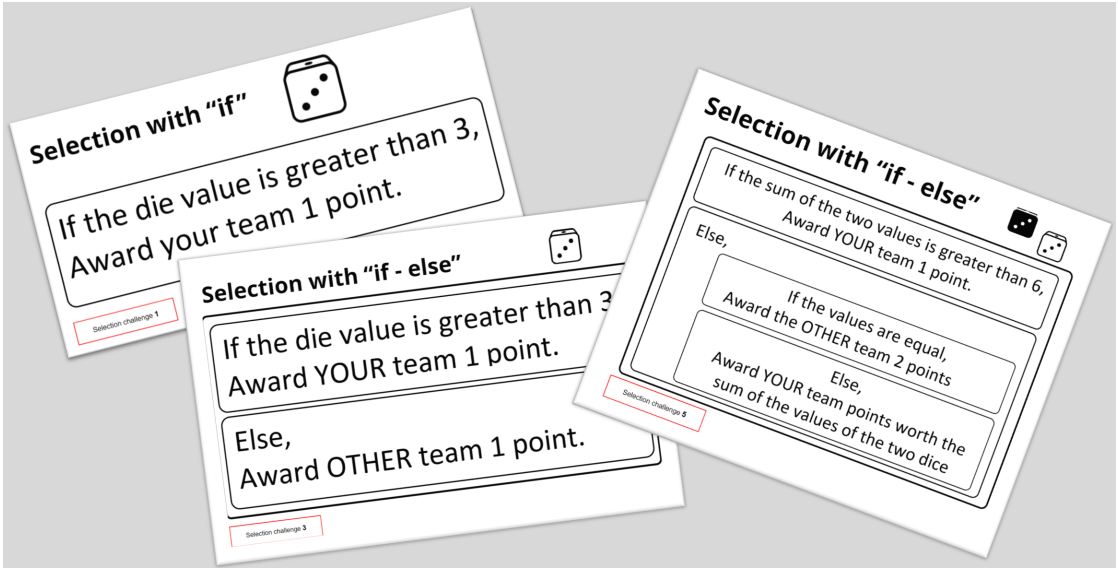

Figure 8.9: Conditional Dice Games with game rules with increasing difficulty

It is important that the learner develop a correct basic mental models of how a condition works in a program for them to use the concept effectively in increasing complexity, for example 'if' statements are combined, or compound conditions are used. When the learner does not have the correct mental model with 'If-else-if' or with a series of 'If-else' blocks, they may find it hard to construct, trace and debug them when they have series of such constructs in their code. It was observed (both during design discussions and in the PD workshops where the games were trialled) that playing a physical game scaffolds learners to understand "what" such a series of conditions would result in and enables them to rethink the formation of their logic in actual program code. It also makes them think when and where they should use different programming constructs their programming language provide (e.g. 'if' blocks or 'if-then-else' blocks in Scratch). The cards are essentially a combination of PRIMM [206] and Unplugged.

A teacher can provide scaffolding based on the the rules of an unplugged game, and provoke the learners' imagination with an assortment of programming exercises of a similar nature, with rules of increasing complexity. Moreover, it was observed (both during design discussions and in the PD workshops where the games were trialled) that the teacher may not need to use many games, but after a few games the necessary understanding would be already established.

'Conditional Card Games' instead of dice games could open a whole array of logical thinking exercises, that can lead to the use of different programming exercises with various conditional constructs. A deck of playing cards can provide an assortment of

combinations with its four different suits, two different colours, array of numbering and order, and faces & pips. The more demanding conditional rules in a single card could push the player harder to think before making the decision, which helps in building a strong mental model.

Figure 8.10: A Conditional Card Game with only one simple condition

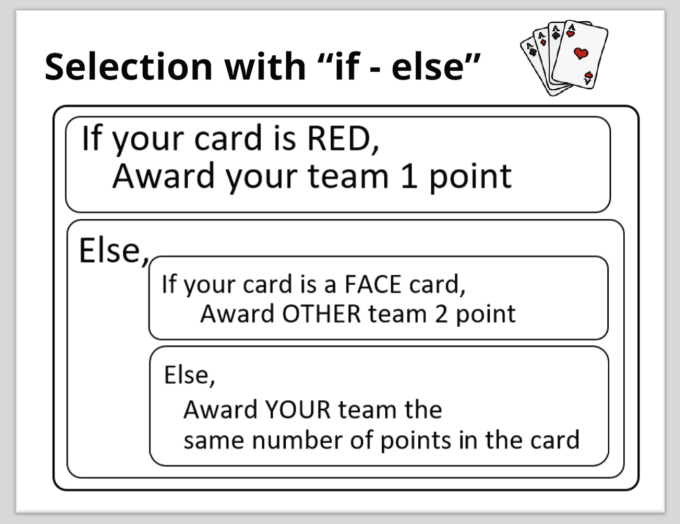

Figure 8.11: A Conditional Card Game

For example, Figures 8.10 and 8.11 show Conditional Card Games using conditional rules for scoring. A simple rule as in Figure 8.10 is not difficult to understand, since the players can automatically disregard every other detail but the colour of the card. However the situation changes when they move to a more challenging game as in Figure 8.11. When playing this physical game, when a player draws a RED KING card, the immediate reaction would be to award 2 marks to the other team. But having to apply the “If” condition *before* taking the FACE into consideration forces the players to think about the

flow of logic and realise that even if the KING is higher than a number card, the fact that it is RED becomes the decisive factor of the game scoring. For a beginner programmer, this may present how a computer would follow instructions blindly to respond to a conditional clearer and much more straight-forwardly than program code. The only major difference between a game card and program code would be the syntax, yet the ability to physically see the detail in the playing card alongside the game card (i.e. condition), saves the learner from the cognitive load of having to imagine the scenario while understanding how conditions work.

The next level of scaffolding for selection would be to introduce Boolean operators within a condition (e.g. "If the card is RED AND a FACE card" or "If a card is a FACE card OR HEARTS"), building up to scaffolding complex logical combinations as conditions that require incorporating more different programming constructs in a program. After playing the physical game for some time, the teacher can also encourage the class to come up with new rules of their own, play them and try to implement them in program code. Moreover, with younger audiences, the game scores can be replaced with a physical activity to create a much more playful and kinaesthetic learning experience (e.g. Figure 8.12). Such variants can set up to a far more meaningful, relatable and engaging learning experience to earners that connects their existing knowledge to new knowledge, to form effective mental models as well as relating them to a robust NM of the programming language they employ.

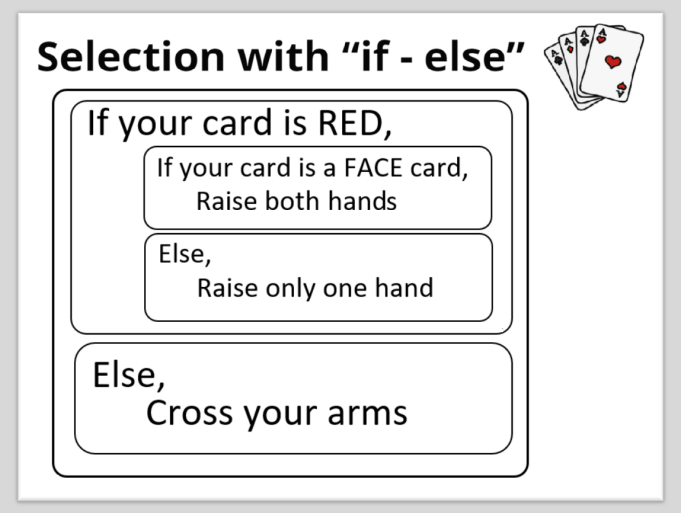

Figure 8.12: A Conditional Card Game with Physical Movements

The indentation in the game cards can stimulate the learner to think more of the syntax that shows the logical formation of their logic and thereby the programming constructs they should use in their code, further refining their mental model. In that

way, the game cards are essentially a combination of PRIMM [206] and Unplugged, where the learner gets to *predict* the outcome of their program before they actually move into coding, and thereby moving into the rest of the phases in the model. Different programming languages provide different constructs for selection (e.g. Scratch has 'If' blocks and 'If-then-else' blocks and no 'If-else-if' blocks). In that case, having indentation in the game cards forces the learners to think about implementing the conditions within the limitations of the programming language they use, or for younger students, the cards could be limited to constructs that map directly to their programming language.

Plugging-in

Implementing selection could be challenging for a beginner programmer. However, the simple conditional games similar to those shown in Figures 8.9, 8.11 or 8.12 can be directly converted to programming exercises. Moreover, the programming exercises can expose the learner to the use of different language constructs gradually, with the increasing difficulty of the games. Figure 8.13 shows Scratch programs that introduce: (a) the 'If' block and (b) the 'If-Else' block respectively, with two of the dice games shown in Figure 8.9.

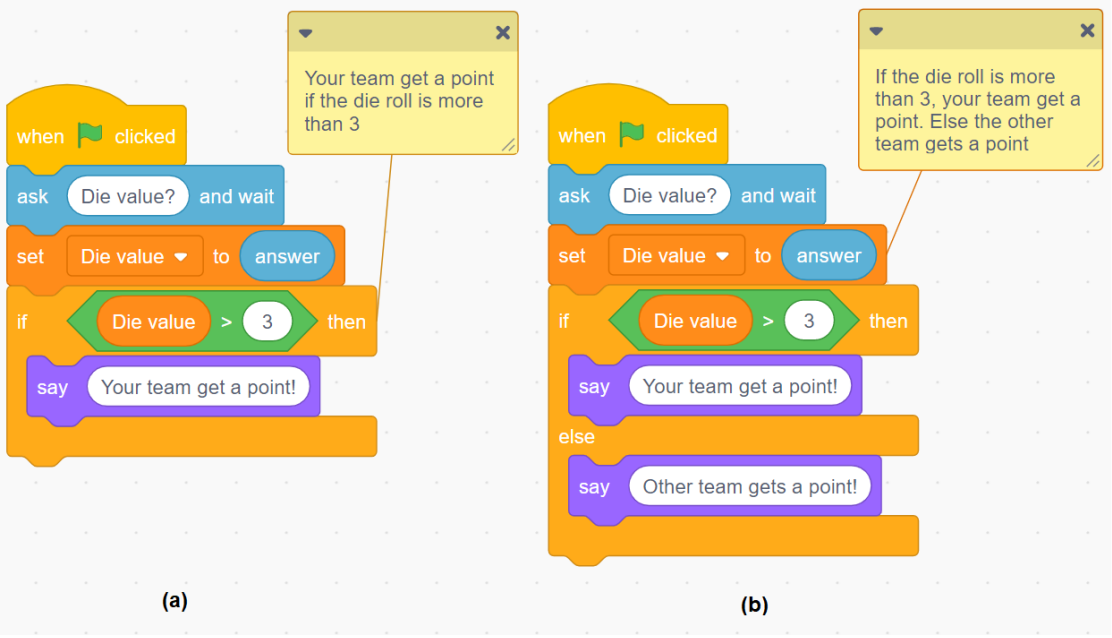

Figure 8.13: A Conditional Dice Game Scratch program code (a) with 'If' block (b) with 'If-else' block

The difficulty in transferring the activity to a programming exercise was the main reason behind the choice of dice over playing cards in these games. It was soon discovered

during the design discussions that, although the wide variety of information available in a deck of cards (i.e. four suits, two colours, range of values and two types) makes them suitable of creating challenging games, the same reason can make things quite complicated for a beginner programmer when they try to convert the game into a program. They may need knowledge of different data types other than integers (numbers) such as strings or assigning a value to represent a string, if they are to convert the game into a program, which is possibly beyond their level. The dice on the other hand, are simple and far more relatable to learners, having only to deal with numbers in their program logic.

When the plugging-in exercise is started off as an attempt to automate a relatable problem solving experience, the learners can be expected to have less difficulty understanding the different constructs a programming language can employ (e.g. 'If' and 'If-Else' blocks in Scratch) and the manner they should be used in a program code when building their rules. Having such understandings cemented at early stages of learning helps to avoid misunderstandings due to the use of jargon (e.g. 'selection' in natural language has many different meanings and its application in the sense of 'choice' is only one of them). Robust basic mental models developed at early stages and them fine-tuned with stronger scaffolding using the learners' existing knowledge as a bridge (e.g. using 'If' for a single game rule at first and extending it to several 'If-Else' statements with more complex rules) could prevent misconceptions from occurring in later stages of learning. It was also observed during the design discussions that exercises of this nature can be extended from a very simple programming exercise to much more complex exercises (e.g. introducing Boolean operators to combine several conditions) quicker than with a plugged-in only approach.

Table 8.3: Some unplugged activities for *selection*

Index No	Activity Name	Description	Author
1	Everyday selection [24]	Discuss everyday understanding of the conceptusing examples such as choosing your socks, using this resource and role-play them. Finally create a programming project that uses the concept.	Code-it.co.uk
2	Lightest and heaviest [25]	An activity originally designed to introduce sorting algorithms. Sort the weights by placing two weights on a double-sided scale, comparing two weights at a time.	CS Unplugged
3	Conditionals with cards [26]	Introduce 'if' and 'if-else' using series of simple card games such as 'scoring a point if you choose a black card'.	Code.org
4	Patterns	A card game. The goal of the game is to find patterns and match cards by certain rules (such as they have the same pattern, should not be of same colour at the same position, should sahre the same colour palette), as fast as possible.	K. Tsarava, K. Moeller, and M. Ninaus [234]
5	Crabs & Turtles - The Treasure Hunt	A grid board game of coloured game pieces played by two teams (turtles and crabs). The goal of the game is to collect a specific number of food-treasures spread across the game-board as fast as possible. Turtles and crabs meet different conditions based on the colour of the items they collect, and meet conditions marked on the grid based on their prescribed abilities and the terrain.	K. Tsarava, K. Moeller, and M. Ninaus [234]
6	Treasure Hunter	A set of unplugged coding with flowblocks of missions with incremental difficulty to hunt a treasure on a 5x5 grid. After completing each mission, players should check the correctness of their own rules. Players proceed from one mission to another, applying the concept that they previously learned.	A.Threekunprapa and P. Yasri [233]

Continued on next page

[24] http://code-it.co.uk/wp-content/uploads/2019/04/everydaycomputingconcepts.pdf

[25] https://classic.csunplugged.org/documents/activities/sorting-algorithms/unplugged-07-sorting_algorithms.pdf

[26] https://code.org/files/ConditionalsHoC.pdf

Table 8.3 – *Continued from previous page*

Index No	Activity Name	Description	Author
7	Full or Fully, Adjective to Adverbs, I before E except after C, Sentence Checker Algorithm [27]	Convert a spelling/grammar rule into a flowchart algorithm or use a flowchart algorithm to work out which spelling to use. Students can practice individually and/or test their input/output understanding by working in groups to 2-3.	Code-it.co.uk
8	Sorting Networks [28]	An activity originally designed to introduce sorting. Six players are given six items respectively in unsorted order (e.g. numbers, a musical bells, letters,etc.). They follow a sorting network drawn on the ground while checking a condition comparing their value with another player they meet at each internal node they walk across. The conditions would direct players to sort themselves in order.	CS Unplugged
9	Imp Computer [29]	Physical execution of a program by students representing instructions in a baton exchange relay. A variety of concepts that have been used in the respective program is explained and discussed.	CS4FN
10	Create-a-face [30]	The class makes an emotional robot face (relating to moods and emotions) by holding cards and tubes themselves. Then they program it to react to different kinds of sounds such as nasty, nice or sudden and show different emotions such as sad, happy, surprised.	CS4FN

[27] http://code-it.co.uk/spellingalgorithms

[28] https://www.csunplugged.org/en/topics/sorting-networks/unit-plan/

[29] https://teachinglondoncomputing.org/resources/inspiring-unplugged-classroom-activities/the-imp-computer-activity/

[30] https://teachinglondoncomputing.org/resources/inspiring-unplugged-classroom-activities/the-create-a-face-activity/

8.4.6 Concept: Iteration

'*Iteration*' in programming is conceptually similar to its dictionary definition, "the action or a process of iterating or repeating" [68], since in programming it is about repeating a sequence of instructions. The amount of repetition depends on what the program segment intends to achieve, and the termination is determined by meeting a condition specified (if no condition is specified or unless it is a special case like a counted repeat, it leads to repeating infinitely). Iteration is implemented through 'loops' in computer programs; different programming language have different loop structures with varying syntax and semantics.

It was observed that the basic concept of iteration is quite easily understood by learners, and implementing simple counted loops is also learnt easily. For example, during Kidbot activities, participants sometimes opted to use '3FR' instead of 'FFFR', which implies 'repeat F three times'. It is natural to replace a long list of Fs with such a notation. During the programming workshops, learners were also seen starting to use simple counted loops (repeating for a certain number of times) in their programs even before the concept was introduced by the instructor.

However, the concept of iteration becomes challenging for beginner learners when it involves conditions or nesting. Skills in Computational Thinking are needed to determine how such situations should be logically formed and to what segment of instructions those conditions should be applied. Implementing loops with conditions and/or how a conditional loop behaves in program code has always been difficult for the beginner learners to understand [53]. They also seem to be the cause leading to most of the common misconceptions about loops in programming. Of conditional loops, the concept of 'repeat-until' seems easier for the learners to understand [157]. However, such seemingly easy understandings may include misconceptions such as that exiting from the loop happens as soon as the condition fails, while it actually completes the remaining instructions in the sequence for that loop, then attempts to re-enter the loop, at which point the condition fails, causing the control to exit the loop.

Another common challenge is to understand the correct interaction between loops within loops. Instead of trying to understand what is happening in both inner and outer loops at the same time, learners need to focus on one loop at a time and understand the inner loop within the controlled condition of the outer loop.

Some Programming Unplugged activities that have been used to model iteration are listed in Table 8.4. It was observed that most unplugged activities available are addressing the grade school[31] curriculum, particularly primary school. The concepts of controlled loops can be beyond junior classes and therefore the many of the activities only focus

[31] By grade school we refer to the elementary, middle and high school levels, also known as K-12, or primary and secondary school.

(e.g. activities 1, 2, 3, 4, 5 and 9) on simple counted repetition. Counted loops could be used as scaffolding to later encounters with loops that use Boolean expressions to control them. Many of these activities are purposely used to communicate the concept of iteration (i.e. used as Programming Unplugged activities, and not as Non-programming Unplugged activities). For example, activities 3, 4, 5 and 6 in Table 8.4 have the term 'loop' or 'repetition' in their title, and they seem to be used successfully when they are used in programming classrooms, mostly because, for a learner, iteration in its simplest form is easy to understand as a concept. Counted loops could be used as scaffolding to later encounters with loops that use boolean expressions to control them.

Table 8.4: Some unplugged activities for *Iteration*

Index No	Activity Name	Description	Author
1	Life-cycle Puzzles [32]	Create animal life-cycles, natural cycles, etc. using cardboard puzzles, and write simple programs for them. In these puzzles the cycles are fitted into a flowchart style diagram familiar to programmers.	CS4FN
2	Getting Loopy [33]	A dancing activity that use dance patterns containing repeated movements. Students are asked to identify repeated patterns and replace them with loops.	Code.org
3	Everyday Repetitions [34]	Discuss everyday situations that contain repetition (i.e. songs, dances, PT exercises). Then role-play them before moving into the actual programming exercises.	Code-it.co.uk
4	Happy Loops [35]	Grid activity to make the "flurb" reach its fruit. Students practice writing precise instructions and then translate them into the symbolic instructions provided. Repeated instructions are replaced with loops.	Code.org
5	My Loopy Robotic Friends Jr. [36]	'Program' a fellow student 'Robot' using a set of "robot" programming instructions agreed by the class. Introduce 'loops' by replacing the repeated instructions with the number of times they are repeated	Code.org

Continued on next page

[32] https://teachinglondoncomputing.org/sequencing-and-looping-puzzles/

[33] https://studio.code.org/s/course1/lessons/12/levels/1

[34] http://code-it.co.uk/wp-content/uploads/2019/04/everydaycomputingconcepts.pdf

[35] https://studio.code.org/s/coursea-2021/lessons/7

[36] https://studio.code.org/s/coursec-2021/lessons/7

Table 8.4 – *Continued from previous page*

Index No	Activity Name	Description	Author
6	Treasure Hunter 5	A set of unplugged coding with flowblocks for a treasure hunt on a grid of 5x5, with incremental missions of complexity. This activity introduces different programminng concepts at different stages and at level 5 it introduces 'loops' with relevant symbols.	A.Threekunprapa and P. Yasri [233]
7	Conditionals with Cards [37]	A series of games first used to Introduce 'if, if-else' using a set of simple card games. 'Loops' are introduced to replace repeated patterns in the games.	Code.org
8	For Loop Funs [38]	Dice game that discusses how the 'for' loop works using a starting value, stop value and intervals. Students are provided with a special sheet to with blanks to fill in with information needed to discuss the loop.	Code.org
9	Knitting [39]	Knitting activities that use knit and purl of different sequences of patterns. Use symbolic notation to indicate repeated patterns.	Code.org
10	Fitness Unplugged [40]	Create a Fitness App using a 'programming language' defined by the students. Design cards with instructions for start, different exercises, and end pose. Write ' Fitness Apps' by arranging the cards in different combinations, using a hula hoop to indicate where to repeat a pattern and a whiteboard to indicate the number of times to repeat, loops within loops are introduced.	CS Unplugged
11	Letter Q [41]	Convert a grammar rule into an algorithm, and use 'loops' where repeated patterns are identified.	Code-it.co.uk

However, as mentioned earlier, many of these activities need advanced programming skills if they are to be directly converted to programming exercises. When a beginner learner moves from unplugged activity to a plugged-in activity, they may become complex programming exercises for them to handle. For example, knitting or jewellery making contain clear examples of repeated actions, and are very useful activities to show the concept. However, the loops that need to capture complex patterns may involve nested or controlled loops, which may be complicated for a beginner programmer understand and to

[37] https://code.org/files/ConditionalsHoC.pdf

[38] https://code.org/curriculum/course4/8/Teacher

[39] http://www.cs4fn.org/regularexpressions/knitters.php

[40] https://www.csunplugged.org/en/topics/kidbots/unit-plan/fitness-unplugged/

[41] http://code-it.co.uk/spellingalgorithms

implement. Programming languages provide a variety of constructs to implement a loop. The counted repetition loops are easier to implement (e.g. in Scratch the 'repeat' block simply implements it), and many examples in Table 8.4 are examples of count-controlled loops. Anything beyond a simple count-controlled loop needs advanced understanding of the concept. Programmers need a deeper understanding of how those constructs would behave in their program for them to use them effectively.

Some existing activities lack meaningful connections to a programming context, despite their success in modeling the concept. Learners may understand the concept of iteration, but their understanding may not be very useful when they attempt to write actual program in a computer. For example, activity 9 successfully models iteration, and teaches the learners how a repeated sequence can be replaced with a loop, but using them as programming exercises may not be straightforward and may need advanced programming skills if complex algorithms are involved.

What suits better?

It was observed that the Fitness Unplugged[42] activity by CS Unplugged can be used to model iteration effectively, alongside the other computing concepts it models. While providing a conceptual understanding of how computer applications are developed, it also helps introducing sequence. The activity uses its own simple programming language with instructions defined by the participants during the activity (thereby its own NM) to introduce sequence as well as iteration; the learners can write 'program' their own Fitness App using this programming language. In a programming classroom, these Fitness programs can be directly converted to an actual programming code, using a programming language familiar to learners. This is discussed further in the following section.

Teachers can use the Fitness Unplugged activity to model different varieties of counted loops, using different 'Fitness programs'. Teachers can scaffold the understanding how of iteration concept works in a program by building learners' understanding using Fitness programs with increasing complexity. For example, Figure 8.14 shows a simple Fitness program with a sequence of instructions: A green symbol to indicate the program start, 3 instruction cards with physical activities and an end symbol card to indicate the end of the program. In Figure 8.15, the 3 identical instructions are replaced with one instruction with a number indicating the number of times it should be repeated. This activity easily shows the use and importance of iteration, while learners can playfully engage in the physical activity given by the program. Teachers can encourage learners to design their own Fitness programs with various actions instructions.

Figure 8.16 shows an advanced Fitness program. In this program, a hula hoop is

[42] Fitness Unplugged: https://www.csunplugged.org/en/topics/kidbots/unit-plan/fitness-unplugged/

used to represent an outer loop in the Fitness program, with two inner loops inside; the number next to the hula hoop (3) indicates the number of times the things inside the hoop should be repeated. Accordingly, a set of 10 star jumps and 5 beanbag catches are repeated 3 times, as controlled by the number of times of the hula hoop. Scratch programs for these examples are given in the next section.

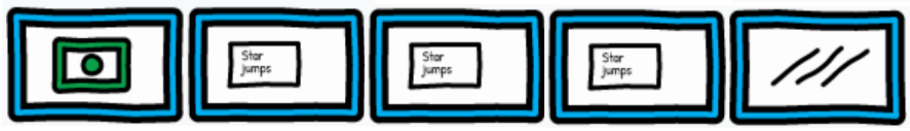

Figure 8.14: Fitness Unplugged program with no loops

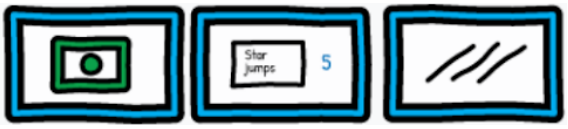

Figure 8.15: Fitness Unplugged program with a simple counted loop. The number before the instruction indicates the number of times the action is repeated

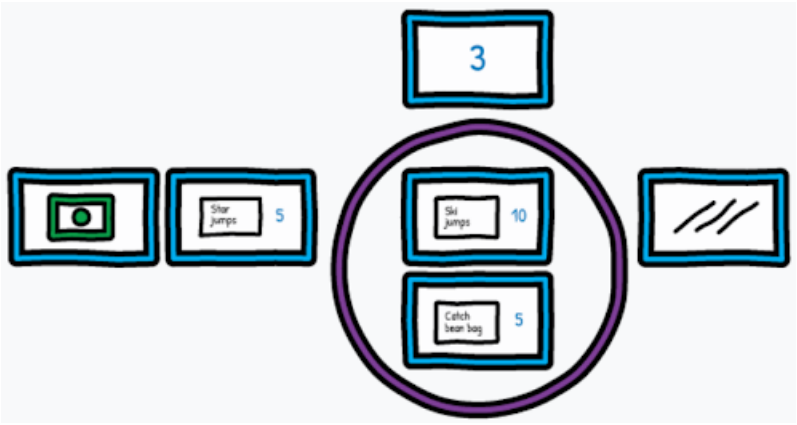

Figure 8.16: Fitness Unplugged program with a nested loops. The hula hoop represents an outer loop with two inner loops inside.

If Programming Unplugged activities that model iteration can also be used to model different varieties of counted of loops distinctively, they can be more useful in a programming classroom. For example, the activity 8 in Table 8.4 is an example of a Programming Unplugged activity that is designed to model the concept of a 'for' loop with different

starting, stopping and interval values. Converting such distinct activities modeling each kind of loop to programs would be much more straightforward and thereby, can be used as tools to make meaningful connections between a learner's mental model and the NM.

A conditional loop can be modeled using the Fitness Unplugged programs by introducing a condition in place of the control variable in the loops. For example, the number 5 in Fitness Unplugged program in Figure 8.15 could be replaced with a condition such as "until a whistle is blown", which effectively communicates the conditional concept of a loop. However, using such non-Boolean conditions may not be useful in a computer program, if the activity is to be used as a programming exercises and it can potentially convey the misconception that the loop is exited as soon as the whistle blows. Moreover, since different programming languages use different kinds of constructs to implement loops (e.g. Scratch only has 'repeat', 'repeat until' and 'forever' blocks to implement loops), a Programming Unplugged activity dedicated to a certain type of a loop construct may cause confusion among novice learners as well as teachers if the programming language does not support that kind of loop structure.

Plugging it in

The Fitness Unplugged programs can be directly used as plugged-in programming exercises in a programming classroom since they contain only count-controlled loops in their published form. However, the implementation of loops depends on the programming language used; different programming languages may use different loop constructs to implement the same logic; most languages use more syntax for counting loops. Figure 8.17 shows three Scratch programs that implement the Fitness Unplugged programs in Figures 8.14, 8.15 and 8.16. The sequence of actions in the program in Figure 8.17 (a) is replaced with a repeat block in (b). The program in Figure 8.17 (c), which uses nested loops, implements the Fitness Unplugged program in Figure 8.16. When the unplugged activity is followed by the plugged-in exercise, the learners are aware of how the values of the control variables have affected the instructions sequences in the respective loops, and therefore, they can easily understand how the different loops work in their program code. Introducing loops on a counter helps to illustrate Denning's point that a idea in CT is the ability to of a CA to do a lot of computation in a short time/fast.

8.5 Using Unplugged Activities for Programming: Concerns and Considerations

With a novice or beginner learner, an immediate next phase from an unplugged activity would ideally be to reflect a sort of mental imaging of the concept that they have just learnt, rather than trying to develop an algorithm mentally. For example, to answer "what would be the number is after 16?" after five cards are opened in the Binary

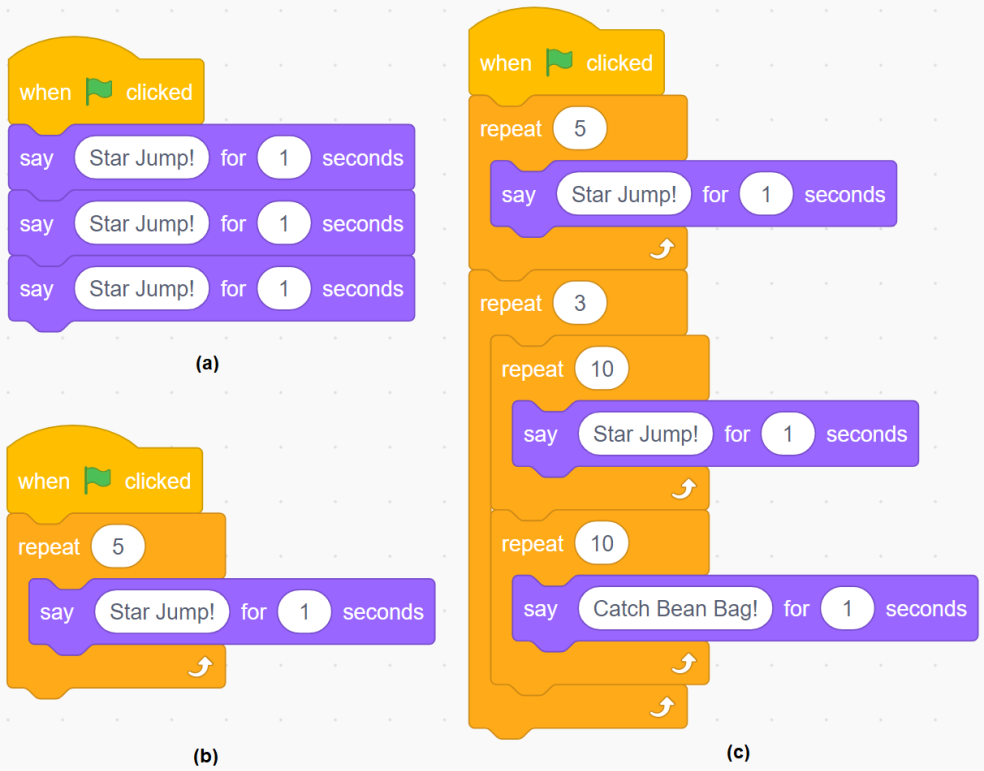

Figure 8.17: Fitness Unplugged program with a nested loops. The hula hoop represents an outer loop with two inner loops inside.

Cards unplugged activity, one learner would say 32 while another would hold up the card with 32 dots to show 32. A concern with unplugged activities would be that, at what point should one move back and forth between unplugged and plugged in learning? The answer may depend on many things like the stage of learners, what they are learning, how complex the question/lesson is, availability of the next resource, etc. A teacher may find it natural to alternate frequently between unplugged and plugged-in to scaffold new concepts and then implement them.

To a beginner, a plugging-in exercise to immediately follow an unplugged activity whenever possible would be more helpful in making more sense of the lesson: the physical, hands-on unplugged activity emphasises the realisation of a concept, while the plugging-in exercise bringing the conceptual understanding to its actual application to cement the learning. Learning that way can prevent some of the common misconceptions from forming. This nature of unplugged learning is rationalised by Fincher et al.'s study on NMs in teaching computing, suggesting that unplugged activities have "often relied on analogy to make salient and visible some aspect of the largely hidden underlying system" [90] .

Detailed descriptions of unplugged activities designed to model a particular programming concept are rare both in the literature and on the internet. Instead, physical analogies or metaphoric references are used and/or given as unplugged examples often. In such situations, it is important to pay attention to avoid misconceptions (particular to programming) from forming, as well as to explore the possibilities of using unplugged experiences to overcome them. Well designed unplugged experiences are a promising approach in avoiding some misconceptions being formed as well as unlearning the existing misconceptions. Furthermore, some programming constructs can almost look alike to a novice learner (e.g. functions and objects have a lot in common, both iteration and recursion use repetition, and repetitions and functions can avoid repeating code). These subtle distinctions may not be apparent to a beginner learner when they see them in program code, due to reasons like their level of understanding, unrefined mental models and possibly overwhelming information load (i.e. syntax and semantics of the programming language). Unplugged activities can act as the go-between in many of these situations, by providing beginner programmers opportunities to distinguish concepts from programming language constructs.

The Programming Unplugged activities[43] refined and developed during this study attempt to model programming concepts in particular, with the aim of being used to introduce a programming concept prior to plugging-in exercises. The activities designed use commonly available, low cost material and are easy to implement. Another concern of using unplugged activities in a classroom is to address time constraints, because doing an unplugged activity can be time consuming. However, it was observed during the introductory programming workshops we conducted (discussed in Chapter 9) that these unplugged activities can be incorporated in a programming lesson within a similar timeframe as a plugged-in only lesson, and with similar results. Further discussion in this regard is in Chapters 9 and 10.

8.6 Conclusions

When using unplugged activities for teaching programming, it is important that the focus stays on the programming concept, thereby the learners would not get overwhelmed by more general computer science concept(s) than the activities can possibly model. This is also why many published unplugged activities do not qualify to be effective in teaching a particular programming concept, since they communicate a general CS concept rather than a specific programming element. While using such activities could be used to teach a particular programming concept, the teacher may need to focus the learners' attention on the (possibly overshadowed) programming aspect, that could result in an excess cognitive

[43] Details of these activities can be found in the working document of Programming Unplugged activities at https://bit.ly/3NrueI5

load on the learner.

When combining an unplugged activity in a programming lesson, the learning happens mostly by the activities playing a representational role (by means of a CA) in a learner's mental model development. Intrinsically, a carefully designed unplugged activity that makes use of learners' existing knowledge to establish new knowledge helps a learner to develop a mental model that is closer to the NM, by physical engagement and/or with the aid of a computational agent (as discussed in Chapter 7, Section 7.4.1). A teacher or an expert's guidance and explanations are naturally necessary when a lesson is building upon learners' prior knowledge, in order to make necessary connections to an underlying computing context. In the absence of such meaningful connections, an unplugged experience could be left abandoned both conceptually and contextually.

Considering the unfamiliar and invisible nature of NMs, the physical and kinesthetic unplugged activities can communicate such concepts to learners with a sense of tangibleness, and more importantly without using a particular programming language. However, the long term effect of unplugged computing in mental model formation in a (possibly young) learner, and the resultant effects in their NMs when they become experts could not be observed in short term studies but should be part of a longitudinal study. Some initial evidence of this was given by Hermans and Aivaloglou in [115].

Chapter IX

Teaching Teachers Programming with Unplugged: Experimental Studies

Computational Thinking concepts are now part of the New Zealand School curriculum. Research has indicated that student engagement in Computer Science and programming lessons in New Zealand schools has been consistently high when adequate support is provided to enable teachers to successfully integrate Computer Science and programming in their lessons [76]. A range of professional development (PD) programmes conducted by the Computer Science Education Research Group (CSERG) of the University of Canterbury aims to provide the support needed for teachers, particularly those who have no computing qualifications or background, and therefore participants of such PD programmes were well suited for the kind of experimental studies of this research.

Accordingly, two main experimental studies were conducted alongside such PD and introductory programming courses in 2021. They were conducted as the evaluation phase (phase 5) of the review study discussed in Chapter 8, and were also used to trial the Programming Unplugged activities that were developed. How the two studies were conducted, the nature of the data collected, and how they were analysed are discussed in this chapter. The experiments attempt to find answers to RQ 2, i.e. how unplugged content used as a professional tool impacts teachers, and to find partial answers to RQ 3 by investigating why unplugged is useful when combined with programming. The possible implications of the results of the individual studies, and their limitations, are also discussed, and conclusions with respect to the research questions are presented in this chapter.

9.1 Aims of the Studies

The main aims of these two studies were: 1) to investigate how the use of unplugged content as a professional development tool impacts teachers' confidence, expectations, and knowledge, 2) to trial the Programming Unplugged activities developed in this project and 3) to investigate why unplugged is useful when combined with programming.

Another aim was to investigate the impacts of alternating unplugged teaching methods with conventional plugged-in methods when providing PD for teachers (RQ 2.1). The investigation on using unplugged activities was two-fold: 1) alternating unplugged

activities in introductory programming, and 2) using unplugged activities to introduce Computational Thinking (CT) concepts.

The two studies were also used to investigate the usability of the Programming Unplugged activities developed in this project (as discussed in Chapter 8). Accordingly, the findings of these studies were also intend to provide partial answers to RQ 3.2 of the research questions. Observational data from these two studies were also used as the main data source for the Semantic Wave analysis discussed in Chapter 10.

9.2 Participants

The participants of the two studies were pre-service teachers and in-service teachers of New Zealand. Section 9.5 of this chapter discusses an introductory programming workshop conducted for the pre-service teachers and Section 9.6 discusses a PD workshop in introductory programming for in-service teachers. All the participants are addressed as the 'learners' in this analysis.

All the participants were requested to respond to a set of identical surveys (as discussed in Section 5.2.3, and the survey instruments are given in Appendix B) at different stages of their workshops: Pre, post, and in one case also between sessions. These surveys included three instruments to measure: 1) the participants' teaching self-efficacy, 2) computer programming self-efficacy and 3) motivation to teach the DT component in the NZ school curriculum.

The same expert team described in Chapter 8 (Section 8.3) participated in the operations of the workshops. All the workshops were conducted by the same Instructor, who is a Computer Scientist (and also one of the supervisors of this research project). He tried to maintain an equal pace of delivery in all the workshops, which was nearly successful. In all workshops, the participants were also supported by two knowledgeable helpers (two expert primary school teachers) who kept their involvement to a minimum unless requested by the learners. They roamed around the room, providing support on request. Helpers kept their interference to a minimum during instructor's presentations.

Two observers (the researcher herself and an expert high-school teacher) took detailed notes of all the sessions. Their observations ranged from general behavior and vibe of the classroom, to noteworthy questions, gestures, 'Ah-ha' moments, engagement, etc. The observational notes were recorded with written entries every 5-7 minutes and/or with any noteworthy observations otherwise. The observations made by the instructor were recorded at the end of each session, as a recorded discussion between the instructor and the researcher.

9.3 Unplugged Materials Used

A similar set of unplugged activities were used in both the studies. The Non-programming Unplugged activities that were used in the CT concepts introduction sessions were the Binary Cards activity, Parity Magic Trick, Bar-code Magic Trick, QR Code Activity, and Sorting Networks. The Programming Unplugged activities used in the introductory programming session were Kidbots (for 'sequence'), Variables Dice battle (for 'variables'), Conditional Dice Games (for 'selection') and Fitness Unplugged (for 'iteration'). The details about these activities are given in Section 5.2.1. Moreover, the Instructor discussed several jargon words that are commonly used in computing and are useful in programming (such as sequence, selection and iteration), which are also either related to or included in the DT curriculum.

9.4 Data Analysis Approach

Given the multiple cases of data for each participant, multi-level modelling (MLM) is used in the quantitative data analysis of the two studies (as explained in Section 5.2.3). A Linear Mixed Effect Model (LME) was used to analyse the data of three surveys used in both the studies. LME is capable of estimating the imbalances of the dropouts in the repeated measures while adjusting the correlation. Model selection included checking for interactions among the fixed effects, and checking the necessity of including the random intercept. The three surveys used were teaching self-efficacy (TE), programming self-efficacy (CPE) and motivation to teach CT topics (M) (see Appendix B.) The statistical mean of each survey were considered separately.

In the LME model, participants (P) was considered as the random effect, since the subjects vary randomly and presumably have their own personal values. By including the participant effect in the model, a hierarchy in the experiment was taken into account; because there are repeated measures over time (within subjects), the model separates the noise between participants (i.e. subjects) as a random effect. In other words, presumably the participants' personal values on things such as a level of understanding of an individual survey item or the distribution of Likert vary randomly, the participants (P) are considered as a random effect on the model.

When assigning numerical representations to Likert scale measures, the choices were assumed as equally distanced values. The individual mean score of each participant under each survey is used as the dependant variable(s) (i.e. Mean-Teaching Self-efficacy, Mean-Computer Programming self-efficacy and Mean-Motivation) in individual analyses. By calculating a mean we assume that they are on a continuous scale (i.e. the mean can fall somewhere in between categories). However, this may not have been the perspective of the participants. They may have not responded considering the scale to be equally

distanced and the thresholds (for the feeling of confidence over something) between two individual participants may vary as mentioned earlier. Therefore, by giving a numeric value to a qualitative response, we assume that the responses are equally distanced, and that everyone responses around the same scale. Any missing responses (caused by participants skipping some questions without answering) were filled with the average over the sums of the available data.

9.5 Study I: Introductory Programming Course for Pre-Service Teachers

9.5.1 Overview

The first of the two experimental studies was an introductory programming workshop for the Bachelor of Education degree program and Postgraduate Diploma in Education students (herein after appropriately referred to as the 'student type') of the University of Canterbury, School of Education. All the participants were pre-service teachers, preparing to leave for their internship in schools. Although the workshop series was a compulsory part of their academic program, participation in the workshops was voluntary.

Collectively 152 education students participated in the experiment, out of which 63 were Bachelor's students and 89 were Postgraduate students. 30.1% of the Bachelor's students (19) and 26.9% of the Postgraduates students (24) indicated that they had prior knowledge of some programming language. 33.3% of the Bachelor's students (21) and 55% of the Postgraduate students (49) had indicated to have no prior programming knowledge A vast majority of the participants opted not to reveal their gender information (which was given as a response option in the demographic survey).

9.5.2 Method

Each batch of students were separated into three groups, totalling to six groups. All the groups had a couple of two hour workshop sessions each, adding up to a total of 12 2-hour workshops. The experiments were structured to cover introductory programming content in one session, and basic CT concepts in the other, across two weeks. The session arrangement for the six groups were as shown in Figure 9.1. Two approaches were used to introduce programming: 1) Alternating unplugged activities with plugged-in exercises (marked in mixed colour in Figure 9.1), and 2) Traditional plugged-in only approach without using any unplugged material (marked in red in Figure 9.1). The CT concept introductory sessions are marked in green in Figure 9.1.

Three different teaching approaches (hereinafter referred to as 'treatments') were used to introduce the basic programming concepts, so that one group from each student type (Bachelor and Postgraduate) were exposed to each treatment. The three treatments used

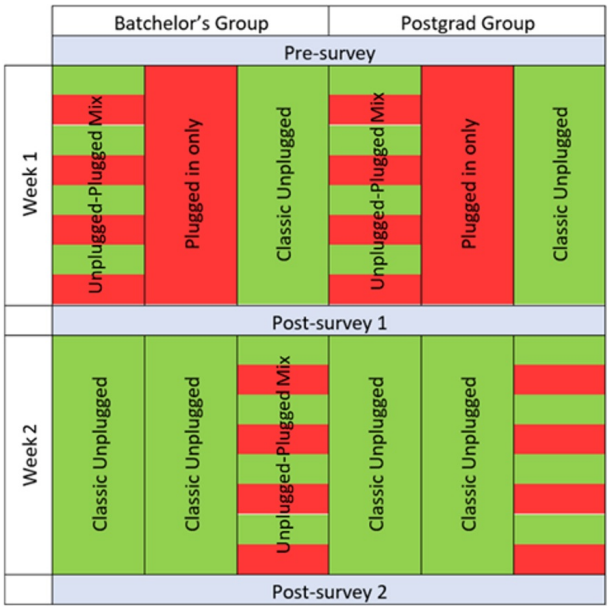

Figure 9.1: Grouping of participants into classes and treatments; unplugged components are shown in green, and plugged-in components in red

in this experiment were:

Treatment 1 Introduce programming concepts using alternating unplugged activities with plugged-in exercises in week 1, and CT concepts in week 2,

Treatment 2 Introduce programming concepts using a traditional plugged-in only approach in week 1, and CT concepts in week 2,

Treatment 3 Introduce CT concepts in week 1, and programming concepts using alternating unplugged activities with plugged-in exercises in week 2.

The overall structure of the class arrangement for different groups is shown in Table 9.1.

Table 9.1: The Experiment Set-up

Group	Treatment 1	Treatment 2	Treatment 3
Bachelors (B)	Class B1	Class B2	Class B3
Postgraduate (PG)	Class PG1	Class PG2	Class PG3

All the participants were requested to respond to a set of identical surveys prior to the workshops (Pre-survey), after the first week (Post-survey 1) and after the second week of the workshop (Post-survey 2).

9.5.3 Limitations

These workshops were conducted as part of an ongoing academic course for Education students at the University of Canterbury, School of Education. The experiments were used to trial the Programming Unplugged activities, and to study the effectiveness of alternating unplugged with plugged-in versus plugged-in only approaches. Considering the limitations of time and availability of participants, the three different treatments used were chosen as best combination for capturing the effectiveness of delivery as well as fair coverage of content for all participants. Thus, it is noted that all possible combinations of treatment formats could not be trialled. Accordingly, introducing CT concepts prior to introducing programming concepts could be achieved with only one group, and could not be tested with both introductory programming approaches. Further, the workshop sessions were limited to only four hours (2 hour sessions in two weeks), and participants' programming skills were not evaluated using any test or examination.

Several observations regarding the overall conduct of the class made at the very first workshop session were identified to provide some critical adjustments to the remaining sessions. For the same reason, the first session outcomes did not run efficiently as hoped. These observations and the respective adjustments are detailed under general observations.

These workshops were conducted as the last remaining compulsory sessions of the semester. It was also observed that some other assignment due dates were fast approaching for the learners, which was barrier for maintaining their full attention during the sessions. Attendance was observed to be lower and less regular than expected in all six groups.

9.5.4 Workshop Observations

General Observations

Some attendees openly expressed their reluctance/dislike in participating in a workshop on programming targeting the Digital Technologies (DT) component of the NZ curriculum. The majority of the participants were not very much aware of the DT curriculum or its Progress Outcomes.

Several observations by the Instructor and observers (i.e.general behaviour, gestures noticed and responses given by the participants) about the learning process were noted during the first session (Class PG1: Treatment 1) that needed to be adjusted for the remaining sessions. One main observation was defining the role of two dedicated note takers. Another notable adjustment was to include Parson's problems[1], which made

[1] Parson's Programs provide learners with instructions of a program in a jumbled order, where they have to rearrange in an order that would make the program work. The puzzle-like format allows

introducing the novice learners to programming significantly easier and more effective, while introducing them to the Scratch programming environment at the same time. For this reason, less weight was put on the outcome of the first session.

Participants who were new to Scratch seemed to spend the majority of their time figuring out the basics of the programming environment rather than the programming concepts. Some participants were fascinated by the environment design tools (i.e. backgrounds, Sprites, costumes, etc.) and seemingly distracted from the Instructor's discussion by them. Some seem to assume that such background design work was an integral part of computer programming, and the introduction of Parson's problems in the remainder of the workshops was very effective in overcoming this issue. With the use of Parson's problems, the instructor could focus the learners objectively towards the expected outcome of the lesson, without them being carried away by the background effects or being distracted by trying to figure-out the Scratch environment.

Programming concepts were introduced at a much higher pace in the plugged-in only treatment (treatment 2) than the alternating unplugged treatments (treatments 1 and 3). The participants with no Scratch awareness had more time to experience the interactive programming environment before moving on to deeper concepts, which may have resulted in a faster learning process later on. Although much more content was delivered during the alternating unplugged sessions (i.e. unplugged content, introduction to the programming environment and plugged-in exercises) within the same time span, the approach may have overwhelmed some of the participants.

At some points during the plugged-in only sessions (treatment 2), it was observed that the participants were losing interest. Despite the observers noting positive engagement and a reasonable pace, learners looked tired towards the end of the session. At times it was difficult to judge by their expressions whether their engagement was synchronising with the Instructor 's discussion. Most of the plugged-in session participants needed support from the helpers.

Compared to treatment 1 (i.e. programming concepts introduced in week 1), treatment 3 (i.e. CT concepts introduced in week 1) participants seem to embrace alternating unplugged activities more smoothly and showed no sign of being overwhelmed during the programming sessions. Treatment 3 introduced CT concepts prior to the programming concepts and therefore the participants were familiar with the unplugged style. The transition to plugged-in exercises from the unplugged activity was also smoother with the treatment 3 groups, than with the treatment 1 groups.

In both alternating unplugged treatments, most of the learners participated actively in the activities and were engaged in constructive discussions among themselves as well as with the Instructor. It was observed that some learners started to successfully work

learners to practice basic programming in an entertaining manner.

with the Parson's problems ahead of the Instructor's guidance. However, the overall enthusiasm among the Bachelor's groups was less than the PG groups that participated in similar treatments. The average age of the Bachelor's groups was also observed be lower than the PG group. The PG groups showed higher enthusiasm (i.e. active participation, asking questions about both the concepts and their applicability in their own classrooms, etc.) despite their lack of prior programming knowledge.

The time limitation and additional material delivery did not seem to synchronise well for many of the participants who were new to the Scratch environment.

Observations on Non-programming Unplugged Activities

There were no noticeable distinctions observed between the two types of students during the CT concepts introduction sessions (i.e. during Non-programming Unplugged activities). The initial enthusiasm was a bit low in all groups, with a few less responsive and unengaged students at the beginning. The engagement visibly improved with time where responses like "I get it", "Hmm!" (sound of understanding something), "Yay!", "Cool!" and sounds of awe could be heard from among the class. Almost everyone was fully engaged as the classes progressed, in every group. The level of engagement varied slightly depending on the ongoing activity. However, overall engagement and enthusiasm was higher during learning CT concepts than during programming sessions.

Learners asked good questions related to unplugged activities ranging from how to use them in general computing teaching, to their use in special situations such as with autistic students, which indicated that they have realised the objective of the activities and their pedagogical application in a classroom context. The sorting Network activity[2] was observed to be the most engaging activity. Due to the range of items sorted using the Sorting Network (e.g. words, musical notes), it was rather easier for the participants to initiate discussions of how computing concepts can be integrated with other curriculum areas (broader context and deeper thinking).

The Binary Cards activity[3] had similar good responses, however probably due to the lesser opportunity for curriculum integration (possibly limited to maths mostly) fewer learners were actively participating in the discussion. Some learners seem confused during the activity. Their questions indicated that they were confused with the order of the bit arrangement (least and most significant bit), the purpose of the activity, and for one student, the binary system itself. However, simple discussions with the Instructor or among peers seemed to solve these issues easily.

During the Parity Magic activity[4], members of some groups were observed to be

[2] Sorting Networks: https://classic.csunplugged.org/activities/sorting-networks/

[3] Binary Numbers: https://classic.csunplugged.org/activities/binary-numbers/

[4] Parity Magic: https://www.csunplugged.org/en/topics/error-detection-and-correction/

participating without a clear understanding of the purpose of the activity. The Instructor had the full attention of the class during the product code explanations, probably due to the relatedness of the concept to real life drawing the participants' attention. The QR Code activity[5] was engaging as a group activity and received their full attention during the Instructor's explanations. Discussions about the applicability of CT as well as about the activity could be overhead among group members during the activities. However, these discussions were only limited to general application of the error-detection concepts, and not directed towards programming or implementation of a QR code reader. One person asked a pointed question 'what are we trying to achieve by showing this to kids?', indicating the teachers' confusion in connecting the unplugged computing activity context to the curriculum context.

Overall enthusiasm (observed through their active participation in the discussions, objective questions asked, noticeable reactions made, etc.) was higher in all PG groups than their respective Bachelor's groups.

Observations on Programming Unplugged Activities

The Programming Unplugged activities were introduced by the Instructor in the order he was introducing the basic programming concepts. Accordingly, the intended order of activities were Kidbots (sequence), Variable Dice Battle (variables), Conditional Dice Games (selections) and Fitness Unplugged (Iteration). This may seem like an ambitious agenda, but the instructor did not rush to accommodate all the concepts in one session, yet managed to cover most of them. However, with the time limitation of the workshops, the Fitness Unplugged activity could not be used in any of the classes. Nevertheless, the Instructor introduced the basic idea of the concept of 'iteration' to the learners during the final review discussion. Each activity was followed by a corresponding plugged-in exercise.

Moreover, many learners had already started using the 'loop' constructs (i.e. the 'repeat' blocks) of Scratch during the Conditional Dice Game activity's plugged-in exercises, intuitively trying to use them to improve their programs even without a specific introduction to either the 'iteration' concepts or the 'loop' constructs. Therefore, the Instructor could introduce the concept of iteration towards the end of the lesson without using the intended Programming Unplugged activity or its corresponding programming exercise. Consequently, no proper observations could be made about using the improved version of the Fitness Unplugged activity for iteration. Nevertheless, the learners had advanced to a level where they could successfully understand the concept, but how they

`unit-plan/parity-magic/`

[5] QR Code activity: Scanning a printed QR Code, and gradually changing the dots until it failed to scan.

would implement the concept in a program could not be objectively observed.

Observations on Kidbots - The activity and the subsequent plugged-in exercise received full engagement throughout. A few learners (2-3) did not actively participate in *doing* the activity but were observing objectively (perhaps feeling the activity was too childish to actually participate or just prefer to observe not participate). The whole class actively participated answering questions and having discussions both with Instructor and among themselves. Few learners had previous experience with the activity. Some groups asked questions from helpers mainly clarifying the execution of the activity. When the Instructor explained the reason to have a separate 'programmer' and 'tester', and explained the concept of 'debugging', some learners' facial expressions showed new understanding/realisation.

Observations on Variable Dice Battle - Despite one or two learners' slight disengagement at the beginning of the activity, all learners seemed well engaged and successfully completed both the activity and the corresponding Parson's problem that followed. Several big "Ah-ha!" moments regarding understanding a variable's behaviour were overheard among learners. At one point, one learner started using the whiteboard instead of flip-cards, which opened a small discussion on realising the possibility of alternative props, about whether the whiteboard is a good alternative. One group started playing without labelling the flip-cards, leading to a discussion on the importance of proper labelling of variables in computer programs. In general there was visible enthusiasm among group members with questions like "How do you do that?", "Is this how this [variable in Scratch] works?" and clarifications among themselves. A smooth transition to programming was observed.

Observations on Conditional Dice Games - Learners approached this more like a scaffolding from the Variable Dice Battle activity. The transition from the Variable Dice Battle activity to its corresponding plugged-in exercise was observed to be very smooth. Only a few learners needed scaffolding during the plugged-in exercises (e.g. some learners were trying to introduce a new 'variable' instead of 'if-else block'). A high level of peer discussions within and outside groups was observed. This activity was designed to stepwise scaffold ideas, and some learners seem to miss this due to their own distractions. Moreover, it was observed that several groups mixed up the scaffolding order of the cards, yet with no noticeable disadvantage. The Instructor introduced the corresponding Parson's problems while the groups were working on game cards (i.e. moved to plugged-in exercise before wrapping-up the unplugged activity) but the transition was smooth, and almost all the learners caught up with the pace. The introduction of Parson's Problems was observed to be very effective when connecting the Programming Unplugged activity to plugged-in exercises. This improvement effectively helped learners' understanding of the environment without feeling overwhelmed or being carried away with less important background detail while retaining their attention on the programming task. Despite most

of the participants' very low previous programming experience and most of them being new to Scratch, Parson's problems were observed to maintain continuous high attention and engagement.

9.5.5 Results Analysis

The following parameters were used in the Linear Mixed Effect model for the survey results analysis of this study.

- Survey types (ST): pre-survey, post-survey 1 (after first week), post-survey 2 (at the end)
- Treatment types (TT): treatment 1, treatment 2, treatment 3 (as explained in Section 9.5.2)
- Group types (GT): Bachelor's and Postgraduate.

In all individual analyses, the survey type (ST), treatment type (TT) and group (GT) (with inter-dependencies GT x TT, ST x TT, and GT x ST x TT) were considered to have fixed effects on the model. The reasons for including the inter-dependencies into the model are to consider the influence of the following relationships to the measured parameter:

1. GT x TR - the influence of a particular peer group in a particular treatment,
2. ST x TR - the influence of response over the course of the two workshops in a particular treatment, and
3. GT x ST x TR - the influence of a particular peer group over the course of the two weeks in a particular treatment.

In each individual model, fixed effects are considered to have levels that are of primary interest and would be used again if the experiment were repeated. Random effects have levels that are not of primary interest, but rather are thought of as a random selection from a much larger set of levels. Accordingly, the following model (equation 9.1) has been derived and adapted in all three individual analyses.

$$\begin{aligned} Y_{ijk} = {} & \alpha + \beta_i \times group_i + \gamma_j \times survey_j + \delta_k \times treatment_k \\ & + \theta_{ij} \times group_i \times survey_j + \lambda_{ik} \times group_i \times treatment_k + \eta_{jk} \times survey_j \times treatment_k \\ & + \mu_{ijk} \times group_i \times survey_j \times treatment_k \end{aligned} \tag{9.1}$$

SPSS[6] software was used for data analyses and by default, the parameter value that comes last in the alphabet is considered to be the reference level. Accordingly, the model has considered GT=PG, ST=Pre-survey, and TR=Treatment 3 as the references. Table 9.2 contains a summary of the three estimated fixed effect models used in this analysis. The intercepts of all three models are significant (p=0.00). The detailed models are given in Appendix D.

Initial analyses were conducted to test the effect of the weekly order of the workshop formats used within the treatments (i.e. introduce computational thinking in the first week and programming concepts in the second week). The workshop order did not effect any of the models (i.e. TE, CPE and M), thus the workshop order difference effect was dropped from the subsequent analysis. It is worth mentioning that the practical limitations faced in conducting the experiment (i.e. lack of time and access to larger number of participants) resulted in a lack of data points for some parts of the model. Many of the observations did not show statistical significance due to this reason.

When assigning numerical representations to Likert scale measures (i.e. values from 1 to 4, effect varying from the least to the most), the choices were assumed to be equally distanced values. The individual mean score of each participant under each survey is used as the dependant variable(s) (i.e. Mean-Teaching Self-efficacy, Mean-Computer Programming self-efficacy and Mean-Motivation) in individual analyses. By calculating a mean, it was assumed that they are on a continuous scale (i.e. the mean can fall somewhere in between categories; so the model assumes that the scale is continuous). However, this may not have been the perspective of the participants. They may have not responded considering the scale to be equally distanced and the thresholds (for the feeling of confidence over something) between two individual participants may vary. Therefore, by giving a numeric value to a qualitative response, it was assumed that the responses are equally distanced, and that everyone responded around a similar scale. The missing responses (i.e. questions the participants has not answered) were filled with the average over the sums of the available data.

[6] IBM SPSS software: https://www.ibm.com/analytics/spss-statistics-software

Table 9.2: Summary of Estimates and Standard Errors (in parentheses) of Fixed Effects by Model - Pre-service Teachers

	Group type	Survey Type	Treatment	Group x Survey Type	Group x Treatment	Survey Type x Treatment	Group x Survey Type x Treatment
Model 1: Teaching Self-efficacy *Intercept 2.041 (0.109)*	B ↓ -0.105 (0.215)†	P1 ↑ 0.386 (0.136)* P2 ↑ 0.425 (0.149)*	T1 ↑ 0.185 (0.154)‡ T2 ↑ 0.340 (0.168)*	B P1 ↑ 0.294 (0.245)† B P2 ↑ 0.586 (0.260)*	B T1 ↑ 0.007 (0.283)† B T2 ↓ -0.420 (0.280)†	P1 T1 ↓ -0.14 (0.190)† P1 T2 ↓ -0.43 (0.206)* P2 T1 ↑ 0.139 (0.209)† P2 T2 ↑ 0.119 (0.217)†	B P1 T1 ↑ 0.397 (0.259)† B P1 T2 ↑ 0.994 (0.234)* B P1 T3 ↑ 0.294 (0.245)† B P2 T1 ↑ 0.359 (0.247)† B P2 T2 ↑ 0.622 (0.269)* B P2 T3 ↑ 0.586 (0.260)*
Model 2: Programming Self-efficacy *Intercept 1.885 (0.260)*	B ↑ 0.535 (0.516)†	P1 ↑ 0.966 (0.331)* P2 ↑ 2.309 (0.362)*	T1 ↑ 0.120 (0.369)† T2 ↑ 0.159 (0.401)†	B P1 ↓ -0.240 (0.595)† B P2 ↓ -0.176 (0.631)†	B T1 ↓ -0.130 (0.678)† B T2 ↑ 0.152 (0.672)†	P1 T1 ↓ -0.057 (0.462)† P1 T2 ↓ -0.216 (0.501)† P2 T1 ↓ -0.822 (0.506)† P2 T2 ↓ -0.612 (0.526)†	B P1 T1 ↑ 0.818 (0.625)† B P1 T2 ↑ 0.440 (0.569)† B P1 T3 ↓ -0.240 (0.595)† B P2 T1 ↑ 0.717 (0.596)† B P2 T2 ↑ 0.654 (0.653)† B P2 T3 ↓ -0.176 (0.631)†
Model 3: Motivation *Intercept 3.290 (0.239)*	B ↓ 0.941 (0.486)*	P1 ↓ -0.199 (0.516)† P2 ↑ 0.208 (0.540)†	T1 ↓ -0.103 (0.537)† T2 ↓ -0.134 (0.508)†	B P1 ↓ -0.748 (0.624)† B P2 ↑ 0.772 (0.660)†	B T1 ↑ 0.545 (0.636)† B T2 ↑ 0.387 (0.627)†	P1 T1 ↑ 0.786 (0.743)† P1 T2 ↑ 0.233 (0.678)† P2 T1 ↑ 0.205 (0.718)† P2 T2 ↑ 0.402 (0.771)†	B P1 T1 ↓ -1.288 (0.888)† B P1 T2 ↓ -0.4 (0.861)† B P2 T1 ↑ 1.008 (0.892)† B P2 T2 ↓ 1.178 (0.950)†

1. Reference Levels:
   Treatment = Treatment 3 (T3); Group Type = Postgraduate group (PG); Survey Type = Pre survey (Pre)
   Group x Survey Type = PG x Pre; Group x Treatment = PG x T3; Survey Type x Treatment = Pre x T3; Group x Survey Type x Treatment = GP x Pre x T3
2. Content: Estimates and standard errors (in parentheses)
3. $^{\dagger}p>0.1$, $^{\ddagger}p<0.10$, $^{*}p<0.05$

9.5.5.1 Computational Thinking Teaching Self-Efficacy

Row one of the Table 9.2 contains a summary of the estimated fixed effect model for Computational Thinking Teaching self-efficacy. The full estimated fixed effects table, which gives the p-values plus estimates of the effect sizes and 95% confidence intervals for those estimates for the CT teaching self-efficacy is given in Appendix D. The following observations were made:

- The Mean-score of the Bachelor's group is 0.105 scores lower than the PG group. However, this difference is not statistically significant (p=0.624).
- Compared to the pre-survey, the mean-score for the post-survey 1 is 0.386 higher with a significance of p=0.05 and to Post-survey 2 is 0.425, with a significance of p=0.05. Both post surveys show a statistically significant increase compared to the pre-survey results for teaching efficacy.
- Compared to treatment 3, the mean-score for treatment 1 is 0.185 higher with a significance of p=0.230. There is no statistically significant difference between treatment 3 and treatment 1. However, the mean difference between treatment 2 and treatment 3 is 0.340, with a significance p value of 0.043, which is less than 0.05, and therefore, there is a significant difference between treatment 3 and treatment 2.
- Compared to the PG group, the Bachelor's Group shows a 0.294 mean difference between their pre-survey results and post survey 1 results with a p value of 0.233. This is not a significant difference. The difference between pre-survey and post survey 2 shows a 0.586 difference with a p=0.026 significance. This difference is significant, with a significance of p<0.05.
- Compared to the PG group's self-efficacy after treatment 3, the treatment 1 Bachelors' group shows a slight increase in teaching self-efficacy (0.007).
- Compared to PG after treatment 3, Bachelors' show a higher average of -0.42 with treatment 2. However, as mentioned earlier, this is not a significant difference (p=0.135).
- Compared to treatment 3 pre-survey, the post-survey1 responses of the both of the other treatments show a decrease in self-efficacy (i.e. -0.14 for treatment 1 and -0.43 for treatment 2), but none of these observations are statistically significant. The same comparative observation for post-survey2 shows and increased self-efficacy for both treatments 1 and 2 (0.139 and 0.119 respectively) but with no statistical significance.

Accordingly, by including the above observations in model equation 9.1 we get:

$$\begin{aligned} Y_{ijk} &= 2.04 + 0.386 \times \textit{post-survey}1 + 0.425 \times \textit{post-survey}2 \\ &\quad + 0.340 \times \textit{treatment}3 + 0.586 \times \textit{Bachelors} \times \textit{post-survey}2 \\ &\quad - 0.43 \times \textit{post-survey}1 \times \textit{treatment}2 + 0.994 \times \textit{Bachelors} \times \textit{post-survey}1 \times \textit{treatment}2 \\ &\quad + 0.622 \times \textit{Bachelors} \times \textit{post-survey}2 \times \textit{treatment}2 + 0.586 \times \textit{post-survey}2 \times \textit{treatment}3 \end{aligned} \tag{9.2}$$

9.5.5.2 Computer Programming Self-Efficacy

Row two of the Table 9.2 contains a summary of the estimated fixed effect model for Computational Thinking Teaching self-efficacy. The full estimated fixed effects table, which gives the p-values plus estimates of the effect sizes and 95% confidence intervals for those estimates for the CT teaching self-efficacy is given in Appendix D. The following observations were made:

- The Mean-score of the Bachelor's group is 0.535 scores lower than the PG group. However, this difference is not statistically significant (p=0.301).
- Compared to the pre-survey, the mean-score for the post-survey 1 is 0.965 higher with a significance of p=0.004 and Post-survey 2 is 2.309, with a significance of p<0.05. Both post surveys show a statistically significant increase compared to the pre-survey results for teaching efficacy.
- Compared to the treatment 3, the mean-score for treatment 1 is 0.12 higher with a significance of p=0.745. Mean difference between treatment 2 and treatment 3 is 0.159, with a p value of 0.692. There is no statistically significant difference between treatment 3 and treatment 1, or treatment 2 and treatment 3.
- Compared to the PG group, the Bachelor's Group shows a -0.24 mean difference between their pre-survey results and post survey 1 results with a p value of 0.687 and between pre-survey and post survey 2 the difference is -0.176 with a p=0.781 significance, which are both not significant.
- Compared to the PG group's self-efficacy after each treatment, none of the treatments of the Bachelor's group show a significant difference in teaching self-efficacy (p=0.007).
- Compared to the treatment 3 pre-survey, the post-survey1 responses of the both of the other treatments show a decrease in self-efficacy (i.e. -0.216 for treatment 1 and -0.611 for treatment 2). But none of these observations are statistically significant. The same comparative observation for post-survey2 shows an increased self-efficacy for both treatments 1 and 2 (0.056 and 0.216 respectively) but with no statistical significance.

Accordingly, by including the above observations in model equation 9.1 we get:

$$Y_{ijk} = 1.885 + 0.966 \times \textit{post-survey}1 + 2.309 \times \textit{post-survey}2 \tag{9.3}$$

9.5.5.3 Motivation to Teach Computational Thinking

Row three of the Table 9.2 contains a summary of the estimated fixed effect model for Computational Thinking Teaching self-efficacy. The full estimated fixed effects table, which gives the p-values plus estimates of the effect sizes and 95% confidence intervals for those estimates for the CT teaching self-efficacy is given in Appendix D. The following observations were made:

- The Mean-score of the Bachelor's group is -0.94 scores lower than the PG group. However, this difference is not statistically significant (p=0.054).
- Compared to pre-survey, the mean-score for post-survey 1 is -0.199 higher with a significance of p=0.7 and post-survey 2 is 0.208, with a significance of p=0.7. Both post-surveys show no statistically significant difference compared to the pre-survey results for teaching efficacy.
- Compared to treatment 3, the mean-score treatment 1 is -0.103 and for treatment 2 is -0.134, both with no statistical significance (p>0.05).
- Compared to the PG group, the Bachelor's Group shows a 0.747 mean difference between their pre-survey results and post survey 1 results with a p value of 0.232, and between pre-survey and post survey 2 shows a difference of -0.772. None of these observations are statistically significant (p>0.05).
- Compared to the PG group's motivation after treatment 3, both treatments 1 and 2 of the Bachelors' group show an increase (0.545 and 0.386 respectively) but with no statistical significance.
- Compared to treatment 3 pre-survey, the post-survey1 responses of the both of the other treatments show an increased motivation (i.e. 0.786 for treatment 1 and 0.223 for treatment 2). But none of these observations are statistically significant. The same comparative observation for posts-survey2 shows and increased self-efficacy for both treatments 1 and 2 (0.205 and 0.407 respectively) but with no statistical significance.

Accordingly, by including the above observations in model equation 9.1 we get:

$$Y_{ijk} = 3.29 + 0.941 \times \textit{Bachelors} + 1.581 \times \textit{post-survey}2 \times \textit{treatment}2 \tag{9.4}$$

9.5.6 Discussion and Conclusion

The Bachelor's group indicated lower self-efficacy in both teaching CT as well as in programming, and they also showed lower motivation to teach DT as well. Although not statistically significant, this observation agrees with the in-class observations of low engagement and enthusiasm among the same group. The Postgraduate group consisted of more mature students with prior work experience, who have joined the education diploma with a more directed purpose of taking up teaching as a career option, which could be the reason for these differences.

Both post-surveys showing an increase in teaching and in programming self-efficacy (statistically significant in the former and not significant in the latter), indicate that the workshop has positive impact on all the learners' self-efficacy in teaching CT and in programming. This continuous increase over the course of the workshop, although slightly, may indicative of how workshops of this nature would positively motivate the teachers to teach DT subjects.

Between treatments analysis indicates that the traditional approach to introducing programming was more effective, as a treatment. However, with the absence of the unplugged content to introduce programming, treatment 2 carried less content compared to the other two treatments. Both alternating unplugged approaches covered more content than the traditional approach in a similar time-frame. Considering the nature of the participants (i.e. lack of experience in programming and the workshops were a compulsory part of their semester, regardless of the participation was voluntary), this could have been influential in this statistical indication of the response, due to the possibility of the participants' feeling overwhelmed by the content. Further, an approach of learning programming using plugged-in only followed by learning CT with unplugged was absent in the model, and it restricted the optimal utilisation of the statistical model as well as obtaining more insightful observations.

Introducing general computing concepts first and then moving to a programming approach (treatment 3) appeared to be slightly less effective than introducing programming first. Introducing programming before introducing CT concepts seems a better approach to increase learners' programming self-efficacy. However, their motivation to teach digital technologies increased when the CT concepts were introduced before the programming concepts.

When CT concepts were introduced using unplugged in teachers' PD courses prior to programming, teachers felt motivated to teach the subject. However, their teaching self-efficacy was higher (not significantly with alternating unplugged, and significantly with plugged-in only) when they were introduced to programming concepts prior to CT concepts. We conclude that, using unplugged content motivated teachers to teach DT subjects, and their self-efficacy towards teaching CT increased when they are equipped

with programming knowledge. The traditional approach increased teachers' programming self-efficacy more than alternating it with unplugged.

Introducing programming concepts in the first week (with or without unplugged content) has negatively effected the participants' teaching self-efficacy. However, after the second week, this has been changed to positive. Interestingly, the negative impact of introducing programming concepts in the first week on their programming self-efficacy recovered after they were exposed to CT content in the second week. Their motivation towards teaching DT has continuously been increasing through the two weeks, indicating that the more content knowledge they get for the subject, the more motivated they are to teach it. This indicates that when teachers are knowledgeable about the general CT content, they are more self-efficacious of their own programming ability as well as teaching CT.

The overall lower mean self-efficacy of the postgraduate students' towards teaching CT topics could be due to their realisation of either the importance and depth of the CT content in the curriculum, or the level of their understanding about teaching CT in a classroom. The significant increase indicated in post-survey2 responses shows that the workshop has been successful in improving Bachelor's groups the self-efficacy towards teaching CT.

These workshops covered a large amount of content knowledge in a short period of time. The postgraduate group showing comparatively lower efficacy towards teaching could be due to feeling overwhelmed, especially the more mature participants. The postgraduate group included students with experience in computing, teaching subject areas other than DT, or professional/work experience in some other area. Comparatively, the younger and inexperienced Bachelor's group could perhaps be less sensitive or feeling less concerned about their sense of responsibility for teaching.

Treatment 1 with the postgraduates was the first time the Instructor trialled the alternating unplugged approach with a newly developed set of activities with a learner group. Therefore, several observations leading to on-the-fly improvements were inevitable. Nevertheless, it must be noted that a better experimental design would have piloted it first to shake out these problems. However, as mentioned in the limitations, the experiment was run under severe external constraints, so this was not possible.

Many of the participants were new to the Scratch programming environment. Since unplugged activities need engagement, the initial distraction observed with this first group put them at a disadvantage compared to the other groups, which could have been reflected in the survey responses and effected the overall affects in the statistical model.

The Bachelor's group consisted of more participants with prior programming experience (mostly of a programming language other than Scratch) compared to the PG group (30.1% and 26.9% respectively). It was observed in the programming-only groups that those with prior programming experience were helping others during the workshops, and

also in that set up the participants' attention was not compromised for any unplugged activity, thereby they had more time for discussions among them. This could have been reflected in their post-survey responses. The participants with prior programming experience among the Bachelors' group could have either found the workshop less interesting or have developed a sense of responsibility in teaching programming.

9.6 Study II: Introductory Programming Course for In-Service Teachers

This section reports on a second set of workshops that took advantage of the learnings from the first ones, but were conducted for in-service teachers.

9.6.1 Overview

Two workshops for introductory programming for in-service, associate teachers were conducted as a PD support session for the associate teachers of the University of Canterbury, School of Education. The participation in the workshops was voluntary. The workshop sessions were four hours (2-hours introductory programming session and 2-hours CT concept session).

Thirty one (31) associate teachers participated in the workshops. Participants were grouped into two groups, as chosen by them at their programme registration. 83.9% of the participants (26) were female and 16.1% (5) were male. 74% (23) had prior experience in programming and 26% (8) had no programming exposure. Group 1 consisted of 15 participants and group 2, 16 participants.

9.6.2 Method

The two groups were given two different treatments. In the introductory programming session, alternating unplugged approach with plugged-in exercises was used with Group 1 and for Group 2, a plugged-in only approach was used with no unplugged activities. In the second session, Non-programming Unplugged activities were used to introduce the computational thinking CT concepts to both groups. For both groups, the introductory programming session was followed by the CT concept session. The Instructor tried to maintain an equal pace and delivery at each treatment type, which was a success. Table 9.3 shows the group composition.

All the participants were requested to respond to a set of identical surveys prior to the workshops (Pre-survey), and after the introductory programming session (Post-survey). These in-service teachers participants were invited to a follow up telephone interview with the researcher at the end of their school term, both to obtain their views of the usefulness of the resources used in the workshops, as well as to obtain any feedback if they had used any of the Programming Unplugged activities in their classes.

Table 9.3: Associate Teachers' Group Compositions

Group	Treatment	Gender	Prior Programming Knowledge
Group 1	T1 - Alternating Unplugged	Male: 2, Female: 13	Yes: 12, No:3
Group 2	T2 - Plugged-in Only	Male: 3, Female: 13	Yes: 11, No:5

Moreover, as part of this study, the expert teachers of CSERG, who also actively participated as helpers and an observer during the sessions, have been interviewed. These expert teachers of CSERG had many years of experience using unplugged activities in their classrooms and in teachers' PD in the Digital Technologies curriculum. Although they had limited chances to use the newly developed Programming Unplugged activities, their prior experience in using unplugged provided valuable insights in this discussion. Their opinions were recorded from two perspectives: 1) as an in-service teacher who uses unplugged activities in their programming classrooms, and 2) as observers of the PD workshops.

The interviews were semi-structured and were conducted in a similar way to the interviews described in Chapter 5. Questions were open ended, where teachers were allowed to freely express their views. However, sometimes the interviewer had to provide further clarifications to questions and/or terms used in the questions.

9.6.3 Limitations

Soon after the workshops, the first delta variant COVID19 case was reported in New Zealand, which led to several restrictions imposed within the country including moving back to online-schooling. The sudden pressure on the teachers' everyday schedule changes adversely affected the feedback interviews of this study. Participation in the interviews were very low, and those few teachers who responded did not have time to use many of the resources from the workshops in their classrooms, thus could not give any feedback on their practical experience.

9.6.4 Results

9.6.4.1 Surveys

Three LME models were used in this study to find the relationship between the dependant variables teaching self-efficacy (TE), programming self-efficacy (CPE) and motivation of participants (M) to teach CT topics, respectively.

The statistical means of TE, CPE and M were analysed individually. In all individual analyses, Treatment Type (Group) and Survey Type (ST) were considered as fixed effects in the models. The following model equation was fitted accordingly.

$$Y_{ij} = \alpha + \beta_i \times treatment_j + \gamma_j \times survey_j \tag{9.5}$$

The estimated fixed effects of all three models are summarised in Table 9.4.

Table 9.4: Summary of Estimates and Standard Errors (in parentheses) of Fixed Effects by Model - Associate Teachers

	Treatment		Survey Type	
Model 1: Teaching Self-efficacy Intercept 3.028 (0.139)	T1 ↑	0.042 (0.194)†	Pre ↓	-0.493 (0.067) *
Model 2: Programming Self-efficacy *Intercept 5.243 (0.32)*	T1 ↓	-0.377 (0.438) †	Pre ↓	-2.212 (0.195) *
Model 3: Motivation *Intercept 4.163 (0.568)*	T1 ↑	0.875 (0.347) *	Pre ↓	-0.971 (0.268) *

1. Reference Levels:
   Treatment = Treatment 2 (T2) and Survey Type = Post as reference level
2. Note: Estimates and standard errors (in parentheses).
3. $^{\dagger}p>0.1$, $^{\ddagger}p<0.10$, $^{*}p<0.05$

The table shows the following observations:

- The average score of Mean-TE of the treatment 1 group is 0.042 higher than the other group. However, this difference is not statistically significant (p=0.83).
- The pre-survey score is 0.493 lower than the post survey and this observation is statistically significant (p<0.05).
- The average score of Mean-CPE of the treatment 1 group is 0.377 scores lower than the other group. However, this difference is not statistically significant (p=0.396).
- The pre-survey score is 2.212 lower than the post survey and this observation is statistically significant (p<0.05).
- The average score of Mean-M of the treatment 1 group is 0.875 higher than the other group. However, this difference is not statistically significant (p=0.018).
- The pre-survey score is 0.971 lower than the post survey and this observation is statistically significant (p<0.05).

Accordingly, by including the above observations in model equation 9.1, for the three models CT teaching self-efficacy, computer programming self-efficacy, and motivation to teach DT subjects respectively we get:

$$Y_{ij} = 3.028 - 0.493 \times \textit{pre-survey} \tag{9.6}$$

$$Y_{ij} = 5.243 - 2.212 \times \textit{pre-survey} \tag{9.7}$$

$$Y_{ij} = 4.163 + 0.875 \times \textit{treatment}1 - 0.971 \times \textit{pre-survey} \tag{9.8}$$

9.6.4.2 Interviews

Collectively, six (6) teachers were interviewed, to obtain their views and feedback of the usefulness of the workshops and/or the resources used. Only three (3) learners (herein after referred to as associate teachers) responded to the follow up interviews, of which two (2) were from alternating unplugged workshops (treatment 1 or 3) and one from plugged-in only workshop (treatment 2). One of the 3 learners, a primary school teacher, had no prior experience in programming. The other three interviewees were expert teachers from CSERG: one high school teacher, and two middle and primary school teachers (herein after referred to as expert teachers). Since the expert teachers are in-service teachers who actively involve in teaching similar to the participants, their observations in the perspective of a teacher who would use unplugged tools was considered in this study.

The process of interviewing, and the qualitative analysis process used for the interview transcripts and feedback form, are detailed in Section 5.2.3. A deductive approach was used to analyse the transcripts of the interviews. The original focus of the interview questions was on feedback about the workshops, usability of the teaching approach and/or resources of the workshop in the participants' actual classroom setting, and any feedback they could provide on using the unplugged activities (particularly the new Programming Unplugged activities). However, all the interviewees had limited or no opportunity to use the unplugged resources in their classrooms due to the COVID19 restrictions. Consequently, they expressed their free views based on their prior experience and/or personal opinion. Additionally, the expert teachers were asked to share their experience and views about using the unplugged resources in teaching teachers. Interviews were tagged by the researcher to assist with analysis. Interviewees' answers to direct questions of the theme and/or their explanations to capture any indirect connection to the theme outside the key question were used as the criteria for tagging.

Non-programming Unplugged Activities in Teaching Computing

All six participants were asked about using unplugged activities in their classrooms. All of them had used some Computing Unplugged activity in their Digital Technologies

(DT) classrooms at some point. None of the associate teachers interviewed had taught DT content during the period after the workshop, thus had no opportunity to use Programming Unplugged activities in their classes. The expert teachers had used at most three of the Programming Unplugged activities (i.e. Kidbots, Variable Dice Battle and Conditional Dice Games) with students in classrooms and/or with teachers in training sessions outside of the studies of this project.

The associate teachers expressed their views based on the unplugged activities they used in their classes. However, their usage was limited to teaching and learning Computational Thinking and not related to teaching programming, thereby the activities of interest in the discussions are Non-programming Unplugged activities and not Programming Unplugged activities. Even the expert teachers who had used the Programming Unplugged activities a few times in their programming lessons shared similar views that using Non-programming Unplugged activities in a general context prior to moving to programming lessons has a positive effect on learners. However, this observation seems to contradict the observation made in the previous study that introducing CT concepts using unplugged prior to introducing programming concepts was less effective introducing to programming first. An important difference was the latter scenario was teachers learning, where as expert teachers were working with young students. The following themes were identified regarding unplugged computing activities.

Provide an authentic context for a lasting understanding: Teachers highlighted the importance of providing a familiar context for the computing content in order to establish a focused and long-lasting understanding in a learner. They believed that unplugged activities making use of existing knowledge to introduce new knowledge essentially provides a degree of authenticity for the learner:
"it make use of your knowledge, the existing knowledge, and that's how the unplugged activities had been. That is, passing the message better than just using the big words and try to pretend ..." , *"a young student who's got no experience might find it difficult to understand those things [computing context]. If you give them a practical, hands on, sort of unplugged activity to start them off with, it gives them a good foundation from which they can build."*, *"that actually demystifies what's happening in their everyday world"*

Provide a 'physical' representation to what is 'unseen': Teachers pointed out how unplugged activities provide a tangible representation to computing concepts which are otherwise intangible and thereby difficult to explain:
"Those physical representations of those, you know, intangible things, make it so much easier for a child to understand" , *"That's a good way to see something physical, that isn't something that you can normally see."*, *"I think when you when you use a combination of something that's unplugged with a concept, then you get that*

connection between - well, because ... again it's tangible - An unplugged activity is something that you can do with your hands or ... body or whatever it is. And it makes that concept easily understood.", *"Instead of the Beebot, to use the person you give them instructions - that's a tangible thing a younger student can get in their head."*

Engaging and encourage further studying: The engaging nature of unplugged activities was appreciated by the teachers while emphasising how that could lead to students wanting to continue studying computing in their higher studies.
"They were all very interested. Very engaged. Yeah, even the 'cool kids' were proactive" , *"it's more fun, it's more interesting, it's more engaging ... it's something that they can do. It's not overwhelming"*, *"The whole point of this is to encourage them and get them to want to carry on with it and the future. They talk about users with joy, about wonderment and I want to inspire them to go "well, this is really cool""*, *"It's more about big picture stuff. So it's more fun, it's more interesting, it's more engaging. It's not highly detailed in some areas. It's not difficult to understand."*

One teacher expressed how unplugged activities had been useful in identifying and providing opportunities for the 'logical thinkers' in the classroom, whose skill is difficult to identify and highlight compared to other skills like art or sports.
"I noticed that students who hadn't yet necessarily have a place in the classroom has a place because of unplugged. An example of that is you'd have your academic leaders, your sporty leaders, your artistic children, but the logical thinkers - they didn't really have a home until we started doing the unplugged. And then their lights went ON, and they've asked so many amazing questions. And I thought that, because of teaching unplugged, we were actually seeing their thinking."

Despite the fact that unplugged activities use learners' existing knowledge and therefore provide familiar contexts, it is essential for the teacher to make connection to the computing context. The teachers pointed out the importance of this in their discussions. *"You have to do some extent of hand holding obviously. You have to carry them through the learning process, because otherwise it will be just doing some activity like playing in the playground, ... unless you give them the right kind of, what do you call it: making the connections, as the teacher. Otherwise unplugged activities are going to be just a play thing."* , *"You have to [show] some concept behind it. There has to be some understanding, some computational thinking behind whatever the activity is. That has to sort of 'marry up'."*, *"Unplugged activities are basically just the physical version of creative tools such as Scratch or any other programming language [interface]. So that would be that link".*

Combining Unplugged with Programming, and Programming Unplugged

The next theme that emerged from the interviews was the feedback and opinions about using unplugged in teaching and learning programming in particular. Having had no opportunity to use the Programming Unplugged activities in their own lessons, the associate teachers expressed their views about the combining and/or alternating Programming Unplugged activities based on their experience during the workshops. The expert teachers had used a few Programming Unplugged activities in their classrooms as well as in teacher training sessions.

Using Programming Unplugged Activities in a Programming Lesson

The most common opinion among the interviewees on the best way to combine Programming Unplugged activities in introductory programming lessons was to do the unplugged activity first and then move to the plugged-in exercise.
"I think unplugged definitely does support plugging it in. And if you're able to do the unplugged activities before the plugged in ones.", *" I like the idea of doing unplugged activities first. Specifically, because I think it gives a good visual and an experience of understanding those concepts in a textual way, so people can actually have something to link the virtually abstract ideas"*, *"Plugging it coupled together is the best. Those unplugged things and those Parson's problems [combined together] are really good examples of how to get some foundation understanding started.*, *"I think having them side by side, yeah. Do the activity for 20 minutes, half an hour whatever. And then being able to actually apply it plugged in. I think that, would work really well.* , *"Trying to do wee bit more unplugged before you do that [plugged-in]. If the tools are new, then they need that unplugged activity first. So they're not learning two different things at once."*, *[Do the] Unplugged and then directly take it to plugged in or within the next couple of days, rather than waiting a week. That would work better."*

However, one expert teacher was of the view that the order of what should follow what is not important, as long as they are alternating. She expressed how some students may already be knowledgeable about how to program but without effective understanding about the programming concepts, and how the order of the unplugged and plugged-in experience may not be important for such students. She pointed out that Programming Unplugged activities can help students with such incomplete understandings or misunderstandings to solidify and/or clarify their understandings.
"I think they go together. And you can swap in and out of them. I don't think it's just do unplugged and plugged in. You might need to come back to unplugged again. Because something's not quite working in your plugged in version, so you've got to work out what it is. So it's a matter of alternating and the order doesn't matter. Like what should come first and what not, or could come next does not matter. ... They might already

have some prior knowledge of having had a go on different apps on their own time or be programming in Minecraft or doing something. But not fully understanding why they don't know why."

Programming lessons involve frequent use of devices as well as interfaces that draw the learners' attention, often more than necessary, away from understanding the concepts (for example, drawing Sprites or designing background images). The interviewees highlighted how Programming Unplugged activities can essentially draw the learners' attention to the programming concept, and not be distracted by the device and/or the programming language or environment they use to implement it.

"When you start programming in Scratch, kids get very distracted with the sprites and things like that, whereas with unplugged you're definitely focusing on the concepts. And then ... plugging it back in. So I think there's some pairing in it."

Feedback on Programming Unplugged activities

As mentioned earlier, only the expert teachers had few opportunities to trial the new Programming Unplugged activities with students in their classroom. Although the Kidbots activity has been available and used for some period of time, they had not used it to model a programming concept (i.e. sequence) for a plugged-in programming lesson.

Two expert teachers had used the Variable Dice Battle in their classes, which they have observed to be very effective, including helping students to realise the role of variable initialisation.

"The flip book for the variables, I did use that one and that went really well. ... When we did it with the variables, they would question why we had to put the variable back to zero at the beginning, or anything. And they'd remember to do that. I remember that it was quite successful. And it was ... within a half hour lesson. ... I think it supports the learning. They reinforce so that, when they come into the actual programming they already know.", *"I think the hardest thing to teach is variables. So I think having it so that unplugged to plugged in the approach of writing down the score and then flipping the cards, I think that helps cement that understanding of what a variable does."*

One teacher expressed her appreciation regarding the activity, indicating how her earlier unplugged attempts for variables had failed.

"We've always experimented with unplugged variables and if statements and they hadn't found a successful [method] like yours, or they were playing with it. ... I've had little buckets with names on it and you put the number in ... It doesn't quite replicate it very well. But the flip cards are good.".

These expert teachers had also used the Conditional Dice Games, where one had positive feedback while the other had limited time to complete the activity, but had a positive response in the time available. *"I think that IF-ELSE statements was very, very*

clear and very easy to use. ... I think that IF-ELSE statements, they worked a lot better.", "I wanted to just trial it and see if it they get any better understanding of it, whereas in the past I've just draw them straight into the Scratch if statement".

Another expert teacher has had a similar experience when she used the same activity with teachers during a training session.
"The teachers I've been working with ... we had a real breakthrough because we've been doing unplugged activities. We're probably to a point where I could possibly, well when we get to programming ... some of the activities you created. But at that moment it was just a real breakthrough, because they were kind of like "Oh, this isn't what I thought it was"."
She appreciated how the programming specific unplugged activities can scaffold the Computational Thinking understandings the teachers have been training with thus far, with a meaningful focus.
" ... then when we had our whole staff meeting afterwards ... they're like 'oh this isn't what I thought it was!' ... So far we've been focusing on computational thinking which is developing those initial building blocks to the programming."

Unplugged Activities Supporting Learners' Mental Model Development

The expert teachers as well as one experienced associate teacher discussed how unplugged activities, Programming Unplugged in particular, can support learners in developing good mental models. This was an emerging theme from discussions, rather than from directed questions, which is an indicator of the teacher's realisation of connections between learners' mental models and unplugged activities. This theme emerging with no specific mention of mental models by the interviewer (i.e. the researcher herself) provides a strong indication that teachers are aware of learners' mental models and about Notional Machines (although unknowingly) and that they believe unplugged activities can support in that regard, which also supports the discussion in Chapter 7. They also highlighted how unplugged activities can help young learners who are at a stage where they are still developing their abstract thinking, to understand the intangible contexts of computing at an early stage.
"Using unplugged activities, the teacher can facilitate a student, or teacher teaching a teacher - a novice teacher, can facilitate building these mental model", "Especially the students who ... are still developing their ability to think abstractly. You know they don't [have the ability fully developed yet].. you know, that's something we grow into. Not something that we naturally posess".

They expressed their view on how such mental model development could help increasing their confidence in learning programming, by providing a representation that relates to the abstract concept.

"They [students] don't understand what's going on inside the computer, you know, the hardware - the electronics. They don't understand those concepts like the variables because they can't physically see them. So we have to create some kind of representation for them to be able to build those visions.", *"The more we can have an unplugged representation of it, the easier it is for them to be able to make some connections with their prior knowledge. Because they can't just see it happening inside the computer."*

Unplugged also supports learners to interpret the new learning using their own vocabulary (using existing knowledge) and relate the understanding to computational jargon, thereby supporting them to overcome barriers in using computing specific vocabulary.
"Especially with people who would consider themselves not having this natural kind of strengths in their abilities, you know to be able to visualize those abstract concepts without having something to [relate to]", *"It gives them that ability to understand the concepts, but also be able to build on their language ... you know, the language of the computer science discipline, in particular, in a manner of the those words being related to computer science concepts."*.

One teacher suggested that unplugged helps teachers to "see what learners think". In other words, doing the activity allows the teachers to physically see how well learners understand the concept they intend to communicate, and where learners make mistakes, it helps teachers to diagnose the errors and/or mistakes of the learners' mental model.
"Unplugged gets people thinking in that logical way, in a different way. ... Because of teaching unplugged, we were actually seeing they are thinking."
Another viewed this ability to 'visualise the learners' mental model' as important for the teacher's own understanding about the nature of their class.
"You know how sometimes when you're teaching and you think "I know they know it" and you're totally blindsided by all the ways they get the answers wrong? And [I would like] if I can have a go, if I can figure out how ... how I went wrong, and this is why."

Unplugged and Teachers' TPACK

When asked how Programming Unplugged would help with teachers' Technological Pedagogical Content Knowledge (TPACK), the expert teachers pointed out that it could be a "nice stepping stone" to work for teachers who are not ready to learn programming yet or are afraid of programming. Unplugged provides a framework for such teachers to "actually teach some computational thinking in the safety of the unplugged realm". For the novice teachers, Programming Unplugged activities could be an "uplifting experience" that provides a comfortable zone to operate in teaching computer programming while supporting their students. Having unplugged activities can show them how those concepts go across different programming languages and that the general concepts are not specific to any particular programming language.

One associate teacher answering the same question pointed out that it helps them to focus on understanding the concepts without having the device (i.e. the programming language) interfering in the tedious learning process.
"I don't have to focus on using two different things. I don't have to try and do it with a computer as well, so it's not too many things.". Although unplugged does not necessarily teach them anything 'new', another teacher admitted that it definitely helped "refining" their understanding.
"Rather than it being just conceptual and on the screen, it was a lot easier for me to understand. And the way that the lecturer ran through it just going one by one, each concept that selection, ... input/output and and linking them to an activity, I was able to see the difference as well".
However, there is a risk that teachers do not go beyond unplugged, and miss out on the full experience of computing, therefore plugging-it-in is an important connection.

9.6.5 Discussion and Conclusion

The Study II participants' self-efficacy towards teaching CT as well as programming, and their motivation to teach DT has increased after the PD intervention. This indicates that PD interventions in any of the formats used positively affects in-service teachers' self-efficacy and motivation to teach computing subjects in classrooms.

The alternating unplugged group showed higher self-efficacy towards teaching CT than the plugged-in only group. This approach also provided more material than the plugged-in only approach, which relate well to a classroom setting. Therefore, alternating unplugged seems a more successful approach for teachers' professional development in introductory programming. Moreover, the majority of the participants had prior programming experience. Therefore, the availability of more material and discussion points could be the reason behind the difference.

Although not significant statistically, it was observed that the teachers who were exposed to the alternating unplugged approach showed lower self-efficacy towards programming than that of the traditional plugged-in group after the workshop. The possible reasons for this could be that, since many of the teachers had experience in Scratch programming, the plugged-in only approach must have seen more efficient, and been comparatively more meaningful for them. Moreover, the alternating unplugged approach delivered more material within the same time-frame and possibly could have overwhelmed the participants. The sense of rush may also have caused frustration. Nevertheless, the statistical insignificance may also imply that the observation is merely a random occurrence.

The availability of more material that relates to classroom use (more useful with younger students in the classrooms) with unplugged style activities to introduce pro-

gramming is a motivating factor for an experienced teacher, as they understand how the content can be used to make connections in the classroom compared to the conventional plugged-in approach. Provided that the in-service teachers already have some context of teaching DT, their motivation to teach is an indicator of how PD programmes of this nature can have a positive influence. This could be the reason why the alternating approach was significantly more successful with in-service teachers than with the beginner teachers of Study I.

Overall, the participants' feedback and opinions showed that unplugged activities can help teachers to provide a familiar approach to computing with their students by being able to use student's existing knowledge to introduce new knowledge. Moreover, they also help teachers to provide tangible interpretations to the abstract and/or intangible contexts of computing in their classrooms, which is an important pedagogical aspect, especially with young students who are yet to improve their abstract thinking skills. Unplugged computing provides a pathway to introduce deep, abstract computing concepts to younger learner groups to be exposed to such abstract knowledge simply and effectively.

While the majority of the interviewees were of the view that introducing a programming concept first and then moving into plugged-in is the most effective way to combine Programming Unplugged with plugging-it-in, there was a view that suggested that as long as the two are alternating, the order may not really matter, because the chances of students having prior exposure to programming are high now that it is more common in schools. Because unplugged activities effectively communicate the programming concepts, having them alternating could help learners to either establish or refine their understanding, depending on their prior level of understanding.

Despite the limited opportunities for using Programming Unplugged activities in their classrooms, teachers showed a positive response towards having unplugged activities that can be directly transferred to a programming exercise. They confirmed the decision made in developing the activities (as discussed in Chapter 8), for example using a flip-card holder as a prop to represent variables over other candidate props. The use of flip-cards was observed to be an effective prop for understanding and avoiding misconceptions about variables, which could also lead to avoiding some important misconceptions later on, at an early stage of learning. The teachers appreciated having Programming Unplugged activities that can model programming concepts that are also directly transferable to programming exercises.

The unplugged alternating approach helped with both cementing a concept in the learner better and provided additional useful material for teachers to use in their classrooms. We speculate that, had the participants had more awareness of the Scratch environment, the additional material delivery within the time span would have been more effective.

9.7 Conclusion

The Programming Unplugged activities designed and tested during this study were the activities developed during the study discussed in Chapter 8 and the discussion in this chapter relates to Phase 5 of the Study Phases (See 8.3). The expert teachers' observations indicated that Programming Unplugged activities can, particularly for the novice teachers, reduce unease towards teaching programming as well as positively influence their programming skills. All three aspects tested among the teachers (i.e. teaching self-efficacy, programming self-efficacy and motivation) were increased after the workshops, indicating that PD of this nature can support teachers in DT.

Overall, the alternating unplugged approach has increased both pre-service and in-service teachers' teaching self-efficacy and motivation to teach DT. However, the alternating unplugged approach used has decreased the computer programming self-efficacy of both kinds of teachers. This is possibly due to the teacher feeling overwhelmed with the time restriction to cover the more content, where the novice majority had less time to get used to the Scratch programming environment. Although using the programming environment is not the same as a person's programming skill, at this stage it could affect the novice participants, programming self-efficacy adversely. Where there was a higher number of participants with prior exposure to programming, this large amount of content might have brought about new understandings of their own knowledge.

The associate teachers' feedback indicated that they have good awareness about the NZ DT Curriculum as well as the impact and influence of both Computational Thinking and programming in this content. However, the observational notes indicated that this case is not common with the pre-service teachers, where the number of participants who had even read the Digital Technologies learning area of the curriculum prior to the workshop was very low. The associate teachers pointed out the effectiveness of Non-programming Unplugged activities in modeling Computational Thinking concepts and were enthused and appreciative of having programming-specific unplugged activities that can be directly transferred to programming exercises (i.e. Programming Unplugged activities).

The teachers who provided their opinion pointed out an important connection in unplugged computing activities supporting learners developing good mental models regarding computing concepts, and particularly programming concepts. They mentioned that unplugged activities provide tangible interpretations to the otherwise intangible or highly abstract concepts of computing as well as programming, which is helpful in introducing these concepts to younger students. This discussion partially supports the discussions in Chapter 7. It indicates that teachers have both identified a connection between learners' mental models and the Notional Machine (albeit unknowingly) as well as recognising how unplugged activities can facilitate teachers' pedagogy for supporting

learners.

The teachers indicated that unplugged also supports learners to interpret new learning using their own vocabulary (using existing knowledge) and relating that understanding to computational jargon, thereby supporting them to overcome the barriers in using computing-specific vocabulary. This observation also partially supports the possible applicability of unplugged activities in crystallising the learning as discussed in Dual Purpose Theory discussion in Chapter 4. Only weak claims can be made in both the studies based on survey data as well as other observations, because the sample size was small and the results were not statistically significant. The observations, findings, and the limitations encountered throughout the studies indicate that further work may be warranted with a larger sample size to see the actual nature of such claims, as the lack of passing the significance test also means they could be the result of random processes.

The Programming Unplugged activities used in the experiments need refining before they are completely ready to serve the purpose as they have so far been evaluated with teachers rather than using with young students. They must be used in actual classrooms and adjusted based on feedback from teachers. This is a longer process than the length of this project. Nevertheless, the successful use of unplugged activities effectively in a classroom depends on several parameters ranging from the level of the learners to the way the activities are conducted/incorporated in a lesson. Further, if the Programming Unplugged activities are not followed by plugged-in exercises and/or such links integrated into the activity in semantic waves to cement the conceptual understanding the learner obtains from them, the activities may become irrelevant or seen as mere playful activity.

Chapter X

Programming with Unplugged - Understanding the Semantic Profiles

This chapter discusses the usefulness of Unplugged computing activities in introductory programming for teachers' professional development, by studying the behavior of the 'semantic waves' [150] of two different teaching strategies: alternating unplugged activities with plugged-in programming, and plugged-in only programming. The semantic waves of three Unplugged activities used to model three programming concepts are analysed in detail, and compared to the semantic profile of a plugged-in only lesson that teaches the same concepts. Alternating Unplugged activities with a plugged-in experience successfully covers a wider semantic range, indicating the possibility of avoiding both learner anxiety as well as boredom, enabling teachers to find optimal teaching strategies that suit their classrooms. The discussion in this chapter intends to provide partial answers to research question RQ 3 described in Section 1.3, particularly RQ 3.3 which intends to investigate the features of unplugged that are useful when combined with programming and RQ 3.6 that studies the possible impacts programming-focused unplugged activities may have for teachers and learners.

10.1 Introduction

As explained under the educational framework of Legitimation Code Theory (Section 4.3), a 'Semantic Wave' is a concept that describes a purported good learning journey of a novice learner over their course of learning, while shifting between expert and novice understanding, abstract and concrete context, and technical and simple meanings [178]. LCT [149, 150] describes several terms associated with the semantic wave concept:

Semantic Gravity (SG): The context dependence of knowledge (the context of meanings and how much meaning depends on everyday context).

Semantic Density (SD): The degree of complexity of knowledge (the complexity of meanings rather than their context).

Semantic Wave (SW): The movement between concrete, simpler knowledge and more abstract, complex knowledge in a wave pattern to master a subject.

Semantic Profile (SP): The changes in the context-dependence and complexity of the knowledge being taught (or learnt) during a specific learning experience [59].

During a lesson where typically novice learners' knowledge is low in SD at the beginning, this journey is a cooperation between the teacher and learners where the teacher determines the phase of the curve between SD and SG in transferring conceptual knowledge using learners' existing knowledge, to bring the learners to a higher SD level while at times also aiming to get them to work in a lower SG context. SD and SG are different dimensions that are superimposed in semantic profiles to give a simple intuitive picture of this idea.

In a typical lesson, there are always multiple semantic profiles in play. For example, a lesson plan can be analysed and will have a semantic profile. Any lesson based on it can be observed and the profile of what is delivered could be the same or different. Moreover, any individual learner's experience will also follow their own unique profile, for example depending on whether they are listening/doing activities expected or not, how far they get, etc. The semantic profiles of individual learners may or may not be measurable or observable.

Waite et al. [241] brought the notion of Semantic Waves into the discussion around Unplugged computing activities in both teaching computing to school students and teachers' professional development. They explain how a good Unplugged activity based lesson plan follows a Semantic Wave and suggest that Semantic Waves are useful in developing and reviewing lesson plans. Their argument that moving back and forth between concrete/simpler and abstract/complex knowledge helps to increase the usefulness of an Unplugged activity reflected the discussions of an Unplugged activity development process discussed in Chapter 8 as well as in the study on a series of introductory programming professional development workshops that followed (discussed in Chapter 9).

In order to understand the effectiveness of the two teaching strategies that were used in the studies of the previous chapter to introduce programming concepts to teachers in a professional development experience, their Semantic Profiles were studied. These analyses were conducted based on the observations made during the activity development process in Chapter 9 and lesson planning for the two programming introduction strategies of the studies in that chapter. The two strategies used were: 1) alternating Unplugged activities with plugged-in exercises and 2) conventional plugged-in exercises. Those experiences are discussed here in the light of how changes between SG and SD influenced them. The following sections of this chapter contain a discussion of: 1) an analysis of the semantic profiles of the unplugged activities used and 2) a comparative analysis of the semantic profiles of two of the teaching strategies used in the introductory programming courses discussed in Chapter 9.

10.2 Unplugged Computing and Semantic Waves

An unplugged activity that is used to explain a computing concept(s) can create a wide semantic range that embed the nature of the context (for semantic gravity) and the meanings being condensed (for semantic density), providing a good opportunity to create semantic waves in learning. Determining the Semantic Profile of a learning experience or a lesson can depend on several circumstances specific to that particular lesson.

The nature of the participants in an Unplugged activity can vastly vary depending on the purpose of usage (e.g. in grade school[1] classroom activity with children vs professional development for teachers). Executing the same Unplugged activity may also need to vary according to the background of the audience (e.g. children vs adults, students vs teachers, novices vs experts, etc.). Further, an expert teacher can be responsive to the audience by adjusting the activity execution to match the moment, where as a non-expert teacher may need to execute the activities adhering to pre-arranged instructions. Learners' motivation also affects the Unplugged activity execution, especially with the nature of learner-initiated questions that cause variation in the discussions during the activity. Unplugged computing needs a certain degree of flexibility and freedom in execution that may not be supported by some curricula. Thus the flexibility of the curriculum can also play a role in activity execution; a teacher may have lesser freedom in adapting Unplugged style material in their lessons in a formal and structure-bound curriculum than in a flexible curriculum.

In their attempt to understand the semantic profile of an Unplugged activity in computing, Waite et al. [241] discuss a teaching and learning activity that involves definition, exercise and explanation. The semantic profile of the lesson appears to have three corresponding regions that includes both downward and upward shifts (also referred to as down-shifting/unpacking and up-shifting/repacking), as shown in Figure 10.1.

How a teacher handles the definition, exercise and explanation to achieve a learning outcome is determined by individual lesson execution, which is influenced by many attributes such as the audience, resources available, teacher's expertise, etc., and therefore the distinctions between these regions can often be vague. It can be seen that how an Unplugged activity is incorporated in a lesson and how it is executed in these regions largely determines the overall shape of the semantic profile of a lesson [59].

10.2.1 The 'Unpacking'

The 'unpacking' refers to the learning and knowledge exchange by attempting to explain/describe complex ideas in terms of more everyday knowledge, and using simple

[1] By grade school we refer to the elementary, middle and high school levels, also known as K-12, or primary and secondary school.

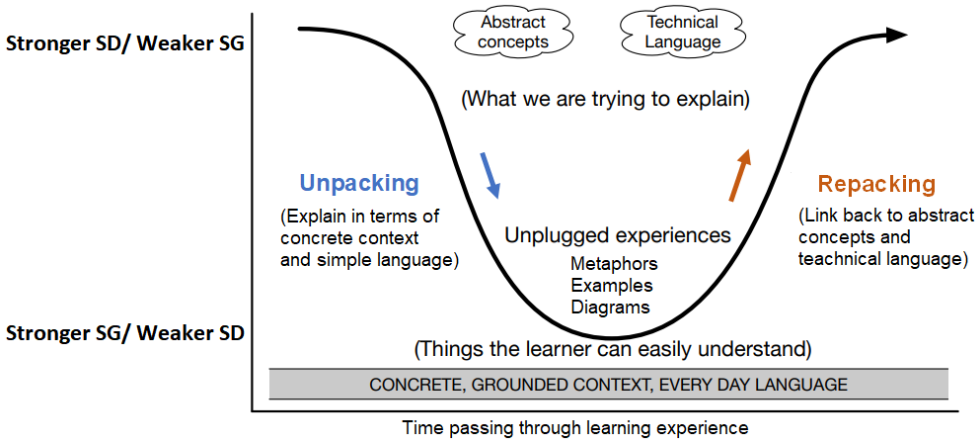

Figure 10.1: Semantic profile of a learning experience (the formation of a semantic wave), adapted from [241]

language. Maton [150] describes this as "weakening semantic density (SD↓), such as moving from a highly condensed symbol to one that involves fewer meanings". Many of the meanings of knowledge involve compositional structures, taxonomic structures or explanatory processes imbuing the terms that are used to communicate those knowledge with a far greater range of meanings. Unpacking is diluting those meanings along the continuum of strength.

Both teachers and learners may be involved in unpacking; in an instructivist approach the teacher's involvement could be higher than in a constructivist approach. It is a rather straightforward contextual reference to the decreasing SD and increasing SG, and would be expected to show as a down-shift in the curve. However, in the actual context, 'unpacking' could be only setting the context to the lesson with the expectations of helping the learner to get ready for high SD knowledge from the lesson.

The teacher's starting approach to a lesson (i.e. whether with an introductory definition or moving straight to an exercise) determines the starting point of a Semantic Profile curve. In other words, the accent of the curve is actually determined by the approach the teacher would employ to blend the complex ideas with everyday knowledge. A lesson plan that starts with a definition, that is, *the teacher explaining the conceptual knowledge to the learner up front* and relating it to more contextual knowledge gradually prior to an exercise (e.g. teaching how to declare an integer), will have a down-shift as the start of its Semantic Profile curve. Such an approach to a lesson would be more instructivist.

Figure 10.2 shows the shapes of possible semantic profiles for the two different lesson starts. As explained in the Unplugged activity example in [59], providing conceptual knowledge up front (e.g. by introducing the learning outcomes) indicates high SD, thus the curve starts at a higher point in the continuum as in Figure 10.2 (a). Unpacking

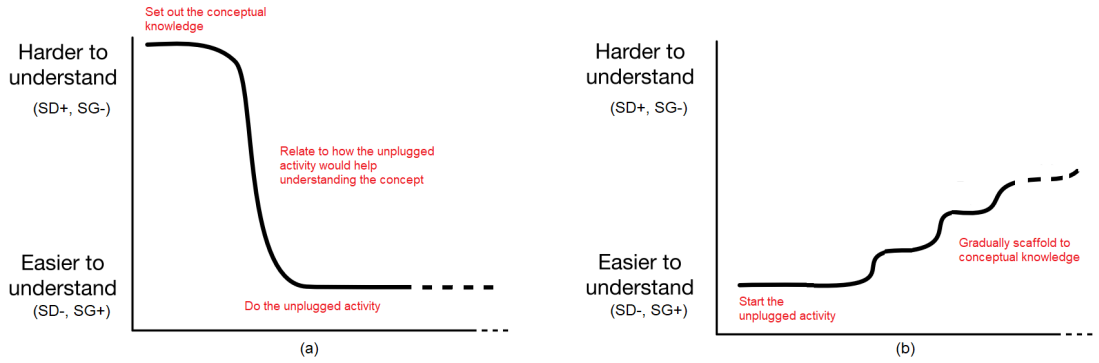

Figure 10.2: Starting point of a semantic profile: (a) teacher explains concept upfront, then relate to context (b) teacher moves to doing the exercise to draw out ideas to scaffold context to concept

later by relating to everyday knowledge/natural language explanation draws a downward slope curve through the continuum.

In the case studies used by Curzon et al. [59], they consider setting out the learning outcome of the lesson at the beginning of the lesson and relate how the activity would help reasoning as the initial step of the Unplugged activity they have studied. Thus the authors plot the start of the semantic curve at a point higher in SD and lower in SG (Figure 10.2 (a)) following Maton [150]. They consider this 'describing about the activity up front, explaining how it would help realising those learning outcomes' as changing from decreasing semantic density and increasing semantic gravity in the continuum, thus plotting the experience as a downshift (rather rapid, considering that this is a very short part of the activity) in the semantic profile from a higher point to a point far lower. This approach has caused the "U shape" of the overall semantic profile that matches the one in Figure 10.1.

Although the learning process covered by a semantic profile involves both teacher and learner [241], the teacher plays a much bigger role in 'unpacking' than in 'repacking', because the learner may not start with any conceptual knowledge (therefore a low SD). Loading with knowledge of high SD at the beginning of a lesson essentially means that the teacher's influence and/or contribution in the discussion is much higher (i.e. instructivist approach) than the learner. As mentioned earlier, deciding the best strategy to start a lesson can be subjective depending on aspects such as the nature of the audience, expertise of the teacher, and structure and flexibility of the curriculum. Providing a formal explanation of the conceptual knowledge up front and gradually relating it to contextual knowledge would be effective for a mature audience like teachers in a professional development programme or high-school students; they may be sufficiently knowledgeable to

comprehend a sudden change by decreasing SD to increasing SG without feeling anxious or lost. Less mature audiences like children or novice learners may sometimes benefit from a more informal approach with gradual scaffolding from contextual knowledge towards conceptual knowledge. The effective learning of a less mature audience may also involve redoing the activity several times with slightly different instruction cycles and/or regular reference to the experience in their future learning until the learners cement the conceptual understanding.

As suggested above, the expertise of the teacher is also a significant factor in deciding the unpacking strategy. A less expert teacher (i.e. an expert teacher with less technical knowledge or a technical expert with less pedagogical knowledge) may find the first strategy more useful for an Unplugged computing activity, as it purposely allows the teacher to introduce the concept and relate it to the activity (possibly with clear instructions to follow). The structure of the curriculum could also be a decisive factor in the 'unpacking'; more structured and formal curricula like the UK curriculum may require laying out the learning outcomes upfront, whereas a flexible and loosely specified curriculum like the New Zealand curriculum may not require such a formal approach.

10.2.2 The 'Exercise'

The part of the lesson when students are engaged with the physical unplugged activity is considered as 'exercise'. Understandably, the lesson operates at a high SG level during an unplugged exercise.

Unplugged computing activities are kinaesthetic, largely involve games and/or a sense of story, and were originally designed intentionally for outreach [167]. The learners' concrete and contextual knowledge is regularly used by the teacher to explain or to link to a concept and to communicate the conceptual knowledge through to them. A frequent movement between contextual and conceptual knowledge, in many forms such as a learner asking simple questions and teacher or a knowledgeable peer answering, teacher and/or learner explaining, using metaphors/analogies and making connections to concepts [59], are inevitable in Unplugged computing.

Because of the extent of the interplay between SG and SD that is involved in an Unplugged computing activity, the semantic profile during the activity may have several micro sinusoidal or step-up shapes caused by explicit teacher design of the activity or moments such as smaller realisations, questions and answers, and understandings or "Aha!" moments during a lesson (Figure 10.3). Such moments can be caused either as part of a predetermined plan of the lesson, peer learning during the activity, or spontaneous, personal realisations of a learner.

During an exercise, these micro-oscillations can indicate several perspectives from a learner's cognition and interventions of attention. The upward micro oscillations might be

when the teacher intervenes to say "look, now you're seeing an example of this concept", or it might be an intuition of the learner themselves who realises that they have just observed what the teacher was talking about before. A micro downward slide might be the learner realising "Oh, now that I understand what that means, I'm going to go and try some more examples". It could either be a teacher pondering a question, a learner asking spontaneous questions, or a learner realising on their own.

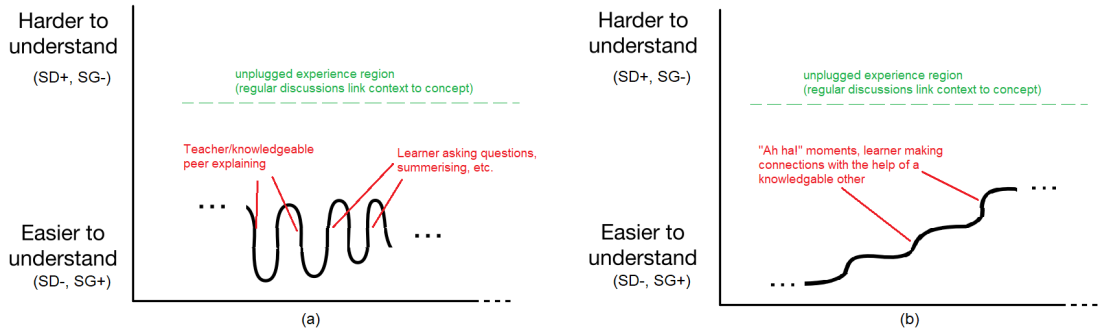

Figure 10.3: Details of possible semantic profiles during an Unplugged activity: (a) moving back and forth between context and concept knowledge (b) Using context knowledge to understand concept(s)

These micro-discussions may not necessarily be detailed in a full semantic profile of an Unplugged computing activity. Further, not every participant may actively engage in such micro discussions, and also the "Aha!" moments may not be common to all learners. However, during an activity, a teacher may notice such moments and make use of them for the benefit of the entire group, thereby causing these micro-semantic waves to become common to the individual learning experiences of most learners. Such predetermined moments (i.e. the teacher pushing the learners by simple hints or pondering simple questions) and peer learning moments (that involve a knowledgeable peer) during a learning process, which are very common in an Unplugged computing activity, seem to have connections to the concept of Zone of Proximal Development (ZPD) (discussed in Section 4.1).

Maton [150] points out that both starting low or high on a semantic profile can work, as can various shapes, as long as a curve shape is formed, which is the critical aspect. This aligns with the micro-oscillations we observe in unplugged activity. It should be also noted that, in learning, a topic/subject may involve several lessons in isolation, and therefore have waves within and following waves across lessons.

Due to the nature of utilising existing knowledge to introduce computing concepts, learning through Unplugged activities naturally and essentially includes many (prominent) up-shifts in the semantic profiles. Curzon et al. [59] suggests that Unplugged style

activities can exhibit a series of semantic waves within a larger wave, caused by those micro moments and discussions that occur during the activity. They suggest that by allowing the learners to do more repacking during Unplugged activities, preferably on their own (e.g. role play, questioning in everyday language) can result in learners making abstract, tangible questions expressed in a simpler language understood by them that consequently relate to more complex, technical knowledge which they are yet to master (and may not yet be equipped to express in technical language). Novice learners get the opportunity to ask questions about what they do not understand, without requiring the technical mastery to form the question. The answers can generate semantic waves by linking them with complex, technical knowledge, creating a series of small semantic waves.

10.2.3 The 'Repacking'

In [241] and [59], the authors analyse the semantic profiles of several Unplugged style activities related to computing, where the semantic wave is seen as "broadly a U shape". Waite et al. [241] observe a "staged return" coming out of the U shape, indicating a series of smaller step-like up-shiftings locally along a globally upward shift. Maton [150] describes 'repacking' as moving back into the pedagogic discourse of the subject through re-wrapping explicated meanings and examples into terms or ideas. He shows that it is "more than a summary of the preceding passage of 'unpacking' " and is "moving expressed knowledge up the semantic scale".

The 'staged return' as observed by Waite et. al [241] indicates the participants (teacher and learners alike) making connections between the Unplugged experiences such as metaphors, physical activities and analogies (i.e. commonsense with high semantic gravity) and contextual knowledge, by scaffolding the existing knowledge into newer knowledge, pushing them towards deeper conceptual understanding.

"Repacking" refers to moving up the curve; it can exist either where there was an 'unpacking' earlier in the lesson (e.g. teacher mentioning/introducing the expected conceptual knowledge of the lesson upfront) or when learners construct the technical knowledge themselves from the parts of a real world activity (e.g. on a dice game learners realising that a loop needs a 'test' and a 'body' leading to a repacking for students to "invent their own loop construct"); if not, the SD knowledge may not be available for the learners to 'repack' by making connections. During 'repacking', the learner involvement may be higher (unless in a very instructivist style lesson), because that is the region where the learner connects their SG knowledge with the high SD knowledge learnt to cement and move themselves towards the high SD level.

The 'repacking' experience of an Unplugged activity also depends on the nature of the audience. A novice audience with limited technical knowledge may find it difficult

to make straightforward connections that link the concrete knowledge they gain from an Unplugged activity to the conceptual understanding the activity intends to communicate. Such audiences may benefit more from a "staged return" with the teacher providing most of the necessary scaffolding (i.e. a step-wise gradual incline in the "repacking region"), than straightforward explanations (i.e. steep inclines in the "repacking region"). Mature audiences with a pre-existing technical awareness (perhaps as part of their everyday knowledge) may not require such directed support, but can afford more straightforward linking between abstract and technical contexts, which could trace steeper inclines in semantic profiles in the "repacking region" than that of a younger or novice audience.

The teachers' feedback after our workshops confirmed that teacher direction to link concrete learning into a technical context is essential during an Unplugged activity with children. Regular reference back to a corresponding Unplugged activity experience their students had earlier is very common among teachers who have used Unplugged computing activities with children. The observations made during professional development workshops for teachers where novice participants made queries such as *"Why are we doing this?"* or *"How am I going to use this with my students?"* indicates a need for certain degree of specific support even among adult learners despite their maturity and higher contextual knowledge. Therefore, "staged repacking" that can create a series of micro semantic steps as shown in Figure 10.3 (b) is highly beneficial for novice audiences during an Unplugged computing activity. These questions mainly came later, and related to general concerns about why the curriculum was necessary, although some could potentially have been generated because some activities started with the profile as in Figure 1.2, and this can risk not having concepts to repack.

10.2.4 A 'Packing'?

When carefully analysing how an 'unpacking' could occur during an unplugged computing lesson, particularly at the beginning, with a steep downward shift in the Semantic Profile as explained by Waite et al. in [241] (Figure 10.2 (a)), it can be seen that it is only based on the authors' observation of a particular lesson where the teacher was explaining the learning outcome *before* the Unplugged exercise, which might not be true for all situations. Even relating to their observation, a question arises as to who that downward slope is for, if the learner does not hold high SD knowledge. If that slope represents a situation such as the teacher saying "today we're going to talk about algorithms and we're going to derive all the notation for various classes of sorting algorithms" at the beginning of the lesson, then that may still not mean anything to the learner, even if the teacher finds something from learners' everyday knowledge to connect it to (which could possibly form the downward slope). Moreover, a 'repacking' can only meaningfully exist in a lesson if there was an 'unpacking' of that nature earlier in the lesson or if the learner

realising a connection on their own (such as with metaphors). Otherwise the relevant conceptual knowledge of high SD is not available for the learners to connect their SG to.

In an Unplugged lesson where the teacher moves directly to the exercise, purposely avoiding the introduction of any high SD knowledge, and *draws out ideas* from the learners and use them to gradually scaffold contextual knowledge to more conceptual knowledge (in a constructivist manner), the Semantic Profile starts at a very low point in the continuum (high SG) as shown in Figure 10.2 (b). Consequently, during an Unplugged computing activity executed with a constructivist approach, where the abstract knowledge is presented gradually, the actual transition from 'unpacking' to 'repacking' may not be very obvious. Since there has not been any notable 'unpacking', there is no downshift of the curve anywhere at the beginning, and with no conceptual knowledge present, by definition 'repacking' does not exist. The learning happens as a gradual shift from SG towards SD, thus the shape of the curve could be as shown in Figure 10.2 (b), and for clarity, this particular shape we define as 'packing'.

This approach of the teacher's drawing out of ideas from learners to gradually scaffold to higher SD that we define as 'packing', is blending concrete knowledge with conceptual knowledge simultaneously, and conceptual knowledge is presented by the teacher at the point of learning rather than prior. In the absence of upfront explanations, the Semantic Profile would have followed a step-wise incline starting from the bottom of the continuum and climbing upward. Curzon et al. [59] have experienced a similar situation by accident, but it is an approach that a teacher could purposefully use, that potentially has benefits for the learner. Curzon et al. use the term 'packing', in particular when it was needed to discuss the 'accident' where there was no 'unpacking', but they generally use 'packing' to discuss the movement in general as in curving the semantic profile (both 'unpacking' and 'repacking'). They also refers to this as "staged return", i.e. if the curve begins at a higher SG level, and thus at a lower point in the continuum. However, unlike "staged unpacking", where conceptual knowledge has already been provided for the learner to connect to, here both the conceptual knowledge and concrete knowledge exist *simultaneously*. In this approach of an up escalator, the overall shape of the Semantic Wave may not look like anything close to a sinusoidal shape, but it traverses across (probably the full) breadth of the continuum.

This strategy of starting the unplugged activity with no connection to conceptual knowledge and gradually drawing out ideas from learners may need a reasonable blend of both technical expertise and pedagogic knowledge and on-the-spot decision making by the teacher while executing the Unplugged activity. Otherwise, the learners may find making connections difficult and feel lost if there are long technical explanations at the beginning. However, it potentially allows the effective use of learners' existing knowledge, which could be very useful in building their confidence and cementing their conceptual knowledge.

10.3 Zone of Proximal Development and Semantic Profiles of Unplugged Computing Activities

During an Unplugged computing activity, the teacher essentially attempts to communicate a technical concept to a learner, using the learner's existing knowledge such as a physical activity or device, metaphor or analogy. The technical concept that is communicated is likely to be less known knowledge to the learner as well as high in semantic density. The Unplugged activity itself is designed based on knowledge already familiar to the learner, therefore high in semantic gravity. Therefore, (frequent) traversing between low and high semantic density while maintaining a higher degree of semantic gravity over time is inevitable during an Unplugged activity, and that creates a semantic wave (Figure 10.3), as well as situating the learning process well within the Zone of Proximal Development (ZPD) (detailed in Section 4.1). This phenomenon can also relate to the non-linearity of the learners' trajectory as well as its fluctuation between the two axes during the space of learning within the ZPD in Figure 4.1.

A "staged return" in a Semantic Profile of an Unplugged activity as discussed in [59], which involves the teacher asking meaningful questions that help learners to develop answers based on specific yet simple contexts, draws the learners from a high SG region (where they started at with their concrete knowledge) towards a now high SD region (with the scaffolding provided by the teacher for conceptual knowledge) in the SG/SD continuum. This process also matches Hedegaard's interpretation of ZPD [113]. Therefore, the 'packing' (as i n staged return) or 'unpacking' and 'repacking' (as in a wave) of a well planned and executed Unplugged computing activity's Semantic Profile, especially a "staged return", essentially situates the learner in the ZPD region as shown in Figures 10.3 (b) and Figure 4.1. Assuming that a person's level of competence cannot decrease over time, heuristically the ZPD curve can also possibly be seen as a differentiation (rate of change) of a semantic profile of an Unplugged computing activity, that essentially shifts the learner back and forth between their level of SG and SD at a point of time of learning, while the teacher (or more knowledgeable other) pushes them towards a higher SD level over the course of the learning activity.

A connection between different slopes in a Semantic Wave to ZPD that we propose is shown in Figure 10.4. The more semantic range a learning activity covers, preferably with the aid of a more knowledgeable other, the further it moves up curve and situates a learner firmly within the ZPD, and mixing Unplugged learning into a lesson helps improve the semantic range of a lesson. Nevertheless, it should be noted that covering a larger semantic range too fast may push the learner into anxiety and too slow, into boredom, and there will be other links between what is happening in a lesson and these two curves.

A learner's Semantic Profile remaining horizontal indicates stable knowledge (techni-

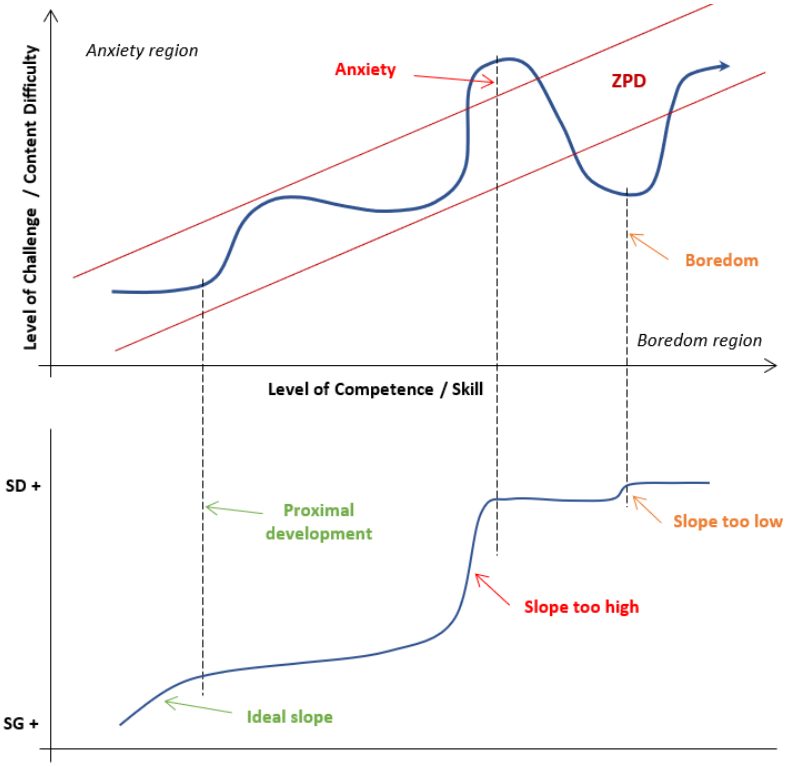

Figure 10.4: A connection between the ZPD and different slopes in a Semantic Wave

cal or abstract). The teachers' unpacking of the technical learning/language is essential in a lesson at some point, in order to make the required knowledge lasting and for the learning to be stable [149]. However, unpacking and then repacking of technical learning (e.g. an instructivist approach of teaching programming in a coding first strategy) may leave learners feeling abandoned [116] if the Semantic Wave is too steep (anxiety in ZPD). Similarly, without the support of a teacher or of scaffolding resources in making connections to the technical concepts (i.e. 'packing' or 'repacking'), a learner can be lost during an unplugged computing activity where they would merely enjoy the experience without seeing the reason or making meaningful connections (i.e. they are flatlining low). This shallow Semantic Wave corresponds to ZPD in the boredom range. Inability to make connections could also create anxiety in an enthusiastic learner. Thus, a well performed 'packing' or an optimal balance between 'unpacking' and 'repacking' during an Unplugged computing lesson, if achieved, would place the learners' trajectory comfortably within the ZPD, without bringing them toward the anxiety zone or boredom zone.

10.4 Programming Unplugged and Semantic Waves

In this section, three Unplugged activities that were used to introduce three key basic programming concepts (sequence, variables and selection) are analysed to understand their semantic profiles. They were used in the experimental studies discussed in Chapter 9. Each Unplugged activity was immediately followed by a plugged-in coding exercise where the learners used the Scratch programming language to directly implement their Unplugged experience in a computer program.

Although the participants in the workshops were teachers (pre-service and in-service), that were mostly novice or beginner programmers, and are identified as the 'learners' in the analysis. All the workshops were conducted by the same instructor, who is both a Computer Scientist and a teaching expert. In all workshops, the learners were also supported by two to three knowledgeable helpers. Two observers took detailed notes of all the sessions.

The three activities used were Kidbots, Variable Dice Game, and Conditional Dice Battles respectively. Learners were requested to keep their computer screens closed during the Unplugged activities, so learning during the activities was totally Unplugged with zero device interruption. The first activity, Kidbots, has been developed and used popularly in promoting Computational Thinking. In this session the instructor focused on using this activity to introduce a programming concept (i.e. sequence). The concepts introduced in the other two activities were also directly related to programming. Although Unplugged computing activities similar to the other two activities (i.e. Variable Dice Game and Conditional Dice Battles) are available elsewhere, these new activities were created after carefully considering the limitations of the previously proposed activities (as detailed in Chapter 8).

10.4.1 Method

A similar method explained by Waite et al. in [241] is followed to analyse the individual activities and the lesson plans to create the semantic profiles; semantic profiles were created heuristically by examining the activities step-by-step, based on the lesson plan and observational notes taken during the sessions. Although the method followed in the analysis is the same as Waite et al.'s, the experiment context of the data gathered in this project on which the method is applied to is different from their learning exercise. Moreover, the analysis in this study is based on combining both lesson plan and a lesson observation. Therefore, it can be seen as a hybrid of the two rather, than applying the method to only a lesson plan.

We found no similar experiments in the literature that had analysed a semantic profile of a programming lesson of any context. Waite et al.'s method that analyses an unplugged computing lesson seemed sufficiently elaborate and efficiently covered the key aspects of

a semantic profile of a lesson programming lesson as well. Nevertheless, a slight deviation was made to capture the alternating unplugged with plugged-in, as well as plugged-in only context. The analysis of the observational notes were additionally helpful in this regard.

The following three key questions from Waite et al. [241] were then asked to further explain the learning.

> *Q1: Does the shape of the profile plotted follow a rough wave shape (either 'u'– or 'n'–shaped), avoiding 'flatline' and 'down escalator' profiles, in order to support learners to move between concrete, simpler knowledge and more abstract, complex knowledge?*
>
> *Q2: How far up and down does the semantic profile move?*
>
> *Q3: Who is doing the packing and/or unpacking that moves knowledge up and down the profile: the teacher or the learner (or both)?*

The instructor followed a constructivist approach during the entire session and often used simple language carefully in explanations. Even when the technical content was introduced, he avoided the use of deep technical explanations or use of jargon. Therefore, all the activities were kept in a fairly lower range in the SD/SG continuum (i.e. high SG) at the beginning, and the instructor deliberately scaffolded and pushed the learners towards a higher SD range of the semantic profile during the activity. The overall objective of all activities was to mostly relate to a semantic profile similar to Figure 10.2 (b); an overall up-shifting with micro steps at places where the instructor purposely introduced or made connections to a technical context. Overall, the phase of the activities were observed as reasonably suited for all the participants to 'pack' their own understanding on the spot, when the instructor introduced a new technical content for 'packing'.

For each follow-up plugged-in exercise, some basic Scratch code was provided for the learners as a Parson's problem[2] (Figures 10.8, 10.9 and 10.10). Learners had to develop their basic program using the Parson's problem provided and improve it to obtain the complete program that captures the objective of the respective Unplugged activity. Learners were asked not to use their computer (or any other device) during the Unplugged activity sessions, and were allowed to use them only during plugged-in exercise sessions, which was toward the end of each Unplugged activity.

Example individual activities are analysed in detail and discussed below. Note that the Semantic Profiles illustrated and discussed are abstractions (they are not based on specific measurements). The intention is to explore the concept of Semantic Profiles in general through idealised examples.

[2] Parson's problems provide learners with instructions of a program in a jumbled order, where they have to rearrange in an order that would make the program work.

10.4.2 Activity 1: Kidbots

The Lesson Plan and Execution

Kidbots is an Unplugged computing activity that focuses on the programming concept 'sequencing'. The activity involves three volunteers from the class: a 'programmer', 'tester', and a 'bot'. The instructor first explained the objective of the activity, i.e. reaching a target object placed on a grid by following three basic instructions: move forward: F, turn left: L and turn right: R. As explained earlier, in this collaborative activity, the developer writes a program using the instructions to guide the bot from location A to B on a grid, and the tester then reads out the instructions to the bot, who blindly follows them to reach the target, while the rest of the class participate as spectators. No initial computing knowledge is required to do the activity.

This activity also introduces the concepts of 'algorithms' and 'debugging' in a basic context. Also the concept that there are multiple correct programs for a given requirement, and the concept of a 'complete' language, i.e. that only F+R can be used to get anywhere on the grid, but a limiting to R+L is not complete are covered. The instructor introduced the relevant concepts as well as the related computational terms in a scaffolded manner as the activity progressed. He took a constructivist approach to 'unpack' the technical concepts to suit the audience as the activity unfolded; the moments of introducing the concepts were pre-planned yet were not necessarily pre-set, but rather determined by the expert judgment of the instructor.

After considering the pre-workshop lesson planning and analysing the observational notes during activity execution, the following three Stages that seemed to be influential in the shape of the overall semantic profile were identified:

Stage 1: Introduction

S1.1: Set out the grid and target and explain the objective of the Kidbot on the grid.

S1.2: Select three players and explain their roles.

S1.3: Explain the three instructions (F,L,R), have the bot demonstrate them and relate it to programming languages.

Stage 2: Doing the activity

S2.1: Volunteers wrote and tested the first program.

S2.2: The instructor drew out ideas - the order within the sequence matters; instructions are unambiguous; there are infinite solutions available; introduce debugging; restricted instruction set (e.g. only F and L is complete; only L and R in insufficient).

S2.3: Repeat the activity, bringing in new volunteers.

Stage 3: Wrapping up

S3.1: The instructor drew out ideas - concepts of programming languages; Turing-completeness.

S3.2: The instructor summarised, followed by a plugged-in exercise; related it to the

curriculum; discussed considerations when teaching (i.e. the usefulness, applicability and other concerns when using the activity in a classroom context).

The Semantic Profile

The semantic profile of the Kidbot activity is shown in Figure 10.5. No reference to any computing context (including programming) was made at Stage 1 (S1), except that one of the students was referred to as a programmer. With the instructor's extensive experience in conducting the activity, the overall pacing and introducing technical concepts and/or terms were adjusted on-the-go to suit the audience. The notes of the two observers also indicated that the activity execution was smooth and meaningful for the learners, and therefore no notable alterations were needed during the repeated workshops.

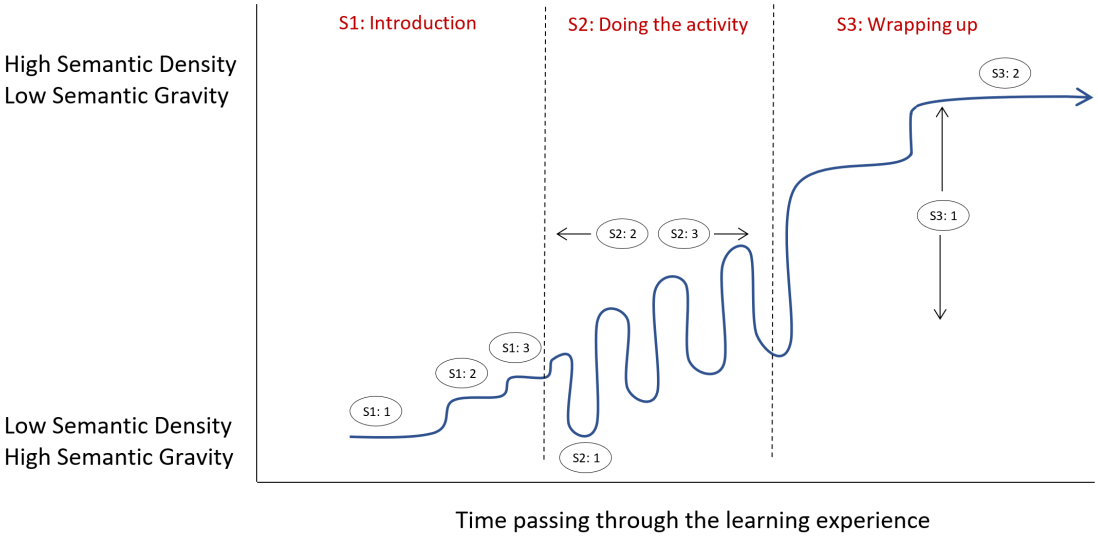

Figure 10.5: Semantic profile of the Kidbot activity

Q1: Is it a wave?

The over-all Kidbot semantic profile follows a step-wise incline, and not a wave structure; it does not have an overall '∩' or '∪' shape. However, the observational notes confirmed that it has micro-waves made locally. It also covers a large semantic range. As mentioned earlier, the curves are drawn in a heuristic and broad-brush manner, thus no quantitative values or absolute values can be inferred from it. The instructor did not introduce any computing context up front, therefore the overall semantic profile has been plotted starting from high semantic gravity region.

During Stages 1 and 3, the instructor took the lead in linking the real world knowledge into technical knowledge. However, he purposely avoided introducing any concept beforehand and linking them back to the real experience, but rather did this in the opposite

order. Therefore, the semantic profile in S1 does not contain any prominent 'downshifts'. Moreover, the instructor's linking and summarizing were intentionally paced to be gradual, therefore the wave-like profile in the S2 region do not indicate 'steep inclines' that cover a large semantic range. The technical context exposed during the introduction was kept to a very minimum: few jargon words were used and the activity was carried out more like a playful game.

Since the participants were adult learners who are also teachers themselves, the nature of the questions from the audience were directed towards the technical concept knowledge expected to be taught, and were concerning how the real world context can be related during teaching in a classroom, which resulted in discussions causing both 'unpacking' by the instructor as well as 'repacking' by learners. The peer support and peer discussions also contributed to learners making connections and linking knowledge during the activity. Accordingly, the semantic wave during Stage 2 was more sinusoidal than step-like.

By Stage 3, the instructor directly linked the activity's context into deeper technical concepts such as 'debugging' and 'programming languages'. Considering the nature of the group (i.e. majority novice to programming) and the main objective of the activity (i.e. introduce the concept 'sequence'), such additional conceptual knowledge could be considerably high in semantic density. Moreover, the plugged-in exercise at the end of the activity, which necessarily implemented the sequence of actions carried out during the Unplugged session, was a direct packing of simple knowledge into a technical context. However, it was observed that, towards the end of the activity, such knowledge transfer was possible, and was smooth and not overwhelming for the participants despite a substantial increase in SD. This leap is indicated by the steep up-shift in the semantic profile. The plugged-in experience at the end helped the learners to cement their understanding despite the fact that the technical knowledge was fully exposed at a very late stage. The semantic profile flattening at the highest SD region indicates the learners' concrete understanding of the intended learning objective. Thus, Stage 3 in Figure 10.5 includes sharper inclines in the semantic profile, indicating the instructor's packing towards higher semantic densities. However, even the over-all flattening curve towards the end of the lesson also involves learners acquiring new knowledge (particularly related to the programming language at this point), thus the flat line hides a small wave itself. This is not a very strong wave, as it only uses concrete examples to unpack the concepts just understood and continue to do so. At this point students are doing it themselves not the teacher doing it.

Q2: How high and low do you go? The activity itself was simple, easily understandable with no computational knowledge required, and could be explained and executed using simple, everyday language (with the exception of occasional words such as 'programmer' and 'bot', which do not need to be understood because their role is defined

during the activity). The profile thus starts off very low in the continuum. By the end of the activity, several computing concepts that directly relate to the activity were successfully communicated and learners were doing hands-on programming, bringing the curve to the highest semantic density level during the session and stabilising. Therefore, the overall shape is a continuous incline, rather than a '∩' or and '∪' shape. The semantic range of a lesson is subjective, and depends on the details of the particular lesson, so the curve here is based on a particular delivery, and might be different for other teachers and students.

Q3: Who is unpacking?

Both 'unpacking' and 'repacking' were largely driven by the instructor rather than the learners. However, a considerable amount of peer discussion and learners asking for explanations or support from helpers was observed as expected in a constructivist approach. This raised valuable questions and answers for the learners. At times the instructor used important questions raised by the participants for explanations to the entire class.

Lesson Reflections

The traversing between SG and SD regions in the semantic profile during the Kidbot activity is also similar to progressing through a conventional explanation of a ZPD (see Section 4.1, Figure 4.1). It is observed that this is largely due to the way the activity was executed, and the expertise of the instructor, which helped orient the participants to share knowledge from the MKO efficiently. The pacing of the necessary scaffolding during the activity was determined largely by the instructor's expert judgement, which resulted in a gradual inclining semantic profile. This way of scaffolding by linking the physical activity with computing concepts, in a manner that causes a overall incline from high SG towards high SD in the overall semantic profile, while having micro-oscillations locally, increased the conceptual knowledge of the learners without making them feel bored or abandoned.

A less expert instructor or a more formal execution setting may need a certain degree of 'unpacking' up-front and a more instructions-based approach to executing the activity, in which case the semantic profile would have looked more '∪' shaped, with a downshift at the beginning. Moreover, the expertise of the instructor and the availability of knowledgeable peers in the learner audience could also result in the much steeper slope in the 'repacking' region (Stage 3). The Kidbot activity was executed in a gradual, well-paced 'upward inclining' semantic profile that seems suitable to an audience consisting of novice and limited-knowledgeable learners. However, effectively executing the activity in such a manner requires a level of expertise of the instructor in order to provide scaffolding optimally during the activity.

10.4.3 Activity 2: Variable Dice Game

The Lesson Plan and Execution

The "Variable Dice Game" is the second activity used in the workshop, with the objective of introducing and establish a firm understanding of the concept of 'variables' and key basic concepts related to variables in a computer program such as value storage and reuse, initialization and assignment.

This activity is a group activity that involves two players, two 'scorers' and a 'counter'. Three sets of similar looking sets of flip cards marked incrementally from 0 to 10 are given to each group, and are used as scores and a counter respectively as shown in Figure 8.1 on page 8.1. The two 'players' each roll a die to battle against each other; whoever rolls the higher value gets a point, and neither scores if the values are equal. The scorers keep track of scores, using the two sets of flip cards assigned to each player respectively, and the counter keeps track of the number of remaining rolls. The dice are rolled 10 times; whoever has the highest score when the countdown flip card reaches 0 is the winner.

The game was played repeatedly, while the instructor gradually linked the activity to computing concepts. These links were based on both initiatives taken by the instructor as well as questions and concerns raised by the learners. The instructor took a constructivist approach to introduce and link the experience during the activity to the programming context. His expertise as well as the nature of the learners largely determined the pace of the activity and the micro discussions that caused packing during the activity. For example, initially the three flip cards were identical, and the learners soon realised the value of assigning a meaningful name to a variable such as using the name "counts remaining" instead of "counter" for the count variable. The observational notes indicated three stages influencing the shape of the semantic profile as:

Stage 1: Introduction and First Round S1.1: The instructor provided a very basic introduction about why storing values is necessary in a computer program and introduced the word 'variables' to the audience.

S1.2: The instructor explained the rules of the game; selected three players and explained their roles, and related them to computing.

S1.3: The learners played one cycle of the game (with no linking to any technical context).

Stage 2: Repeated Play

S2.1: Repeated the activity preferably swapping the people in the roles.

S2.2: Learners question/discuss - they realised the importance of naming the variables; having different initial values (for scores and turns remaining); changing the values in the variables

S2.3: The instructor drew out ideas - explained the importance of naming the variables; discussed initialisation and value assignments in a program; linked several key

concepts of variables with the nature of the flip cards to make meaningful understandings (e.g. how the flip card retains its values until the person change it, and link the idea into a programming context, how they can only store one value at a time, and don't store their "history").

Stage 3: Wrapping up

S3.1: Instructor summarised, and learners moved to plugged-in exercise with variables - summarised the key concepts discussed during Stage 2 and linked them to programming context.

The Semantic Profile

The semantic profile of the Variable Dice Game activity as shown in Figure 10.6 has a slightly '∪' shaped overall structure. In all Unplugged activities used during the workshop, the instructor purposely tried to avoid 'unpacking' technical content as much as possible. However, the very basic reference to the technical context of variables at the beginning has been considered when drawing the semantic wave, and is indicated by the slight downward shift at the beginning of the curve (S1, Figure 10.6). Towards the end of the activity, the learners were more familiar with the objective of the game as well as the purpose of the activity. As a result, the instructor discussed the use of variables in a computer program in a much deeper technical context, which brought the learners to a higher semantic density region within a short period of time. This observation is indicated by the steep upward shift at the beginning of S3 in Figure 10.6.

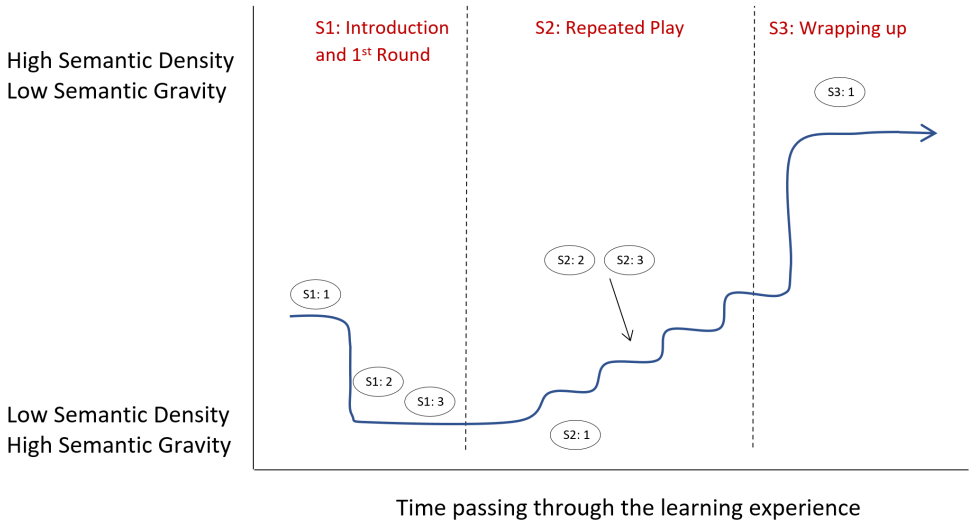

Figure 10.6: Semantic profile of the Variable Dice Game activity

Q1: Is it a wave? As explained in the previous activity, the over-all flattening curve

towards the end of the lesson also involves learners acquiring new knowledge during the plugged-in exercise, thus the flat line is a micro-sinusoidal wave itself. Moreover, the semantic behaviour of the plugged-in exercise that followed is not reflected explicitly in the profile shown. Had it been incorporated, it would likely have as rich a profile as the unplugged part of the activity, but we have not analysed this.

The semantic profile does follow a wave shape that is a '∪' shape overall, but starting with low SD because the conceptual knowledge (e.g. variables) was not introduced immediately. It can be seen that the semantic profile during 'doing the activity' (Stage 2) follows a step-like 'staged return', rather than a sinusoidal wave as in the Kidbot activity (Figure 10.5). Although the activity (the game itself) has no technical context, technical concepts were gradually introduced by the instructor during repeated plays, gradually relating the game to a deeper technical context. Learners noticing knowledge directly related to technical context (e.g. the need to name the flip card holders with unambiguous name such as 'player 1', 'player 2' and 'turns remaining') was observed even during the first cycle of the game. The instructor intentionally took advantage of such existing knowledge to link them to a technical context. Several "Aha!" moments, and responses like "I get it!" and "*Now*, I undertand" were heard among the learners during repeated plays. Therefore, the whole 'doing the activity' region of the semantic profile is a clear 'staged return' to a much higher semantic density. The plugged-in exercise at the end was a direct linking of the physical situation into its technical application, and therefore the repacking could be considered as rapid, indicated by the steep upward shift in the semantic profile. However, as discussed in the previous activity, this rapid repacking did not overwhelm the learners. Conversely, learners seemed to meaningfully apply and use the variables in their program, which indicated that the learners had cemented the concept within a very short period of time, steadying the semantic wave at the highest semantic density region of the session. Doing this involves applying the conceptual understanding in a (now) technical context so presumably some new kind of wave is going on if the plugged part was actually mapped.

Q2: How high and low do you go?

The activity started off with minimal technical knowledge, thus at a reasonably lower region in the continuum. However, every repeated play linked the learners to a higher level of technical understanding, which gradually increased the semantic density. Therefore, no noticeable 'flat lines' were seen either with respect to simpler knowledge or abstract knowledge. Further, despite not having flat lines, upward shifts were also observed to be gradual and not steep. At the wrapping up stage, the instructor could repack several key concepts related to variables in programming, while helping the learners to avoid common misconceptions (e.g. that variables in programming are similar to variables in mathematics). During the plugged-in exercise, the semantic density of the lesson was at its highest. Therefore, this activity successfully traversed from low SD to high SD and

high SG to low SG, indicating the good use of simple knowledge to explain technical knowledge.

Q3: Who is unpacking?

Due to the nature of the learners and the novelty of the concept among them, the 'unpacking' and 'repacking' were largely instructor-driven. However, the "Aha!" moments and simple realisations by the learners during play (often with no support from either the instructor, helpers or peers) were used by the instructor in scaffolding and linking the actions to technical context, as expected in the constructivist approach. This kept the learners engaged, despite their maturity and the simplicity of the game.

Lesson Reflections

If the instructor had chosen to explain the game rules and let the learners play the first round only, without a basic introduction as in S1.1, the semantic profile would have followed an up-shifting only. This could have caused the audience to wonder why they are playing the game, as well as making them less prepared for or feeling lost in repacking to the technical context of variables in a program. Moreover, the introduction at S1.1 made the adult learners in the class enthused and engaged in this simple game more naturally, otherwise it would have been seen as nothing more than playing a very simple dice game. Having a basic understanding at the beginning (i.e. following a '∪' shape in the overall semantic profile with a less accented downshifting) seemed more suitable with these kind of activities.

It was observed that some of the learners (who were also novices to programming) were feeling lost during Stage 1, either not understanding the rules of the game, having difficulty to understand the basic idea of a 'variable', or both. Despite the instructor trying to keep the technical context at the possible lowest, this behavior could be related to the learners' 'anxiety' as explained in the ZPD, where they feel 'what is unknown' is too difficult to comprehend compared with 'what is known'. Somewhat similar behaviours were also observed during Stage 3, where some learners appeared uninterested or distracted from the instructor's summarising. Although the nature of the activity required 'repacking' by the instructor rather than the learner, it was observed that, for some learners who are either novice or generally uninterested, this was probably due to boredom. However, it was observed during the plugged-in session that the majority of the learners immediately started using variables in their programs meaningfully, with minimal support by the instructor or helpers. Nevertheless, a simple variation would be to ask the learners to summarise what they knew about variables before the instructor did. They would then be doing the (re-)packing.

10.4.4 Activity 3: Conditional Dice Battles

The Lesson Plan and Execution

"Conditional Dice Battles" are a set of carefully selected dice games that are designed to introduce the concept of 'selection'. They are designed to start with a very simple game and increase in complexity in each successive game. Each game directly corresponds to a key idea related to how conditionals (i.e. 'if', 'if-else', 'if-else-if' and nested 'if-else' statements) work in a computer program. However, they can be played individually as well. In the case that the games are played in a random order rather than the expected order of increasing complexity, the instructor's involvement in linking the simple knowledge to abstract knowledge may be highly necessary.

The games are played between two teams. One person from the group rolls the dice to battle against another from the other team. Points are earned according to the game rules given in the respective game cards. For example, a simple rule was "if the die value is greater than 3 award your team 1 point". Two people play 'scorer' and 'counter' respectively; the scorer use two sets of flip cards for 'scores' assigned for each team, incrementing from 0; the counter uses one set of flip cards for the 'counter', counting decrementally starting from 10. The dice are rolled 10 times. When the 'counter' flip card reaches 0, the highest scoring team wins the game. Game rules are presented in 'Game Cards', which are numbered according to the increasing complexity of the games. The instructor has the choice to deliver all the game cards at once or deliver one after the other, considering the nature of the audience, time constraints and/or lesson objectives.

During the workshop, all the game cards were distributed to the teams as sets, and the instructor directed a team to pick one card (i.e. one game) and play one game at a time. This activity immediately followed the "Variable Dice Battle" activity, and the learners seem to embrace the activity as an extension to their previous exercise and accepted the reason for playing an Unplugged game to learn a technical concept rather easily. As a result, the instructor could unpack the introductory technical knowledge smoothly and easily. The activity execution steps were as follows.

Stage 1: Introduction

S1.1: The instructor provided a very basic introduction to the concept of 'selection' and set out the the basic instructions for the games (i.e. teaming up, use of flip cards, playing the games by the rules given in the game cards, etc.).

S1.2: The instructor explained how conditions in real world relates to programming using the 'if-else' challenges with examples such as comparing temperature values to give weather information or control a heater.

Stage 2: Playing the games

S2.1: Teams were requested to play the games according to the rules given in the card of their choice.

S2.2: Learners question/discuss the rules - From this they realise how to deal with increasingly complex Boolean conditions in the rules (as well as gaining further experience with variables).

S2.3: The instructor drew out ideas - discussed the concept of nested if statements and how they work.

Stage 3: Plugging-it-in and Wrapping up

S3.1: Plugging-it-in - Learners attempted to code a simple Scratch program to match one of the games of their choice.

S3.2: The instructor summarised the key concepts discussed during Stage 2 and linked them to a programming context.

The Semantic Profile

The semantic profile of the Conditional Dice Games activity also followed a slightly '∪' shaped wave as shown in Figure 10.7. Whenever an Unplugged computing activity was used throughout the workshop, the instructor intentionally started the activities introducing as little technical knowledge as possible. Therefore, the beginning of the curve is closer to the high SG region than the high SD region in the continuum. However, towards the end of the activity, the learning objectives of the activity were observed to be achieved, bringing the end of the semantic profile to the expected high SD region. For this reason, the overall '∪' shape has a shorter downshift at the beginning (i.e. low technical knowledge in 'unpacking') than the upward shift (i.e. linking to higher level of technical knowledge in 'repacking') towards the end.

It must be noted that the low flattening curve at the S1.2 region of the curve includes some unpacking of the game (and so the concepts in the concrete game context) to understand the game. This would have caused micro-oscillations with respect to the knowledge of the game, and is not reflected in the curve. Further, the lesson was extended to the plugged-in exercise, which also included learning the Scratch environment. Thus the end of the semantic profile is indicated as higher than the start point given that terms/concepts were explained at the start.

Q1: Is it a wave?

The semantic profile clearly followed a wave. The downward shift at the beginning of the curve indicated the simple unpacking the instructor did as an introduction. However, he took a constructivist approach, which at this stage was largely discussing the concept in everyday language and not connecting it strongly to computational language. Although the games were presented with an orderly increment in complexity (in the game cards), because the game cards were distributed as a set rather than one after the other (in the intended scaffolding order), the teams used games of their own choice rather than the order they are numbered in the set (e.g. some chose to play the game that was

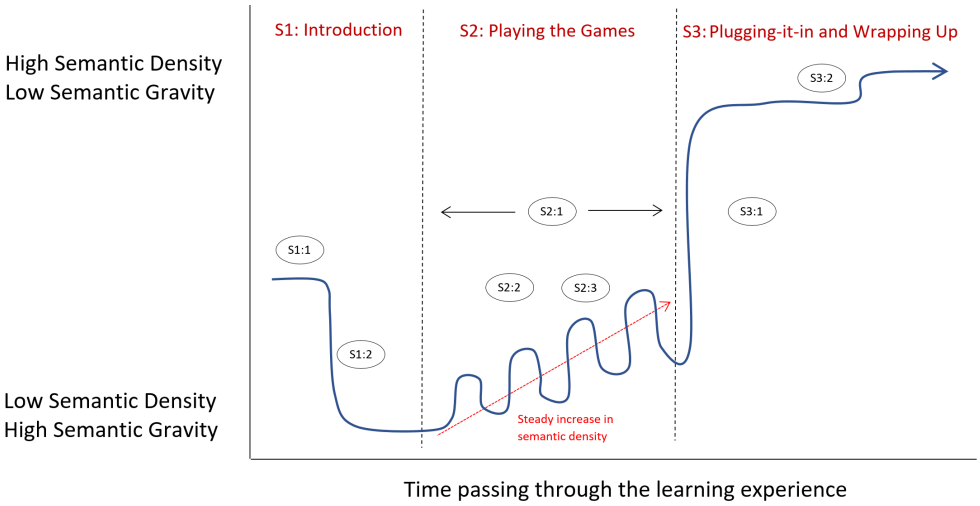

Figure 10.7: Semantic profile of the Conditional Dice Games activity

intended to introduce 'if-then-else' before the game intended for simple 'if' statement). This caused discussions with the instructor as well as among the peers that went back and forth in in SG/SD continuum, instead of a step-wise scaffolding from high SG region towards high SD region. Therefore, the semantic profile during stage 2, where the learners were actually doing the activity, shows a sinusoidal shape rather than a step-wise escalation.

The plugging-it-in exercise at the end of the activity helped the learners to directly link the abstract knowledge obtained during the activity into technical knowledge, causing a steep upward shift in the semantic profile. This contextual difference in a single lesson could be reflected in various ways in a semantic profile since a new context could introduce a downward profile, but based on the observations in this case, we have chosen an upward profile based on increased SD and moving away from SG. It was observed that the steep incline did not affect the learners by causing confusion or feeling lost, due to the nature of the activity (i.e. a collection of simple games of a similar nature) and their close correspondence to the structure of conditionals in a programming language. This was well received by the audience. The instructor's wrapping up summary steadies the curve as it brought the class to a conclusion connecting the technical concept intended to be established by the activity.

Q2: How high and low do you go?

After the initial simple introduction that included only a little technical knowledge, the games in the activity were based on real life knowledge. Therefore, the curve started somewhat higher in the continuum, yet remained fairly low towards the high semantic gravity region during Stage 1 and 2. However, during Stage 2, the learners were

traversing between simpler knowledge and technical knowledge while scaffolding their knowledge to higher technical contexts while playing the different games. Accordingly, the sinusoidal wave also includes an up-shift during Stage 2. The plugging-it-in exercise that followed gave an in-depth technical knowledge to learners, and therefore during Stage 3 the semantic profile reached the highest region of SD during the activity.

This activity was built on a foundation of the previous game, and therefore did not need to go as low in the sense of it now being a known context (hence students seeing it as a continuation). These games are perfectly understandable as the games are based on students' everyday experience. Going to a low level has been noted in that context here. Using very everyday language, despite using words like 'ELSE' and the structure in the game, prevent a really low position. As it was enacted rather than explained, the curve is positioned lower in the continuum as a concrete experience but not abstract discussion of such a game.

Q3: Who is unpacking?

The initial introduction and the simple 'unpacking' was done by the instructor. Since the participants consisted largely of novice or beginner programmers, the learners' direct contribution to unpacking was limited. However, it was observed that the learners actively involved in meaningful discussions among themselves, and knowledgeable peers leading the others during the play. This largely helped the plugging-it-in (S3.1) and instructor's repacking (S3.2) to go smoothly, without overloading the weaker learners.

Lesson Reflections

Considering the nature of the concept to be introduced (i.e. a concept more relating to a programming context than general computing) and the composition of the audience (novice to beginner programmers), a basic unpacking of technical context at the beginning seems more appropriate than directly moving to doing the activity and gradually scaffolding later. Had the game cards been distributed in order of their increasing complexity with the instructor scaffolding at each stage, the shape of the curve would have been a step-wise incline rather than the sinusoidal wave as shown in Stage 2 of Figure 10.7. Even so, an initial basic level unpacking seems effective for this kind of activity.

The concept of 'selection' (if-else statements) was new technical knowledge to many of the participants. However, it was observed that the learning did not seem to overwhelm most of the participants, despite having several games played during the activity and the games (and therefore, the concepts) being chosen at random rather than the originally expected scaffolding order. Several "Ah ha!" moments as well as attempts at micro discussions to cement the understandings were observed among the learners. This activity was the third Unplugged activity during the workshop and was preceded by the Variable

Dice Battle. Therefore, the learners were already familiar with playing games of this nature for learning purposes. The observations (and thus, the shape of the semantic profile), especially during Stage 2, could be different if the activity was used on its own.

10.5 Programming Without Unplugged

This section discusses the semantic profile of a programming lesson that did not use any Unplugged computing activity and followed the conventional, directly-into-programming strategy. The method followed was similar to that explained in Section 10.4.1, attempting to answer the same questions. The audience of interest was of a similar nature (i.e. adult learners consisting of pre-service or in-service teachers; all novice or beginner programmers) and the same expert instructor conducted the class. He followed a similar constructivist approach to conduct the lessons by drawing out ideas from the learners before introducing and/or connecting conceptual and technical ideas to them. However, the simple, everyday knowledge used was largely limited to the use of everyday language in explanations in making connections with computational concepts. The introduction of the three programming concepts: sequence, variables and selection, is analysed to identify the semantic profile(s). The total duration of one session was the same (two hours) as an alternating Unplugged session, and observations of two sessions were recorded. The pace that a new concept was introduced, as well as the time spent on introducing each concept was also observed as being closely similar to the Unplugged workshops.

The plugged-in only programming exercises were based on the same set of Scratch programming Parson Problems as the alternating Unplugged workshops, with additional problems introduced to reinforce the concepts. The learners were introduced to the Scratch programming environment at the beginning of the session, before opening the Parson's problems related to the three concepts. Figures 10.8, 10.9 and 10.10 show the initial view of the Parson's problems for the three concepts sequence, variable and selection respectively.

Each concept was introduced starting with a simple introduction by the instructor that mostly unpacked the connection between the everyday terms and the computational language. The instructor then provided a simple introductory Parson's problem that initially contained only a basic set of Scratch commands that would lead to creating a program to generate an output similar to the exercises in the corresponding Unplugged sessions discussed in Sections 10.4.2, 10.4.3 and 10.4.4. Accordingly, learners created programs for the three concepts; for 'sequence': moving the Sprite to a certain target; for 'variables': implement a game similar to "Variable Dice Game"; for 'selection': implement game(s) similar to "Conditional Dice Battles". The instructor provided the necessary scaffolding for the learners to achieve the expected learning outcomes by completing each of the programming exercises. The following Stages were identified during every

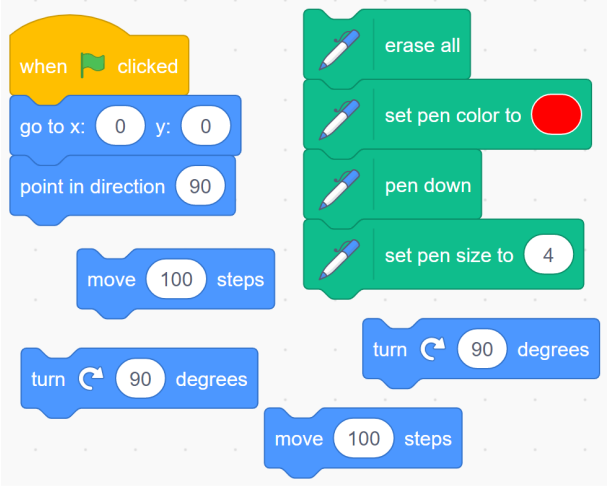

Figure 10.8: Scratch programming Parson's problem for 'sequence'

exercise.

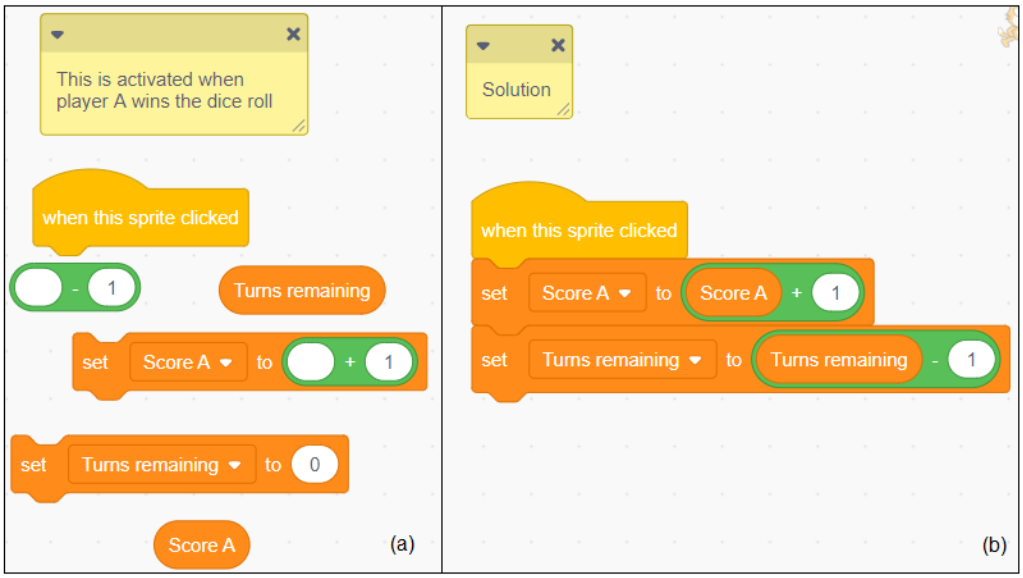

Figure 10.9: Scratch programming Parson's problem for 'variables' - 'Player A' sprite of Variables Dice Game: (a) Problem (b) Solution

Stage 1: Introduction

S1.1: The instructor provided a very basic introduction to the concept, mostly unpacking the connection between the everyday terms in use and the computational language.

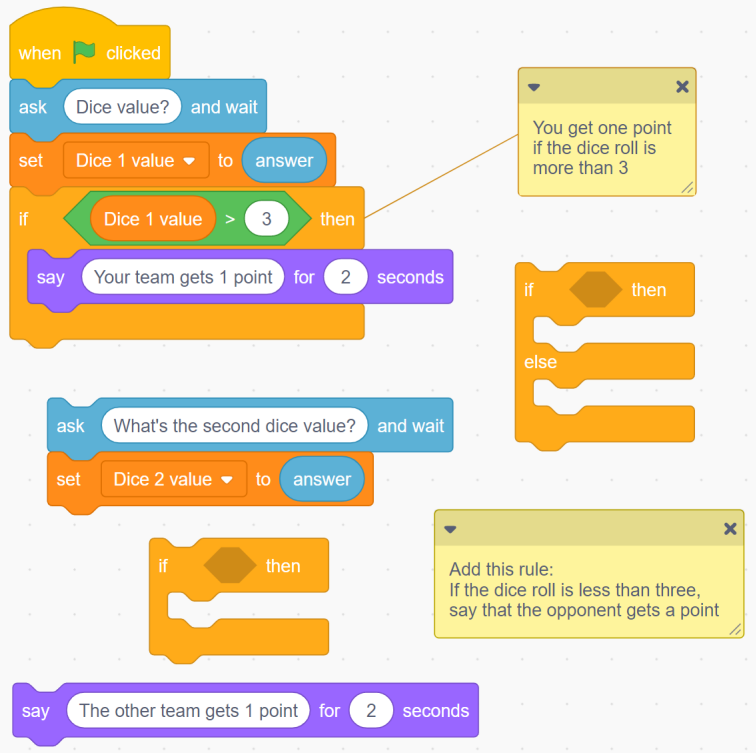

Figure 10.10: Scratch programming Parson's problem for 'selection'

S1.2: He introduced the objective of the Parson's problem and basic instructions provided.

Stage 2: Coding exercise

S2.1: Learners started coding by achieving the initial objective set out in the Parson's problem.

S2.2: The instructor provided instructions to improve the learners' code the next level.

S2.3: Learners improved their code to incorporate the new instructions.

Stage 3: Wrapping up

S3.1: The instructor summarised the key concepts discussed during Stage 2.

The Semantic Profile

The semantic profile of the plugged-in only programming lesson is given in Figure 10.11. The profile curve stayed at the higher semantic density region of the continuum with very little movement towards high semantic gravity. Although the instructor's explanations and unpacking generated wave-like variations during the lessons, the variations are not highly accentuated, as the connection to everyday knowledge during the learning process

was limited and the learning revolved mostly around technical and conceptual knowledge.

Q1: Is it a wave?

The instructor purposely attempted to relate the technical and conceptual programming knowledge discussed in the class to everyday examples and experiences, and tried to keep his explanations simple so as not to load the learner in a manner that the conceptual instruction makes them feel overwhelmed or uncomfortable to learn it. His attempt was successful, mostly due to his expertise in the subject and experience in working with similar groups. For this reason, slight wave-like variation is included in the semantic profile. However, the semantic profile was a less-accented, flatter curve compared to the semantic waves discussed in Sections 10.4.2, 10.4.3 and 10.4.4.

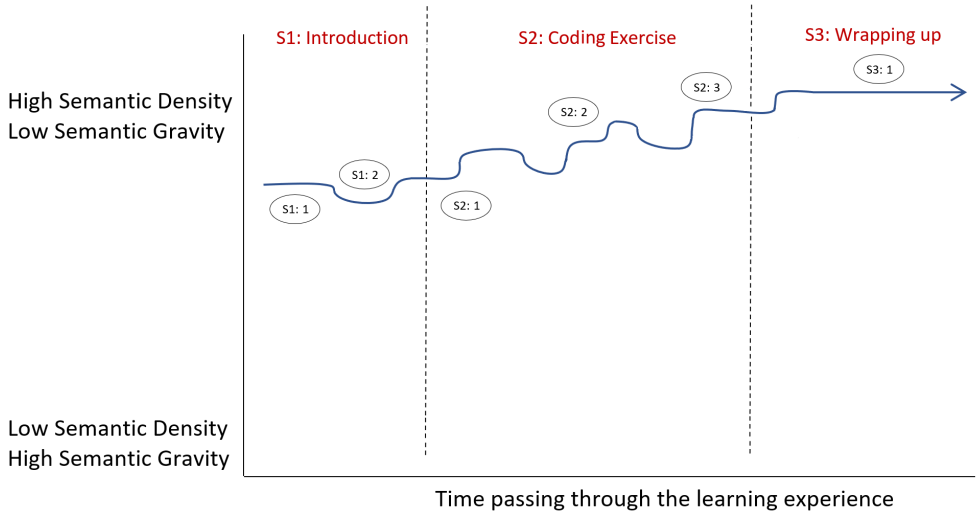

Figure 10.11: Semantic profile of introductory programming without Unplugged examples

Q2: How high and low do you go?

Overall, the curve remained at a higher semantic density region. Moreover, it lacked steep upward or downward shifts. There was limited use of everyday knowledge in the discussions, and thereby less high semantic gravity region reach in the profile. And as explained in Q1 above, the curve stayed at the higher SD region and was rather flat (cover small semantic range).

Q3: Who is unpacking and repacking?

The instructor was responsible for the majority of unpacking. However, because the exercises were programming, it was observed that knowledgeable peers (i.e. the participants who were familiar with programming in Scratch) actively engaged in discussions and helping novices with their coding. References to everyday knowledge were limited

and therefore, the need for repacking that linked knowledge of high semantic gravity to high semantic density with the support of an instructor was limited.

Lesson Reflections

Despite the constructivist approach used throughout the lessons, knowledge delivered during the plugged-in only programming session revolved mostly around technical knowledge and conceptual knowledge. The instructor and the helpers supported the learners in familiarising themselves with programming in Scratch prior to any conceptual knowledge being introduced. The semantic profiles of the programming experience therefore remained at the higher SD region in fairly flat curves. However, the instructor introduced key ideas to the class gradually through challenges to improve the initial Parson's problem. Therefore, the semantic profile contained slight yet step-wise up-shifts during Stage 2.

10.6 Discussion

A comparison of the shapes of the overall semantic profiles of the two strategies are given in Figure 10.12. S1, S2 and S3 are the 'introduction' stage, 'doing the exercise' stage and 'wrapping up' stage respectively, for all three concepts. In the alternating Unplugged strategy, S2 indicates learners doing the Unplugged activity, and the actual programming exercises were introduced afterwards at S3, whereas in the plugged-in only strategy, S2 is coding exercises that extend to S3. Accordingly, the alternating Unplugged groups spent relatively less time in actual programming than the plugged-in only groups. However, as mentioned earlier, in the alternating unplugged lesson the semantic behavior of the programming exercise at the end (S3 region) has not been reflected explicitly.

Only the strategy of introducing the programming concepts was different (i.e. alternating Unplugged activities with plugged-in exercises vs plugged-in only strategy). Other elements of the lessons were kept very similar, if not the same; the same three programming concepts were introduced; the programming exercises used (i.e. Scratch programming Parson's problems) were the same for both groups; a similar constructivist pedagogic approach was used by the same instructor in both groups; and the nature of the learner audience was similar. Therefore, the two overall semantic profiles in Figure 10.12 can be considered as reflective of the two strategies used. The semantic profiles during Unplugged computing activities and their follow-up programming exercises are indicated by blue (lower) curve in Figure 10.12 and the plugged-in only exercises are indicated by the red (higher) curve.

In all three unplugged activities and the plugged-in exercises following, the original intention of the instructor was to initially use the learners' existing knowledge and gradually push the learners towards deeper technical/conceptual knowledge. Such a strategy

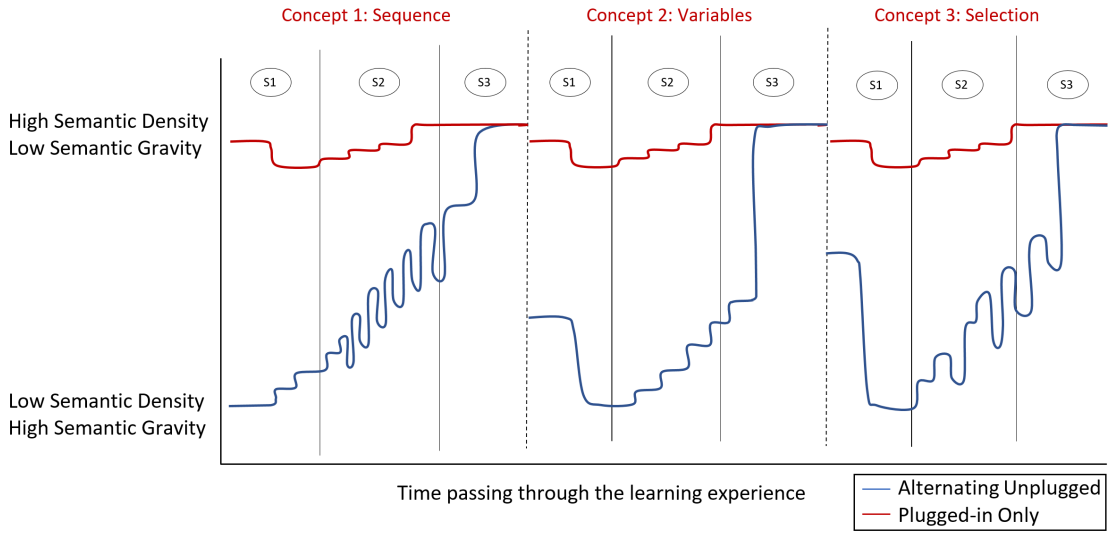

Figure 10.12: Semantic profiles of introductory programming with and without Unplugged examples

also relates closely to the concepts of ZPD. Preferably, such a strategy would have resulted in a semantic profile that starts at a high SG region and works towards a high SD region of the continuum, thereby creating an up-shift in the semantic profile at the starting point of every S1. However, this was accomplished only in the Kidbot activity, in which no reference to any technical context was used at the beginning of the activity (other that the use of words like 'bot', which were used to label students, rather than being defined in a technical sense). In the other two activities, a simple yet basic introduction to the technical concepts was given up-front by introducing some technical terms and/or linking them to general knowledge. Moreover, after the first activity, the learners were more familiar with computing and programming. These basic differences during the introduction stages of the second and third activities affected the overall shape of their individual semantic waves.

However, overall, the alternating Unplugged groups started off with much higher semantic gravity than plugged-in only groups. All have been shown as ending with the same SD, since learners essentially achieved understanding of the same programming concepts in both cases, and this matches the observations by Hermans and Aivaloglou [115]. Nevertheless, this observation was not evaluated with any kind of a programming assessment such as a test, and the semantic profiles are drawn heuristically based on the observational notes of the 'general feel' of the two classes. Therefore, it must be noted that in Figure 10.12, although the semantic profiles of the two classes have been drawn as ending at a matched SD level, in reality it may have had gaps between the two profiles.

During all three Unplugged activities the learning during S2 was mostly micro dis-

cussions both between instructor and learners, and among learners. Often the instructor linked the Unplugged experiences to technical knowledge. Whether the shape of the curve during those stages was sinusoidal or a step-wise incline, learners' knowledge was moving from simple knowledge to a deeper conceptual and technical knowledge. All three subsections of the semantic profile of the alternating Unplugged strategy (i.e. the blue line) are the shape of a wave that cover a large semantic range in the continuum. Noticeably, none have steep downward movement. The semantic profiles, particularly during S2, could be seen as a close, direct interpretation of concepts from the ZPD, where the simple/concrete knowledge known to learners was used to introduce new abstract/conceptual knowledge (by a more knowledgeable other). Therefore, alternating Unplugged to introduce programming concepts seem to: 1) cover a much larger semantic range reflecting cumulative knowledge building and 2) help situating the learner well within the ZPD.

During the alternating Unplugged sessions, the instructor introduced the programming exercises only towards the end of each activity, along with the necessary scaffolding to repack the Unplugged knowledge into a programming context. Learners spent comparatively less time using the Scratch programming environment to do actual coding for each of the exercises compared to the plugged-in only groups. The steep upward shift in S3 of all three alternating Unplugged semantic waves (blue curves) indicates the learners moving from an Unplugged activity to a Scratch programming exercise. Although much less time was spent at the high semantic density region, it was observed that learners did not find difficulty in cementing the conceptual knowledge or embracing the technical knowledge, despite being exposed to actual programming at a much later stage. Moreover, several audible exclamations of understanding something such as "Aha!" could be overheard among the participants. Therefore, this strategy seems to be an easier yet productive strategy to introduce programming to novice and beginner programmers, compared with pushing them through a frustration zone, and could possibly prevent them from feeling disengaged or anxious. However, due to the fact that the connection between the Unplugged experience and the technical knowledge is not obvious to the learner themselves, an instructor's involvement in scaffolding ideas and relating the Unplugged experience to technical context is essential in Unplugged computing activities. In the absence of such meaningful instruction, an Unplugged activity can leave the learners in a low-flattening semantic profile, stagnating them in a boredom zone, only improving their competency in doing the Unplugged activity and not progressing through the ZPD.

The semantic profile of the plugged-in only strategy (indicated by the red curve in Figure 10.12) was comparatively flatter than that of the other strategy. This indicates a smaller coverage in semantic range. Moreover, due to the low use of everyday knowledge of high semantic gravity during the plugged-in only lessons, the curve remained in the high semantic density region. Given the limited everyday knowledge used, the higher

flattening semantic profile is also an indicator of learners falling out of the ZPD, pushing them towards a zone of a higher level of challenge and content difficulty (anxiety zone). However, the learners spent more time doing hands-on programming, thus had comparatively more time to master the tool as well as developing conceptual understanding using actual programming.

At the end of both kinds of introductory programming sessions, based on observations such as conclusive remarks made by learners, questions asked at the end of the class, etc., the learners were observed to have achieved a fairly similar level of understanding of the use of the three basic concepts in a Scratch program. However, the alternating Unplugged strategy lessons covered a much larger amount of material compared to the plugged-in only lessons within a similar period of time. A pre and post survey (that was used to measure the participants' level of self-efficacy towards teaching Computational Thinking as well as computer programming, though not testing their technical ability just self assessment of believed efficacy) indicated that both of the groups had improved their self-efficacy, but there was no notable difference in the level of achievement between the two groups. Both the semantic profiles flattening at a similar SD level is indicative of this observation. However, the alternating Unplugged group was able to spend a much larger time traversing a longer semantic range (i.e. stayed within the ZPD) and arrived at the same semantic density level later, yet cemented their understanding within a much shorter period of time. Moreover, some participants from the plugged-in only group mentioned that although the workshop was useful for their own personal programming knowledge, it was less useful in adapting the material into their own teaching in a classroom. Considering that the participants were teachers, this meant that they found it less useful from a professional development perspective and that they saw more benefit for young learners from teaching unplugged.

The authentic context of a learning objective in a programming classroom can vary vastly depending on many parameters such as grade, level of knowledge, purpose of the lesson, etc. A teacher can determine the strategy they may use to introduce a programming concept by considering the semantic range they intend to cover during a lesson and taking into account that alternating Unplugged can cover a larger semantic range than plugged-in. Given that an Unplugged activity works directly on a learner's existing knowledge (i.e. high semantic gravity and low semantic density), it is at the commonsense extreme in a commonsense-uncommon sense continuum, whereas a plugged-in exercise that requires a degree of technical knowledge (i.e. high semantic density and and low semantic gravity) places at the opposite extreme. The larger semantic range in an alternating Unplugged strategy therefore could justify why the Unplugged activities are known to be more engaging and equally productive in teaching programming than the conventional plugged-in only approach [115].

The plugged-in approach on the other hand, can be useful in introducing technical

and conceptual knowledge very quickly (high-flattening semantic profile), and is thereby capable of generating new knowledge in a shorter period of time if the learners have a suitable mindset. A teacher may also have the flexibility to determine the semantic range of a lesson depending on the motivation of the audience to learn programming. For example, an intrinsically motivated audience may be ready to accept a content of high SD compared to an extrinsically motivated audience. If the learners are intrinsically motivated, they may be curious about computing and that makes it easier to work with them compared to extrinsically motivated learners, who may be equally ready to work yet with no such inherent enthusiasm. Despite their motivation to attend the class, an extrinsically motivated audience may not be as ready for high SD content, and therefore starting from high SG rather than high SD for such an audience may be helpful in course delivery as well as effective learner engagement and better understanding. However, its worth making a note of the fact that they may also still find it difficult/impossible if they have no appropriate conceptual underpinning to build on however motivated they are.

A plugged-in strategy may be more productive for an intrinsically motivated group, considering the ability to cover a larger conceptual/technical knowledge faster than the alternating Unplugged strategy, whereas the latter strategy may be far more productive in providing scaffolding to lesser or mixed motivated groups. While Unplugged lessons could be useful for a more engaging classroom, provoking Computational Thinking (mostly in the early stages of programming learning) and helping curriculum integration, the plugged-in only programming lessons could be useful to identify specific talent, creating computational output and providing much more focused technical knowledge. This latter observation also suggests that plugged-in only might be less effective at reaching those who do not see themselves as talented at programming.

10.7 Conclusion

Computer programming is a highly technical subject with abstract concepts and specific terminology. Computing concepts tend to be invisible, intangible and largely new and unknown to beginner learners. When learning to program, learners face difficulties in understanding concepts and technologies and relating the use of everyday language to computer terminology. Relating technical and conceptual knowledge to learners' existing knowledge becomes a valuable as well as effective teaching strategy when introducing programming concepts to novice and beginner programmers. Analysing the semantic profiles of different teaching strategies used in an introductory programming course provides an insight into the usefulness of using learners' everyday knowledge in establishing new, conceptual understandings. The use of Unplugged computing activities is an effective way to incorporate learners' existing knowledge to introduce new computing conceptual knowledge. Two strategies for teaching an introductory programming course were ex-

plored and analysed to study their semantic profiles to understand the effectiveness of each strategy. The strategies were used in a series of teachers' professional development workshops, where we speculated that the nature of the learners and their background (e.g. adults/children, novice/expert, students/teachers, etc.) as well as the nature of the teacher (e.g. expertise in computing, experience in teaching, etc.) become determining factors to the shape of the semantic profile of a lesson.

Unplugged activities provide sufficient semantic gravity (using everyday knowledge and unpacked density) that teachers are enabled to find effective pedagogical strategies for effective lesson planning in their classrooms, especially with younger or novice students. For non-expert teachers, this helps deepen their own understanding, building confidence in teaching computational or programming content in classrooms. The analysis of the semantic profiles show that traversing in the high SD/low SD level and high SG/low SD continuum can be closely related to the concept of a learners' movement in the ZPD during learning. A relatively flatter semantic profile could be an indicator of either anxiety (if remains too long at a higher semantic density region), or boredom (if it remains too long at a higher semantic gravity region). Lessons with flat semantic profiles similar to the plugged-in only strategy could make novice and beginner learners anxious and lost. The alternating Unplugged strategy successfully covers a wider semantic range than the plugged-in only strategy. It provides a possible explanation for the effectiveness of Unplugged computing, as it indicates the possibility of avoiding both learner anxiety as well as boredom, or at least long periods in these states.

Even though the plugged-in only approach was successful for professional development for a teacher, it may not provide the best model for teachers to use in their classroom, particularly with younger students. Alternating Unplugged would have given them a model of how to teach programming to their students. Research has indicated that children learn programming better with incorporating Unplugged activities [115], and this analysis provides some insight into why this is the case. Therefore, alternating Unplugged in introducing programming to teachers not only is an effective strategy to teach programming to them but it also exposes them to a better and effective model that they can use with the learners in their classrooms.

This semantic wave study provides some descriptive reasoning for the success of using Unplugged activities in programming education, while giving a detailed insight to what and how they can be incorporated in a programming lesson effectively. The simple, explanatory nature of a semantic profile (i.e. the movement of SG and SD) provides a graphical understanding to the different teaching approaches (while taking into consideration the parameters like the level and/or nature of the audience, instructor's expertise, etc.), that could provide insights to teachers/instructors, advisors and researchers how to approach/organise their Unplugged programming lessons/research, according to the nature of their audience.

In an alternating Unplugged strategy, where the semantic profile traverse across the SD/SG continuum, the indications of returning cycles between a learners' System 1 and System 2 cognition as explained by the DPT also rationalises this difference. From a DPT perspective, using Programming Unplugged activities adds a few things in a lesson:

- It makes programming accessible to younger and/or novice groups of learners, who are either not well equipped with or could not yet cope with rule based systems, in which the traditional plugged-in only approach might not be successful. It provides a format accessible to the younger/novice groups. However, the data most studies have about this alternating approach is much too short term to capture the true effect of Unplugged computing experience of learners. Introducing programming to young and novice students in a form that they can handle with System 1 cognition may result in much better outcomes over time, which would require a longitudinal study and cannot be captured in a short period of time.

- Unplugged computing helps learners in crystallizing the knowledge. Moreover, as explained, it does this is by bringing learner's SG and SD into the ZPD very effectively. During Programming Unplugged activities, a teacher probably takes knowledge of high SD (System 2 type knowledge) for learners currently outside of the ZPD and brings them into the zone by giving them more points of reference from the knowledge of high SG (System 1 type knowledge) from what is already known and already in crystallized intelligence and associative memory.

Semantic profiles are a useful lens through which to view a learning activity, and mapping a semantic profile raises useful questions. At what semantic level (or in what context, technical or practical) does an activity begin? Does its progression draw on learners' authentic practical knowledge / experience in order to support its goals at the technical / abstract level? Is the intended profile adequately reflected in the structure of the task, and its instructions and resources? Are there steep transitions that need to be carefully managed? Is the activity as a whole situated within the target learners' ZPD? Using semantic profiles to explicitly explore and answer such questions may help teachers to design and deliver effective learning experiences. While the focus here is on the Unplugged framework, other kinds of activity offer similarly rich opportunities to span a wide semantic range, combining practical and abstract elements. These include tangible computing and robotics, and any kind of task that is based on personal contexts, real-world case studies or examples, problem-based learning, or similar approaches.

Chapter XI

Conclusions

The widely used unplugged approach in computing education can evidently be used for teaching and learning programming, despite the paradox that it does not use a device or any programming language. Understanding the nature and application of unplugged activities in Computational Thinking and introductory programming could be helpful to address the challenges faced by teachers teaching computing in the classroom (especially since most are novices to computing). It could also be useful for developing resources that help improve their knowledge and self-efficacy for teaching these topics. This thesis has aimed to address these possibilities by analysing the use of unplugged computing in depth, and specifically its applicability in teaching and learning programming, and teachers' professional development.

This was done by first studying the relationship between Computational Thinking, the concepts of Notional Machines in programming, and unplugged computing; second, by investigating how unplugged impacts on teachers' professional development; and finally, studying *why* they are useful when combined with programming. To do this we carried out a thorough theoretical analyses of educational theories and other literature, and experimental studies with New Zealand teachers on using unplugged as a tool to teach programming. The experiments included creating unplugged activities that specifically model programming concepts, trialling them in introductory programming lessons as part of teachers' professional development, and studying teachers' self-efficacy towards teaching Computational Thinking and programming after using them. The findings of this research have contributed a rationale for the success of unplugged computing in teaching and learning Computational Thinking and programming topics, have produced a set of programming specific unplugged activities that can be directly converted to plugged-in programming exercises, and propose an effective approach to combine unplugged activities with teaching and learning programming.

11.1 Research Outcomes

The goal of this research was to understand the success of unplugged computing in computing education by looking at it through the lens of some important educational theories, and with experimental studies using unplugged activities in introductory programming,

investigate their usage in teachers' professional development. This thesis has made significant progress towards achieving these goals. This section integrates our answers to the research questions this thesis aimed to address.

RQ 1: *How does CT relate to the mental models that students are forming?*
The Computational Agent (CA) in CT provides a way to establish links between computational and programming concepts, and robust Notional Machines (NM). The nature and flexibility of its definition that allows the CA to be a human as well as machine could be useful to students in forming accurate mental models, as it allows learners to have tangible interpretations to some intangible aspects of computing, especially at early programming learning stages, and this is very much in line with what an unplugged approach does. The more the CA becomes closer to the NM, the more helpful it is for the students to create accurate mental models.

A CA facilitates teachers' consciousness of their own mental model against the NM they are expected to teach, especially if the teachers are computing novices. The CA's flexible nature potentially eliminates the need for novice teachers to be aware of NMs altogether, while supporting a similar learning context. When an abstract NM is defined without reference to an actual machine, a CA presented with a set of precise yet less detailed instructions can be useful for pedagogical explanations.

RQ 1.1: *How can models for student learning inform our understanding of learning CT?*

In the theoretical analysis we showed a transition from CA to NM as a continuum, as discussed in Chapters 4 and 7. This context allowed us to interpret our understanding in terms of a basic cognitive psychology learning theory, the Dual Process Theory, which matches this transition from a System 1 type learning (associations arising from long-term memory) to gradually shifting the balance until at the end (which may be as late as the time of undergraduate level studies) when it becomes System 2 type learning process (that involves conscious reasoning with explicit rules in working memory).

RQ 1.2: *How does the view of CT vary with the stage that teachers/learners are working at?*

Several stages were identified that the relationship between the CA and NM encompasses in computing and learning to program, and is discussed in Section 7.6, Chapter 7. We argued how learning to exercise CT begins with a CA but with no relatable NM initially, and as the learning proceeds and as CT is scaffolded to incorporate programming into the learning process, the roles of the CA and NM also converge subsequently.

RQ 2: *How does the use of unplugged as a PD tool impact teachers' confidence, expectations, and knowledge?*
Based on the results of the main experimental studies, the alternating unplugged approach (where unplugged activities are used to introduce programming concept) increased both pre-service and in-service teachers' teaching self-efficacy and motivation to teach Digital Technologies. However, in our experiments, their self-efficacy towards programming decreased, indicating the possibility that the amount of material when alternating unplugged activities in a programming lesson may overwhelm learners. Through interviews, teachers pointed out that the unplugged activities can provide tangible interpretations of the otherwise intangible or highly abstract concepts of computing, indicating that they are aware of a connection between learners' mental models with the Notional Machine (albeit unknowingly) and recognised unplugged activities as a useful pedagogical tool for that. They also pointed out that tangible unplugged activities could give the teacher more insight into students' mental models.

RQ 2.1: *What are the impacts of alternating or combining unplugged teaching methods with conventional "plugged-in" (computer based) methods when providing PD for teachers?*

The use of unplugged computing activities is an effective way to incorporate learners' existing knowledge to introduce new computing conceptual knowledge. Several impacts of alternating unplugged activities were identified during the experimental studies and discussed in Chapter 9. Chapter 10 looks at the two approaches through the lens of Semantic Waves and provides a rationale for the success of alternating unplugged in programming lessons, based on the observations during experimental studies.

RQ 2.2: *What are teachers' expectations for students who experience unplugged in a CT curriculum?*

The qualitative data analysis findings of the Pilot Study and the two experimental studies (in Chapters 5 and 9) indicate that teachers appreciate, use and have positive expectations about unplugged content as a useful tool in a CT curriculum. They expect unplugged activities to model programming concepts in a more focused manner than unplugged activities that model more general computing concepts, because with the unplugged activities that do not focus on programming specifically they found it difficult to make meaningful connections to programming by themselves. Accordingly, a set of unplugged activities that focus on modeling common basic programming concepts were created. Chapter 8 covers a detailed discussion on Program-

ming Unplugged activity development, that covers a range of concerns and considerations in this regard.

RQ 2.3: *What is the nature of teachers' understanding of computational terms (jargon) related to CT concepts, and how can a relevant professional development intervention help to resolve issues related to them?*

The Pilot Study findings indicated that the teachers find the vocabulary used in computational contexts difficult to understand, even if the meanings of the words and phrases used in them are known to them from other contexts. An appropriate intervention that can essentially relate the technical language to more familiar contexts can resolve language difficulties in computational contexts. A detail discussion in this regard is given in Chapter 6.

RQ 3: *Why is unplugged useful when combined with programming?*
Using Semantic Profiles from Legitimation Code Theory, we were able to explain the success of unplugged activities when combined with programming. We showed that, heuristically, the Zone of Proximal Development (ZPD) can be seen as a differentiation (rate of change) of a semantic profile of an unplugged computing activity, essentially shifting learners back and forth between existing and new knowledge at an appropriate rate, while learning a programming concept. Our theoretical explorations could connect unplugged computing with a basic cognitive psychology learning theory (Dual Process Theory) which fitted with the Semantic Wave explanations. This opened insights to how learners retain knowledge better when they connect it to other things that they already know or when they can elaborate new knowledge with personal experience and other pre-existing knowledge. Unplugged computing enables this, building strong connections from a learners' existing knowledge to the new knowledge, and supporting a process of crystallising their learning.

RQ 3.1: *What is the role of unplugged style activities in the context of the computing concepts they address?*

A careful analysis showed that unplugged computing activities are versatile in modeling computing concepts from two perspectives: the general computing context and a programming context. We identified these as two separate roles, which we named as Non-programming Unplugged activities and Programming Unplugged activities respectively (discussed in Chapter 3). A novice teacher may not be able to see all the connections an unplugged computing activity can have to programming and may feel at a loss to support their learners to make such connections. Yet, making such connections available can possibly be overwhelming for them as well. Therefore, regardless

of the computing concept they model, maintaining simplicity and allowing time for engagement is a key consideration in unplugged activities. The context of developing useful Programming Unplugged activities are discussed in Chapter 8.

RQ 3.2: *Is there a need for a separate, programming focused set of unplugged activities that connects the concepts through "plugging it in"?*

A theme that emerged from the interviews with teachers in the Pilot Study was that they had difficulties in connecting unplugged activities to programming. A set of programming-focused unplugged activities that can be directly converted to programming exercises was needed to address this issue, since general computing concepts (such as data representation or sorting algorithms) to overshadow the unplugged activities' ability to communicate basic programming concepts. Chapter 8 covers the development process of such focused unplugged activities, with detailed discussions about the concerns and considerations, and resulted in new activities that addressed the need for programming-focused material). Chapter 9 discusses the evaluation of the applicability of such programming focused activities alongside non-programming ones in the context of teachers' professional development in introductory programming, and the findings confirming that combining unplugged with plugged-in is more useful in teaching programming.

RQ 3.3: *What are the features of unplugged that are useful when combined with programming?*

Observation based, heuristically drawn Semantic Profiles showed that formal and traditional plugged-in teaching in programming do not use a large semantic range. When an unplugged activity is used (or also a metaphor, an analogy or any similar reference to the learner's concrete knowledge), a wider semantic range is introduced, which could potentially pull topics from far away into the learners' Zone of Proximal Development. This is discussed in Chapter 10.

RQ 3.4: *Does unplugged help teachers build a useful Notional Machine?*

Experimental findings indicated that the teachers have seen unplugged activities as providing tangible interpretations to the otherwise intangible or abstract concepts of computing. Even without any specific knowledge about the Notional Machine and having it not being even slightly mentioned in the professional development programmes, teachers seem to have realised a

need for a connection to a machine-like conceptual presence when learning to program, and that unplugged activities provide learners with a tangible connection to that conceptual model. Having unplugged activities incorporated in lessens may have triggered these realisations. The findings of the theoretical study indicated how a CA in CT, particularly a human/physical CA, can provide a very physical interpretation to (at least some parts of) a NM. These discussions (in Chapter 7, 9 and 10) combined show how unplugged computing can potentially be helpful for teachers to build a useful NM.

RQ 3.5: *How can a more programming-focused set of unplugged activities help teachers build useful Notional Machines?*

The representational and abstract nature of such Programming Unplugged activities indicate that comprehensions are needed by the learners that relate to their everyday knowledge, in order to understand the computing concepts. As discussed in Chapter 7, the correspondence of CA in CT and NM made through unplugged activities, and its connection to learners' mental model development, provides a good proxy to support a teacher's self-efficacy in communicating a robust NM to a learner.

RQ 3.6: *What impacts do programming-focused unplugged activities have for teachers and learners?*

Programming-focused unplugged activities allow learners to cover a larger semantic range and teachers to use more teaching and learning material during programming lessons. We have analysed theoretically and proved experimentally (in Chapters 4, 7, 9 and 10) how alternating unplugged activities with programming concepts, or trying to connect programming concepts with unplugged activities wherever possible, can evidently be successful in teaching both Computational Thinking and programming concepts. Both qualitative and quantitative analysis of the experimental studies show that introducing the programming concepts using an unplugged activity first and then moving to plugged-in exercises was an effective order to combine unplugged in programming lessons. However, some views from experienced teachers indicate that, as long as the lesson combines unplugged with plugged-in exercises, this order may not matter greatly.

11.2 Potential Directions for Future Research

Unplugged activities can make lessons more enjoyable and fun, which feed students motivation and engagement. Although that might not necessarily be reflected in a measure of knowledge, it might well be reflected in a measure of future engagement with the field. People who learnt computing with unplugged approaches and enjoyed it may be more likely to carry on with future studies in the field than those who had a very traditional experience that they did not enjoy. However, this is a long term argument that cannot be tested in short-term snapshot studies. A longitudinal study is needed to measure the long-term effects of unplugged approaches in early computing education.

This thesis has explored the connection between alternating unplugged activities in learning to program and cognitive learning theories (rather theoretically) that could potentially explain their success and popularity among the grade school[1] community. Experimental research aiming to further explore these links could be useful in understanding the relationship between socio-cultural experiences, everyday knowledge, and Computational Thinking and programming aptitude. During our discussions, one expert teacher highlighted how unplugged activities provided opportunities to young 'logical thinkers' in her classroom, who otherwise would not have had proper recognition. Such studies may be useful in identifying and nurturing students who are particularly good at computing, from a young age. Unplugged computing could also be useful in communicating programming concepts to more 'non-rational thinkers' and special groups in classes, so that they may increase their enthusiasm and feel less uncomfortable towards learning computing topics.

The Programming Unplugged activities we have created and used, that can be directly converted to programming exercises, propose an evidently successful approach to combining unplugged activities with programming lessons. These activities take unplugged computing a step further by focusing on modeling and delivering programming concepts. They have shown success in a teachers' professional development context. However, due to the limitations experienced by COVID19 during the course of this study, the findings could not be supported with strong quantitative evidence. Therefore, further experiments with larger sample sizes are needed in support of these findings. Moreover, their applicability in a classroom context with younger learner audiences also needs to be studied.

With school curricula focusing on moving students from being *users* of digital devices to *creators* of computing technologies and digital outcomes, the importance of teaching Computational Thinking in grade schools is growing. Many tools and technologies (with and without the use of digital devices) are available to support teaching and learning

[1] By grade school we refer to the elementary, middle and high school levels, also known as K-12, or primary and secondary school.

CT, and therefore, evaluating how effective these pedagogical tools and technologies are has also become important. Considering how the intense involvement of the computer in a much broader spectrum of purposes overshadows the different usages of the device in pedagogical purposes in computing, synergising the many pedagogic methods rather than focusing on one method or the other would be more effective. Computing education research should focus on experimenting to find ideal blends of different pedagogic approaches that can effectively communicate the CT concepts, and equipping teachers with relevant content and pedagogy, so that they can embrace teaching CT with more self-efficacy.

Appendix A

Ethics Approval Documents

- Approval letter for the Pilot Study
- Amendment approval for the Experimental Study with pre-service teachers
- Amendment approval for the Experimental Study with associate teachers

UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

HUMAN ETHICS COMMITTEE

Secretary, Rebecca Robinson
Telephone: +64 03 369 4588, Extn 94588
Email: human-ethics@canterbury.ac.nz

Ref: 2019/78/ERHEC

5 December 2019

Bhagya Munasinghe
Computer Science and Software Engineering
UNIVERSITY OF CANTERBURY

Dear Bhagya

Thank you for providing the revised documents in support of your application to the Educational Research Human Ethics Committee. I am very pleased to inform you that your research proposal **"**How Does CS Unplugged (CSU) Can Help Teachers Build Confidence for Teaching Computational Thinking (CT) Topics?" has been granted ethical approval.

Please note that this approval is subject to the incorporation of the amendments you have provided in your emails of 13th, 21st, and 26th November and 4th December 2019.

Should circumstances relevant to this current application change you are required to reapply for ethical approval.

If you have any questions regarding this approval, please let me know.

We wish you well for your research.

Yours sincerely

pp. R. Robinson

Dr Patrick Shepherd
Chair
Educational Research Human Ethics Committee

Please note that ethical approval relates only to the ethical elements of the relationship between the researcher, research participants and other stakeholders. The granting of approval by the Educational Research Human Ethics Committee should not be interpreted as comment on the methodology, legality, value or any other matters relating to this research.

F E S

HUMAN ETHICS COMMITTEE

Secretary, Rebecca Robinson
Telephone: +64 03 369 4588, Extn 94588
Email: human-ethics@canterbury.ac.nz

Ref: 2019/78/ERHEC Amendment 2

28 May 2021

Bhagya Munasinghe
Computer Science and Software Engineering
UNIVERSITY OF CANTERBURY

Dear Bhagya

Thank you for your request for an amendment to your research proposal "How Does CS Unplugged (CSU) Can Help Teachers Build Confidence for Teaching Computational Thinking (CT) Topics?" as outlined in your email dated 25th May 2021. I am pleased to advise that this amendment has been considered and approved by the Human Research Ethics Committee.

Please note that should circumstances relevant to this current application change you are required to reapply for ethical approval.

If you have any questions regarding this approval, please advise.

We wish you well for your continuing research.

Yours sincerely

Dr Dean Sutherland
Chair
Human Research Ethics Committee

Please note that ethical approval relates only to the ethical elements of the relationship between the researcher, research participants and other stakeholders. The granting of approval by the Human Research Ethics Committee should not be interpreted as comment on the methodology, legality, value or any other matters relating to this research.

F E S

UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

HUMAN ETHICS COMMITTEE

Secretary, Rebecca Robinson
Telephone: +64 03 369 4588, Extn 94588
Email: human-ethics@canterbury.ac.nz

Ref: 2019/78/ERHEC Amendment 4

1 September 2021

Bhagya Munasinghe
Computer Science and Software Engineering
UNIVERSITY OF CANTERBURY

Dear Bhagya

Thank you for your request for an amendment to your research proposal "How Does CS Unplugged (CSU) Can Help Teachers Build Confidence for Teaching Computational Thinking (CT) Topics?" as outlined in your email dated 13th August 2021. I am pleased to advise that this amendment has been considered and approved by the Human Research Ethics Committee.

Please note that should circumstances relevant to this current application change you are required to reapply for ethical approval.

If you have any questions regarding this approval, please advise.

We wish you well for your continuing research.

Yours sincerely

Dr Dean Sutherland
Chair
University of Canterbury Human Research Ethics Committee

Please note that ethical approval relates only to the ethical elements of the relationship between the researcher, research participants and other stakeholders. The granting of approval by the Human Research Ethics Committee should not be interpreted as comment on the methodology, legality, value or any other matters relating to this research.

F E S

Appendix B

Survey Instruments and Sample Interview Questions

B.1 Survey Items - Teaching Self-Efficacy in Teaching Computational Thinking

Table B.1: Teaching Self-Efficacy Survey

1. I am able to teach all the relevant subject content of Computational Thinking and make it understandable for typical students
2. I am able to teach all the relevant subject content of Computational Thinking and make it understandable for all of my students
3. As time goes by, I will continue to become more and more capable of helping to address my students' Computational Thinking knowledge needs
4. Even if I get disrupted while teaching Computational Thinking during a class, I am confident that I can maintain my composure and continue to teach well
5. I am confident in my ability to be responsive to my students' Computational Thinking knowledge needs even if I am having a bad day
6. I can exert a positive influence on both the personal and academic development of my students with Computational Thinking
7. I can develop creative and innovative ways to cope with system constraints (such as limited resources) and continue to teach Computational Thinking well
8. I can motivate my students to participate in innovative projects using Computational Thinking knowledge
9. I can help students overcome their misconceptions around Computational Thinking
10. I can carry out innovative Computational Thinking projects even when I am opposed by sceptical colleagues
11. I can maintain a positive relationship with parents and community even when concerns related to Computational Thinking arises

B.2 Survey Items - Computer Programming Self-Efficacy

Table B.2: Computer Programming Self-Efficacy Survey

1. I can understand whether a problem is a programming problem or not
2. I can solve complex programming problems by breaking them into smaller sub problems
3. I can seek out knowledge that is required for solving a programming problem
4. I can select the most appropriate programming techniques for solving a programming problem
5. I can suggest different solutions to solve programming problems
6. I can determine solutions to a programming problem confidently
7. I can show the steps of a solution by drawing it or writing it on paper
8. I can explain the process I use to solve a programming problem with my colleagues
9. I can correct a program that has a bug
10.I can explain the different steps needed to develop a solution to a programming problem
11.Given some requirements, I can design a program that meets the requirements
12.I know how to use variables when programming
13.I can use loops instead of typing in instructions multiple times
14.I know what the operators +, -, *,/,<,>, = mean in a program
15.I know what the logical operators 'and', 'or', and 'not' mean in a program
16.I can run a program I have developed
17.I can write a program that has no bugs
18.I can write a program that meets the requirements I've been given
19.I can make changes to someone else's program
20.I can find bugs in a program that I've written
21.I can fix a bug I have found in a program
22.I can save a program I have developed
23.I can share a program with others
24.I can estimate how difficult my students will find a particular problem

B.3 Survey Items - Motivation Towards Teaching Computational Thinking Topics

Table B.3: Motivation Survey

1. I like to teach Digital Technologies because it is enjoyable/satisfying to teach the subject
2. I seek Professional Development in Digital Technologies because it is important for my teaching/content knowledge
3. I opted to teach the new Digital Technologies curriculum content because I would feel guilty not doing it
4. I intend to teach Digital Technologies because my school requires me to do it
5. I feel unmotivated to teach Digital Technologies because I don't always see the relevance of the subject in the curriculum.

B.4 Sample Interview Questions of the Pilot Study

Table B.4: Sample Interview Questions of the Pilot Study

1. In your understanding, what is Computational Thinking?
2. In your understanding, does Computational Thinking differ from computer programming? If yes, how?
3. In your opinion, what is the best way to incorporate CT concepts in the school curriculum [prompt if needed e.g. through computer programming (Scratch, Python, etc), problem solving tasks, through incorporating the concepts in other subjects and real life experience, or through combination of these?
4. In your experience/opinion, what are the best pedagogical strategies that can help students to develop Computational Thinking?
5. In your experience, what are the best computing tools to support the learning of Computational Thinking?
6. Can you please indicate the most difficult material presented in the DT curriculum for which you would like additional support to understand?

B.5 Sample Interview Questions of the Experimental Study II

Table B.5: Sample Interview Questions of the Experimental Study II

1. Have you used any techniques from the PD programme? a. Name activities from the workshops that you have used or intend to use in your classroom. b. Would you like to elaborate why you thought of using them? c. If unplugged, any feedback about it? d. Which ideas from the workshop were not useful in teaching in your class?
2. What is your opinion about using unplugged activities to introduce programming concepts to your class? a. Do you think it is useful? Care to elaborate how?
3. If you use unplugged activities, what concepts do you expect your students would understand better?
4. In what ways do they think the students *experience* learning through CSU activities?
5. In what ways do teachers think that students *perceive* learning programming concepts through unplugged?
6. Do you think unplugged activities help improving your own pedagogical, conceptual and technological knowledge?

Appendix C

Unplugged Activities Related to Programming Found in the Survey

The activities listed in this appendix cover the main published activities that were considered for teaching programming. It isn't an exhaustive list of general unplugged activities, and some of the activities listed here appear in a variety of forms in resources. New unplugged activities are appearing (e.g. code.org often has new activities), and some are lost as projects finish (e.g. CS Inside had some good activities, but the project ran from 2005-2009, and the activities no longer seem to be available). The list appears on the next page.

List of Unplugged Activities Related to Programming

(The websites include in the table were accessed during the during the period from January 2020 to October 2020)

	Activity	Author	Source
1	Treasure Hunter 1, 2, 3, 4 and 5	Arinchaya Threekunprapa and Pratchayapong Yasri	Arinchaya Threekunprapa and Pratchayapong Yasri. "Unplugged coding using flowblocks for promoting computational thinking and programming among secondary school students". English. In: International journal of instruction 13.3 (2020), pp. 207–222
2	Dance Move Algorithms	Barefoot Computing	https://www.barefootcomputing.org/resources/dance-move-algorithms
3	Algorithmic Doodle Art	cs4fn authors: Paul Curzon, Peter McOwan, Gabriella Kazai, Jonathan Black, Chrystie Myketiak, Jo Brodie, and Nicola Plant. Most of these activities are authored by Paul Curzon, and *The Invisible Palming Activity* is co-authored by Paul Curzon with Peter McOwan. Original (earlier) version of *Create a Face Activity* was developed with Quintin Cutts of University of Glasgow team, as part of the CS Inside project.	https://abitofcs4fn.org/art/algorithmic-doodle-art/
4	Origami		https://abitofcs4fn.org/art/computer-science-and-craft/origami-algorithms/
5	Lifecycle Puzzles		https://abitofcs4fn.org/sequencing-and-looping-puzzles/
6	Assignment Dry Run		https://teachinglondoncomputing.org/resources/inspiring-unplugged-classroom-activities/the-assignment-dry-run-activity/
7	Box Variables		https://teachinglondoncomputing.org/resources/inspiring-computing-stories/box-variables-understanding-variables-and-assignment/
8	Intelligent Paper		http://www.cs4fn.org/teachers/activities/intelligentpaper/intelligentpaper.pdf
9	Knitting		http://www.cs4fn.org/regularexpressions/knitters.php
10	Rock-Paper-Scissors		http://www.cs4fn.org/ai/robotalwayswins.php
11	Tour Guide		https://teachinglondoncomputing.org/the-tour-guide-activity/
12	The Knight's Tour		https://teachinglondoncomputing.org/resources/inspiring-unplugged-classroom-activities/the-knights-tour-activity/
13	The Intelligent Piece of Paper Activity		http://www.cs4fn.org/teachers/activities/intelligentpaper/intelligentpaper.pdf
14	The Invisible Palming Activity		https://teachinglondoncomputing.org/resources/inspiring-unplugged-classroom-activities/the-invisible-palming-activity/
15	Imp Computer		https://teachinglondoncomputing.files.wordpress.com/2014/03/activity-impcomputer.pdf
16	Create-a-face Activity		https://teachinglondoncomputing.org/resources/inspiring-unplugged-classroom-activities/the-create-a-face-activity/
17	The Swap Puzzle Activity		https://teachinglondoncomputing.org/resources/inspiring-unplugged-classroom-activities/the-swap-puzzle-activity/
18	Sorting Unplugged		https://teachinglondoncomputing.org/11-sorting-unplugged/
19	Unplugged Variables	Code-it.co.uk	http://code-it.co.uk/book/mq6/mq6
20	Everyday Variables		http://code-it.co.uk/wp-content/uploads/2018/06/Variableasbox.pdf
21	Everyday selection		http://code-it.co.uk/wp-content/uploads/2019/04/everydaycomputingconcepts.pdf
22	Ful or Fully		http://code-it.co.uk/wp-content/uploads/2015/08/fulorfully.pdf
23	Adjective to Adverbs		http://code-it.co.uk/wp-content/uploads/2015/08/adjectivestoadverbs.pdf
24	I before E except after C		http://code-it.co.uk/wp-content/uploads/2015/08/ibeforee.pdf
25	Sentence Checker Algorithm		http://code-it.co.uk/wp-content/uploads/2018/09/sentencecheckerv2.pdf
26	Everyday Sequences		http://code-it.co.uk/wp-content/uploads/2019/04/everydaycomputingconcepts.pdf
27	Human Crane		http://code-it.co.uk/wp-content/uploads/2015/05/humancraneplan.pdf
28	Everyday repetitions		http://code-it.co.uk/wp-content/uploads/2019/04/everydaycomputingconcepts.pdf
29	Letter Q		http://code-it.co.uk/wp-content/uploads/2015/08/letterq.pdf
30	Spelling Algorithms and Grammar Algorithms		http://code-it.co.uk/spellingalgorithms
31	Exchange Sort Investigation		http://code-it.co.uk/wp-content/uploads/2015/05/exchangesortplan.pdf
32	Singular to Plural		http://code-it.co.uk/wp-content/uploads/2015/08/singulartoplural.pdf
33	Human Crane		http://code-it.co.uk/wp-content/uploads/2015/05/humancraneplan.pdf
34	Jam Sandwich Algorithm		http://code-it.co.uk/unplugged/jamsandwich
35	Envelope variables	Code.org	https://studio.code.org/s/course4/lessons/4/levels/1
36	Functional Suncatchers		https://code.org/curriculum/course3/4/Teacher
37	Conditionals with cards		https://code.org/files/ConditionalsHoC.pdf

38	Move it, Move it		https://curriculum.code.org/csf-20/courseb/2/
39	Happy Maps		https://curriculum.code.org/csf-20/coursea/3/
40	My Robotic Friends Jr		https://curriculum.code.org/csf-20/coursec/3/
41	Graph Paper Programming		https://code.org/curriculum/course2/1/Teacher
42	Getting Loopy		https://studio.code.org/s/course1/lessons/12/levels/1
43	Happy Loops		https://studio.code.org/s/coursea-2021/lessons/7
44	My Loopy Robotic Friends Jr.		https://studio.code.org/s/coursec-2021/lessons/7
45	The Big Event Jr.		https://curriculum.code.org/csf-20/coursea/11/
46	The Big Event		https://curriculum.code.org/csf-20/coursec/14/
47	Everyday procedures		http://code-it.co.uk/wp-content/uploads/2019/04/everydaycomputingconcepts.pdf
48	Song Writing		https://code.org/curriculum/course3/9/Teacher
49	Song Writing with Parameters		https://code.org/curriculum/course4/13/Teacher
50	Rock-Paper-Scissors		https://code.org/files/PGUTSRockPaperScissors.pdf
51	Crowdsourcing		https://code.org/curriculum/course3/19/Teacher
52	Relay Programming		https://code.org/curriculum/course2/9/Teacher
53	Tangrams		https://studio.code.org/s/course4/lessons/1/levels/1
54	Plant a seed		https://code.org/curriculum/course1/6/Teacher.pdf
55	Paper Airplanes		https://code.org/curriculum/course2/2/Teacher
56	For Loop Fun		https://studio.code.org/s/coursef-2021/lessons/13
57	Sorting Network	CSUnplugged.org	https://classic.csunplugged.org/activities/sorting-networks/
58	Fitness unplugged		https://www.csunplugged.org/en/topics/kidbots/unit-plan/fitness-unplugged/
59	Lightest and Heaviest		https://classic.csunplugged.org/activities/sorting-algorithms/
60	KidBots		https://www.csunplugged.org/en/topics/kidbots/
61	Rock-Paper-Scissors		https://www.101computing.net/rock-paper-scissors/
62	Crabs & Turtles - The Race	Katerina Tsarava, Korbinian Moeller, and Manuel Ninaus	Katerina Tsarava, Korbinian Moeller, and Manuel Ninaus. “Training Computational Thinking through board games: The case of Crabs & Turtles”. *In: International Journal of Serious Games 5.2 (June 2018)*, pp. 25–44. doi: 10.17083/ijsg. v5i2.248. url: https://journal.seriousgamessociety.org/index.php/IJSG/article/view/248.
63	Patterns		
64	Crabs & Turtles - The Treasure Hunt		
65	Describing with variables	micro:bit	https://microbit.org/teach/lessons/getting-active-describing-variables/
66	Emergency Room example	R. A. Alamer, W. A. Al-Doweesh, H. S. Al-Khalifa, and M. S. Al-Razgan	R. A. Alamer et al. “Programming Unplugged: Bridging CS Unplugged ActivitiesGap for Learning Key Programming Concepts”. *In: 2015 Fifth International Conference on e-Learning (econf). 2015*, pp. 97–103. doi: 10.1109/ECONF.2015.27.
67	Minecraft Man		
68	Carpet	Yucnary-Daitiana Torres-Torres, Marcos Román-González, and Juan-Carlos Pérez-González.	Yucnary-Daitiana Torres-Torres, Marcos Román-González, and Juan-Carlos Pérez-González. 2019. Implementation of Unplugged Teaching Activities to Foster Computational Thinking Skills in Primary School from a Gender Perspective. *In Proceedings of the Seventh International Conference on Technological Ecosystems for Enhancing Multiculturality (TEEM'19)*. Association for Computing Machinery, New York, NY, USA, 209–215. DOI:https://doi-org.ezproxy.canterbury.ac.nz/10.1145/3362789.3362813

The book “Computing without Computers” by Paul Curzon of Queen Mary University of London available at https://teachinglondoncomputing.files.wordpress.com/2014/02/booklet-cwc-feb2014.pdf is a write up of introduction to programming, data structures and algorithms for complete novices that avoids programming notation, instead focusing on helping learners understand the concepts. It explains computing in terms of everyday things, strange links, and thought-provoking metaphors in the cs4fn ‘Computer Science for Fun’ style.

Appendix D

Detailed Linear Mixed Effect Models

- Estimates of Fixed Effects (Study I) – Teaching Self-Efficacy
- Estimates of Fixed Effects (Study I)– Computer Programming Self-Efficacy
- Estimates of Fixed Effects (Study I)– Motivation to Teach Computational Thinking
- Estimates of Fixed Effects (Study II) – Teaching Self-Efficacy
- Estimates of Fixed Effects (Study II)– Computer Programming Self-Efficacy
- Estimates of Fixed Effects (Study II)– Motivation to Teach Computational Thinking

Estimates of Fixed Effects (Study I) – Teaching Self-Efficacy							
Parameter	Estimate	Std. Error	df	t	Sig.	95% Confidence Interval	
						Lower Bound	Upper Bound
Intercept	2.041497	0.109	207.810	18.763	0.000	1.827000	2.255994
[Group=B]	-0.10549	0.215	244.208	-0.490	0.624	-0.529106	0.318132
[Group=PG]	0[b]	0.000					
[SurveyType=Post1]	0.386117	0.136	136.944	2.830	0.005	0.116335	0.655900
[SurveyType=Post2]	0.424501	0.149	146.561	2.848	0.005	0.129944	0.719058
[SurveyType=Pre]	0[b]	0.000					
[Treatment=1]	0.185223	0.154	216.503	1.204	0.230	-0.118110	0.488555
[Treatment=2]	0.340524	0.168	209.816	2.032	0.043	0.010211	0.670837
[Treatment=3]	0[b]	0.000					
[Group=B] * [SurveyType=Post1]	0.293619	0.245	150.352	1.197	0.233	-0.191048	0.778287
[Group=B] * [SurveyType=Post2]	0.585857	0.260	153.043	2.251	0.026	0.071585	1.100130
[Group=B] * [SurveyType=Pre]	0[b]	0.000					
[Group=PG] * [SurveyType=Post1]	0[b]	0.000					
[Group=PG] * [SurveyType=Post2]	0[b]	0.000					
[Group=PG] * [SurveyType=Pre]	0[b]	0.000					
[Group=B] * [Treatment=1]	0.007126	0.283	240.002	0.025	0.980	-0.549471	0.563724
[Group=B] * [Treatment=2]	-0.42018	0.280	232.895	-1.498	0.135	-0.972545	0.132312
[Group=B] * [Treatment=3]	0[b]	0.000					
[Group=PG] * [Treatment=1]	0[b]	0.000					
[Group=PG] * [Treatment=2]	0[b]	0.000					
[Group=PG] * [Treatment=3]	0[b]	0.000					
[SurveyType=Post1] * [Treatment=1]	-0.14055	0.190	145.672	-0.738	0.462	-0.516845	0.235814
[SurveyType=Post1] * [Treatment=2]	-0.43101	0.206	139.229	-2.087	0.039	-0.839229	-0.022781
[SurveyType=Post1] * [Treatment=3]	0[b]	0.000					
[SurveyType=Post2] * [Treatment=1]	0.138500	0.209	151.557	0.663	0.508	-0.273933	0.550932
[SurveyType=Post2] * [Treatment=2]	0.118817	0.217	140.688	0.549	0.584	-0.309387	0.547022
[SurveyType=Post2] * [Treatment=3]	0[b]	0.000					
[SurveyType=Pre] * [Treatment=1]	0[b]	0.000					
[SurveyType=Pre] * [Treatment=2]	0[b]	0.000					
[SurveyType=Pre] * [Treatment=3]	0[b]	0.000					
[Group=B] * [SurveyType=Post1] * [Treatment=1]	0.396605	0.259	202.961	1.534	0.127	-0.113331	0.906541
[Group=B] * [SurveyType=Post1] * [Treatment=2]	0.993964	0.234	157.065	4.239	0.000	0.530847	1.457081
[Group=B] * [SurveyType=Post1] * [Treatment=3]	0.293619	0.245	150.352	1.197	0.233	-0.191048	0.778287
[Group=B] * [SurveyType=Post2] * [Treatment=1]	0.359115	0.247	209.293	1.456	0.147	-0.127118	0.845348
[Group=B] * [SurveyType=Post2] * [Treatment=2]	0.622120	0.269	148.579	2.312	0.022	0.090439	1.153802
[Group=B] * [SurveyType=Post2] * [Treatment=3]	0.585857	0.260	153.043	2.251	0.026	0.071585	1.10013
[Group=B] * [SurveyType=Pre] * [Treatment=1]	0[b]	0.000					
[Group=B] * [SurveyType=Pre] * [Treatment=2]	0[b]	0.000					
[Group=B] * [SurveyType=Pre] * [Treatment=3]	0[b]	0.000					
[Group=PG] * [SurveyType=Post1] * [Treatment=1]	0[b]	0.000					
[Group=PG] * [SurveyType=Post1] * [Treatment=2]	0[b]	0.000					
[Group=PG] * [SurveyType=Post1] * [Treatment=3]	0[b]	0.000					
[Group=PG] * [SurveyType=Post2] * [Treatment=1]	0[b]	0.000					
[Group=PG] * [SurveyType=Post2] * [Treatment=2]	0[b]	0.000					
[Group=PG] * [SurveyType=Post2] * [Treatment=3]	0[b]	0.000					
[Group=PG] * [SurveyType=Pre] * [Treatment=1]	0[b]	0.000					
[Group=PG] * [SurveyType=Pre] * [Treatment=2]	0[b]	0.000					
[Group=PG] * [SurveyType=Pre] * [Treatment=3]	0[b]	0					

a. Dependent Variable: Mean_TE.

b. This parameter is set to zero because it is redundant.

Estimates of Fixed Effects (Study I) – Computer Programming Self-Efficacy							
Parameter	Estimate	Std. Error	df	t	Sig.	95% Confidence Interval	
						Lower Bound	Upper Bound
Intercept	1.885826	0.261	215.870	7.239	0.000	1.372366	2.399286
[Group=B]	0.535287	0.516	247.356	1.037	0.301	-0.481392	1.551966
[Group=PG]	0[b]	0.000					
[SurveyType=Post1]	0.965871	0.331	147.394	2.916	0.004	0.311265	1.620476
[SurveyType=Post2]	2.308865	0.362	157.179	6.385	0.000	1.594580	3.023151
[SurveyType=Pre]	0[b]	0.000					
[Treatment=1]	0.120020	0.369	223.624	0.326	0.745	-0.606499	0.846538
[Treatment=2]	0.159058	0.401	217.670	0.396	0.692	-0.631741	0.949857
[Treatment=3]	0[b]	0.000					
[Group=B] * [SurveyType=Post1]	-0.240158	0.595	160.727	-0.404	0.687	-1.415209	0.934892
[Group=B] * [SurveyType=Post2]	-0.176177	0.631	163.557	-0.279	0.781	-1.422790	1.070436
[Group=B] * [SurveyType=Pre]	0[b]	0.000					
[Group=PG] * [SurveyType=Post1]	0[b]	0.000					
[Group=PG] * [SurveyType=Post2]	0[b]	0.000					
[Group=PG] * [SurveyType=Pre]	0[b]	0.000					
[Group=B] * [Treatment=1]	-0.129842	0.678	243.726	-0.192	0.848	-1.465183	1.205500
[Group=B] * [Treatment=2]	0.151633	0.672	237.829	0.226	0.822	-1.172986	1.476253
[Group=B] * [Treatment=3]	0[b]	0.000					
[Group=PG] * [Treatment=1]	0[b]	0.000					
[Group=PG] * [Treatment=2]	0[b]	0.000					
[Group=PG] * [Treatment=3]	0[b]	0.000					
[SurveyType=Post1] * [Treatment=1]	-0.056627	0.462	156.086	-0.123	0.903	-0.969272	0.856018
[SurveyType=Post1] * [Treatment=2]	-0.216078	0.501	149.657	-0.431	0.667	-1.206463	0.774306
[SurveyType=Post1] * [Treatment=3]	0[b]	0.000					
[SurveyType=Post2] * [Treatment=1]	-0.821689	0.506	162.153	-1.623	0.107	-1.821520	0.178142
[SurveyType=Post2] * [Treatment=2]	-0.611829	0.526	151.213	-1.164	0.246	-1.650586	0.426927
[SurveyType=Post2] * [Treatment=3]	0[b]	0.000					
[SurveyType=Pre] * [Treatment=1]	0[b]	0.000					
[SurveyType=Pre] * [Treatment=2]	0[b]	0.000					
[SurveyType=Pre] * [Treatment=3]	0[b]	0.000					
[Group=B] * [SurveyType=Post1] * [Treatment=1]	0.8178531	0.625	210.578	1.308	0.192	-0.414919	2.0506255
[Group=B] * [SurveyType=Post1] * [Treatment=2]	0.4398216	0.569	167.322	0.774	0.440	-0.682545	1.5621884
[Group=B] * [SurveyType=Post1] * [Treatment=3]	-0.240158	0.595	160.727	-0.404	0.687	-1.415209	0.9348923
[Group=B] * [SurveyType=Post2] * [Treatment=1]	0.7165231	0.596	216.159	1.202	0.231	-0.458553	1.8915995
[Group=B] * [SurveyType=Post2] * [Treatment=2]	0.6542047	0.653	159.33	1.002	0.318	-0.634926	1.9433353
[Group=B] * [SurveyType=Post2] * [Treatment=3]	-0.176177	0.631	163.557	-0.279	0.781	-1.422790	1.0704365
[Group=B] * [SurveyType=Pre] * [Treatment=1]	0[b]	0.000					
[Group=B] * [SurveyType=Pre] * [Treatment=2]	0[b]	0.000					
[Group=B] * [SurveyType=Pre] * [Treatment=3]	0[b]	0.000					
[Group=PG] * [SurveyType=Post1] * [Treatment=1]	0[b]	0.000					
[Group=PG] * [SurveyType=Post1] * [Treatment=2]	0[b]	0.000					
[Group=PG] * [SurveyType=Post1] * [Treatment=3]	0[b]	0.000					
[Group=PG] * [SurveyType=Post2] * [Treatment=1]	0[b]	0.000					
[Group=PG] * [SurveyType=Post2] * [Treatment=2]	0[b]	0.000					
[Group=PG] * [SurveyType=Post2] * [Treatment=3]	0[b]	0.000					
[Group=PG] * [SurveyType=Pre] * [Treatment=1]	0[b]	0.000					
[Group=PG] * [SurveyType=Pre] * [Treatment=2]	0[b]	0.000					
[Group=PG] * [SurveyType=Pre] * [Treatment=3]	0[b]	0.000					

a. Dependent Variable: Mean_CPE.

b. This parameter is set to zero because it is redundant.

Estimates of Fixed Effects (Study I) – Motivation to Teach Computational Thinking							
Parameter	Estimate	Std. Error	df	t	Sig.	95% Confidence Interval	
						Lower Bound	Upper Bound
Intercept	4.231347	0.423	258.650	10.006	0.000	3.398619	5.064074
[Group=B]	-0.940574	0.486	257.147	-1.936	0.054	-1.897347	0.016199
[Group=PG]	0[b]	0.000					
[SurveyType=Post1]	-0.199010	0.516	190.551	-0.386	0.700	-1.216259	0.818239
[SurveyType=Post2]	0.208159	0.540	193.076	0.386	0.700	-0.855966	1.272284
[SurveyType=Pre]	0[b]	0.000					
[Treatment=1]	-0.103845	0.537	257.477	-0.194	0.847	-1.160624	0.952934
[Treatment=2]	-0.134629	0.508	256.928	-0.265	0.791	-1.134062	0.864804
[Treatment=3]	0[b]	0.000					
[Group=B] * [SurveyType=Post1]	0.747792	0.624	183.982	1.198	0.232	-0.483242	1.978825
[Group=B] * [SurveyType=Post2]	-0.772093	0.660	190.120	-1.170	0.243	-2.073582	0.529396
[Group=B] * [SurveyType=Pre]	0[b]	0.000					
[Group=PG] * [SurveyType=Post1]	0[b]	0.000					
[Group=PG] * [SurveyType=Post2]	0[b]	0.000					
[Group=PG] * [SurveyType=Pre]	0[b]	0.000					
[Group=B] * [Treatment=1]	0.545376	0.636	255.762	0.858	0.392	-0.706121	1.796874
[Group=B] * [Treatment=2]	0.386655	0.627	254.201	0.616	0.538	-0.848992	1.622302
[Group=B] * [Treatment=3]	0[b]	0.000					
[Group=PG] * [Treatment=1]	0[b]	0.000					
[Group=PG] * [Treatment=2]	0[b]	0.000					
[Group=PG] * [Treatment=3]	0[b]	0.000					
[SurveyType=Post1] * [Treatment=1]	0.786150	0.743	218.829	1.058	0.291	-0.678708	2.251007
[SurveyType=Post1] * [Treatment=2]	0.223897	0.678	196.389	0.330	0.742	-1.113121	1.560916
[SurveyType=Post1] * [Treatment=3]	0[b]	0.000					
[SurveyType=Post2] * [Treatment=1]	0.205041	0.718	219.214	0.285	0.776	-1.210551	1.620633
[SurveyType=Post2] * [Treatment=2]	0.402784	0.771	197.070	0.522	0.602	-1.117565	1.923134
[SurveyType=Post2] * [Treatment=3]	0[b]	0.000					
[SurveyType=Pre] * [Treatment=1]	0[b]	0.000					
[SurveyType=Pre] * [Treatment=2]	0[b]	0.000					
[SurveyType=Pre] * [Treatment=3]	0[b]	0.000					
[Group=B] * [SurveyType=Post1] * [Treatment=1]	-1.288427	0.888	207.041	-1.450	0.148	-3.039639	0.462784
[Group=B] * [SurveyType=Post1] * [Treatment=2]	-0.400842	0.861	187.034	-0.466	0.642	-2.099081	1.297396
[Group=B] * [SurveyType=Post1] * [Treatment=3]	0[b]	0.000					
[Group=B] * [SurveyType=Post2] * [Treatment=1]	1.008060	0.892	209.025	1.130	0.260	-0.751169	2.767288
[Group=B] * [SurveyType=Post2] * [Treatment=2]	1.178512	0.950	189.704	1.240	0.216	-0.695684	3.052708
[Group=B] * [SurveyType=Post2] * [Treatment=3]	0[b]	0.000					
[Group=B] * [SurveyType=Pre] * [Treatment=1]	0[b]	0.000					
[Group=B] * [SurveyType=Pre] * [Treatment=2]	0[b]	0.000					
[Group=B] * [SurveyType=Pre] * [Treatment=3]	0[b]	0.000					
[Group=PG] * [SurveyType=Post1] * [Treatment=1]	0[b]	0.000					
[Group=PG] * [SurveyType=Post1] * [Treatment=2]	0[b]	0.000					
[Group=PG] * [SurveyType=Post1] * [Treatment=3]	0[b]	0.000					
[Group=PG] * [SurveyType=Post2] * [Treatment=1]	0[b]	0.000					
[Group=PG] * [SurveyType=Post2] * [Treatment=2]	0[b]	0.000					
[Group=PG] * [SurveyType=Post2] * [Treatment=3]	0[b]	0.000					
[Group=PG] * [SurveyType=Pre] * [Treatment=1]	0[b]	0.000					
[Group=PG] * [SurveyType=Pre] * [Treatment=2]	0[b]	0.000					
[Group=PG] * [SurveyType=Pre] * [Treatment=3]	0[b]	0.000					

a. Dependent Variable: Mean_M.

b. This parameter is set to zero because it is redundant.

Estimates of Fixed Effects (Study II) – Teaching Self-Efficacy							
Parameter	Estimate	Std. Error	df	t	Sig.	95% Confidence Interval	
						Lower Bound	Upper Bound
Intercept	3.028024	0.139464	32.582	21.712	0.000	2.744144	3.311905
[Group=1]	0.042083	0.194560	29	0.216	0.830	-0.355836	0.440002
[Group=2]	0[b]	0					
[SurveyType=1]	-0.493548	0.067352	30	-7.328	0.000	-0.631099	-0.355998
[SurveyType=2]	0[b]	0					

a. Dependent Variable: TE.

b. This parameter is set to zero because it is redundant.

Estimates of Fixed Effects (Study II) – Computer Programming Self-Efficacy							
Parameter	Estimate	Std. Error	df	t	Sig.	95% Confidence Interval	
						Lower Bound	Upper Bound
Intercept	5.243952	0.320150	34.914	16.380	0.000	4.593955	5.893949
[Group=1]	-0.377500	0.438261	29	-0.861	0.396	-1.273845	0.518845
[Group=2]	0[b]	0					
[SurveyType=1]	-2.212903	0.195528	30	-11.318	0.000	-2.612224	-1.813583
[SurveyType=2]	0[b]	0					

a. Dependent Variable: CPE.

b. This parameter is set to zero because it is redundant.

Estimates of Fixed Effects (Study II) – Motivation to Teach Computational Thinking							
Parameter	Estimate	Std. Error	df	t	Sig.	95% Confidence Interval	
						Lower Bound	Upper Bound
Intercept	4.163609	0.568061	806.915	7.330	0.000	3.048557	5.278661
[Group=1]	0.875208	0.347887	29	2.516	0.018	0.163699	1.586718
[Group=2]	0[b]	0					
[SurveyType=1]	-0.970968	0.268296	30.000	-3.619	0.001	-1.518901	-0.423035
[SurveyType=2]	0[b]	0					

a. Dependent Variable: M.

b. This parameter is set to zero because it is redundant.

References

[1] Ashish Aggarwal, Christina Gardner-McCune, and David S. Touretzky. "Evaluating the Effect of Using Physical Manipulatives to Foster Computational Thinking in Elementary School". In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '17. Seattle, Washington, USA: Association for Computing Machinery, 2017, pp. 9–14. ISBN: 9781450346986. DOI: 10.1145/3017680.3017791. URL: https://doi.org/10.1145/3017680.3017791.

[2] Alfred V. Aho. "Computation and Computational Thinking". In: *The Computer Journal* 55.7 (July 2012), pp. 832–835. ISSN: 0010-4620. DOI: 10.1093/comjnl/bxs074. eprint: https://academic.oup.com/comjnl/article-pdf/55/7/832/971573/bxs074.pdf. URL: https://doi.org/10.1093/comjnl/bxs074.

[3] R. A. Alamer et al. "Programming Unplugged: Bridging CS Unplugged Activities Gap for Learning Key Programming Concepts". In: *2015 Fifth International Conference on e-Learning (econf)*. 2015, pp. 97–103. DOI: 10.1109/ECONF.2015.27.

[4] Reem A. Alamer et al. "Programming Unplugged: Bridging CS Unplugged Activities Gap for Learning Key Programming Concepts". In: *2015 Fifth International Conference on e-Learning (econf)*. 2015, pp. 97–103. DOI: 10.1109/ECONF.2015.27.

[5] Peter R. Albion. "Self-Efficacy Beliefs as an Indicator of Teachers' Preparedness for Teaching with Technology". In: *Proceedings of Society for Information Technology & Teacher Education International Conference 1999*. Ed. by J.D. Price et al. Waynesville, NC USA: Association for the Advancement of Computing in Education (AACE), 1999, pp. 1602–1608. URL: https://www.learntechlib.org/p/8156.

[6] *Algorithm*. URL: https://en.wikipedia.org/wiki/Algorithm (visited on 03/12/2020).

[7] Gary L. Anderson and Bonnie Page. "Narrative Knowledge and Educational Administration: The Stories That Guide Our Practice". In: *The knowledge base in educational administration: multiple perspectives*. Ed. by Robert Donmoyer, Michael Imber, and James J. Scheurich. State University of New York Press, 1995, pp. 124–131.

[8] George Aranda and Joseph P. Ferguson. "Unplugged Programming: The future of teaching computational thinking?" English;Czech; in: *Pedagogika* 68.3 (2018).

[9] Aristotle and Sir Barker Ernest. *The politics of Aristotle*. English. [1st American]. New York: OxfordPress, 1958.

[10] Michal Armoni. "COMPUTING IN SCHOOLS

Computer Science, Computational Thinking, Programming, Coding: The Anomalies of Transitivity in K-12 Computer Science Education". In: *ACM Inroads* 7.4 (Nov. 2016), pp. 24–27. ISSN: 2153-2184. DOI: 10.1145/3011071. URL: https://doi.org/10.1145/3011071.

[11] Michal Armoni and Judith Gal-Ezer. "Early Computing Education: Why? What? When? Who?" In: *ACM Inroads* 5.4 (Dec. 2014), pp. 54–59. ISSN: 2153-2184. DOI: 10.1145/2684721.2684734. URL: https://doi.org/10.1145/2684721.2684734.

[12] Computer Science Teachers Association. *Standards for Computer Science Teachers*. 2020. URL: https://csteachers.org/teacherstandards (visited on 01/09/2022).

[13] *Australian CURRICULUM*. URL: https://www.australiancurriculum.edu.au/f-10-curriculum/technologies/structure/ (visited on 01/09/2022).

[14] Albert Bandura. "Self-efficacy: Toward a unifying theory of behavioral change". In: *Advances in Behaviour Research and Therapy* 1.4 (1978). Perceived Self-Efficacy: Analyses of Bandura's Theory of Behavioural Change, pp. 139–161. ISSN: 0146-6402. DOI: https://doi.org/10.1016/0146-6402(78)90002-4. URL: https://www.sciencedirect.com/science/article/pii/0146640278900024.

[15] David Barr, John Harrison, and Leslie Conery. "Computational thinking: a digital age skill for everyone: the National Science Foundation has assembled a group of thought leaders to bring the concepts of computational thinking to the K-12 classroom". English. In: *Learning and leading with technology* 38.6 (2011), p. 20.

[16] Yavar Bathaee. "The Artificial Intelligence Black Box and the Failure of Intent and Causation". In: *Harvard Journal of Law & Technology* 31 (2018), pp. 890–939.

[17] Fatma Batur and Jan Strobl. "Discipline-Specific Language Learning in a Mainstream Computer Science Classroom: Using a Genre-Based Approach". In: ACM, 2019, pp. 1–4. ISBN: 9781450377041;1450377041;

[18] Tim Bell and Michael Lodi. "Constructing Computational Thinking Without Using Computers". In: *Constructivist foundations*. Special Issue "Constructionism and Computational Thinking" 14.3 (July 2019), pp. 342–351. URL: https://hal.inria.fr/hal-02378761.

[19] Tim Bell and Josie Roberts. "Computational thinking is more about humans than computers". English. In: *Set: research information for teachers (Wellington)* 2016.1 (2016), pp. 3–7. DOI: 10.18296/set.0030. URL: http://dx.doi.org/10.18296/set.0030.

[20] Tim Bell, Frances Rosamond, and Nancy Casey. "Computer Science Unplugged and Related Projects in Math and Computer Science Popularization". In: *The Multivariate Algorithmic Revolution and Beyond: Essays Dedicated to Michael R. Fellows on the Occasion of His 60th Birthday*. Ed. by Hans L. Bodlaender et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 398–456. ISBN: 978-3-642-30891-8. DOI: 10.1007/978-3-642-30891-8_18. URL: https://doi.org/10.1007/978-3-642-30891-8_18.

[21] Tim Bell and Jan Vahrenhold. "CS unplugged — How is it used, and does it work?" English. In: *Lecture notes in computer science* 11011 (2018), pp. 497–521.

[22] Tim Bell et al. *Computer Science Unplugged: An enrichment and extension programme for primary-aged children*. English. University of Canterbury. Computer Science and Software Engineering, 2005. URL: https://ir.canterbury.ac.nz/handle/10092/247.

[23] Michael Berry and Michael Kölling. "The design and implementation of a notional machine for teaching introductory programming". In: *Proceedings of the 8th Workshop in Primary and Secondary Computing Education*. 2013, pp. 25–28.

[24] Sylvia Berryman. "Ancient Automata and Mechanical Explanation". In: *Phronesis* 48.4 (2003), pp. 344–369. DOI: https://doi.org/10.1163/156852803772456083. URL: https://brill.com/view/journals/phro/48/4/article-p344_3.xml.

[25] Marina Umaschi Bers. *Designing Digital Experiences for Positive Youth Development: From Playpen to Playground*. English. Oxford University Press, 2012. ISBN: 9780199757022. DOI: 10.1093/acprof:oso/9780199757022.001.0001.

[26] Corrado Böhm and Giuseppe Jacopini. "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules". In: *Commun. ACM* 9.5 (1966), pp. 366–371. ISSN: 0001-0782. DOI: 10.1145/355592.365646. URL: https://doi.org/10.1145/355592.365646.

[27] Corrado Böhm and Giuseppe Jacopini. "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules". In: *Commun. ACM* 9.5 (May 1966), pp. 366–371. ISSN: 0001-0782. DOI: 10.1145/355592.365646. URL: https://doi.org/10.1145/355592.365646.

[28] Benedict Du Boulay. "Some Difficulties of Learning to Program". In: *Journal of Educational Computing Research* 2.1 (1986), pp. 57–73. DOI: 10.2190/3LFX-9RRF-67T8-UVK9.

[29] Matt Bower and Katrina Falkner. "Computational Thinking, the Notional Machine, Pre-service Teachers, and Research Opportunities." In: *17th Australasian Computing Education Conference, ACE 2015, Sydney, Australia*. Vol. 160. CRPIT. Australian Computer Society, 2015, pp. 37–46.

[30] Virginia Braun and Victori Clarke. *Thematic analysis: a practical guide*. English. London: SAGE Publications Ltd, 2022. ISBN: 9781473953239; 9781473953246.

[31] Virginia Braun and Victoria Clarke. "Thematic analysis". In: *APA handbook of research methods in psychology, Vol 2: Research designs: Quantitative, qualitative, neuropsychological, and biological*. American Psychological Association, 2012, pp. 57–71. ISBN: 9781433810053. DOI: 10.1037/13620-004.

[32] A. Butterfield and Gerard E. Ngondi. *A dictionary of computer science*. Seventh. New York, NY, United States of America;Oxford, United Kingdom; Oxford University Press, 2016. ISBN: 0199688974; 9780199688975.

[33] Elisa Nadire Caeli and Aman Yadav. "Unplugged Approaches to Computational Thinking: a Historical Perspective". In: *TechTrends* 64.1 (2020), pp. 29–36. DOI: 10.1007/s11528-019-00410-5. URL: https://doi.org/10.1007/s11528-019-00410-5.

[34] Helen Caldwell and Neil Smith, eds. *Teaching Computing Unplugged in Primary Schools: Exploring Primary Computing through Practical Activities Away from the Computer*. 55 City Road, London: SAGE Publications, Inc., 2017. DOI: 10.4135/9781473984332. URL: https://sk.sagepub.com/books/teaching-computing-unplugged-in-primary-schools.

[35] Pacific Policy Research Center. *21st Century Skills for Students and Teachers*. 2010.

[36] Browne Charles, Culligan Brent, and Phillips Joseph. *New Academic Word List — New General Service List*. URL: http://www.newgeneralservicelist.org/nawl-new-academic-word-list.

[37] Browne Charles, Culligan Brent, and Phillips Joseph. *New General Service List*. URL: http://www.newgeneralservicelist.org/.

[38] Per Christensson. *Glossary of Computer and Internet Terms*. URL: https://pc.net/glossary/.

[39] Vincent A. Cicirello. "A CS Unplugged Activity for the Online Classroom". In: *J. Comput. Sci. Coll.* 28.6 (June 2013), pp. 162–168. ISSN: 1937-4771.

[40] Vincent A. Cicirello. "A CS unplugged activity for the online classroom". In: *Journal of Computing Sciences in Colleges* 28 (2013), pp. 162–168.

[41] Beverly Clarke. *Computer science teacher: insight into the computing classroom*. English. Swindon, UK: BCS, The Chartered Institute for IT, 2017. ISBN: 9781780173948; 1780173946.

[42] T. R. Colburn and G. M. Shute. "Metaphor in computer science". In: *Journal of applied logic* 6.4 (2008), pp. 526–533.

[43] David Cole. “The Chinese Room Argument”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Winter 2020. Metaphysics Research Lab, Stanford University, 2020.

[44] Kari Gwen Coleman. “Android arete: Toward a virtue ethic for computational agents”. In: *Ethics and Information Technology* 3.4 (2001), pp. 247–265. DOI: 10.1023/A:1013805017161.

[45] Andrew M. Colman. *Zone of proximal development*. 2015. DOI: 10.1093/acref/9780199657681.013.9017. URL: https://www.oxfordreference.com/view/10.1093/acref/9780199657681.001.0001/acref-9780199657681-e-9017.

[46] Deborah R. Compeau and Christopher A. Higgins. “Computer Self-Efficacy: Development of a Measure and Initial Test”. In: *MIS Q.* 19.2 (June 1995), pp. 189–211. ISSN: 0276-7783. DOI: 10.2307/249688. URL: https://doi.org/10.2307/249688.

[47] *Computer Jargon and terms explained*. URL: http://www.pcrescue.com.au/jargon.htm.

[48] Jack B. Copeland. “The Church-Turing Thesis”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Summer 2020. Stanford, USA: Metaphysics Research Lab, Stanford University, 2020.

[49] Isabella Corradini, Michael Lodi, and Enrico Nardelli. “Computational Thinking in Italian Schools: Quantitative Data and Teachers’ Sentiment Analysis after Two Years of "Programma Il Futuro"”. In: *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’17. Bologna, Italy: Association for Computing Machinery, 2017, pp. 224–229. ISBN: 9781450347044. DOI: 10.1145/3059009.3059040. URL: https://doi.org/10.1145/3059009.3059040.

[50] Isabella Corradini, Michael Lodi, and Enrico Nardelli. “Conceptions and Misconceptions about Computational Thinking among Italian Primary School Teachers”. In: *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ICER ’17. Tacoma, Washington, USA: Association for Computing Machinery, 2017, pp. 136–144. ISBN: 9781450349680. DOI: 10.1145/3105726.3106194. URL: https://doi-org.ezproxy.canterbury.ac.nz/10.1145/3105726.3106194.

[51] Isabella Corradini, Michael Lodi, and Enrico Nardelli. “Learning difficulties of ‘object-oriented programming paradigm using Java’: students’ perspective”. In: *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’17. Bologna, Italy: Association for Computing Machinery, 2017, pp. 224–229. ISBN: 9781450347044. DOI: 10.1145/3059009.3059040. URL: https://doi.org/10.1145/3059009.3059040.

[52] John W. Creswell. "Chapter 18 - Mixed-Method Research: Introduction and Application". In: *Handbook of Educational Policy*. Ed. by Gregory J. Cizek. Educational Psychology. San Diego: Academic Press, 1999, pp. 455–472. DOI: https://doi.org/10.1016/B978-012174698-8/50045-X. URL: https://www.sciencedirect.com/science/article/pii/B978012174698850045X.

[53] Professional Development for CS (PD4CS) Principles Teaching Team. *The New Zealand Curriculum Online*. URL: http://www.pd4cs.org/loops-student-misconceptions-and-challenges/ (visited on 07/30/2021).

[54] *CS Unplugged*. URL: https://csunplugged.org (visited on 11/09/2020).

[55] Paul Curzon. "Cs4fn and Computational Thinking Unplugged". In: *Proceedings of the 8th Workshop in Primary and Secondary Computing Education*. WiPSE '13. Aarhus, Denmark: Association for Computing Machinery, 2013, pp. 47–50. ISBN: 9781450324557. DOI: 10.1145/2532748.2611263. URL: https://doi.org/10.1145/2532748.2611263.

[56] Paul Curzon et al. "Computational Thinking". In: *The Cambridge Handbook of Computing Education Research*. Ed. by Sally A. Fincher and Anthony V.Editors Robins. Cambridge Handbooks in Psychology. Cambridge University Press, 2019, pp. 513–546. DOI: 10.1017/9781108654555.018.

[57] Paul Curzon et al. "Introducing Teachers to Computational Thinking Using Unplugged Storytelling". In: WiPSCE '14. Berlin, Germany: Association for Computing Machinery, 2014, pp. 89–92. ISBN: 9781450332507. DOI: 10.1145/2670757.2670767. URL: https://doi.org/10.1145/2670757.2670767.

[58] Paul Curzon et al. "Introducing Teachers to Computational Thinking Using Unplugged Storytelling". In: WiPSCE '14. Berlin, Germany: Association for Computing Machinery, 2014, pp. 89–92. ISBN: 9781450332507. DOI: 10.1145/2670757.2670767. URL: https://doi.org/10.1145/2670757.2670767.

[59] Paul Curzon et al. "Using Semantic Waves to Analyse the Effectiveness of Unplugged Computing Activities". In: WiPSCE '20. Virtual Event, Germany: Association for Computing Machinery, 2020. ISBN: 9781450387590. DOI: 10.1145/3421590.3421606. URL: https://doi.org/10.1145/3421590.3421606.

[60] Quintin I Cutts et al. "Enthusing and informing potential computer science students and their teachers". In: *ACM SIGCSE Bulletin* 39.3 (2007), pp. 196–200.

[61] Edward L. Deci and Richard M. Ryan. *Intrinsic motivation and self-determination in human behavior*. English. New York: Plenum, 1985.

[62] Peter J. Denning. "Computing is a Natural Science". In: *Commun. ACM* 50.7 (July 2007), pp. 13–18. ISSN: 0001-0782. DOI: 10.1145/1272516.1272529. URL: https://doi.org/10.1145/1272516.1272529.

[63] Peter J. Denning. “Remaining Trouble Spots with Computational Thinking”. In: *Commun. ACM* 60.6 (May 2017), pp. 33–39. ISSN: 0001-0782. DOI: 10.1145/2998438. URL: https://doi.org/10.1145/2998438.

[64] Peter J. Denning and Matti Tedre. *Computational Thinking*. English. Cambridge, USA: MIT Press, 2019. ISBN: 0262353415;9780262353410;

[65] Peter J. Denning and Matti Tedre. “Computational Thinking: A Disciplinary Perspective”. English. In: *Informatics in education* 20.3 (2021), pp. 361–390.

[66] Amanda Catherine Dickes et al. “Development of Mechanistic Reasoning and Multilevel Explanations of Ecology in Third Grade Using Agent-Based Models”. In: *Science Education* 100.4 (2016), pp. 734–776. DOI: https://doi.org/10.1002/sce.21217. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/sce.21217. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/sce.21217.

[67] Paul E. Dickson, Neil C. C. Brown, and Brett A. Becker. “Engage Against the Machine: Rise of the Notional Machines as Effective Pedagogical Devices”. In: *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’20. Trondheim, Norway: Association for Computing Machinery, 2020, pp. 159–165. ISBN: 9781450368742. DOI: 10.1145/3341525.3387404. URL: https://doi.org/10.1145/3341525.3387404.

[68] *Dictionary by Merriam-Webster: America’s most-trusted online dictionary*. URL: https://www.merriam-webster.com/.

[69] Ira Diethelm and Juliana Goschler. “On Human Language and Terminology Used for Teaching and Learning CS/Informatics”. In: WiPSCE ’14. Berlin, Germany: Association for Computing Machinery, 2014, pp. 122–123. ISBN: 9781450332507. DOI: 10.1145/2670757.2670765. URL: https://doi.org/10.1145/2670757.2670765.

[70] Ira Diethelm and Juliana Goschler. “Questions on Spoken Language and Terminology for Teaching Computer Science”. In: ITiCSE ’15. Vilnius, Lithuania: Association for Computing Machinery, 2015, pp. 21–26. ISBN: 9781450334402. DOI: 10.1145/2729094.2742600. URL: https://doi.org/10.1145/2729094.2742600.

[71] J. Diethelm I.and Goschler and T. Lampe. “Language and Computing”. In: *Computer Science Education: Perspectives on Teaching and Learning in School*. Ed. by Sue Sentance, Erik Barendsen, and Carsten Schulte. Bloomsbury Publishing Plc, 2018, pp. 207–219. ISBN: 9781350057111;135005710X;1350057118;9781350057104;

[72] Edsger W. Dijkstra. “On the Necessity of Correctness Proofs”. URL: https://www.cs.utexas.edu/users/EWD/ewd03xx/EWD360.pdf.

[73] Slobodan Dmitrović. *Modern C for Absolute Beginners: A Friendly Introduction to the C Programming Language*. English. Berkeley, CA: Apress. ISBN: 9781484266427.

[74] Yuemeng Du, Andrew Luxton-Reilly, and Paul Denny. "A Review of Research on Parsons Problems". In: ACE'20. Melbourne, VIC, Australia: Association for Computing Machinery, 2020, pp. 195–202. ISBN: 9781450376860. DOI: 10.1145/3373165.3373187. URL: https://doi.org/10.1145/3373165.3373187.

[75] Benedict Du Boulay, Tim O'Shea, and John Monk. "The black box inside the glass box: presenting computing concepts to novices". In: *International Journal of Human-Computer Studies* 51.2 (1999), pp. 265–277. ISSN: 1071-5819. DOI: https://doi.org/10.1006/ijhc.1981.0309.

[76] Caitlin Duncan. "Computer science and computational thinking in primary schools". English. PhD thesis. 2019.

[77] Caitlin Duncan, Tim Bell, and Steve Tanimoto. "Should Your 8-Year-Old Learn Coding?" In: *Proceedings of the 9th Workshop in Primary and Secondary Computing Education*. WiPSCE '14. Berlin, Germany: Association for Computing Machinery, 2014, pp. 60–69. ISBN: 9781450332507. DOI: 10.1145/2670757.2670774. URL: https://doi.org/10.1145/2670757.2670774.

[78] Rodrigo Duran, Juha Sorva, and Otto Seppälä. "Rules of Program Behavior". In: *ACM Trans. Comput. Educ.* 21.4 (Nov. 2021). DOI: 10.1145/3469128. URL: https://doi.org/10.1145/3469128.

[79] Hilary Dwyer et al. "Identifying Elementary Students' Pre-Instructional Ability to Develop Algorithms and Step-by-Step Instructions". In: SIGCSE '14. Atlanta, Georgia, USA: Association for Computing Machinery, 2014, pp. 511–516. ISBN: 9781450326056. DOI: 10.1145/2538862.2538905. URL: https://doi.org/10.1145/2538862.2538905.

[80] New Zealand Ministry of Education. *Glossary / Welcome to Technology Online - Technology Online*. URL: https://technology.tki.org.nz/Glossary.

[81] Simple English Wikipedia the free encyclopedia. *Computer jargon*. URL: https://simple.wikipedia.org/wiki/Computer_jargon.

[82] Simple English Wikipedia the free encyclopedia. *Programming Language*. URL: https://en.wikipedia.org/wiki/Programming_language.

[83] Barbara Ericson, James Foley, and Jochen Rick. "Evaluating the Efficiency and Effectiveness of Adaptive Parsons Problems". English. In: ACM, 2018, pp. 60–68. ISBN: 9781450356282;1450356281.

[84] Peggy A. Ertmer and Anne T. Ottenbreit-Leftwich. "Teacher Technology Change: How Knowledge, Confidence, Beliefs, and Culture Intersect". In: *Journal of research on technology in education* 42.3 (2010), pp. 255–284.

[85] Barohny Eun. "The zone of proximal development as an overarching concept: A framework for synthesizing Vygotsky's theories". In: *Educational Philosophy and Theory* 51.1 (2019), pp. 18–30. DOI: 10.1080/00131857.2017.1421941.

[86] Hylke H. Faber et al. "Teaching Computational Thinking to Primary School Students via Unplugged Programming Lessons". In: *Journal of the European Teacher Education Network* 12 (2017), pp. 13–24.

[87] Yvon Feaster et al. "Teaching CS unplugged in the high school (with limited success)". In: *Proceedings of the 16th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2011, Darmstadt, Germany, June 27-29, 2011*. Ed. by Guido Rößling, Thomas L Naps, and Christian Spannagel. ACM, 2011, pp. 248–252. ISBN: 978-1-4503-0697-3.

[88] Claude Fernet et al. "The Work Tasks Motivation Scale for Teachers (WTMST)". English. In: *Journal of career assessment* 16.2 (2008), pp. 256–279.

[89] Sally Fincher et al. "Notional Machines in Computing Education: The Education of Attention". In: *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. ITiCSE-WGR '20. Trondheim, Norway: Association for Computing Machinery, 2020, pp. 21–50. ISBN: 9781450382939. DOI: 10.1145/3437800.3439202. URL: https://doi.org/10.1145/3437800.3439202.

[90] Sally A. Fincher, Yifat Ben-David Kolikant, and Katrina Falkner. "Teacher Learning and Professional Development". In: *The Cambridge Handbook of Computing Education Research*. Ed. by Sally A. Fincher and Anthony V.Editors Robins. Cambridge Handbooks in Psychology. Cambridge University Press, 2019, pp. 727–748. DOI: 10.1017/9781108654555.026.

[91] Sally A. Fincher, Yifat Ben-David Kolikant, and Katrina Falkner. "Teacher Learning and Professional Development". In: *The Cambridge Handbook of Computing Education Research*. Ed. by Sally A. Fincher and Anthony V.Editors Robins. Cambridge Handbooks in Psychology. Cambridge University Press, 2019, pp. 727–748. DOI: 10.1017/9781108654555.026.

[92] Vitaly Ford et al. "Capture the Flag Unplugged: An Offline Cyber Competition". In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '17. Seattle, Washington, USA: Association for Computing Machinery, 2017, pp. 225–230. ISBN: 9781450346986. DOI: 10.1145/3017680.3017783. URL: https://doi.org/10.1145/3017680.3017783.

[93] Michal Forišek and Monika Steinová. *Explaining algorithms using metaphors*. 2013th ed. London;New York: Springer, 2013. ISBN: 2191-5768.

[94] Michal Forišek and Monika Steinová. "Metaphors and analogies for teaching algorithms". In: ACM, 2012, pp. 15–20. ISBN: 1450310982;9781450310987;

[95] Stan Franklin and Art Graesser. "Is It an agent, or just a program?: A taxonomy for autonomous agents". In: *Intelligent Agents III Agent Theories, Architectures, and Languages*. Ed. by Jörg P. Müller, Michael J. Wooldridge, and Nicholas R. Jennings. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 21–35. ISBN: 978-3-540-68057-4.

[96] *Future Learn: Unplugged activities*. URL: https://www.futurelearn.com/info/courses/creating-an-inclusive-classroom-approaches-to-supporting-learners-with-send-in-computing/0/steps/68311 (visited on 01/09/2021).

[97] Andrzej T. Gałecki and Tomasz Burzykowski. *Linear mixed-effects models using R: a step-by-step approach*. English. 1. Aufl.;2013; New York, NY: Springer, 2013;2015; ISBN: 1431-875X.

[98] Michail N. Giannakos et al. "Examining and mapping CS teachers' technological, pedagogical and content knowledge (TPACK) in K-12 schools". In: *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*. 2014, pp. 1–7. DOI: 10.1109/FIE.2014.7044406.

[99] Colin Gibbs. "Explaining effective teaching: self-efficacy and thought control of action". In: *The Journal of Educational Enquiry* 4.2 (2003).

[100] Ronald N. Giere. *Explaining science: a cognitive approach*. English. Chicago: University of Chicago Press, 1988.

[101] Avelino J. Gonzalez. *Computer Programming in C for Beginners*. English. Cham: Springer International Publishing. ISBN: 9783030507497;3030507491;

[102] Judith Good and Kate Howland. "Programming language, natural language? Supporting the diverse computational activities of novice programmers". In: *Journal of visual languages and computing* 39 (2017), pp. 78–92.

[103] Anne Gravells and Susan Simpson. "Delivering Education and Training". In: *The Certificate in Education and Training*. Sage Publishing Company, 2014.

[104] Martin Greenberger, ed. *Management and the Computer of the Future*. English. London and New York: The M.I.T. Press and John Wiley & Sons, inc., 1962.

[105] Shuchi Grover. *The 5th 'C' of 21st Century Skills? Try Computational Thinking (Not Coding)*. URL: https://www.edsurge.com/news/2018-02-25-the-5th-c-of-21st-century-skills-try-computational-thinking-not-coding (visited on 02/20/2020).

[106] Shuchi Grover and Satabdi Basu. "Measuring Student Learning in Introductory Block-Based Programming: Examining Misconceptions of Loops, Variables, and Boolean Logic". In: SIGCSE '17. Seattle, Washington, USA: Association for Computing Machinery, 2017, pp. 267–272. ISBN: 9781450346986. DOI: 10.1145/3017680.3017723. URL: https://doi-org.ezproxy.canterbury.ac.nz/10.1145/3017680.3017723.

[107] Shuchi Grover and Roy Pea. "Computational Thinking in K–12: A Review of the State of the Field". In: *Educational Researcher* 42.1 (2013), pp. 38–43. DOI: 10.3102/0013189X12463051. eprint: https://doi.org/10.3102/0013189X12463051. URL: https://doi.org/10.3102/0013189X12463051.

[108] Mark Guzdial. "Education

Paving the Way for Computational Thinking". In: *Commun. ACM* 51.8 (Aug. 2008), pp. 25–27. ISSN: 0001-0782. DOI: 10.1145/1378704.1378713. URL: https://doi.org/10.1145/1378704.1378713.

[109] Mark Guzdial. "Software-Realized Scaffolding to Facilitate Programming for Science Learning". In: *Interactive Learning Environments* 4.1 (1994), pp. 001–044. DOI: 10.1080/1049482940040101. eprint: https://doi.org/10.1080/1049482940040101.

[110] Mark Guzdial et al. "Notional Machines and Programming Language Semantics in Education (Dagstuhl Seminar 19281)". In: *Dagstuhl Reports* 9.7 (2019), pp. 1–23. DOI: 10.4230/Daep.9.7.1. URL: https://doi.org/10.4230/DagRep.9.7.1.

[111] D. Harel et al. "A universal flowcharter". In: *2nd Computers in aerospace conference*. AAIA A79-54378/24-59 (1979), pp. 218–224.

[112] Orit Hazzan et al. *Guide to Teaching Computer Science: An Activity-Based Approach*. English. 3rd 2020. Cham: Springer International Publishing, 2020. ISBN: 9783030393601;9783030393595.

[113] Mariane Hedegaard. "The zone of proximal development as basis for instruction". In: *An Introduction to Vygotsky*. Ed. by Harry Daniels. Routledge, Jan. 1996. DOI: https://doi.org/10.4324/9780203434185.

[114] David Hemmendinger. "A Plea for Modesty". In: *ACM Inroads* 1.2 (June 2010), pp. 4–7. ISSN: 2153-2184. DOI: 10.1145/1805724.1805725. URL: https://doi.org/10.1145/1805724.1805725.

[115] Felienne Hermans and Efthimia Aivaloglou. "To Scratch or Not to Scratch? A Controlled Experiment Comparing Plugged First and Unplugged First Programming Lessons". In: *Proceedings of the 12th Workshop on Primary and Secondary Computing Education*. WiPSCE '17. Nijmegen, Netherlands: Association for Computing Machinery, 2017, pp. 49–56. ISBN: 9781450354288. DOI: 10.1145/3137065.3137072. URL: https://doi.org/10.1145/3137065.3137072.

[116] Anna Maria Hipkiss and Pernilla Andersson Varga. "Spotlighting pedagogic metalanguage in Reading to Learn – How teachers build legitimate knowledge during tutorial sessions". In: *Linguistics and Education* 47 (2018), pp. 93–104. ISSN: 0898-5898. DOI: https://doi.org/10.1016/j.linged.2018.08.002. URL: https://www.sciencedirect.com/science/article/pii/S0898589817303996.

[117] C. Hollings, U. Martin, and A.C. Rice. *Ada Lovelace: The Making of a Computer Scientist*. MIT paperback series. Bodleian Library, 2018. ISBN: 9781851244881.

[118] Faber H. Hylke et al. "Teaching Computational Thinking to Primary School Students via Unplugged Programming Lessons". In: *Journal of the European Teacher Education Network* 12 (2020), pp. 13–24.

[119] Andri Ioannidou et al. "Computational Thinking Patterns". In: *American Educational Research Association Annual Meeting 2011*. Oct. 2011. URL: https://www.learntechlib.org/p/108975.

[120] *It's Early Days for the New Digital Technologies Curriculum Content*. URL: https://ero.govt.nz/our-research/its-early-days-for-the-new-digital-technologies-curriculum-content (visited on 05/06/2020).

[121] Sattar Izwaini. "A corpus-based study of metaphor in information technology". In: Centre for Computational Linguistics, 2003, pp. 1–8.

[122] Association for Computing Machinery (ACM) Joint Task Force on Computing Curricula and IEEE Computer Society. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. New York, NY, USA: Association for Computing Machinery, 2013. ISBN: 9781450323093.

[123] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. English. 2nd. Englewood Cliffs., N.J: Prentice Hall, 1988. ISBN: 9780131103627; 9780131103702.

[124] *Kidbots - CS Unplugged*. URL: https://csunplugged.org/en/at-home/kidbots/.

[125] Donald E. Knuth. "Computer Science and its Relation to Mathematics". In: *The American Mathematical Monthly* 81.4 (1974), pp. 323–343. DOI: 10.1080/00029890.1974.11993556. eprint: https://doi.org/10.1080/00029890.1974.11993556. URL: https://doi.org/10.1080/00029890.1974.11993556.

[126] Andrew J. Ko. "What is a Programming Language, Really?" In: *PLATEAU 2016*. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 32–33. ISBN: 9781450346382. DOI: 10.1145/3001878.3001880. URL: https://doi.org/10.1145/3001878.3001880.

[127] Matthew J. Koehler and Punya Mishra. "Technological Pedagogical Content Knowledge: A Framework for Teacher Knowledge." English. In: *Teachers College Record* 108.6 (2006), pp. 1017–1054.

[128] Matthew J. Koehler, Punya Mishra, and William Cain. "What Is Technological Pedagogical Content Knowledge (TPACK)?" English. In: *Journal of education (Boston, Mass.)* 193.3 (2013), pp. 13–19.

[129] Özgen KORKMAZ. "The Effects of Scratch-Based Game Activities on Students' Attitudes, Self-Efficacy and Academic Achievement". In: *International Journal of Modern Education and Computer Science (IJMECS)* 8.11 (2016), pp. 16–23. ISSN: 2075-017X. DOI: 10.5815/ijmecs.2016.01.03.

[130] Shriram Krishnamurthi and Kathi Fisler. "Programming Paradigms and Beyond". In: *The Cambridge Handbook of Computing Education Research*. Ed. by Sally A. Fincher and Anthony V.Editors Robins. Cambridge Handbooks in Psychology. Cambridge University Press, 2019, pp. 377–413. DOI: 10.1017/9781108654555.014.

[131] Volkan Kukul, Şahin Gökçearslan, and Mustafa Serkan Günbatar. "Computer programming self-efficacy scale (CPSES) for secondary school students: Development, validation and reliability". In: *Eğitim Teknolojisi Kuram ve Uygulama* 7.1 (2017), pp. 158–179.

[132] Bill Kules. "Computational thinking is critical thinking: Connecting to university discourse, goals, and learning outcomes". In: *Proceedings of the Association for Information Science and Technology* 53.1 (2016), pp. 1–6. DOI: https://doi.org/10.1002/pra2.2016.14505301092. eprint: https://asistdl.onlinelibrary.wiley.com/doi/pdf/10.1002/pra2.2016.14505301092. URL: https://asistdl.onlinelibrary.wiley.com/doi/abs/10.1002/pra2.2016.14505301092.

[133] Richard Ladner. *Computer Science Unplugged (AccessComputing News Feb 2009)*. URL: https://www.washington.edu/doit/book/export/html/5288.

[134] Edmund A. Lamagna. "Algorithmic Thinking Unplugged". In: *J. Comput. Sci. Coll.* 30.6 (June 2015), pp. 45–52. ISSN: 1937-4771.

[135] Irene Lee. "Reclaiming the roots of CT". In: *CSTA Voice: The Voice of K–12 Computer Science Education and Its Educators* 12.1 (2016), pp. 3–4.

[136] Colleen M. Lewis, Michael J. Clancy, and Jan Vahrenhold. "Student Knowledge and Misconceptions". In: *The Cambridge Handbook of Computing Education Research*. Ed. by Sally A. Fincher and Anthony V.Editors Robins. Cambridge Handbooks in Psychology. Cambridge University Press, 2019, pp. 773–800. DOI: 10.1017/9781108654555.028.

[137] Leo Liberti and Fabrizio Marinelli. "Mathematical programming: Turing completeness and applications to software analysis". In: *Journal of combinatorial optimization* 28.1 (2014), pp. 82–104.

[138] Cher Ping Lim and Ching Sing Chai. "Teachers' pedagogical beliefs and their planning and conduct of computer-mediated classroom lessons". In: *British Journal of Educational Technology* 39.5 (2008), pp. 807–828. DOI: https://doi.org/10.1111/j.1467-8535.2007.00774.x. URL: https://bera-journals.onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8535.2007.00774.x.

[139] Guan-Yu Lin. "Self-Efficacy Beliefs and Their Sources in Undergraduate Computing Disciplines: An Examination of Gender and Persistence". In: *Journal of Educational Computing Research* 53.4 (2016), pp. 540–561. DOI: 10.1177/0735633115608440.

[140] Alex Lishinski et al. "Learning to Program: Gender Differences and Interactive Effects of Students' Motivation, Goals, and Self-Efficacy on Performance". In: *Proceedings of the 2016 ACM Conference on International Computing Education Research*. ICER '16. Melbourne, VIC, Australia: Association for Computing Machinery, 2016, pp. 211–220. ISBN: 9781450344494. DOI: 10.1145/2960310.2960329. URL: https://doi.org/10.1145/2960310.2960329.

[141] Chee-Kit Looi et al. "Analysis of linkages between an unplugged activity and the development of computational thinking". In: *Computer Science Education* 28.3 (2018), pp. 255–279. DOI: 10.1080/08993408.2018.1533297. eprint: https://doi.org/10.1080/08993408.2018.1533297. URL: https://doi.org/10.1080/08993408.2018.1533297.

[142] Tony A. Lowe. "Misconceptions and the Notional Machine in Very Young Programming Learners (RTP)". English. In: Atlanta: American Society for Engineering Education-ASEE, 2018.

[143] James J. Lu and George H.L. Fletcher. "Thinking about Computational Thinking". In: *SIGCSE Bull.* 41.1 (Mar. 2009), pp. 260–264. ISSN: 0097-8418. DOI: 10.1145/1539024.1508959. URL: https://doi.org/10.1145/1539024.1508959.

[144] Douglas A. Luke. *Multilevel modeling*. English. Vol. no. 07-143;7-143; Thousand Oaks, Calif: Sage Publications, 2004. ISBN: 0761928790;9780761928799;

[145] Saga Magazine. *Computer jargon explained - a glossary for beginners - Saga*. URL: https://www.saga.co.uk/magazine/technology/computing/how-to-guides/understanding-computer-jargon.

[146] Hiroki Manabe et al. "CS Unplugged Assisted by Digital Materials for Handicapped People at Schools". In: *Proceedings of the 5th International Conference on Informatics in Schools: Situation, Evolution and Perspectives*. ISSEP'11. Bratislava,

Slovakia: Springer-Verlag, 2011, pp. 82–93. ISBN: 9783642247217. DOI: 10.1007/978-3-642-24722-4_8. URL: https://doi.org/10.1007/978-3-642-24722-4_8.

[147] Andrew Manches and Lydia Plowman. "Computing education in children's early years: A call for debate." In: *British Journal of Educational Technology* 48.1 (2017), pp. 191–201. ISSN: 00071013. URL: http://ezproxy.canterbury.ac.nz/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=a9h&AN=120689225&site=ehost-live.

[148] J.R. Martin, Karl Maton, and Yaegan Doran. "Academic discourse: An interdisciplinary dialogue". In: *Accessing Academic Discourse: Systemic Functional Linguistics and Legitimation Code Theory*. Ed. by J.R. Martin, Karl Maton, and Yaegan Doran. London: Routledge, 2019. Chap. 1, pp. 1–28.

[149] Karl Maton. *Knowledge and knowers: towards a realist sociology of education*. English. London: Routledge, 2014. ISBN: 0415479991;9780415479998.

[150] Karl Maton. "Making semantic waves: A key to cumulative knowledge-building". In: *Linguistics and education* 24.1 (2013), pp. 8–22.

[151] S. A. McLeod. *What Is the zone of proximal development?* 2019. URL: www.simplypsychology.org/Zone-of-Proximal-Development.html.

[152] Peter McOwan and Paul Curzon. *The Create-a-face Activity*. URL: https://teachinglondoncomputing.files.wordpress.com/2014/02/activity-create-a-face.pdf.

[153] L. Meddings and S. Thornbury. *Teaching Unplugged: Dogme in English Language Teaching*. Delta Teacher Development Series. Delta Publishing, 2017. ISBN: 9783125013568.

[154] Neil Mercer. *Words and Minds: How We Use Language to Think Together*. English. 1st ed. London: Routledge, 2000. ISBN: 9780415224758; 0415224756; 0203464982; 9780203464984.

[155] Shari J. Metcalf et al. "Assessing computational thinking through the lenses of functionality and computational fluency". In: *Computer Science Education* 31.2 (2021), pp. 199–223. DOI: 10.1080/08993408.2020.1866932. eprint: https://doi.org/10.1080/08993408.2020.1866932.

[156] Iain Milne and Glenn Rowe. "Difficulties in Learning and Teaching Programming—Views of Students and Tutors". In: 7.1 (Mar. 2002), pp. 55–66. ISSN: 1360-2357. DOI: 10.1023/A:1015362608943. URL: https://doi.org/10.1023/A:1015362608943.

[157] Monika Mladenović, Žana Žanko, and Ivica Boljat. "Programming Misconceptions at the K-12 Level". In: *Encyclopedia of Education and Information Technologies*. Ed. by Arthur Tatnall. Cham: Springer International Publishing, 2020, pp. 1383–1395. ISBN: 978-3-030-10576-1. DOI: 10.1007/978-3-030-10576-1_234. URL: https://doi.org/10.1007/978-3-030-10576-1_234.

[158] Patricia Morreale and David A. Joiner. "Reaching future computer scientists". In: *Communications of the ACM* 54.4 (2011), pp. 121–124. DOI: 10.1145/1924421.1924448. URL: https://doi.org/10.1145/1924421.1924448.

[159] *Most Common Computer Jargons List*. URL: http://kpush.tripod.com/tweaking/id49.html.

[160] D. Moursund. *Computational thinking and math maturity: improving math education in K-12 schools*. University of Oregon Press, 2006.

[161] Chrystalla Mouza et al. "Resetting educational technology coursework for pre-service teachers: A computational thinking approach to the development of technological pedagogical content knowledge (TPACK)". In: *Australasian Journal of Educational Technology* 33.3 (July 2017). DOI: 10.14742/ajet.3521. URL: https://ajet.org.au/index.php/AJET/article/view/3521.

[162] Bhagya Munasinghe, Tim Bell, and Anthony Robins. "Teachers' Understanding of Technical Terms in a Computational Thinking Curriculum". In: *Australasian Computing Education Conference*. ACE '21. Virtual, SA, Australia: Association for Computing Machinery, 2021, pp. 106–114. ISBN: 9781450389761. DOI: 10.1145/3441636.3442311. URL: https://doi.org/10.1145/3441636.3442311.

[163] Tom Murray and Ivon Arroyo. "Toward Measuring and Maintaining the Zone of Proximal Development in Adaptive Instructional Systems". In: *Intelligent Tutoring Systems*. 2002.

[164] Harley R. Myler. *Fundamentals of Engineering Programming with C and FORTRAN*. USA: Cambridge University Press, 1998. ISBN: 0521620635.

[165] *National curriculum in England: computing programmes of study*. URL: https://www.gov.uk/government/publications/national-curriculum-in-england-computing-programmes-of-study (visited on 01/09/2022).

[166] Greg L. Nelson, Benjamin Xie, and Amy J. Ko. "Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1". In: *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ICER '17. Tacoma, Washington, USA: Association for Computing Machinery, 2017, pp. 2–11. ISBN: 9781450349680. DOI: 10.1145/3105726.3106178. URL: https://doi.org/10.1145/3105726.3106178.

[167] Tomohiro Nishida et al. "A CS Unplugged Design Pattern". In: *SIGCSE Bull.* 41.1 (Mar. 2009), pp. 231–235. ISSN: 0097-8418. DOI: 10.1145/1539024.1508951. URL: https://doi.org/10.1145/1539024.1508951.

[168] Mark Noone and Aidan Mooney. *First Programming Language: Visual or Textual?* 2017. arXiv: 1710.11557 [cs.CY].

[169] Mark Noone and Aidan Mooney. "Visual and textual programming languages: a systematic review of the literature". In: *Journal of Computers in Education* 5.2 (Mar. 2018), pp. 149–174. ISSN: 2197-9995. DOI: 10.1007/s40692-018-0101-5. URL: http://dx.doi.org/10.1007/s40692-018-0101-5.

[170] Jalal Nouri et al. "Development of computational thinking, digital competence and 21st century skills when learning programming in K-9". In: *Education Inquiry* 11.1 (2020), pp. 1–17. DOI: 10.1080/20004508.2019.1627844. eprint: https://doi.org/10.1080/20004508.2019.1627844. URL: https://doi.org/10.1080/20004508.2019.1627844.

[171] Steven O'Bryan. "Quantifying student learning within the Zone of Proximal Development: Application in an accelerated program". In: *Proceedings of the 9th Workshop in Primary and Secondary Computing Education*. STARS, 2019. URL: https://vuir.vu.edu.au/39104/.

[172] Büşra Özmen and Arif Altun. "Undergraduate Students' Experiences in Programming: Difficulties and Obstacles". English. In: *Turkish online journal of qualitative inquiry* 5.3 (2014).

[173] Frank Pajares. "Self-Efficacy Beliefs in Academic Settings". In: *Review of Educational Research* 66.4 (1996), pp. 543–578. DOI: 10.3102/00346543066004543. eprint: https://doi.org/10.3102/00346543066004543. URL: https://doi.org/10.3102/00346543066004543.

[174] Seymour Papert. *Mindstorms: children, computers and powerful ideas*. English. Vol. 14. Brighton: Harvester Press, 1980. ISBN: 9780855271633;0855271639;

[175] F. Paraskeva, H. Bouta, and Aik. Papagianni. "Individual characteristics and computer self-efficacy in secondary education teachers to integrate technology in educational practice". In: *Computers & Education* 50.3 (2008), pp. 1084–1091. ISSN: 0360-1315. DOI: https://doi.org/10.1016/j.compedu.2006.10.006. URL: https://www.sciencedirect.com/science/article/pii/S0360131506001643.

[176] Dale Parsons and Patricia Haden. "Parson's Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses". In: *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52*. ACE '06. Hobart, Australia: Australian Computer Society, Inc., 2006, pp. 157–163. ISBN: 1920682341.

[177] Roy D. Pea. "Language-Independent Conceptual "Bugs" in Novice Programming". In: *Journal of Educational Computing Research* 2.1 (1986), pp. 25–36. DOI: 10.2190/689T-1R2A-X4W4-29J2. eprint: https://doi.org/10.2190/689T-1R2A-X4W4-29J2. URL: https://doi.org/10.2190/689T-1R2A-X4W4-29J2.

[178] *Pedagogy Quick Reads: Improving explanations and learning activities in computing using semantic waves*. URL: https://raspberrypi-education.s3-eu-west-1.amazonaws.com/Quick+Reads/Pedagogy+Quick+Read+6+-+Semantic+Waves.pdf (visited on 01/30/2022).

[179] Paul R. Pintrich and Elisabeth V. de Groot. "Motivational and self-regulated learning components of classroom academic performance." In: *Journal of Educational Psychology* 82.1 (1990), pp. 33–40. ISSN: 0022-0663.

[180] Martinha Piteira and Carlos Costa. "Learning Computer Programming: Study of Difficulties in Learning Programming". In: *Proceedings of the 2013 International Conference on Information Systems and Design of Communication*. ISDOC '13. Lisboa, Portugal: Association for Computing Machinery, 2013, pp. 75–80. ISBN: 9781450322997. DOI: 10.1145/2503859.2503871. URL: https://doi.org/10.1145/2503859.2503871.

[181] Alessandro Pluchino et al. "Agent-Based Simulation of Pedestrian Behaviour in Closed Spaces: A Museum Case Study". In: *Journal of Artificial Societies and Social Simulation* 17.1 (2014), p. 16. ISSN: 1460-7425. DOI: 10.18564/jasss.2336. URL: http://jasss.soc.surrey.ac.uk/17/1/16.html.

[182] David L. Poole and Alan K. Mackworth. "Artificial Intelligence and Agents". In: *Artificial Intelligence: Foundations of Computational Agents*. 2nd ed. Cambridge University Press, 2017. Chap. 1, pp. 3–48. DOI: 10.1017/9781108164085.002.

[183] Terry Pratchett, Ian Stewart, and Jack Cohen. *The Science of Discworld*. London,UK: Ebury Press, 1999. ISBN: 978-0091865153.

[184] Yizhou Qian and James Lehman. "Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review". English. In: *ACM transactions on computing education* 18.1 (2017), pp. 1–24.

[185] Jake A. Qualls and Linda B. Sherrell. "Why Computational Thinking Should Be Integrated into the Curriculum". In: *J. Comput. Sci. Coll.* 25.5 (2010), pp. 66–71. ISSN: 1937-4771.

[186] Vennila Ramalingam, Deborah LaBelle, and Susan Wiedenbeck. "Self-Efficacy and Mental Models in Learning to Program". In: *SIGCSE Bull.* 36.3 (June 2004), pp. 171–175. ISSN: 0097-8418. DOI: 10.1145/1026487.1008042. URL: https://doi.org/10.1145/1026487.1008042.

[187] Vennila Ramalingam and Susan Wiedenbeck. "Development and Validation of Scores on a Computer Programming Self-Efficacy Scale and Group Analyses of Novice Programmer Self-Efficacy". English. In: *Journal of educational computing research* 19.4 (1998), p. 367.

[188] Alexander Repenning. "Programming Goes Back to School". In: *Commun. ACM* 55.5 (May 2012), pp. 38–40. ISSN: 0001-0782. DOI: 10.1145/2160718.2160729. URL: https://doi-org.ezproxy.canterbury.ac.nz/10.1145/2160718.2160729.

[189] Luis Reynoso, Sergio Romero, and Fernando Romero. "Towards a formal model of the pedagogic discourse and the Zone of Proximal Development (ZPD) of Vygotsky". In: *2012 IEEE 11th International Conference on Cognitive Informatics and Cognitive Computing*. 2012, pp. 510–517. DOI: 10.1109/ICCI-CC.2012.6311201.

[190] Anthony Robins, Janet Rountree, and Nathan Rountree. "Learning and Teaching Programming: A Review and Discussion". In: *Computer Science Education* 13.2 (2003), pp. 137–172. DOI: 10.1076/csed.13.2.137.14200. eprint: https://doi.org/10.1076/csed.13.2.137.14200. URL: https://doi.org/10.1076/csed.13.2.137.14200.

[191] Anthony Robins, Janet Rountree, and Nathan Rountree. "Learning and teaching programming: A review and discussion". In: *Computer science education* 13.2 (2003), pp. 137–172.

[192] Anthony V. Robins. "Dual Process Theories: Computing Cognition in Context". In: *ACM Trans. Comput. Educ.* (Sept. 2021). Just Accepted. DOI: 10.1145/3487055. URL: https://doi.org/10.1145/3487055.

[193] Anthony V. Robins. "Novice Programmers and Introductory Programming". In: *The Cambridge Handbook of Computing Education Research*. Ed. by Sally A. Fincher and Anthony V.Editors Robins. Cambridge Handbooks in Psychology. Cambridge University Press, 2019, pp. 327–376. DOI: 10.1017/9781108654555.013.

[194] Brandon Rodriguez, Cyndi Rader, and Tracy Camp. "Using Student Performance to Assess CS Unplugged Activities in a Classroom Environment". In: *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE '16. Arequipa, Peru: Association for Computing Machinery, 2016, pp. 95–100. ISBN: 9781450342315. DOI: 10.1145/2899415.2899465. URL: https://doi.org/10.1145/2899415.2899465.

[195] Brandon Rodriguez et al. "Assessing Computational Thinking in CS Unplugged Activities". In: SIGCSE '17. Seattle, Washington, USA: Association for Computing Machinery, 2017, pp. 501–506. ISBN: 9781450346986. DOI: 10.1145/3017680.3017779. URL: https://doi.org/10.1145/3017680.3017779.

[196] Barbara Rogoff. *The cultural nature of human development*. English. Oxford [UK]; New York: Oxford University Press, 2003. ISBN: 9780195131338;0195131339.

[197] Stuart Rowlands. "Vygotsky and the ZPD: Have we got it right?" In: *Research in Mathematics Education* 5.1 (2003), pp. 155–170. DOI: 10.1080/14794800008520120. eprint: https://doi.org/10.1080/14794800008520120. URL: https://doi.org/10.1080/14794800008520120.

[198] Gerdamarie Schmitz. "Entwicklung der Selbstwirksamkeitserwartungen von Lehrern; Development of teachers' self-efficacy beliefs". ger. In: *Unterrichtswissenschaft* 26.2 (1998), pp. 140–157. ISSN: 0340-4099. DOI: https://doi.org/10.25656/01:7770.

[199] Ralf Schwarzer, Gerdamarie S. Schmitz, and Gary T. Daytner. *Teacher Self-Efficacy*. URL: https://userpage.fu-berlin.de/~health/teacher_se.htm (visited on 02/20/2020).

[200] Cynthia Selby and John Woollard. *Computational thinking: the developing definition*. Tech. rep. 2013.

[201] Pratim Sengupta et al. "Integrating computational thinking with K-12 science education using agent-based computation: A theoretical framework". English. In: *Education and information technologies* 18.2 (2013), pp. 351–380.

[202] Sue Sentance, Erik Barendsen, and Carsten Schulte. *Computer science education: perspectives on teaching and learning in school*. English. 2018. ISBN: 9781350057111.

[203] Sue Sentance and Andrew Csizmadia. "Computing in the Curriculum: Challenges and Strategies from a Teacher's Perspective". In: *Education and Information Technologies* 22.2 (Mar. 2017), pp. 469–495. ISSN: 1360-2357. DOI: 10.1007/s10639-016-9482-0. URL: https://doi.org/10.1007/s10639-016-9482-0.

[204] Sue Sentance and Andrew Csizmadia. "Computing in the curriculum: Challenges and strategies from a teacher's perspective". In: *Education and information technologies* 22.2 (2016;2017), pp. 469–495.

[205] Sue Sentance and Andrew Csizmadia. "Teachers' perspectives on successful strategies for teaching Computing in school". English. In: *IFIP TC3 Working Conference 2015*. July 2015.

[206] Sue Sentance and Jane Waite. "PRIMM: Exploring Pedagogical Approaches for Teaching Text-Based Programming in School". In: *Proceedings of the 12th Workshop on Primary and Secondary Computing Education*. WiPSCE '17. Nijmegen, Netherlands: Association for Computing Machinery, 2017, pp. 113–114. ISBN: 9781450354288. DOI: 10.1145/3137065.3137084. URL: https://doi.org/10.1145/3137065.3137084.

[207] Giovanni Serafini. "Teaching Programming at Primary Schools: Visions, Experiences, and Long-Term Research Prospects". In: ISSEP'11. Bratislava, Slovakia: Springer-Verlag, 2011, pp. 143–154. ISBN: 9783642247217. DOI: 10.1007/978-3-642-24722-4_13. URL: https://doi.org/10.1007/978-3-642-24722-4_13.

[208] Karim Shabani, Mohamad Khatib, and Saman Ebadi. "Vygotsky's zone of proximal development: Instructional implications and teachers' professional development." In: *English language teaching* 3.4 (2010), pp. 237–248.

[209] Clifford A. Shaffer et al. "Algorithm Visualization: The State of the Field". In: *ACM Trans. Comput. Educ.* 10.3 (Aug. 2010). DOI: 10.1145/1821996.1821997. URL: https://doi.org/10.1145/1821996.1821997.

[210] Margot Lee Shetterly. *Hidden Figures: The American Dream and the Untold Story of the Black Women Who Helped Win the Space Race*. USA: William Morrow and Company, 2010. ISBN: 978-0-06-236360-2.

[211] Valerie J. Shute, Chen Sun, and Jodi Asbell-Clarke. "Demystifying computational thinking". English. In: *Educational research review* 22 (2017), pp. 142–158.

[212] Teemu Sirkiä and Juha Sorva. "Exploring Programming Misconceptions: An Analysis of Student Mistakes in Visual Program Simulation Exercises". In: *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*. Koli Calling '12. Koli, Finland: Association for Computing Machinery, 2012, pp. 19–28. ISBN: 9781450317955. DOI: 10.1145/2401796.2401799. URL: https://doi.org/10.1145/2401796.2401799.

[213] M. Sivasakthi and R. Rajendran. "Learning difficulties of 'object-oriented programming paradigm using Java': students' perspective". In: *Indian Journal of Science and Technology* 4.8 (2011), pp. 983–985. DOI: 10.17485/ijst/2011/v4i8.9.

[214] Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich. *Cognitive strategies and looping constructs: an empirical study*. 1983.

[215] Juha Sorva. "Notional Machines and Introductory Programming Education". In: *ACM Trans. Comput. Educ.* 13.2 (July 2013). DOI: 10.1145/2483710.2483713. URL: https://doi.org/10.1145/2483710.2483713.

[216] Juha Sorva. "Visual program simulation in introductory programming education". English. Doctoral thesis. School of Science, 2012, p. 428. ISBN: 978-952-60-4626-6 (electronic), 978-952-60-4625-9 (printed). URL: http://urn.fi/URN:ISBN:978-952-60-4626-6.

[217] Juha Sorva, Ville Karavirta, and Lauri Malmi. "A Review of Generic Program Visualization Systems for Introductory Programming Education". In: *ACM Trans. Comput. Educ.* 13.4 (Nov. 2013). DOI: 10.1145/2490822. URL: https://doi-org.ezproxy.canterbury.ac.nz/10.1145/2490822.

[218] Gary S. Stager and Sylvia Martinez. "Thirteen Considerations for Teaching Coding to Children". In: *Creating the Coding Generation in Primary Schools : A Practical Guide for Cross-Curricular Teaching*. Ed. by Steve Humble. Routledge, 2018, pp. 36–52. ISBN: 9781138681187.

[219] Louise Starkey. "Teachers' pedagogical reasoning and action in the digital age". In: *Teachers and Teaching* 16.2 (2010), pp. 233–244. DOI: 10.1080/13540600903478433. URL: https://doi.org/10.1080/13540600903478433.

[220] Max Stephens and Djordje M. Kadijevich. "Computational/Algorithmic Thinking". In: *Encyclopedia of Mathematics Education*. Ed. by Stephen Lerman. Cham: Springer International Publishing, 2020, pp. 117–123. ISBN: 978-3-030-15789-0. DOI: 10.1007/978-3-030-15789-0_100044. URL: https://doi.org/10.1007/978-3-030-15789-0_100044.

[221] Jon Storm. *Dictionary of computer jargon*. URL: http://www.jonstorm.com/glossary/.

[222] Woonhee Sung, Junghyun Ahn, and John B. Black. "Introducing Computational Thinking to Young Learners: Practicing Computational Perspectives Through Embodiment in Mathematics Education". In: *Technology, Knowledge and Learning* 22.3 (2017), pp. 443–463. ISSN: 2211-1670. DOI: 10.1007/s10758-017-9328-x. URL: https://doi.org/10.1007/s10758-017-9328-x.

[223] Edda Sveinsdottir and Erik Frøkjær. "Datalogy — The copenhagen tradition of computer science". In: *BIT Numerical Mathematics* 28.3 (1988), pp. 450–472. ISSN: 1572-9125. DOI: 10.1007/BF01941128. URL: https://doi.org/10.1007/BF01941128.

[224] Alaaeddin Swidan, Felienne Hermans, and Marileen Smit. "Programming Misconceptions for School Students". English. In: ACM, 2018, pp. 151–159. ISBN: 9781450356282;1450356281;

[225] Sebastian Szyjka. "Understanding research paradigms: Trends in science education research". In: *Problems of Education in the 21st Century* 43 (2012), p. 110.

[226] Phit-Huan Tan, Choo-Yee Ting, and Siew-Woei Ling. "Learning Difficulties in Programming Courses: Undergraduates' Perspective and Perception". In: *2009 International Conference on Computer Technology and Development*. Vol. 1. 2009, pp. 42–46. DOI: 10.1109/ICCTD.2009.188.

[227] Rivka Taub, Michal Armoni, and Mordechai Ben-Ari. "CS Unplugged and Middle-School Students' Views, Attitudes, and Intentions Regarding CS". In: *ACM Trans. Comput. Educ.* 12.2 (Apr. 2012). DOI: 10.1145/2160547.2160551. URL: https://doi.org/10.1145/2160547.2160551.

[228] Mohsen Tavakol and Reg Dennick. "Making sense of Cronbach's alpha". In: *International journal of medical education* 2 (2011), p. 53.

[229] Matti Tedre and Peter J. Denning. "The Long Quest for Computational Thinking". In: *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*. Koli Calling '16. Koli, Finland: Association for Computing Machinery, 2016, pp. 120–129. ISBN: 9781450347709. DOI: 10.1145/2999541.2999542. URL: https://doi.org/10.1145/2999541.2999542.

[230] *The New Zealand Curriculum Online*. URL: https://nzcurriculum.tki.org.nz/The-New-Zealand-Curriculum/Technology/Learning-area-structure. (visited on 01/09/2022).

[231] Renate Thies and Jan Vahrenhold. "On Plugging "Unplugged" into CS Classes". In: SIGCSE '13. Denver, Colorado, USA: Association for Computing Machinery, 2013, pp. 365–370. ISBN: 9781450318686. DOI: 10.1145/2445196.2445303. URL: https://doi.org/10.1145/2445196.2445303.

[232] David Thompson et al. "The Role of Teachers in Implementing Curriculum Changes". In: SIGCSE '13. Denver, Colorado, USA: Association for Computing Machinery, 2013, pp. 245–250. ISBN: 9781450318686. DOI: 10.1145/2445196.2445272. URL: https://doi-org.ezproxy.canterbury.ac.nz/10.1145/2445196.2445272.

[233] Arinchaya Threekunprapa and Pratchayapong Yasri. "Unplugged coding using flowblocks for promoting computational thinking and programming among secondary school students". English. In: *International Journal of Instruction* 13.3 (2020), pp. 207–222.

[234] Katerina Tsarava, Korbinian Moeller, and Manuel Ninaus. "Training Computational Thinking through board games: The case of Crabs & Turtles". In: *International Journal of Serious Games* 5.2 (June 2018), pp. 25–44. DOI: 10.17083/ijsg.v5i2.248. URL: https://journal.seriousgamessociety.org/index.php/IJSG/article/view/248.

[235] A. M. Turing. "I.— Computing Machinery and Intelligence". In: *Mind* LIX.236 (Oct. 1950), pp. 433–460. ISSN: 0026-4423. DOI: 10.1093/mind/LIX.236.433. eprint: https://academic.oup.com/mind/article-pdf/LIX/236/433/30123314/lix-236-433.pdf. URL: https://doi.org/10.1093/mind/LIX.236.433.

[236] Alan Mathison Turing. "On computable numbers, with an application to the Entscheidungsproblem". In: *Journal of Mathematics* 58.345-363 (1936), p. 5.

[237] Rebecca Vivian and Katrina Falkner. "Identifying Teachers' Technological Pedagogical Content Knowledge for Computer Science in the Primary Years". In: ICER '19. Toronto ON, Canada: Association for Computing Machinery, 2019, pp. 147–155. ISBN: 9781450361859. DOI: 10.1145/3291279.3339410. URL: https://doi.org/10.1145/3291279.3339410.

[238] L.S. Vygotskii and Michael Cole. *Mind in society: the development of higher psychological processes*. English. Cambridge: Harvard University Press, 1978. ISBN: 9780674576285; 9780674576292;

[239] L.S. Vygotskii and A. Kozulin. *Thought and Language*. MIT paperback series. MIT Press, 1986. ISBN: 9780262720106. URL: https://books.google.co.nz/books?id=bkz7stcdbKgC.

[240] Jane Waite. "Pedagogy in teaching computer science in schools: A literature review". In: *London: Royal Society* (2017). URL: https://royalsociety.org/-/media/policy/projects/computing-education/literature-review-pedagogy-in-teaching.pdf.

[241] Jane Waite et al. "Unplugged Computing and Semantic Waves: Analysing Crazy Characters". In: *Proceedings of the 1st UK and Ireland Computing Education Research Conference*. UKICER. Canterbury, United Kingdom: Association for Computing Machinery, 2019. ISBN: 9781450372572. DOI: 10.1145/3351287.3351291. URL: https://doi.org/10.1145/3351287.3351291.

[242] Mark K. Warford. "The zone of proximal teacher development". In: *Teaching and Teacher Education* 27.2 (2011), pp. 252–258. ISSN: 0742-051X. DOI: https://doi.org/10.1016/j.tate.2010.08.008. URL: https://www.sciencedirect.com/science/article/pii/S0742051X10001447.

[243] Mary Webb and Margaret Cox. "A review of pedagogy related to information and communications technology". In: *Technology, Pedagogy and Education* 13.3 (2004), pp. 235–286. DOI: 10.1080/14759390400200183.

[244] Mary Webb et al. "Computer Science in the School Curriculum: Issues and Challenges". In: *Tomorrow's Learning: Involving Everyone. Learning with and about Technologies and Computing*. Ed. by Arthur Tatnall and Mary Webb. Cham: Springer International Publishing, 2017, pp. 421–431. ISBN: 978-3-319-74310-3.

[245] Webopedia. *Programming Language Definition*. URL: https://www.webopedia.com/TERM/P/programming_language.html.

[246] Susan Wiedenbeck. "Factors Affecting the Success of Non-Majors in Learning to Program". In: *Proceedings of the First International Workshop on Computing Education Research*. ICER '05. Seattle, WA, USA: Association for Computing Machinery, 2005, pp. 13–24. ISBN: 1595930434. DOI: 10.1145/1089786.1089788. URL: https://doi.org/10.1145/1089786.1089788.

[247] *Wilcoxon Signed Ranks Test*. URL: https://www.sciencedirect.com/topics/medicine-and-dentistry/wilcoxon-signed-ranks-test (visited on 03/12/2020).

[248] Jeannette M. Wing. “Computational Thinking”. In: *Commun. ACM* 49.3 (Mar. 2006), pp. 33–35. ISSN: 0001-0782. DOI: 10.1145/1118178.1118215. URL: https://doi.org/10.1145/1118178.1118215.

[249] Jeannette M. Wing. *Computational Thinking: What and Why?* 2010. URL: http://www.cs.cmu.edu/~CompThink/papers/TheLinkWing.pdf.

[250] Thomas Nelson Winter. “The Mechanical Problems in the Corpus ofAristotle”. In: *Faculty Publications, Calssics and Religious Studies Department* 68 (July 2007). ISSN: 10018417. URL: http://digitalcommons.unl.edu/classicsfacpub/68.

[251] Benjamin Wohl, Barry Porter, and Sarah Clinch. “Teaching Computer Science to 5-7 Year-Olds: An Initial Study with Scratch, Cubelets and Unplugged Computing”. In: WiPSCE ’15. London, United Kingdom: Association for Computing Machinery, 2015, pp. 55–60. ISBN: 9781450337533. DOI: 10.1145/2818314.2818340. URL: https://doi.org/10.1145/2818314.2818340.

[252] Gary K.W. Wong and Shan Jiang. “Computational Thinking Education for Children: Algorithmic Thinking and Debugging”. In: *2018 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*. 2018, pp. 328–334. DOI: 10.1109/TALE.2018.8615232.

[253] S. Yadav A.and Gretter, J. Good, and T. McLean. “Computational Thinking in Teacher Education”. In: *Emerging Research, Practice, and Policy on Computational Thinking*. Ed. by Peter Rich and Charles Hodges. Springer, 2017, pp. 205–220. ISBN: 9781350057111;9781350057104.

[254] Tetsuo Yokoyama, Holger B. Axelsen, and Robert Glück. “Fundamentals of reversible flowchart languages”. In: *Theoretical computer science* 611 (2016), pp. 87–115. ISSN: ISSN 0304-3975. DOI: 10.1016/j.tcs.2015.07.046. URL: https://doi.org/10.1016/j.tcs.2015.07.046.

[255] Barry J. Zimmerman. “Self-Efficacy: An Essential Motive to Learn”. In: *Contemporary Educational Psychology* 25.1 (2000), pp. 82–91. ISSN: 0361-476X. DOI: https://doi.org/10.1006/ceps.1999.1016. URL: https://www.sciencedirect.com/science/article/pii/S0361476X99910160.