How Can the Teaching of Programming Be Used to Enhance Computational Thinking Skills?

by

Cynthia Collins Selby

Thesis for the degree of Doctor of Philosophy

April 2014
HOW CAN THE TEACHING OF PROGRAMMING BE USED TO ENHANCE COMPUTATIONAL THINKING SKILLS?

Cynthia Collins Selby

The use of the term computational thinking, introduced in 2006 by Jeanette Wing, is having repercussions in the field of education. The term brings into sharp focus the concept of thinking about problems in a way that can lead to solutions that may be implemented in a computing device. Implementation of these solutions may involve the use of programming languages.

This study explores ways in which programming can be employed as a tool to teach computational thinking and problem solving. Data is collected from teachers, academics, and professionals, purposively selected because of their knowledge of the topics of problem solving, computational thinking, or the teaching of programming. This data is analysed following a grounded theory approach. A Computational Thinking Taxonomy is developed. The relationships between cognitive processes, the pedagogy of programming, and the perceived levels of difficulty of computational thinking skills are illustrated by a model.

Specifically, a definition for computational thinking is presented. The skills identified are mapped to Bloom's Taxonomy: Cognitive Domain. This mapping concentrates computational skills at the application, analysis, synthesis, and evaluation levels. Analysis of the data indicates that the less difficult computational thinking skills for beginner programmers are generalisation, evaluation, and algorithm design. Abstraction of functionality is less difficult than abstraction of data, but both are perceived as difficult. The most difficult computational thinking skill is reported as decomposition. This ordering of difficulty for learners is a reversal of the cognitive complexity predicted by Bloom's model. The plausibility of this inconsistency is explored.

The taxonomy, model, and the other results of this study may be used by educators to focus learning onto the computational thinking skills acquired by the learners, while using programming as a tool. They may also be employed in the design of curriculum subjects, such as ICT, computing, or computer science.
Contents

ABSTRACT .................................................................................................................. i

Contents ................................................................................................................... iii

List of tables ............................................................................................................. ix

List of figures ........................................................................................................... xi

DECLARATION OF AUTHORSHIP ...................................................................... xv

Acknowledgements ................................................................................................. xvii

Definitions and Abbreviations ............................................................................... xix

Chapter 1. Introduction .......................................................................................... 1

1.1 Background ....................................................................................................... 1

1.2 Research questions .......................................................................................... 3

Chapter 2. Frameworks ......................................................................................... 5

2.1 Theoretical framework .................................................................................... 5

2.2 Bloom’s Taxonomy ......................................................................................... 5

2.3 The revised Taxonomy ..................................................................................... 7

2.4 The SOLO taxonomy ....................................................................................... 10

2.5 Bloom’s Digital Taxonomy ............................................................................. 13

2.6 Conclusion ....................................................................................................... 15

2.7 Conceptual framework ..................................................................................... 18

Chapter 3. Literature review .................................................................................. 21

3.1 Rationale for literature selection ...................................................................... 21

3.2 Computational thinking .................................................................................... 22

3.2.1 Defining computational thinking .................................................................. 22

3.2.1.1 Is a definition required? ........................................................................ 23

3.2.1.2 Computing, computer science, or computational thinking .................. 24

3.2.1.3 What computational thinking is not ..................................................... 24

3.2.1.4 Basic terminology ................................................................................ 25

3.2.1.5 A developing definition ........................................................................ 27

3.2.1.5.1 A thought process ........................................................................... 27

3.2.1.5.2 Abstraction ....................................................................................... 28

3.2.1.5.3 Decomposition ................................................................................ 29

3.2.1.5.4 Thinking terms ................................................................................ 29
4.8.2.1  Next actions ............................................................................................... 104
4.8.2.2  Conclusion .................................................................................................. 104
4.8.3  Milestone 3: Additional concepts ................................................................. 105
  4.8.3.1  Next actions ............................................................................................... 107
  4.8.3.2  Conclusion .................................................................................................. 107
4.8.4  Milestone 4: Initial queries ............................................................................. 108
  4.8.4.1  Next actions ............................................................................................... 110
  4.8.4.2  Conclusion .................................................................................................. 110
4.8.5  Milestone 5: Defining a lexicon, hierarchies, and models ......................... 111
  4.8.5.1  Model generation and discussion ............................................................... 117
  4.8.5.2  Next actions ............................................................................................... 121
  4.8.5.3  Conclusion .................................................................................................. 122
4.8.6  Milestone 6: Interviews and additional models ........................................... 122
  4.8.6.1  Model generation and discussion ............................................................... 126
  4.8.6.2  Next actions ............................................................................................... 136
  4.8.6.3  Conclusion .................................................................................................. 137
4.8.7  Milestone 7: Additional data and refinement of models ............................. 138
  4.8.7.1  Model generation and discussion ............................................................... 143
  4.8.7.2  Next actions ............................................................................................... 147
  4.8.7.3  Conclusion .................................................................................................. 147
4.8.8  Milestone 8: Model clarification ................................................................... 148
  4.8.8.1  Model generation and discussion ............................................................... 155
  4.8.8.2  Next actions ............................................................................................... 164
  4.8.8.3  Conclusion .................................................................................................. 165
4.8.9  Milestone 9: Taxonomy of terms ................................................................... 165
  4.8.9.1  Model generation and discussion ............................................................... 166
  4.8.9.2  Next actions ............................................................................................... 169
  4.8.9.3  Conclusion .................................................................................................. 169
4.8.10 Milestone 10: Final issues ............................................................................ 169
  4.8.10.1 Model generation and discussion .............................................................. 176
  4.8.10.2 Conclusion .................................................................................................. 178
4.9  Final model ......................................................................................................... 179
4.10 Final data set statistics ..................................................................................... 181
4.11 Conclusion ......................................................................................................... 184

Chapter 5.  Discussion ............................................................................................. 187

5.1 Taxonomy of computational thinking skills ..................................................... 187
5.2 Hierarchy of difficulty of computational thinking skills ................................. 189
6.2 Reflection on reliability and validity.................................................................218
6.3 Limitations .........................................................................................................223
6.4 Contribution .......................................................................................................224
6.5 Future work .......................................................................................................226
6.6 High-level summary ..........................................................................................228
  6.6.1 Summary of findings .....................................................................................228
  6.6.2 A relationship model ....................................................................................229
  6.6.3 Future research .............................................................................................231
  6.6.4 Advice for classroom practitioners ...............................................................231

Appendices .............................................................................................................233
  Appendix 1. Interview revision (original) .............................................................233
  Appendix 2. Interview revision (af) .....................................................................237
  Appendix 3. Interview revision (ag) .....................................................................243
  Appendix 4. Interview revision (ah) .....................................................................249
  Appendix 5. Four Approaches to Teaching Programming ...................................255
  Appendix 6. Teaching Programming – A Theoretical Framework .......................265
  Appendix 7. Promoting Computational Thinking with Programming ...............269
  Appendix 8. Computational Thinking: The Developing Definition ......................273
  Appendix 9. Research into ICT to support computing in education ....................279
  Appendix 11. Computing at School 2012 – Poster .............................................287
  Appendix 12. Computing at School 2011 – Poster .............................................289

References ...............................................................................................................291
List of tables

Table 1: Computational thinking definition terminology ........................................38
Table 2: Questionnaire mapped to question context ..............................................85
Table 3: Interview schedule mapped to question contexts .....................................97
Table 4: Proposed queries and justification .........................................................112
Table 5: First mapping of terminology against an existing hierarchy ....................166
Table 6: Perceived level of difficulty of computational thinking skills .................168
Table 7: Relationship between three taxonomies .................................................178
Table 8: Computational thinking skills to cognitive domain mapping ..................188
Table 9: Computational thinking skills and hierarchy of difficulty .......................189
Table 10: Current model related to Bloom's Taxonomy and the revised Taxonomy .................................................................198
List of figures

Figure 1: Cognitive domain of Bloom’s Taxonomy .............................................6

Figure 2: The revised Taxonomy .....................................................................8

Figure 3: The revised Taxonomy table...............................................................9

Figure 4: SOLO taxonomy (Biggs and Tang 2007, p. 79) ..............................11

Figure 5: The Digital Taxonomy (Churches 2009a) .......................................14

Figure 6: Theory relationships .....................................................................16

Figure 7: Conceptual framework ..................................................................19

Figure 8: Words used in questionnaire responses .........................................101

Figure 9: Words used in community of practice responses ...........................105

Figure 10: Words used in interview responses ..............................................107

Figure 11: Teachers’ use of computational thinking terminology ...............109

Figure 12: Bloom’s cognitive taxonomy and comparatives ..........................114

Figure 13: Computational thinking terminology and comparatives ..........115

Figure 14: Nodes representing relationships ...............................................116

Figure 15: (a) Teaching programming corresponds to Bloom’s cognitive domain 118

Figure 16: (b) Teaching programming corresponds to Bloom’s cognitive domain 119

Figure 17: Computational thinking terminology hierarchy of difficulty ..........120

Figure 18: Responses indicating transition between notations .....................123

Figure 19: Differentiation is a concern for many respondents .....................125

Figure 20: Limiting factors for computational thinking skills ......................126

Figure 21: (a) Teaching programming corresponds to Bloom’s cognitive domain 127

Figure 22: (b) Teaching programming corresponds to Bloom’s cognitive domain 128
Figure 23: (a) Subdivision of programming skills and computational thinking skills across the cognitive domain .................................................. 130

Figure 24: (b) Subdivision of programming skills and computational thinking skills across the cognitive domain .................................................. 131

Figure 25: Computational thinking terminology hierarchy of difficulty .......... 132

Figure 26: Formal notations and difficulty of moving between them ............ 134

Figure 27: Respondents’ interpretations of relationships .............................. 136

Figure 28: Sources indicating nested instructions are difficult to understand .... 138

Figure 29: Sources using words associated with “nesting” ............................ 139

Figure 30: Query to identify usage of analysis and debugging ....................... 139

Figure 31: Results of query identifying usage of analysis and debugging ...... 140

Figure 32: Original references to debugging .............................................. 140

Figure 33: Early introduction to debugging is an effective teaching technique 141

Figure 34: Responses containing the terms trace or dry-run ....................... 141

Figure 35: Responses containing the terms copy and/or paste .................... 142

Figure 36: Nested instructions are difficult for learners to follow ................ 144

Figure 37: (a) Introduction of debugging early and flowcharts after coding .... 145

Figure 38: (b) Introduction of debugging early and flowcharts after coding .... 146

Figure 39: Practice problem solving references ........................................ 149

Figure 40: Practice promotes ideas clicking in learners’ heads ...................... 149

Figure 41: External stakeholders drive pedagogy ........................................ 151

Figure 42: Drivers of pedagogy ................................................................. 152

Figure 43: Factors which may limit computational thinking skills acquisition . 153
Figure 44: Searching for data structure abstractions ........................................154

Figure 45: Results of searching for data structure or equivalents ..................155

Figure 46: (a) Teaching programming corresponds to Bloom’s cognitive domain156

Figure 47: (b) Teaching programming corresponds to Bloom’s cognitive domain157

Figure 48: (a) Subdivision of programming skills and computational thinking
skills across the cognitive domain..........................................................159

Figure 49: (b) Subdivision of programming skills and computational thinking
skills across the cognitive domain..........................................................160

Figure 50: Formal notations and difficulty of moving between them ..........162

Figure 51: Relationships between computational thinking terms ..............164

Figure 52: Text search for debugging terms ..............................................170

Figure 53: Characteristics of debugging....................................................171

Figure 54: Responses incorporating logical or algorithmic thinking ..........175

Figure 55: Relationship between three concepts ....................................177

Figure 56: Final relationship model ..........................................................180

Figure 57: Total number of respondents ..................................................181

Figure 58: Respondents' reported occupations.........................................182

Figure 59: Respondents' reported teaching level .....................................182

Figure 60: Total number of words per data collection instrument ..........183

Figure 61: Average number of words per response ..................................183

Figure 62: Average number of contributed NVivo™ nodes ......................184

Figure 63: A relationship model ...............................................................230
DECLARATION OF AUTHORSHIP

I, Cynthia Collins Selby, declare that this thesis and the work presented in it are my own and has been generated by me as the result of my own original research.

Title of thesis: How Can the Teaching of Programming Be Used to Enhance Computational Thinking Skills?

I confirm that:

1. This work was done wholly or mainly while in candidature for a research degree at this University;

2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;

3. Where I have consulted the published work of others, this is always clearly attributed;

4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;

5. I have acknowledged all main sources of help;

6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;

7. None of this work has been published before submission.

Signed: .................................................................................................................................

Date: .................................................................................................................................

xv
Acknowledgements

Firstly, I would like to acknowledge all the enthusiastic teachers, academics, and professionals who have been eager to share their experiences with me. If they could all come together in a single venue, it would no doubt have a profound effect on classroom practices and policy makers’ decisions.

Secondly, I would like to thank my supervisor for the hours of proofreading and for asking questions that made me think. Hopefully, at the end of 3 years, he stopped cringing at my Oxford commas.

Finally, thanks go to my long-suffering family who made endless cups of coffee, who put up with books and papers all over the house, and who seemed to appreciate family holiday accommodations requiring Internet access.
Definitions and Abbreviations

A-Level or GCE – A General Certificate of Education is a subject specific public examination usually taken after two years’ study at the age of 18.

Alice – A 3D programming suitable for the development of animations, storytelling, and games. Alice is suitable for pupils aged 12 years and upwards. It can be downloaded from http://www.alice.org/.

Bee-Bot – A floor robot, in the shape of a bee, used with young learners. It is programmed by a series of buttons on its back. Available from http://www.tts-group.co.uk/shops/tts/Products/PD1723538/Bee-Bot-Floor-Robot/

CBI – The Confederation of British Industry is a UK independent employers’ organisation, representing both public and private sector interests. Accessible http://www.cbi.org.uk/

DfE – Department for Education. Accessible http://www.education.gov.uk/

GCSE – A General Certificate of Secondary Education is a subject specific public examination usually taken after two years’ study at the age of 16, at the end of Key Stage 4.

Greenfoot – An integrated development environment based on the Java programming language. Greenfoot is suitable for learners aged 14 years and upwards. It can be downloaded from http://www.greenfoot.org.

ICT – Information and Communication Technology

Java – A platform independent object-oriented programming language. Its syntax is similar to the C-family of languages.

Key stages – Are used to describe the divisions between blocks of years in which learners are educated. Key Stage 1 learners are between the ages of 5 and 7; Key Stage 2 learners are between the ages of 7 and 11; Key Stage 3 learners are between the ages of 11 and 14; Key Stage 4 learners are between the ages of 14 and 16.
Logo – A multi-paradigm programming language. It is often used in education to control real moveable devices.

QSR NVivo™ - Software to support qualitative and mixed methods research. Accessible http://www.qsrinternational.com/products_nvivo.aspx

Scratch – A visual programming language suitable for pupils aged 8 to 12. Scratch can be downloaded from http://scratch.mit.edu/.
Chapter 1.  Introduction

1.1 Background

Shortages in science, technology, engineering, and maths (STEM) skills are currently widespread in the work force. Forty-two percent of employers responding to the CBI’s education and skills survey reported difficulty recruiting qualified staff. When asked to forecast the recruiting difficulty in the next three years, they predicted a rise to 45% (Confederation of British Industry 2012). The Royal Academy of Engineering report, “ICT for the UK’s Future” states “It is essential that a significant proportion of the 14-19 age group understands Computing concepts – programming, design, problem solving, usability, communications and hardware” (2009, p. 17). The Rt Hon Michael Gove, Secretary of State for Education (2012) opened BETT 2012, the learning with technology trade show, with a speech exhorting the teaching of programming. The Royal Society (2012) has indicated that computational thinking, the skills necessary for applying the tools of computer science to understanding the world around us, is actually changing the scientific disciplines themselves and the needs of those engaged in those disciplines. These external pressures are not new. Education policy is acknowledged by Dijkstra to be “… hardly influenced by scientific considerations derived from the topics taught, and almost entirely determined by extra-scientific circumstances such as the combined expectations of the students, their parents and their future employers …” (1988, p. 19). These pleas from industry, along with the new national curriculum (DfE 2013) and the report on vocational education (Wolf 2011), highlight the importance of providing opportunities for learners to acquire knowledge, understanding, and skills associated with programming and problem solving. This setting provides the context for an investigation into the relationship between the teaching of programming and its effect on the acquisition of computational thinking skills by learners.

Three different overlapping areas of research literature have influenced the choice of this research topic. These three areas are problem solving, computational thinking, and programming. While the relationship between
these three areas is explored in the literature review, a visual representation is presented, prior to that, in the conceptual framework.

Each area is directly identified in the contributions of influential organisations such as the CBI (2012), The Royal Academy of Engineering (2009), The Royal Society (2012), and the Department for Education (2013). There is a body of knowledge available in the research literature to address the teaching of programming, especially at the undergraduate level (Ma et al. 2011; Lister, Fidge, and Teague 2009; Jenkins 2002; McCracken et al. 2001). The results of this literature include identifying what makes programming difficult to learn, defining the characteristics of an effective programmer, and identifying what steps may be taken to help beginner programmers learn more effectively.

There is also published literature addressing the teaching of programming to young learners, those in primary and lower secondary school. The teaching of programming to very young learners has a long history dating back to Papert’s constructionism (Ackerman 2001). Pane, Ratanamahatana, and Myers (2001) and Seidman (1981) worked with 10 and 11 year old American pupils. However, there is a distinct lack of published research concerning the teaching of programming to secondary pupils and post-16 students, especially focusing on learners in the United Kingdom. Secondary schools in other countries provide the foundation for the work of Schulte and Bennedsen (2006) who worked with a small number of high school teachers from Denmark, Germany, and the USA and Sakhnini and Hazzan (2008), whose work was based in Israel. There is also a considerable body of knowledge available on the topic of teaching thinking, although in its more general guise of problem solving (Fitzgerald, Simon, and Thomas 2005 and Eckerdal and Berguland 2005). Because the term “computational thinking” is a relatively recent introduction (Wing 2006), there is less literature published specifically with this key word. However, much of the literature devoted to problem solving may well be applicable to this area, if a connection between general problem solving skills and computational thinking can be shown. Therefore, the results of the proposed study should help fill the gap between the teaching of programming and computational thinking, specifically in the context of the post-16 age group of the United Kingdom educational system.
This study assumes, in line with Isbell and colleagues (2010), that computational thinking skills are a requirement of the 21st century society, that these skills must be taught, and that the high school and post-16 phases of education is the next logical focus for this instruction. The concept of using programming as a tool to develop computational thinking skills owes its origin, in part, to the work of Denning (2009), Sakhnini and Hazzan (2008), Jenkins (2002), McCracken et al. (2001), and Pólya (1985).

The new research presented in this document contributes to the body of knowledge that may be used to inform the issue of effective teaching strategies for both programming and computational thinking. In the classroom, teachers may employ the results of this study to redesign their own practice to focus on the broader skills of computational thinking, rather than the quite specific skills of mastering a programming language. By identifying a classification of computational thinking skills, curricula could be designed to develop those skills across longer time spans, similar to the teaching of mathematics across twelve years. Identifying programming activities that support particular computational thinking skills could ensure that the full range of computational thinking skills is taught and learned. “Chapter 5 Discussion” presents an analysis of this new work and a reflection on the model derived from the data analysis. In addition, this research responds directly to Guzdial’s (2008) call for more research into how to teach computing in a way that enforces computational thinking.

1.2 Research questions

The overarching question that this research is designed to answer concerns how the teaching of programming might be used to enhance computational thinking skills. Along with the frameworks in the following section, this question forms the basis for beginning the study, undertaken using the grounded theory method.

During the data collection and analysis cycles, concepts and categories emerge which respond to an additional set of questions. These additional questions are assigned to subcategories of the main research question. These subdivisions are taxonomy and definition of computational thinking, pedagogy, and the
difficulties of learning. Taken together, responses to these questions, highlighted in the analysis of the data, provide support for the original research question.

The initial research question and supporting questions are listed below. Responses to individual questions are provided in the “Discussion” chapter.

- Initial research question
  - How can the teaching of programming be used to enhance computational thinking skills?

- Taxonomy and definition of computational thinking
  - Is there a taxonomy of computational thinking skills?
  - Is there a consonance in the terms used to define computational thinking?
  - What is the connection between problem solving, programming, and computational thinking?

- Pedagogy
  - What specific programming activities contribute to the development of computational thinking skills?
  - Can computational thinking be taught without teaching programming?
  - What are the implications of this work for the teaching, in schools, of programming and computational thinking skills in the current context of computer science education?

- Difficulties of learning
  - What beginning programming skills are most difficult for learners to master?
  - What is the role of debugging in learning to program?
  - What computational thinking skills are most difficult for learners to master?
  - What problem-solving skills are most difficult for learners to master?

What factors may limit the acquisition of computational thinking skills?
Chapter 2. Frameworks

2.1 Theoretical framework

The broad theoretical foundations for the proposed research are presented in this section. At the highest level, this research sits within the Computer Science Education strand. In addition, the position of this study in the broad field of education affords the opportunity to introduce additional education theory. Three such educational models, which form the foundations for the proposed research, are discussed in this section, Bloom’s Taxonomy: Cognitive Domain (Bloom 1956), Anderson’s revision of Bloom’s Taxonomy (Anderson et al. 2001), and the SOLO taxonomy (Biggs and Collis 1982). An additional model specifically addresses the digital domain, Bloom’s Digital Taxonomy (Churches 2009a).

2.2 Bloom’s Taxonomy

Bloom’s original objective was to create a taxonomy of education objectives, across 3 domains, the cognitive domain, the affective domain, and the psychomotor domain (Churches 2009a). The domain under discussion in this work is only the cognitive domain of Bloom’s Taxonomy. In his words, “It is intended to provide for classification of the goals of our educational system.” (Bloom 1956, p. 1). He proposed that a taxonomy should be constructed so that the order of the terms reflects the order of the terms in reality, where more complex behaviours are built upon simpler behaviours. The resulting taxonomy contains six major classes, with some classes subdivided.

1. Knowledge
   1. Specifics
   2. Dealing with specifics
   3. Universals and abstractions in a field

2. Comprehension
   1. Translation
   2. Interpretation
   3. Extrapolation
Chapter 2: Frameworks

3. Application
4. Analysis
   1. Elements
   2. Relationships
   3. Organisational principles
5. Synthesis
   1. Production of a unique communication
   2. Production of a plan or proposed set of operations
   3. Derivation of a set of abstract relations
6. Evaluation
   1. Judgements in terms of internal evidence
   2. Judgements in terms of external criteria

Bloom’s Taxonomy is presented in the diagram below. The Taxonomy may be used to design both learning objectives and assessment materials (Bloom 1956). The higher levels, from comprehension to synthesis, are usually identified as the more important objectives of education, because they are associated with the understanding and use of knowledge, rather than simple recall (Krathwohl 2002).

![Diagram of Bloom's Taxonomy]

Figure 1: Cognitive domain of Bloom's Taxonomy

Bloom’s Taxonomy is not without criticism. Biggs and Tang (2007) question the methods used to develop the Taxonomy, “The original Bloom taxonomy was not
based on research on student learning itself, as is SOLO, but on the judgments of educational administrators …” (p. 80). There is also an acknowledged focus on the relationship between the question and the level of the response it is designed to elicit (Hattie and Purdie 1998). However, there may be some difficulty in assigning assessment levels to questions without an effective understanding of how the material has been taught (Johnson and Fuller 2006). Recognition that Bloom’s Taxonomy levels do not correspond well to the ordering used when assessing practical subjects, such as programming, led Fuller et al. (2007) to suggest a refined 2-dimensional taxonomy just for that subject. Another criticism is actually expressed by Anderson (2005) when stating that “… mastery of each ‘lower’ category was a prerequisite for achieving mastery of the next ‘higher’ category.” This may be interpreted as restricting progression to higher levels without prior full mastery of all the sublevels of each lower category. On the other hand, this is viewed, by Lister (2000), as an acceptable restriction when applied to the task of writing computer programs. “Only after they have passed exams at these lower levels, should they then be thrown into writing complete programs.” (Lister 2000, p. 162). Fuller et al. (2007) suggest that the performance of a beginner in a task may be attributed to the analysis or synthesis levels, while performance in the same task may only evidence application for more advanced learners. From the materials cited here it may appear that Bloom’s Taxonomy is not a perfect model and may suffer from inconsistencies and misinterpretations during implementation.

Regardless of the criticism, Bloom’s Taxonomy is still relied on to inform classroom practice (Fitzgerald, Simon, and Thomas 2005; Whalley et al. 2006). The six separate categories can be evidenced by classes of behaviour that cross subject-matter boundaries and are not limited to a particular age group (Bloom 1956). This makes it an appropriate theory to include in an investigation in the domain of programming and computational thinking.

2.3 The revised Taxonomy

In 2001, Lorin Anderson, a student of Bloom, offered a revision of the original taxonomy, which considered the advances in educational research and
cognitive psychology (Anderson 2005). Several changes to the taxonomy were made at this time. The first change involved the replacement of the nouns, used to describe the categories, with verbs. It was felt that because learning objectives are usually presented in phrases involving a noun and a verb, the subject matter and the action to be performed with or to the knowledge, that the levels of the taxonomy should reflect the use of verbs (Anderson et al. 2001). The choice of terms, the verbs, is reflective of the way in which classroom practitioners talk about their work (Anderson 2005). This introduction of noun verb pairs affords an additional dimension to the revised Taxonomy (Krathwohl 2002). One dimension is the knowledge dimension and the second dimension is the cognitive process dimension. In order to better conform to the language used by teachers, the original strict hierarchal boundaries are allowed to overlap (Anderson et al. 2001). At the same time, the top two levels in the original taxonomy, synthesis and evaluation, were reversed, with create taking the most complex and abstract position at the top level (Anderson 2005). These changes are reflected in the following diagram.

![Figure 2: The revised Taxonomy](image)

The two dimensions of the revised Taxonomy may be presented as a table (Anderson et al. 2001). This is illustrated in the following diagram. The new knowledge dimension has been added to those found in Bloom’s original. The meta-cognitive knowledge dimension is a reflection of the importance of learners being able to identify and understand how they learn. It is possible to
map each learning objective into an intersected cell. The coverage of cells, categories of knowledge and categories of cognitive processes, indicates the extent to which complex levels of knowledge and cognitive processes are involved (Krathwohl 2002). Recall that the higher up the categories a learning objective is placed, the more value it is viewed to have because it represents an interaction of knowledge and the use of knowledge, not just recall.

<table>
<thead>
<tr>
<th>The Knowledge Dimension</th>
<th>The Cognitive Process Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Remember</td>
</tr>
<tr>
<td>Factual</td>
<td></td>
</tr>
<tr>
<td>Conceptual</td>
<td></td>
</tr>
<tr>
<td>Procedural</td>
<td></td>
</tr>
<tr>
<td>Meta-cognitive</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3: The revised Taxonomy table

From a practitioner’s perspective, the revised Taxonomy may also be subject to criticism.

The restructuring into two dimensions has the possibility of introducing a perceived level of complexity for practitioners. It is unclear if the added complexity of identifying an applicable knowledge dimension category outweighs the simplicity of Bloom’s original (Fuller et al. 2007). Proponents of the SOLO Taxonomy question the usefulness of the verbs used in the revised Taxonomy when specifying intended learning outcomes.

“Anderson and Krathwohl’s revision is an improvement, but even then under ‘understanding’ you can find ‘identify’, ‘discuss’ and ‘explain’, which represent three different SOLO levels. This is exactly why ‘understand’ and ‘comprehend’ are not helpful terms to use in writing ILOs.” (Biggs and Tang 2007, p. 80).
In assessing whether the revised Taxonomy is suitable for computer science, Thompson et al. (2008) report difficulty when attempting to categorise questions to the cognitive domain dimension. This difficulty was exacerbated when attempting to place questions into the more refined subcategories of the cognitive domain dimension. On encountering the same difficulty, Whalley et al. (2006) suggest that the context of computer programming may be the cause. They indicate that the exemplar material supplied in the revised Taxonomy is not easy to translate to the programming domain. From this evidence, it could be concluded that the modifications offered by the revised Taxonomy might not all be received positively.

Regardless of the criticism, the revised Taxonomy informs classroom practice (Thompson et al. 2008). In common with the original Bloom’s Taxonomy, the top dimension of the revised Taxonomy can be evidenced by classes of behaviour that cross subject-matter boundaries and are not limited to a particular age group. The addition of the Knowledge Domain, as a separate dimension, may facilitate the construction of learning objectives. All of these qualities make the revised Taxonomy an appropriate theory to include in an investigation in the domain of programming and computational thinking.

2.4 The SOLO taxonomy

The SOLO taxonomy, proposed by Biggs and Collis in 1982, asserts that learning becomes more complex as it progresses (Biggs n.d.). SOLO is a mnemonic for the Structure of the Observed Learning Outcome. This taxonomy is an attempt to assess the work of students in terms of quality not as a tick list of tasks. The dimension of complexity increases as more aspects of a task are grasped, integrated into a whole, and finally generalised to new situations. This is illustrated below.
In order to facilitate understanding of the SOLO diagram, the five levels can be further described as below (North East Wales Institute of Higher Education 2007):

- **Pre-structural**
  - Learner does not understand the point
  - Task not approached in an appropriate manner
  - May acquire pieces of unconnected information

- **Uni-structural**
  - Nominal understanding
  - One or few aspects of task identified and utilised
  - Obvious connections may be made
  - Overall significance is not grasped

- **Multi-structural**
  - Several aspects of task appear to be learned
  - Aspects are treated as separate, having no relationship

- **Relational**
  - Components are identified and integrated into a coherent whole
Chapter 2: Frameworks

- Each component contributes to overall meaning

- Extended Abstract
  - Components are integrated and reconceptualised to create an individual perspective
  - Generalisation or transfer beyond the initial task may occur

The purpose of using the SOLO taxonomy to develop intended learning outcomes is to convey the level of performance required by the learner. The more specific the terminology the more effective it will be for the learner. For example, Biggs and Tang (2007) suggest that the word “understand” is too vague. It may well need to be replaced with a more specific alternative, such as “select” or “present”. Using this approach, along with the learner, as a method for raising metacognition has had some success (Maddern 2012).

One criticism of Bloom’s Taxonomy is the association with levels of difficulty, as well as complexity (Hattie and Purdie 1998). This implies that questions requiring behaviour at one level should be answered correctly more often than questions requiring behaviour at the next higher level (Bloom 1956). In the SOLO taxonomy, questions designed for the lower levels may well elicit responses at the higher levels; the converse is also true (Hattie and Purdie 1998).

However, SOLO is also not without criticism. There is also some potential to misunderstand the level of cognition needed to produce the response that the learner may submit (Chick 1998). In other words, a great deal of cognitive effort at the extended abstract level could have been expended to derive the solution to a problem. After the derivation of the solution, it could be presented in a very simple manner that could be interpreted as only having made use of cognitive processes at the relational level. In this case, it is the documenting of the process that should be assessed not the specific problem solution. The grossness of the scale is identified by Chan et al. (2010), who suggest the addition of three sublevels, low, moderate, and high. This, they suggest, would be a more fair reflection of learners’ attainment. In common with Bloom’s Taxonomy and the revised Taxonomy, the SOLO Taxonomy attracts criticism during implementation.
Regardless of the criticism, the SOLO taxonomy is currently being successfully used in classroom practices (Maddern 2012). SOLO attempts to focus on the quality of learners’ outcomes and the levels of cognitive process required to achieve those outcomes. It also provides the possibility for the learner to provide a response at higher levels than those assumed by the design of the question. All of these qualities make the SOLO taxonomy an appropriate theory to include in an investigation in the domain of programming and computational thinking.

2.5 Bloom’s Digital Taxonomy

An attempt to update the revised Taxonomy to incorporate the ubiquitous use of digital technology is made by Churches (2009a). He has titled his revision, Bloom’s Digital Taxonomy. In this taxonomy, each level is assigned a set of associated gerunds, increasing in complexity going up the diagram. For example, hacking appears at the apply level, reverse engineering appears at the analyse level, blog commenting appears at the evaluate level, and video-casting appears at the create level. Churches’s work is focused on supporting educators in their classrooms. Indeed, some may recognise the description of Bloom’s Digital Taxonomy presented in the following diagram.
Chapter 2: Frameworks

Figure 5: The Digital Taxonomy (Churches 2009a)
While the mapping of the digital tasks to the different levels of the revised Taxonomy appears logical and Churches gives some justification for their placement on his website, the simple choice of individual tasks for each level may be deceiving. For example, hacking is justified as application by Churches since he views hacking “…in its simpler forms is applying a simple set of rules to achieve a goal or objective.” (Churches 2009b). However, hacking may also involve testing to see if the hack has worked (evaluating), planning the steps of the attack (creating), and comparing observed results with expected results (analysing). Therefore, when using Bloom’s Digital Taxonomy in the classroom, the practitioner may need to consider the possibility that subcomponents of tasks may be placed on different levels.

Regardless of this criticism, Bloom’s Digital Taxonomy may be used to facilitate learning in the digital classroom. It retains the simplicity of the Bloom’s Taxonomy by concentrating only on the cognitive dimension of the revised Taxonomy and updates it with the addition of verbs representing tasks associated with the digital age. These qualities make Bloom’s Digital Taxonomy an appropriate theory to include in an investigation in the domain of programming and computational thinking.

### 2.6 Conclusion

Although it is not the objective of this research to generate a new education theory, it is possible, at this point, to place the proposed research into a relationship with the theories presented in the previous section. This relationship is illustrated in the following diagram.
Figure 6: Theory relationships
In the previous section, the relationship between the revised Taxonomy and Bloom’s Taxonomy was described. The two models are founded on many of the same concepts and ideas. This is indicated by the intersection between the two, shown above. Bloom’s Digital Taxonomy incorporates the cognitive domain dimension and the non-hierarchal aspect of the revised Taxonomy. This is also illustrated by the intersection between the three, shown above. The SOLO taxonomy, as shown, is not described as specifically overlapping any of these three models. The research proposed here, while not an education theory, can be placed within the same diagram.

These particular theories form the foundations for the proposed research due to the following:

- Cognitive processes can be ordered into taxonomies, indicating increasing complexity (Bloom 1956; Anderson et al. 2001; Biggs, n.d.).
- The tasks associated with the digital world can be assigned to these levels (Churches, 2009a).
- Computational thinking is assumed to be a group of cognitive processes, associated with the digital world (National Research Council 2010, Guzdial 2008, Denning 2007, Wing 2006). Therefore, it may be possible to order this group of processes into a taxonomy.
- Successful programming tasks, also associated with the digital world, are assumed to require some cognitive processes (Fuller et al. 2007; Johnson and Fuller 2006, Lister 2000). It may be possible to order these separate tasks and cognitive processes into a hierarchy.
- Even though there is currently no agreed definition of computational thinking (National Research Council 2010, Guzdial 2011, Wing 2011, National Research Council 2011, Computing at School Working Group 2012), these works may afford structures against which possible definitions can be measured.
- Computational thinking and programming are evidenced by tasks which, to be successful, may need the full range of cognitive processes described in the SOLO taxonomy (Chan et al. 2010; Chick 1998). This would make SOLO suitable for assessing computational thinking and
programming, but not necessarily appropriate for contributing to a search for a definition of computational thinking.

The education theories anticipated to have the greatest influence on the outcome of this study are Bloom's Taxonomy: Cognitive Domain, the revised Taxonomy, the SOLO taxonomy, and Bloom's Digital Taxonomy, each of which has been discussed in this section. The proposed research aims to parallel the use of a taxonomy when attempting to define and describe computational thinking and the way in which learning to program can contribute to its development by learners.

2.7 Conceptual framework

Three areas feature in the conceptual framework described in this section. Broad problem solving skills are used in developing or implementing strategies to solve problems in any context or domain. These skills are often expressed as heuristics (Pólya 1985), appropriate and plausible approaches to a problem. Computational thinking comprises specialised mental skills (Wing 2011) whose application results in solutions directly translatable to a computing device. These skills are also applicable in any domain. Programming skills are the specific technical skills needed to produce specific solutions using a set of defined digital tools, often associated with a programming language (McCracken et al. 2001). Conceptually, as indicated below, computational thinking is a specialisation of the broader topic of problem solving. Programming represents a narrower focus on evidencing the use of computational thinking and problem solving skills. The relationships between these three areas will be explored in a following section.
The above framework can be further illustrated with presentation of the key factors, concepts, and variables that may be studied further in this research. Each is anticipated to have a relationship to or influence one of the components of this framework. It is not yet apparent what these relationships or influences may be.

An aspect of each of the items in the framework is to be investigated. Computational thinking is to be investigated by searching for a common interpretation of the term and as a set of cognitive processes. Problem solving, only as a process that may underpin computational thinking or may have aspects in common with computational thinking, is also to be considered. The pedagogy of programming is to be explored in an effort to identify strategies that may inform classroom practice.

The factors that may influence the pedagogy or content of programming as delivered in classrooms are to be examined. In addition, the factors that may make learning to program difficult (for learners) will be investigated. Although research exists in this area, the effect the difficulties have on the acquisition of computational thinking skills merits investigation.
Chapter 2: Frameworks

In this section, the broad concepts into which this study is placed have been presented. In the widest terms, these include problem solving, computational thinking, and the pedagogy of programming. Varied aspects of each concept will be examined in an effort to form a picture of the overall relationships.
Chapter 3. Literature review

3.1 Rationale for literature selection

Several different threads were followed to determine the choice of published literature to include in this review. The most influential journals in the area of computer science education were identified and examined. These include *Computer Science Education* (2007-2010), *Transactions on Computing Education* (2007-2010), and all on-line accessible issues of *Journal of Educational Computing Research*. The same approach was taken to identify and examine the most influential conferences relating to computer science education. Although proceedings for several years were individually examined, all previous year’s proceedings were searched using key words. These conferences include International Computing Education Research (ICER) (2005-2010), Association for Computing Machinery Special Interest Group on Computer Science Education (SIGCSE) (2010-2011), and Innovation and Technology in Computer Science Education (ITICSE) (2009-2010). Online repositories of e-theses were searched for applicable literature published since 2000. Several surveys of literature papers (Malmi et al. 2010 and Pears et al. 2007) were also used as sources to identify further reading. The use of educational databases, ERIC, CiteSeerx, and Google Scholar provided mechanisms for tracking citations forward and backward. A slightly modified thread was followed during the attempt to uncover a definition for computational thinking. This involved a restricted search to post-2006 literature containing the particular term, computational thinking. This literature search is further defined in a following section. Finally, individual references in some works were followed to illustrate particular points or to clarify understanding.

Some literature, seminal and influential, has been intentionally omitted from the selection presented here. This omission is justified due to the context of the work. For example, it is not the intention of this research to define or identify programming characteristics of languages or paradigms that are or are not appropriate for novice programmers. Therefore, there is a large body of literature concerning language design, imperative versus object-oriented
paradigms, compiled versus interpretive implementation, and graphical versus command line development environments that has been intentionally omitted.

3.2 Computational thinking

The term *computational thinking* was introduced to a wide audience of those with an interest in computing by Jeannette Wing’s (2006) article in the Communications of the ACM. Here she attempted to define computational thinking in terms accessible by the broad reader base and made a plea that every undergraduate student be given the opportunity to engage with a computational thinking course, regardless of their chosen field of study (Wing 2006). Of course, the introduction of a term rooted distinctly in the context of computer science did not go unchallenged. In his reply, Peter Denning argues that the term is neither unique to nor representative of the whole of computer science (Denning 2009). He questions whether a renaming exercise contributes to substance. Wing, Denning, and others have contributed to the understanding of the varied components of computational thinking. The following sections define the term more precisely, discuss whether computational thinking should be taught to every learner, explore which particular concepts should be taught, investigate how it might be taught in classrooms, and establish its connection to this research.

3.2.1 Defining computational thinking

One of the unanswered challenges presented by Wing (2006) in her use of the phrase computational thinking is the actual definition of the term. From the more recent literature (National Research Council 2010, Guzdial 2011, Wing 2011, National Research Council 2011, Computing at School Working Group 2012) it is evident that there is still confusion over an acceptable definition for the term. In order to address this discord, the following subsections will attempt to clarify the terminology used when discussing computational thinking from a computer science education perspective. The first subsection argues that a rigorous definition of computational thinking is actually required. The second subsection will address, at a high level, the confusion between interpretations of computing, computer science, and computational thinking. The third subsection
will discuss what computational thinking is not. The fourth subsection will define computational thinking terminology with which the reader may not be familiar. The final subsection will propose a developing definition of computational thinking based on the consistent use and interpretation of candidate terms found in the literature.

3.2.1.1 Is a definition required?

Some authors/papers/commentaries may assert that a precise definition of computational thinking is not required (Guzdial 2011, Hu 2011). However, the discussion presented in this section is driven by a perceived need to support professionals working in the field of computer science education and the developing curriculums. This need for definition is supported in the literature (Barr and Stephenson 2011, National Research Council 2011, National Research Council 2010).

Guzdial (2011) has suggested that a very broad definition is acceptable. Such acceptance could shift the focus away from what computational thinking is to how computational thinking should be taught and how evidence of its acquisition might be observed in learners. Hu (2011), supports this by recognising that teachers are confident that what they teach in computing does promote computational thinking, even though they may not know exactly how this mechanism works.

This same argument is expressed by some of those who design or influence the design of computer science curriculums. Several curriculums, while acknowledging the vagueness of a computational thinking definition, continue to include a focus on concepts and techniques from computer science (Computing at School Working Group 2012; Computer Science Teachers Association Task Force 2011; Bell, Andreae, and Lambert 2010; Brinda, Puhlmann, and Schulte 2009). In presenting these concepts and techniques, the curriculums include terminology often found in descriptions of computational thinking.

On the other hand, a rigorous and agreed definition might ensure that computational thinking in these new curriculums for the K-12 years will be more than, as Joyce Malyn-Smith argued, “… just a bunch of examples that are
placed into the curriculum at the discretion of individual teachers” (National Research Council 2011, p.33). Further, Jan Cuny suggests that once computational thinking is included in a curriculum, it requires assessment. Without agreement on a common definition of computational thinking, it will be difficult, if not impossible, to develop appropriate assessment tools that actually measure the ability to think computationally (National Research Council 2010).

The balance of argument is still in favour of searching for a robust definition of computational thinking. Although it may be possible, without a robust definition, to identify examples of the practice of computational thinking, the ability to measure computational thinking may be hampered by that same lack.

3.2.1.2 Computing, computer science, or computational thinking

At this point, there are three terms competing for clarification at a high level. These are computational thinking, computing, and computer science. Computational thinking involves strategic thinking skills that are common to many domains, such as the sciences and engineering. These specific skills will be discussed in a later section. Computing may be viewed as a distinct discipline incorporating these skills, regardless of domain. For example, computing is inclusive of the domains of information science and business systems. On the other hand, computer science is a specific discipline, composed of a body of knowledge and which is not distinctly defined by its use of computational thinking. In the context of this research, the distinction between computer science and computing will not be enforced. The terms will be used interchangeably to represent the study of fields such as information technology, information processing, software engineering, computer science, algorithm design, and artificial intelligence, all of which benefit from the use of computational thinking.

3.2.1.3 What computational thinking is not

The 2010 National Research Council’s report not only attempted to define computational thinking, but also went to some length to define what it is not. Specifically, they assert that it is not computer literacy, not programming, and not a focus on applications such as games or simulations (National Research
Council 2010). This perceived need for defining the antithesis may give credence to Denning’s concerns around computational thinking not being new, but only a new word for the skills always employed by computer science (Denning 2009). In the past, there was an accepted, but inappropriate, analogy between computer science and programming (Denning 2009). He goes on to argue that if the same type of analogy is manifested between computer science and computational thinking, then the discipline of computer science will once again suffer. In the search for a rigorous definition of computational thinking, acknowledging what computational thinking is not may be just as important as the resulting definition.

3.2.1.4 Basic terminology

The terms attributed to a definition of computational thinking are almost as numerous as those attempting to define it. However, several of these terms may not be familiar to the reader. In order to establish some common foundations in the use of the terminology associated with computational thinking, some of these terms will be further discussed in this subsection. These terms are abstraction, decomposition, pattern recognition, generalisation, automation, and visualisation. However, it should not be assumed that inclusion in this section implies that the term is appropriate to use in a definition of computational thinking.

Abstraction is defined as the ability to decide what details of a problem are important and what details can be ignored (Wing 2008). In computing, multiple layers of abstraction are often used to reduce the level of complexity of a problem or a representation.

Decomposition is defined as breaking a problem down into smaller, more easily solved, parts. This is not only a suggested component of computational thinking, but also of the classic problem solving techniques espoused by George Pólya (1985).

Being able to identify patterns in both data (Google 2011) and across problems (Pólya 1985) is, by some, offered in the definition of computational thinking. In his 2005 study, Muller found that undergraduates who recognised patterns in
problem solutions while programming games were able to recognise and transfer the solution patterns to science simulations. Pattern recognition may be revealed as a specific type of generalisation.

Generalisation is a powerful component of problem solving that may help define computational thinking. It describes the ability to express a problem solution in generic terms, which can be applied to different problems that share some of the same characteristics as the original. This definition fits Pólya’s description of analogy, the ability to solve a problem based on the known solution to a similar problem (1985).

Automation in its broadest sense means to remove the need for a human to execute repetitive tasks. Automation is usually achieved by some form of mechanisation. Creating automations allows problem solutions to be realised in an actual computational device, including human beings (Wing 2008). Although referred to as algorithm design (Google 2011), the idea of describing a strategy to solve a problem parallels the idea of automation.

Another proposed term is visualisation. This term can be interpreted in different ways. A visualisation is often experienced as a pictorial model of a real world situation which learners may interact with to produce knowledge. For example, a visualisation in a science class might be a graphical software program where learners adjust slider bars representing temperature to see the graphical representation of water change from solid, to liquid, to vapour. On the other hand, especially outside the domain of computer science, a visualisation may be interpreted as an internalisation of mental images necessary to achieve an objective or goal. At least one source (National Research Council 2010) suggests that being able to represent solutions to problems, in terms of visualisations or models may also be a component of computational thinking.

Several terms, anticipated to be unfamiliar to the reader, have been discussed in this subsection. These terms may or may not be appropriate to use in a definition of computational thinking. However, establishing a common definition or interpretation of these terms may facilitate the development of a proposed definition of computational thinking, which is discussed below.
3.2.1.5 A developing definition

In an attempt to develop a proposed definition for computational thinking that remains true to Wing’s original vision (2006), a smaller set of literature has been explored for inclusion in this subsection. Searches for documents containing the terms “Jeannette Wing” or “computational thinking” written in 2006 or later and a selection of curriculum design documents from Israel, Germany, New Zealand, India, England, and the USA have yielded a set of twenty-two distinct works that have influenced the discussion presented here.

The identified publications were read in chronological order to discern the development, over time, of the phrase computational thinking. Descriptions and suggested definitions of computational thinking were identified in each publication. The terminology, common across descriptions and definitions, was collected. Where interpretation allowed, similar terms were grouped together. The most frequently occurring individual terms and groups are presented below. From this basic collection of terms, a definition of computational thinking is formulated and proposed.

Justification for the inclusion or exclusion of terms is presented on a term-by-term basis. Justification is based on consistency of usage and consistency of interpretation across the literature. The resulting definition reflects much of the consensus found in the literature while removing the less well-defined terms.

3.2.1.5.1 A thought process

When introducing the term “computational thinking” Wing (2006) described it as a way that humans think about solving problems. It incorporates the set of mental tools used in computer science. These tools are used to transform a difficult problem into one that can be solved more easily. In adding his voice to Wing’s, calling for the explicit teaching of computational thinking, Guzdial (2008) refers to computational thinking as a way of thinking about computing.

Participants in the workshop on the scope and nature of computational thinking (National Research Council 2010), although not tasked with defining computational thinking, nevertheless agreed that it incorporates a range of mental tools and concepts from computer science. This idea is extended to
Chapter 3: Literature review

represent problems as information processes and solutions as algorithms (Denning 2007). Al Aho (Denning 2007) picks up the idea of problem transformation when he describes computational thinking as the thought processes in formulating problems and solutions that can be expressed as algorithms. These thought processes do have focus; frequently that focus is described as problem solving. Finally, Wing expresses these refinements by defining computational thinking 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” (Cuny, Snyder, Wing, 2010, cited in Wing 2011, p.20). Because of this consensus, a definition of computational thinking should include the concept of a thought process.

3.2.1.5.2 Abstraction

Although the idea of abstraction, hiding complexity, as being part of computational thinking is introduced by Wing in her original article (Wing 2006), it expands over the next few years. She amends the definition to include simultaneous consideration for multiple layers of abstraction and consideration for defining the interfaces between the layers (Wing 2007). Even Peter Denning (Ubiquity 2007) acknowledges that abstraction plays an important part in computing, including programming. However, he points out that the act of abstracting is not unique to computer science. The next year, Wing (2008) defines abstraction as the cornerstone of computational thinking. Several participants in the workshop on the scope and nature of computational thinking concur that computational thinking has a focus around the process of abstraction, creating them and defining the relationships between them (National Research Council 2010). More recently, in their report on workshops sponsored by the Computer Science Teachers Association (CSTA) and the International Society for Technology in Education (ISTE) to incorporate computational thinking into the K-12 curriculum, Barr and Stephenson (2011) also include the ability to abstract in a definition of computational thinking. The concept of abstraction is explored by L’Heureux et al. (2012) where it is one of six aspects of their information technology approach to computational thinking.
Because of this consensus, a definition of computational thinking should include the concept of abstraction.

3.2.1.5.3 Decomposition

Breaking problems down by functionality is identified by Wing (2006, 2007) as part of computational thinking. Decomposition is required when dealing with large problems, complex systems, or complex tasks. The participants in the first NRC workshop also identify the need for problem decomposition (National Research Council 2010). In the next workshop, focusing on pedagogy, participants extend this idea. Robert Tinker views the core of computational thinking as breaking down big problems (National Research Council 2011). Danny Edelson points out that the creation of solutions requires breaking problems down into chunks of particular functionality and sequencing the chunks (National Research Council 2011). Most recently, in refining his own definition of computational thinking, Guzdial (2012) includes the use of tools including abstraction and decomposition. In light of this consensus, a definition of computational thinking should include the concept of decomposition.

3.2.1.5.4 Thinking terms

Although the idea that computational thinking represents a cognitive process attracts consensus, there are suggestions that several specific types of thinking should also be included. These specific types of thinking are logical thinking, algorithmic thinking, engineering thinking, and mathematical thinking. This section explores the viability of incorporating these types of thinking into the definition of computational thinking.

The concept of logical thinking, although not specifically defined, occurs several times in the literature spanning these years. Albeit not perceived exactly as equivalent, terms to describe similar types of thinking are grouped into this category. These include mathematical thinking, engineering thinking, and heuristic thinking. In her original article, Wing (2006) indicates that computational thinking incorporates heuristic reasoning to devise a solution. In addition to abstraction and decomposition, Guzdial (2012) also includes heuristic reasoning as an appropriate tool to use when engaging in
computational thinking. Computational thinking is equivalent to the logical reasoning used by people (Henderson, Cortina, and Wing 2007). Logical reasoning is included by Iyer et al. (2010) in their model computer science curriculum in order to promote high-level thinking skills that are not necessarily subject specific. L’Heureux et al. (2012), in detailing an aspect of their information technology approach to computational thinking, define logical thinking as the ability to develop and test hypotheses.

Computational thinking also intersects with engineering because computer systems interact with the real world. However, computational thinkers can design and create virtual worlds, not limited by physical reality (Wing 2007). Although Wing (2007) states that computer science relies on mathematics as a foundation, Gerald Sussman (National Research Council 2010) affirms that mathematical thinking revolves around abstract structures while computational thinking revolves around abstract methodology. On Becher and Trowler’s (2001) knowledge and disciplinary grouping scale, computer science, based on its subject content, would be placed in the “hard-applied” domain, along with engineering. However, they further suggest that computer science is a result of fission from mathematics, “hard-pure”, and now has an autonomous existence (Becher and Trowler 2001). Computer science and computational thinking could be viewed as bringing science and engineering together. It could be viewed as a meta-science concerned with studying methods of thinking that are applicable to many different disciplines (National Research Council 2010). While the ability to think logically, mathematically, heuristically, and from an engineering perspective are certainly capabilities that a computational thinker may exhibit, references to these terms in this literature are not well expanded.

Although the term logical thinking, as described above, may not be suitable to include in a definition of computational thinking, the potentially analogous term, algorithmic thinking, requires further investigation. In her original article, Wing (2006) does not use the term algorithmic thinking, preferring the word heuristic instead. However, by 2011, she extends her definition of computational thinking to include algorithmic and parallel thinking (Wing 2011). David Moursund (National Research Council 2010) suggests that computational thinking is related to the idea of procedural thinking, as proposed by Seymour Papert in
Mindstorms. He defines a procedure as a step-by-step set of instructions that can be carried out by a device. The same theme is continued by Gerald Sussman (National Research Council 2010), who defines computational thinking as a way of devising explicit instructions for accomplishing tasks. Inclusion of algorithmic thinking in a curriculum for high schools appears prior to Wing’s contribution. In the Israeli computer science curriculum, Gal-Ezer et al. (1995) placed an emphasis on inclusion of the study of algorithmic processes. There appears to be a consensus that computational thinking incorporates aspects of algorithmic thinking. The term algorithm is interpreted as a step-by-step procedure for accomplishing tasks, not just in computer science, but in other disciplines. Because of its wide acceptance and appropriate definition, algorithmic thinking may be applicable for inclusion in a definition of computational thinking.

Not all of the types of thinking proposed for inclusion in the definition of computational thinking bring further refinement to the term. Tying a definition of computational thinking to other terms such as logically or heuristically, with their open-ended interpretation, or to specific disciplines such as mathematics or engineering may not help advance the development of K-12 curriculums and may not aid in the development of computational thinking assessment instruments. For these reasons, terms expressing the idea of logical thinking or equivalence may dilute a definition of computational thinking. On the other hand, algorithmic thinking is represented consistently in literature and its interpretation does not vary. Of all the potential terms associated with thinking, algorithmic thinking is the only candidate that may be suitable for inclusion in a definition for computational thinking.

3.2.1.5.5 Problem solving terms

The idea that computational thinking has some relationship to problem solving appears frequently in the cited literature. The specific terms problem solving, analysis, and generalisation are most frequently employed in discussions of general problem-solving skills. This section explores the interpretation of these terms and the viability of incorporating them into the definition of computational thinking.
Chapter 3: Literature review

Problem solving, in one form or another, appears frequently in the literature presented here. There is agreement for describing computational thinking as a problem-solving activity. However, the literature does not illuminate problem solving in detail. Wing (2006, 2008), of course, incorporates solving problems using computer science concepts in her definition of computational thinking. The broadness of the problem-solving skills employed in computational thinking, in opposition to specific technical skills, is pointed out by Larry Snyder (National Research Council 2010). A requirement for a computing device is introduced by Barr and Stephenson (2011), who state that the essence of computational thinking is solving problems in a way that can be implemented with a computer. Peter Henderson (National Research Council 2011) concisely describes computational thinking as a type of generalised problem solving with constraints. Problem solving is emphasised by Marcia Linn (National Research Council 2010) who includes in the qualities of a successful computational thinker, the ability to engage in sustained investigative processes to generate problem solutions. Although there appears to be a consensus that computational thinking is a type of problem solving, the term may not be sufficiently specific to define it. Due to the broadness of the term, problem solving may not be suitable for inclusion in a definition of computational thinking.

The term “analysis” is included by some commentators in the definition of computational thinking. Interestingly, the term appears in relation to both problems and solutions, as in analyse a problem and analyse a solution. Analyse, in the context of problems, fits the category of problem solving, as defined above. However, analyse, in the context of solutions, could be interpreted as the comparable term evaluate. In her initial article, Wing (2006) expresses the need for a computational thinker to make trade-offs, by evaluating the use of time and space, power and storage. This evaluation of algorithmic processes, including their power and limitations, is foreshadowed by Gal-Ezer et al. (1995). Application of the term to user interfaces is evidenced in the second objective of the New Zealand proposed curriculum, as part of designing programs (Bell, Andreae, and Lambert 2010). In their IT approach, L'Heureux et al. (2012) include the ability to evaluate processes, in terms of
efficiency and resource utilisation, and the ability to recognize and evaluate outcomes. Although the term “analyse” attracts some agreement for inclusion in a definition of computational thinking, descriptions of the term found in this literature imply an evaluative process. Therefore, because of interpretative consensus in the description, the term “evaluate” may be suitable for inclusion in a definition of computational thinking.

A specific term that appears sparingly in the literature definitions is generalisation. It is the ability to move from specific to broader applicability, for example, understanding how to draw a square by defining internal angles, then applying the same algorithm to produce an approximation of a circle. The ability to recognise parts of solutions that have been used in previous situations or that might be used in future situations is included by Kolodner in a definition of computational thinking (National Research Council 2011). These parts, or functional pieces, can be used to solve the current problem or combined in different ways to solve new problems (National Research Council 2011). The term generalisation, itself, is described in a proposed curriculum as recognising common patterns and by sharing common features (Computing at School Working Group 2012). The idea moves forward from decomposition, described above. Generalisation is the step of recognising how small pieces may be reused and reapplied to similar or unique problems. Although the exact term “generalisation” is used sparingly in the literature, the idea of recognising and reusing common parts of a solution is a candidate for inclusion in a definition of computational thinking.

Candidate terms examined in this section include problem solving, analysis, and generalisation. Problem solving is a broad term that, although used consistently throughout the literature, is not well defined. Analysis, used in the context of a problem, is also a broad term, often incorporating the ideas of abstraction and decomposition, as discussed above. Analysis, used in the context of a solution, is analogous to evaluation and is used consistently in the literature. Although the term generalisation is used infrequently in the literature, there are descriptions of analogous processes. Therefore, from this set of candidate terms, the ones used most consistently, with the least disparity of interpretation,
and which may be suitable for inclusion in a definition of computational thinking are evaluation and generalisation.

3.2.1.5.6 Computer science terms

The authors cited here concede that computational thinking has a deep relationship with computer science. Some suggest specific computer science terminology to be included in a definition of computer science. The specific terms include systems design, automation, and more general computer science concepts such as recursion and recovery through redundancy. This section explores the viability of incorporating these terms into the definition of computational thinking.

Systems design, although not mentioned frequently, is still used to describe computational thinking. Designing systems based on concepts used in computer science is mentioned by Wing (2006). Again, this inclusion is foreshadowed by Gal-Ezer et al. (1995) who incorporate the study of the design and implementation of computing systems in their curriculum. One of Peter Denning’s Great Principles of Computing includes a category based on the design and building of software systems (Denning 2007). He goes further in describing systems as one of the four core practices, in which computing professionals engage, along with programming, modelling, and innovating (Ubiquity 2007). The focus in each of these cases is systems design as a product oriented process. It is evidence of the ability to think computationally, not necessarily a definition of it. Therefore, the term systems design may not be suitable for inclusion in a definition of computational thinking.

Another term, popularised by Wing in defining computational thinking, is automation. She connects the term to that of abstraction when discussing the mechanisation of abstraction layers and the relationships between them (Wing 2007). Even Denning acknowledges that this is what happens when programming (Ubiquity 2007). Later, a stronger connection is made by Wing (2008) when defining computing as the “automation of our abstractions” (p. 3718). This introduces the need for a computational device to interpret the abstractions, the need for a computer to execute a program. The process or processes required in the creation of these automations may be candidates for
defining computational thinking. On the other hand, a program artefact, similar to system design as discussed above, is only evidence that computational thinking has taken place. Previously, a consensus was presented that emphasised the thought process aspect of computational thinking. Based on that consensus, automation, interpreted as a program artefact, may not be a useful addition to the definition of computational thinking.

Throughout the literature, terms closely related to the general content of computer science studies appear in descriptions of computational thinking. Wing (2007) herself introduces computer science concepts such as thinking recursively, interpreting code as data and data as code, type checking, prevention, detection, recovery through redundancy, damage containment, error correction, prefetching, and caching. Additional concepts such as parallel processing, testing, debugging, search strategies, algorithmic complexity, and pattern matching are also recognised (National Research Council 2010). Barr and Stephenson (2011) include the abilities to think iteratively and recursively. Closer reading reveals that not all of these concepts are unique to the field of computer science. For example, mathematicians think iteratively and engineers plan for recovery through redundancy. While each of these concepts may be mastered by computational thinkers, none of them uniquely defines or helps narrow a definition of computational thinking. Therefore, terms interpretable as computer science content may not be helpful in defining computational thinking.

Candidate terms examined in this section include systems design, automation, and more general computer science concepts such as recursion and recovery through redundancy. Systems design, resulting in a product, is evidence of the use of computational thinking skills, not a definition of it. Again, automation, as a product or program, evidences the use of computational thinking skills. Finally, those terms that are interpretable as computer science content do not bring focus to the definition of computational thinking. Therefore, none of the suggested candidate terms discussed in this section appears suitable to be included in a definition of computational thinking.
Chapter 3: Literature review

3.2.1.5.7 Imitation terms

Three additional terms, also used in discussions of computational thinking, are modelling, simulation, and visualisation. These terms appear frequently in the cited literature. This section explores the viability of including these terms in a definition of computational thinking.

Wing (2006) began by defining computational thinking as modelling the appropriate parts of a problem to facilitate a solution. Later, Brian Blake (National Research Council 2010) insists that the definition of computational thinking should include modelling and visualisations. Brinda, Puhlmann, and Schulte (2009) have identified, as one achievable curriculum standard, the processes involved in modelling data. On the other hand, Edward Fox and Janet Kolodner (National Research Council 2010) point out that it is the manipulation of abstractions (models, simulations, and visualisations) that contribute to the development of computational thinking skills. Observing the results of changing variable values, forming hypotheses, finding anomalies in data, and identifying invariants can all be achieved by interacting with models, simulations, and visualisations. The manipulation of these representations are agreed to enhance the development of computational thinking skills, but do not necessarily define it.

Although these tools are effective aids in developing computational thinking skills, they may not be suitable for inclusion in a definition of computational thinking.

A diverse group of terms proposed for inclusion in a definition of computational thinking has been presented in this section. Each of these terms has been employed in the literature in attempting to define and describe computational thinking. Support for the inclusion or exclusion of the term in the definition of computational thinking has been investigated and is based on the terms consistency of use and consistency of interpretation across the literature. The following section summarizes the arguments presented above and suggests a definition of computational thinking based on these arguments.
3.2.1.5.8 Proposed definition

The intent of this subsection is to shed new light on the discussions that attempt to develop a definition of computational thinking. The objectives for such a definition, as stated above, are: to define more narrowly, not more broadly; to bring an order to the criteria not necessarily to accommodate all viewpoints; to refine the definition to facilitate assessment; to retain the validity of work that has been done previously, such as the development of curriculums; to separate a definition from those activities that might promote acquisition of computational thinking skills; and to separate a definition from those artefacts and activities that evidence the use of computational thinking skills. Justification for inclusion or exclusion is based on consistency of usage and consistency of interpretation across the literature. The resulting definition reflects much of the consensus found in the literature while removing the less well-defined terms.

Table 1 summarises the justification for each prospective term's inclusion in or exclusion from a proposed definition of computational thinking.

<table>
<thead>
<tr>
<th>Term</th>
<th>Status</th>
<th>Justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>A thought process</td>
<td>Include</td>
<td>Consensus found in the literature</td>
</tr>
<tr>
<td>Abstraction</td>
<td>Include</td>
<td>Consensus found in the literature</td>
</tr>
<tr>
<td>Decomposition</td>
<td>Include</td>
<td>Consensus found in the literature</td>
</tr>
<tr>
<td>Logical thinking</td>
<td>Exclude</td>
<td>Broad term, not-well defined</td>
</tr>
<tr>
<td>Algorithmic thinking</td>
<td>Include</td>
<td>Well-defined across multiple disciplines</td>
</tr>
<tr>
<td>Problem solving</td>
<td>Exclude</td>
<td>Broad term, evidences the use of skills; develops acquisition of skills</td>
</tr>
<tr>
<td>Evaluation</td>
<td>Include</td>
<td>Well-defined across multiple disciplines</td>
</tr>
<tr>
<td>Generalisation</td>
<td>Include</td>
<td>Well-defined concept, although the term may not be familiar</td>
</tr>
<tr>
<td>Systems design</td>
<td>Exclude</td>
<td>Evidences the use of skills</td>
</tr>
</tbody>
</table>
Chapter 3: Literature review

<table>
<thead>
<tr>
<th>Automation</th>
<th>Exclude</th>
<th>Evidences the use of skills</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer science content</td>
<td>Exclude</td>
<td>Evidences the use of skills</td>
</tr>
<tr>
<td>Modelling, simulation, and visualisation</td>
<td>Exclude</td>
<td>Evidences the use of skills in their creation; manipulation develops acquisition of skills</td>
</tr>
</tbody>
</table>

Table 1: Computational thinking definition terminology

As supported by the preceding arguments, computational thinking is an activity, often product oriented, associated with, but not limited to, problem solving. It is a cognitive or thought process that reflects

- the ability to think in abstractions,
- the ability to think in terms of decomposition,
- the ability to think algorithmically,
- the ability to think in terms of evaluations, and
- the ability to think in generalisations.

This proposed definition attempts to incorporate only those terms for which there is a consensus in the literature or those terms that are well defined across disciplines. The intent is to focus on the thinking aspect of the original phrase.

In other words, computational thinking is a focused approach to problem solving, incorporating thought processes that utilise abstraction, decomposition, algorithms, evaluation, and generalisations.

### 3.2.2 Computational thinking for all

There is some consensus that computational thinking skills are a requirement for understanding the 21st century society and that the skills should be taught. Patterns exist in data, whether it is credit card purchases, shopping baskets, or genomes. Clever thinking could replace the need for more powerful hardware in applications like data mining, where large amounts of data are analysed for
trends or patterns (Wing 2008). Bundy (2007) has identified that computational thinking has the ability to bring new powers of investigation to fields from physics, biology and medicine to philosophy, architecture, and education. The National Research Council (2010) has identified computational thinking as “… a fundamental analytical skill that everyone, not just computer scientists, can use to help solve problems, design systems, and understand human behaviour.” (p. vii). This is reflective of Wing’s (2008) original idea of computational thinking being a fundamental skill that everyone must know to be able to function in a 21st century society. If members of a society require a skill to participate in that society, then there must be some obligation to teach that skill. In a more recent work, Wing (2011) assumes that institutions providing graduate and undergraduate education are already beginning to incorporate computational thinking into their curriculum. She goes on to suggest that the teaching of computational skills be addressed at the elementary and high school phases of education. Isbell and colleagues (2010) identify the secondary and post-secondary phases of education as targets for the teaching of computational thinking skills. This research also assumes that computational thinking skills are a requirement of the 21st century society, that these skills must be taught, and that the post-16 phase of education is the next logical focus for this instruction.

3.2.3 Concepts to be taught

While the components of computational thinking can be defined and justification for the teaching of computational thinking can be given, a list of the concepts that should be taught is more elusive. On the other hand, there is some agreement on what should not be taught. The Computer Science Teachers Association (2011) has made some progress on this front. They have attempted to identify a model for how computational thinking concepts and skills can be taught across various subject areas. However, the model’s computer science track relies heavily on learning to program and learning how computers accomplish tasks. The math and science tracks rely heavily on expressing understanding by manipulating and interacting with visualisations. This reliance is not unique to the CSTA model or computational thinking. While there is no
argument that the activities in the model may well contribute to the development of computational thinking skills, it would be improved by the inclusion of more explicit links indicating this development. Yadav et al. (2011) identified five important concepts of computational thinking that were taught to K-12 trainee teachers in non-computer science domains. They identified the concepts of problem identification and decomposition, abstraction, logical thinking, algorithms, and debugging, which they felt could be exemplified in any subject domain either with or without computing devices (Yadav et al. 2011). The results of the study indicated a better understanding of computational thinking as a human activity rather than the use of computers to solve problems. Plans to incorporate more kinaesthetic activities for teachers to use in their classroom, without identifying the connection between the activities and the thinking, could well lead to collections of classroom resources with no coherent logic to their presentation. While a list of concepts to teach is elusive, there does appear to be some agreement on what not to teach. Both Wing (2008) and Denning (2009), often on opposite sides of an argument, agree that a tool should not be the focus of the learning and teaching. Computers are tools. Programming languages are tools. Kinaesthetic activities are tools. No tool should be the focus of the learning. Any or all of these tools, however, may well be employed, but are not the focus of the teaching of computational thinking skills. In response to the growing number of questions concerning what concepts and skills should be taught, Mark Guzdial (2008) makes a plea for researchers in the field of computing and researchers in the field of education to work together for their resolution. The separation of tool and computational thinking is supported in the context of this research. The objective of teaching is acquisition of computational thinking skills; the tool employed is programming.

3.2.4 How to teach computational thinking

As with the identification of specific concepts to teach, there is no real consensus on how to teach computational thinking skills or whether programming should be incorporated into this teaching. However, there is an identifiable focus on the use of kinaesthetic activities to foster computational thinking skills and engagement in the classroom. Motivational learning takes
place best in hobby-mode according to Curzon et al. (2009). Learning should be fun. The fun in learning, for some, can be experienced with kinaesthetic activities. The activities can make connections between computing and other subjects, such as science or art (Curzon et al. 2009). Further evidence that computational thinking skills can be taught across subjects such as maths, science, social sciences, and languages has been provided by the CSTA (2011) and Lu and Fletcher (2009). The previously mentioned study (Yadav et al. 2011) demonstrated that it is possible to show trainee teachers how to teach computer science concepts with such activities, which they can reproduce in their classrooms. Again, Guzdial (2008) identifies the need for more research into how to teach computer science in a way that provides computational thinking skills. One issue that must be addressed here concerns whether or not to teach programming as part of computational thinking. Certainly, if computational thinking is part of a computer science course, then it seems appropriate to teach programming. Lu and Fletcher (2009) go even further asserting that “… programming is to Computer Science what proof construction is to mathematics, and what literary analysis is to English.” (p. 260). How might this affect those students not studying computer science or those pupils in secondary education who are not exposed to computer science? Eric Roberts, Stanford University, asserts that programming must be taught or it loses its importance (Curzon et al. 2009). He also suggests that the teaching of computational thinking without programming could change the perception of programming from creative and challenging to mundane. While it is laudable to introduce fun and engagement into learning by the use of kinaesthetic activities, the ability of these activities to impart, either implicitly or explicitly, computational thinking skills is yet to be determined. Computational thinking is a human process taught to humans rather than the teaching of how computer processes work. Care should be taken that classroom activities do not degrade to the latter. This research responds directly to Guzdial’s (2008) call for more research into how to teach computer science in a way that enforces computational thinking skills. It also aligns itself with Eric Roberts’ (Curzon et al. 2009) view of programming as a necessary part of computational thinking, although its importance is viewed as a tool.
Chapter 3: Literature review

3.2.5 Connection to this research

There is some disagreement about whether computational thinking is new substance or a repackaging of existing techniques (Wing 2008, Denning 2009). Computational thinking shares strategies with science, maths, and engineering, but is not restricted to those areas (Wing 2008, Yadav et al. 2011). It is made up of several subcomponents including abstraction, decomposition, and generalisation. Algorithmic design, automation, and visualisations are representation of problem solving involving computational thinking skills. The literature presented here indicates that it is possible to teach computational thinking skills, including programming (Curzon et al. 2009), and to teach learners computational thinking skills across subjects (Yadav et al. 2011). The use of kinaesthetic activities shows some promise in the classroom (Curzon et al. 2009), but this use must be well grounded in computational thinking not just a demonstration of how computers or algorithms currently work. As indicated above the term computational thinking comprises specialised mental skills. The computational skill set is now developed, as indicated in the presented literature, to include the ability to think in abstractions, the ability to think in terms of decomposition, the ability to think algorithmically, the ability to think in terms of evaluations, and the ability to think in generalisations. Because this research is set within the study of computer science, programming will be incorporated as a tool. The teaching of programming will be developed further in a following section. From this viewpoint, the derivation of the term computational thinking and its semantic confusion in the domain of computer science is irrelevant. If the term generates interest in thinking skills and kindles teachers', learners', and researchers' imaginations, then it must be received positively.

3.3 Education theory and computer science

The definition above may imply that the way in which a person actually thinks about problem solving and solutions will need to be altered to suit the way in which the resulting solution can be implemented on a computational device. Therefore, students and pupils must “learn” to program; they must change the way they think about the construction of solutions. Viewing programming as an
Chapter 3: Literature review

Educational activity allows general education theories to be applied to students’ behaviours. Several research studies, presented in this section, tie their results to education theories previously presented. These include Bloom’s Taxonomy: Cognitive Domain, the revised Taxonomy, and the SOLO taxonomy.

First year undergraduate students’ thinking skills were investigated by Fitzgerald, Simon, and Thomas (2005). They employed a multiple-choice question instrument and a think aloud problem-solving instrument in an effort to determine how students read and understand code. Their results indicated that, overall, students did use strategies, but that no single one was dominant, that students used multiple strategies for each problem, that students used the same strategy in different ways thereby eliciting different results, and that students used good strategies in poor ways (Fitzgerald, Simon, and Thomas 2005). They mapped the students’ strategies to the different levels of the cognitive domain defined in Bloom’s Taxonomy. As might be expected, the strategies congregated around the comprehension level. However, there were strategies that mapped to all levels. At the highest level, evaluation, were placed those strategies indicating analysis for deeper meaning. This foreshadows the work of Lister, Fidge, and Teague (2009), which identified the explaining of code’s problem-solving purpose as different from understanding at a line-by-line level. The study relies on the students’ ability to articulate their own thinking and reasoning processes. There was, in addition, uncontrolled and unmonitored instructor participation in the think aloud observation process. It is also unclear why Bloom’s original taxonomy was chosen for the mapping in preference to the revised Taxonomy (Anderson et al. 2001).

Thompson et al. (2008) have attempted to provide an interpretation of the revised Taxonomy for computer science in their study with students in first-year programming courses. For each of the six levels of the cognitive domain, they both define the term in relation to computer science and give example assessment questions. They found it difficult to identify which level of the cognitive domain a question should be assigned to. It was more difficult to assign questions to subcategories of the cognitive domain. As a result, academics allocated levels differently to questions. “This was primarily due to difficulty mapping the cognitive tasks described by the taxonomy’s authors into
Chapter 3: Literature review

the programming domain, for which there are no examples.” (Thompson et al. 2008, p. 156). They conclude that an intimate knowledge of how a course is taught is needed in order to be able to assign a cognitive dimension to questions in the computer science domain (Thompson et al. 2008). This study is limited to the top level of the cognitive domain of the revised Taxonomy, excluding subcategories and consideration for the knowledge dimension. Adding provision for their inclusion would have added depth to this study.

Conducted during the time when there was great enthusiasm for the possibility that teaching children to program would actually teach thinking skills, transferable to other domains, the Hawkins and Hedberg (1986) study employed the SOLO taxonomy. The study concluded that SOLO was an effective system for classifying learner responses to problems set in the LOGO environment. As anticipated in the literature review, the researchers had to break down the domain specific tasks into behaviours that could be aligned with the levels of SOLO. These included, at the multistructural level, the ability to type in more than one command before executing, at the relational level, the ability to edit to remove mistakes, and at the extended abstract level, the ability to introduce variables (Hawkins and Hedberg 1986). They conclude that the use of SOLO has afforded the opportunity to maintain an exploratory learning experience while also being able to assess learner responses.

The applicability of using the SOLO taxonomy to assess the way in which programming students read code was tested by Lister et al. (2006) in their study of undergraduate students. They tested novice programmers only, using a “think out loud” technique. They were able to place students’ responses on an appropriate level of the SOLO taxonomy. They concluded that while teachers often focus on aspects of programming associated with the lower levels of the SOLO taxonomy, they should also offer opportunities for eliciting responses at the relational or higher levels. They suggest that this type of response is manifested by “… an ability to read several lines of code and integrate them into a coherent structure …” (Lister et al. 2006, p. 122).

In a study of reading and comprehension skills in novice programmers, Whalley et al. (2006) attempted to create assessment questions using both the revised
Taxonomy and the SOLO taxonomy. When considering the revised Taxonomy, in contrast to another study (Thompson et al. 2008), they did address the subcategories of the cognitive process of the revised Taxonomy. As reported by other researchers (Thompson et al. 2008), Whalley et al. (2006) also identify that "... once a question was written, it was sometimes difficult to formally place it within the revised taxonomy. (p. 3). Using SOLO, the highest level, extended abstract, was not achievable because the questions provided limited opportunities for extended answers. However, questions about summarising code were available for analysis and could yield responses at the next level down, the relational level. Results were as anticipated, with those exhibiting an understanding at the relational level having a deep understanding of the subject content and those with an understanding at the unistructural level appearing in the lowest quartile of subject knowledge.

The number of questions available to be analysed using SOLO was extremely limited. Additional questions designed specifically for analysis by SOLO would have provided additional information about the applicability of using this taxonomy in the domain of computer science. In this study, Whalley et al. (2006) have demonstrated that both the revised Taxonomy and SOLO are suitable assessment strategies in the computer science domain.

The applicability of the various works of education theorists to the teaching of programming has been explored in this section. Research has concluded that both Bloom’s Taxonomy: Cognitive Domain and the revised Taxonomy may be employed effectively when teaching and assessing beginner programmers. The same is true of the SOLO taxonomy, which attempts to analyse responses to assign levels rather than to design questions to elicit responses for different levels, a criticism of Bloom’s Taxonomy. The use of these theories as the basis for research in the computer science education domain was alluded to in the literature review and has been upheld by the studies presented in this section.

### 3.4 Pedagogy of programming

Because the objective of this research is to determine if and how the teaching of programming can be used as a tool to foster computing thinking skills, it is
necssary at this point to discuss the pedagogy of programming. The difficulty of teaching programming is recognised by Carter and Boyle (2002), who indicate that "there is a perennial problem in teaching this essential skill to people who find it genuinely confusing and difficult, whatever their intelligence." (p. 84). The topics, addressed in this section, include students’ own views of learning to program, identifying what makes programming difficult to learn, defining the characteristics of an effective programmer, and identifying what steps may be taken to help beginner programmers learn more effectively. The following sections discuss each of these topics in turn and demonstrate relationships to the conceptual framework.

3.4.1 Student views of learning to program

In an attempt to understand what first year undergraduates believed about learning to program, Eckerdal and Berglund (2005), conducted surveys with their students. These students had already begun their course and had experience of learning to program. They categorise the way students think about learning to program into five different categories, each subsumed by the next higher. The categories briefly define learning to program as (Eckerdal and Berglund 2005):

1. understanding a programming language and writing some code,
2. a difficult to explain way of thinking but is somehow tied to the programming language,
3. understanding existing computer programs,
4. a way of thinking which affords problem solving, a method of thinking,
5. a skill applicable outside the course.

They conclude that to be successful at programming, an effective programmer, it is necessary for the learner to acquire a category four understanding of what it means to program. At this level, the learner demonstrates an understanding of the connection between the general problem-solving thinking and its exemplification by the programming. Although these results were based on student surveys, required the students themselves to be very reflective of their own learning, and required students to demonstrate high levels of articulation,
the resulting categories do demonstrate an increase in thinking sophistication. This implies a possibility to map these increasing levels to one of the existing education theories. Inclusion of questions attempting to extract the triggers for students' changes in category could have resulted in additional interesting insight.

3.4.2 Difficulties of learning to program

Contributors to the difficulty of learning to program, identified in the presented literature, include programming languages themselves, a lack of understanding or misunderstanding of the model of the machine, individual topics and their perceived difficulty, and questionable approaches to problem solving. The next paragraphs address each of these contributors in more detail.

Although anecdotally the choice of programming language may be portrayed as contributing to the difficulty of learning to program, there is no evidence to indicate that currently one language is any better suited for beginners than another is. In a multi-university, international study, using several different languages, no significant difference between the students' problem-solving performances was found that could be attributed to the differences in the programming languages used (McCracken et al. 2001). Jenkins (2002) concludes that the language is unimportant in the scheme of things, because the objective of beginner courses is to learn programming not to learn a specific language. As far as interpretive versus compiled languages is concerned, du Boulay (1989) views the compiled languages as having a distinct disadvantage in that students are required to master many tools at once and to produce complete programs. Butler and Morgan (2007) consider language syntax a low level of conceptual difficulty, which can be overcome regardless of the language choice. However, they also acknowledge that syntax is often the focus of a beginner's course and results in high levels of feedback during the instructional phase (Butler and Morgan 2007). Low levels of concept difficulty could possibly be attributed to the large amount of feedback received; it's easy to explain syntax problems. These findings are based on the results of student surveys. Specifically, the students were asked to assign a level of difficulty of learning and a level of difficulty in implementing a set of programming concepts. There
is, however, no indication of whether respondents have truly mastered a concept, the level of mastery, or whether they were truly able to evidence use of the concept by implementation. The results of the study could be augmented by the addition of an empirical instrument measuring the ability to implement a concept. This could be used to indicate the strength of accuracy in the reported level of difficulty of learning that concept. Because there appears to be no clear evidence that the choice of programming language has a significant effect on a beginner’s ability to learn to program, this research will not focus on any single programming language. The research aligns with Jenkins’s (2002) view that the objective is learning to program, not learning a language.

There is support for the idea that learners, who have little or an inaccurate understanding of how a computing device actually executes a program, find learning to program particularly difficult (Ma et al. 2011; Milne and Rowe 2002). In particular, learners have difficulties when dealing with the fact that the effect of execution of a program is a reflection of the machine’s state. They do not understand and do not create programs that properly handle the fact that any instruction is executed in the state left by the last instruction (du Boulay 1989; Lahtinen, Ala-Mutka, and Järvinen 2005). Of course, it is the teacher’s job to correct these misunderstandings. Du Boulay (1989) suggests that enforcing the idea that there is a strict set of rules governing program execution and avoiding the use of anthropomorphic language should aid in helping learners form an accurate understanding of how the machine works. A significant move from inaccurate to accurate programming concept models was shown by Ma et al. (2011), when using a visualisation tool to introduce cognitive conflict and challenge learners with inappropriate models. This parallels the way in which a one-to-one session with an expert might work, where the expert observes and questions the learner, specifically in the instance when they evidence an inaccurate understanding, to guide their reasoning down a more accurate path. They reported that about half the students with non-viable models moved to a more viable model after using the visualisation tool along with the cognitive conflict technique (Ma et al. 2011). Without doubt, an inaccurate understanding of how a computer executes a program will lead the beginners to great difficulties in learning to program.
There also does appear to be some consistency in the literature with the identification of difficult topics and concepts. The literature also suggests some underlying contributors to this difficulty. Milne and Rowe (2002), in a study of object-oriented programming, found that the top six most difficult topics, as rated by students, involved the use of pointers and memory. Previous work by du Boulay (1989) also found specific issues with variables and assignments, both related to memory manipulation. Memory is also involved in arrays, another topic found to be challenging for learners. Du Boulay (1989) suggests that one of the contributing factors to this is that teachers draw one-dimensional arrays as horizontal when most languages treat the first subscript as vertical. He also found that the execution of loops could be interpreted by students as halting immediately after the first pass, not going back to the test (du Boulay 1989). Two studies (Lahtinen, Ala-Mutka, and Järvinen 2005; McCracken et al. 2001) indicate that both students and teachers identify the same topics as difficult. However, Milne and Rowe (2002) found that teachers rated topics of higher difficulty than students did. This may be justified when considering that students cannot always identify their own lack of knowledge. When focusing specifically on object-oriented programming, Schulte and Bennedsen (2006) found that the topics were seen more difficult by those not teaching them, while at the same time, were viewed as higher level by those who do teach them. Although their instrument was a web-based questionnaire and their sample selection was opportunistic, high schools and colleges were represented as well as universities. The six topics rated as most important to cover in an introductory class all involve programming, rather than high level concepts (Schulte and Bennedsen 2006). Because of the identified shortcomings when questioning learners, for instance the lack of reflective ability and the inability to recognize their own lack of knowledge, this research will focus on the teacher. This assumes that teachers have the depth of knowledge and reflective ability to provide insightful information.

There is some consensus that high-level concepts, abstract thinking, and problem solving are difficult for beginner programmers. Many of these difficulties can be characterised by an inability to join up pieces to form a total solution. The influential study by McCracken et al. (2001) aligned problem
solving in programming to the problem solving strategies of the mathematician, George Pólya, which include decomposition, sub-solution creation, and recombination.

The hardest concepts to understand are high-level, involving larger entities as opposed to individual details. Perhaps this is because students find it difficult to move away from a line-by-line interpretation of the programming process (Lahtinen, Ala-Mutka, and Järvinen 2005). Logical thinking is included as a high-level concept by Butler and Morgan (2007). They have pointed out the connection between the difficulty of topics and the amount of feedback they receive. Design is the most difficult for students and receives the least feedback; syntax is not so difficult but receives large amounts of feedback (Butler and Morgan 2007). They suggest that the emphasis needs to be reversed if students are to master more high-level concepts. In the opinion of Jenkins (2002), students demonstrate an inability to cope with multiple problem-solving issues at once and the precision necessary to instruct the computer to carry out the problem-solving algorithm. This inability is exemplified by learners who can read and interpret code, but that cannot write their own (Jenkins 2002). Sakhnini and Hazzan (2008) conducted one of the few research efforts with high school students using high-level problem solving concepts. They suggest that the students rely heavily on analogy and should be challenged with false analogies, that students should be taught abstract data type behaviours before implementing them, and that students should be exposed to many problems that can be solved using different strategies. Sakhnini and Hazzan (2008) tie their study of abstraction directly to general problem solving skills and strategies such as those advocated by Pólya (1985). This integration of programming, mathematics, and problem solving may lend credence to Denning’s (2009) argument that computational thinking is not a new concept at all. The concept of using programming as a tool to develop computational thinking skills, as described in this research, owes its origin, in part to the supporting work of Denning (2009), Sakhnini and Hazzan (2008), Jenkins (2002), McCracken et al. (2001), and Pólya (1985).

The close tie with mathematics, mentioned above (McCracken et al. 2001), has been observed by other researchers. A lack of exposure during mathematics of
entry level computer science students has been highlighted by Boyle, Carter, and Clark (2002), who lament that the UK A-Level in mathematics has been redesigned in such a way that those topics which are most important in understanding computer science have been removed or reduced in importance to accommodate topics not viewed as so directly applicable. Further discussions about the contribution mathematics may make to successful computer science students is expressed by Alexander et al. (2003) who, in their study of seven international universities, concede that it is not yet conclusive that a good performance in high school mathematics is an indicator for performance in computer science. On the other hand, they do identify that those students with a good performance in those high school areas most closely applicable to the study of computer science are likely to do well where the previous experience applies (Alexander et al. 2003). This is confirmed by Rountree et al. (2004), who agree that students who are successful at higher level mathematics, such as logic and discrete mathematics, may be more successful during the first year of their computer science course.

This section has described contributors to the difficulty of learning to program. These contributors include programming languages, misunderstanding the model of the machine, individual topics and their perceived difficulty, questionable approaches to problem solving, and dependence on levels of mathematics which many students lack. Although languages are often proposed as the focus of students' difficulties, there is no definitive research to indicate that one language is preferable to another. However, there does appear to be a critical mass of evidence to indicate that an inaccurate mental model of the machine can lead to great difficulties for the beginner. Consensus, between teachers and students, identifies the same topics as difficult to learn. The most common contributor to difficulties, as identified in the literature, is the inability of learners to develop and use problem-solving strategies. Problem solving, in the context of programming, will be explored in later sections.

3.4.3 Characteristics of an effective novice programmer

Although lack of problem solving skills (Robins, Rountree, and Rountree 2003) is often cited as causing difficulties for beginners, an alternative explanation
may be that novices simply do not grasp programming principles and tasks (Lister et al. 2004). Several studies have identified that there is a connection between mastery of the basic principles of reading programs, tracing programs, and writing programs (Lister, Fidge, and Teague 2009; Venables, Tan, and Lister 2009; and Lopez et al. 2008). This section describes, based on these differing views, the characteristics of an effective novice programmer.

As with any skill, some learners exhibit higher levels of capability than others do. Often, those with higher capabilities are referred to as experts while beginners, with less capability, are labelled as novices. Even absolute beginner programmers possess some skills, although they may be underdeveloped or incomplete. When teaching programming, the concern is not so much the difference between novice and expert as between the levels of capabilities of groups of novices. Robins, Rountree, and Rountree (2003) prefer the terminology of effective and ineffective programmer. The distinguishing characteristic, as they define it, is the learner’s pre-existing problem solving strategies (Robins, Rountree, and Rountree 2003). Although students may demonstrate some reasoning skills prior to learning to program, they are not always congruent with those possessed by more effective programmers. Chen et al. (2007) in their testing of entry level students, found that students could describe algorithms to sort positive numbers and dates, preferred post-test loops, broke problems down but not sufficiently to produce a correct algorithm, and did not evidence the use of data types or control structures. It is clear that Chen et al. (2007) have attempted to represent three basic sets of respondents, those having completed a programming course or equivalent, those non-computer science majors with no programming experience, and those computer science majors with no programming experience. However, it is unclear that this objective has been met. Twenty of the 118 responses attributed to those having no previous experience of programming may well have not met that criterion (Chen et al. 2007). The results, however, may go some way to enforce the idea that when problem solving for a computer, the effective programmer thinks in a way that the computer can implement, rather than deriving a human-based solution that must be modified before implementation in a machine. This enforces the idea that difficulties in programming are a reflection of inaccurate

Raymond Lister has been involved in trying to explain the relationship between reading, tracing, explaining, and writing code for many years, most recently in the research of Lopez et al. (2008), Venables, Tan, and Lister (2009), and Lister, Fidge, and Teague (2009). When investigating a multilevel hierarchy of programming, based on an analysis of exam papers, strong evidence revealed the association between tracing and writing, especially within the concepts of loops (Lopez et al. 2008). They also found that hierarchically, data and basics were the foundation, which influenced simple tracing and the understanding of sequences. Mastering these concepts influenced the ability to explain and the ability to write code (Lopez et al. 2008). This work was built upon in a further study in which Lister, Fidge, and Teague (2009) found that effective programmers had developed good tracing skills prior to good writing skills, that good students can explain the purpose of code without stating what it does line by line. This led them to conclude that writing good effective code requires both tracing and explaining skills (Lister, Fidge, and Teague 2009). In previous work involving the use of the SOLO taxonomy, Lister et al. (2006), concluded “… students who cannot read a short piece of code and describe it in relational terms are not intellectually well equipped to write similar code” (p. 122). In furthering the work toward establishing a hierarchy of programming, Venables, Tan, and Lister (2009) “… argue that that [sic] some minimal competence at tracing and explaining precedes some minimal competence at systematically writing code.” (p. 128). Regardless of the exact nature of the influences or the order in which the skills are obtained, it is apparent that the skills of reading, explaining, tracing, and writing code must be mastered before becoming, even at a minimal level, an effective programmer.

Although undeveloped problem-solving skills often cause difficulties for beginners (Robins, Rountree, and Rountree 2003), Chen et al. (2007) identified that beginners do possess problem-solving skills. However, these skills may not be refined to an extent sufficient to afford the development of computer-based algorithms. Other researchers, Lister, Fidge, and Teague (2009), Venables, Tan, and Lister (2009), and Lopez et al. (2008) assert that difficulties
are caused by the inability to grasp basic programming principles and tasks. At this point, the evidence supports that to be an effective programmer requires some capability in problem solving and mastery of basic programming concepts and tasks. In this research, the term effective programmer will be used to define those novice learners who have developed sufficient problem-solving skills and mastered sufficient basic programming concepts and tasks to evidence reading, tracing, explaining, and writing simple programs.

3.4.4 Strategies for teaching problem solving

Learners can be taught problem solving by making explicit the connections between different types of problems, by exposure to many and varied problems, and practice (Walker 2010). This theory can be extrapolated to the teaching of computational thinking skills, such as abstraction and generalisation. Learners can also be taught to evaluate their own levels of learning in terms of a defined taxonomy (Fitzgerald, Simon, and Thomas 2005), as described above. This will help learners identify the level of thinking required to solve problems. The outstanding issue with teaching thinking is the relationship, if any, between learning to program and learning transferable problem-solving skills. Unfortunately, some studies, Mayer, Dyck, and Vilberg (1986) and Seidman (1981) indicate that there is no evidence to support the concept of this relationship. However, with the new focus on computational thinking skills, new research will be encouraged which may overturn these findings.

Abstraction is one of the computational thinking skills identified previously. Specifically, in computing, this term has been defined as the ability to determine what elements are important and what elements may be disregarded, a way of hiding complexity. Abstraction is also tied to the other skill previously defined, generalisation. Generalisation is the ability to identify the common features of a set of problems. In a more general problem-solving sense, abstraction has been used to mean identifying the important features of a problem in one domain and recognising those same features in a problem in a new domain or context. This is akin to Pólya’s (1985) analogical reasoning, the identification of a problem with similar characteristics to one already solved. Muller’s (2005) study makes a tie to analogical reasoning. She also suggests that algorithmic
patterns are solutions to basic problems and can form part of a toolbox for developing larger algorithms. This is analogous to the idea of building blocks for use by weaker programmers, as proposed by Lui et al. (2004). One very specific suggestion that could be implemented easily in a classroom for any age of learner is the idea that algorithmic patterns be linked to the type of problem they are used to solve, not the programming construct used to solve them (Muller 2005). In this way, learners should master the ability to classify a problem and be able to choose building blocks to begin a solution. For example, instead of indicating that a problem can be solved using a “for loop”, indicate that the problem belongs to the class of problems where the exact number of required actions is known. Rather than just assuming the ability to change contexts is a by-product of learning to program, this approach makes the concept and the thinking behind the concept explicit. Kramer (2007) adds to this by suggesting that students need repeated exposure to problems that can be solved using abstraction. Disappointingly, for Kramer, no correlation was found between good abstraction ability and final grades in undergraduate computer sciences courses (Bennedsen and Caspersen 2008). They used a version of Piaget’s pendulum test in an attempt to identify higher-level reasoning as indicated when students isolated, controlled, and identified relationships between individual variables that control the swing. It may be argued that this is not a true test of abstraction capability. It is, however, a measurement of cognitive development stage and evidence of some reasoning method. The drawback comes in assuming that those students at the higher levels of cognitive development possess the skill of abstraction and that it is transferable to the computer science domain. A verifiable domain specific test of abstraction ability could have brought more confidence to or even changed the outcomes of this study. Abstraction capability can be evidenced by the successful completion of different problem solutions, which may involve the production or interpretation of a programming artefact.

Although there are claims of a relationship between learning to program and learning to solve problems in different domains, Mayer, Dyck, and Vilberg (1986) conclude that there is no convincing evidence for this. This lack of evidence was presaged by Seidman (1981) who found that only under very
specific conditions did instruction in Logo, to a group of 11 year olds, have any effect on cognitive ability. Specifically, this depended on the pupil’s interpretation of the negative case of “if p then q” being correctly interpreted as “if not p then not q”. In a more positive light, some students were found to understand word problems better after learning Basic (Mayer, Dyck, and Vilberg 1986). Regardless of Mayer, Dyck, and Vilberg’s (1986) results, this research will add to the body of knowledge concerning the relationship between learning to program and learning to solve problems using computational skills.

Another strategy showing some promise is the use of specialised lessons focused, not on computer science, but on learning about learning. Cukierman and Thompson (2009) designed a course for all undergraduates, but targeted to subject domains, including computer science. In the course, they introduced Bloom’s Taxonomy and explained its meaning. The real emphasis of these lessons is understanding the level of thinking that must be achieved to be successful in the class. The tools employed include an analysis of specific computer science questions to identify which levels are involved in answering the question (Cukierman and Thompson 2009). This idea, if employed appropriately in the classroom, could prepare learners for the metacognition required to inform large areas of research about learners’ problem solving skills. This approach may have some lower age limit. Further work here could apply the same learning about learning strategy to novice programmers in secondary schools in a first attempt to find this lower age limit.

Learners do possess the capacity to acquire any of the computational thinking skills, including abstraction as shown above. The difficulty arises in designing verifiable research instruments that measure the level of acquisition. Learners have the capacity to gain knowledge with which to make judgements about their own learning, thinking abilities, and strategies. Learning about learning should help pupils, students, and teachers engage in meaningful discourse about learners’ problem-solving abilities and strategies. Using these techniques will enrich the body of knowledge pertaining to problem solving, computational thinking, and programming.
Chapter 3: Literature review

3.4.5 Promoting effective learning of programming

While there is literature and some agreement to support the ideas that the lack of problem-solving skills and the inability to master basic programming concepts may be the biggest obstacles to becoming an effective programmer, there is no consistent view of strategies that can be used to overcome these obstacles. Techniques that work with one set of learners may well fail to have the same effect when used with another. Strategies that effect progress in one institution may not fulfil that same promise in another. There is a diverse body of literature to be found, especially in conference presentations, providing anecdotal evidence that specific techniques and strategies have had positive impacts in the classroom. A spectrum of strategies, ranging from high levels of support and scaffolding to independent learning, is presented below.

Jenkins (2002) discusses the part that learning style has to play in learning to program. He purports that both a surface learning style, where some facts may be memorised, and a deeper learning style, where some analysis occurs, must be employed by the learner simultaneously (Jenkins 2002). This is a different approach than learners may have encountered before. For example, in biology, learners memorise the parts of the eye then develop that into an understanding of how vision works. It is of course, the job of the teacher to provide enough support to ensure that the correct learning style is being applied at the appropriate time. Teaching by example or teacher modelling is advocated by Robins, Rountree, and Rountree (2003). They suggest the use of live development to show learners the effective use of problem solving strategies. This may be an aspirational objective for the secondary and post-16 classroom where many non-specialist teachers could encounter unforeseen problems that may not be solvable in the allotted class time. Both of these strategies, learning style based and teaching by example, rely heavily on the input and support of the teacher. Success with either may depend largely on the capability of the teacher.

Because learning to program is seen as such a difficult task, Lui et al. (2004) chose to focus their study on weaker students. They build their study on the idea that weak students are particularly susceptible to inaccurate concept
Chapter 3: Literature review

models. This is consistent with literature discussed above concerning machine
models (Ma et al. 2011; Milne and Rowe 2002; du Boulay 1989). They suggest
taking particular care to show real behaviours; to avoid the use of analogies
which lead to confusion; to begin with reading and tracing using paper scripts;
to provide many sample programs which may be edited to develop concepts;
and to encourage learners in the production of key program segments to be
used as building blocks which can be incorporated into larger programs (Lui et
al. 2004). A similar idea to the key program segments is expressed by Eckerdal
and Berglund (2005) when they advocate that students be explicitly trained to
recognise situations that can be solved by these pieces of key code.

All of these suggestions appear appropriate, not just for the weaker
programmer, but for any novice programmer. As with strategies based on
learning style and modelling, this approach requires a large amount of
scaffolding provided by a very capable teacher.

Berges and Hubwieser (2011) took the exact opposite tack in their project to
see just how much university freshmen could learn independently, without any
human help. Students took part in 2.5 days of pre-course work where they
were given precise worksheets explaining programming concepts, set a
problem commensurate with their previous exposure to programming, and were
asked to create a working solution, under the supervision of more experienced
students. Supervisors were not allowed to help the students with their work.
Fifty percent of the programs, submitted by the true beginners, compiled and
worked correctly (Berges and Hubwieser 2011). While this is impressive,
especially considering the object-oriented environment context, the entire focus
of the study is an artefact. There does not appear to be an objective
measurement of exactly how much or what was learned by the students. The
same result may be possible using any good online tutorial in a less intense
environment. The provision for measurement of new knowledge and a follow-
up of the participants to see how the pre-course affected their first programming
course performance would have added depth to the study.

The last word about teaching strategies belongs to Pears et al. (2007), who
based their extensive survey of literature on subtopics such as problem solving,
learning a language, and code production. “We conclude that despite the large volume of literature in this area, there is little systematic evidence to support any particular approach.” (Pears et al. 2007, p. 211). The research described here is an attempt to contribute to the body of knowledge, which may be used to address the issue of effective teaching strategies, not just for programming, but for computational thinking as well.

This section has discussed topics related to the pedagogy of programming. These topics include identifying what makes programming difficult to learn, defining the characteristics of an effective programmer, and identifying what steps may be taken to help beginner programmers learn more effectively. Contributors to the difficulty of learning to program include programming languages, misunderstanding of the model of the machine, individual topics and their perceived difficulty, and questionable approaches to problem solving. The most common contributor to these difficulties is the inability of learners to develop and use problem-solving strategies. Problem solving, in the programming domain, will be explored in the following section. There is also evidence that the ability to read, to trace, to explain, and to write programs have an effect on each other. As a result, to be an effective programmer, a learner must have developed some minimal capability in all these areas. Although there is no consensus as to appropriate teaching strategies to promote learning to program, there is a diverse collection of anecdotal evidence that some teaching strategies promote the learning of programming in some circumstances.

3.5 Teaching problem solving via programming

Because the objective of this research is to determine if and how the teaching of programming can be used to foster computing thinking skills, it is necessary at this point to bring together the previously discussed areas of computational thinking and programming pedagogy. This section addresses topics including the way in which novice programmers solve problems, the way in which novice programmers think, and strategies for teaching problem solving in the context of programming. The following section discusses each of these topics in turn and indicate its connection to the proposed research.
Chapter 3: Literature review

3.5.1 Novices solve problems with natural language

As indicated above, Chen et al. (2007) have found that beginner programmers do indeed possess some problem-solving skills, albeit these skills may not be congruent with the way in which a computer has to be instructed to solve the same problem. A distinct difference has been highlighted between the way novices solve problems, using natural language, and the way in which computers are instructed to solve problems using programming languages.

This disparity was highlighted by Miller (1981) when he asked non-programming undergraduates to write algorithms for solving a set of problems involving file manipulation. He found that the participants were most concerned with creation, access, and manipulation of data and they preferred manipulating entire structures to iteration. Participants were least concerned with flow control constructs, never expressed a bi-conditional (if/else) or a goto, but did use a single-sided true conditional (if then). Miller placed constraints on his participants, such as requiring at least five steps in any solution, limiting the number of characters per line of input to a maximum of 80, and enforcing modification of a line by complete retyping. Pane, Ratanamahatana, and Myers (2001) unjustly criticise Miller’s research for these constraints. They have disregarded the time in which the study was set. The limitation of 80 characters and the retyping of a line were enforced by the implementation tools of the study, a programming language called APL and one of the few input devices that could support that language, an IBM 2741 Selectric® typewriter terminal, based on older 80-column typewriter technology. The requirement for a minimum of five steps was an attempt to extract useful and meaningful data. Regardless of their criticism of Miller (1981), Pane, Ratanamahatana, and Myers (2001) affirmed some of his findings in their study of much younger learners, 10 to 11 years old. Like Miller (1981), they found that pupils used data in sets and subsets, that pupils rarely used looping constructs, although some did use “until”, and that pupils assumed that data structures behaved like lists, without consideration for bounds or memory usage. To these findings, they added minimal use of complex conditionals and “not”, the use of “then” for sequencing, and event-driven rules using “if” or “when”. In a more recent study of non-programmers, Simon et al. (2006) asked computer science students in
their first course and economics students to describe an algorithm for sorting numbers using English. Analysis of their algorithms revealed a preference for post-test loops (Simon et al. 2006). This corresponds to the use of the “until” found in the Pane, Ratanamahatana, and Myers (2001) study. Learners, both undergraduate and primary, employ similar strategies to solve problems using natural language. These strategies are not always afforded by the design of languages used to instruct computers. Recall that computational thinking results in solutions that can be translated to computing devices. As a result, learners are required to change the way they think about solving problems.

None of the studies presented here set out to determine if novices’ problem-solving skills were reflected in object-oriented languages. Indeed Miller’s (1981) study was prior to the introduction of object-oriented programming languages. That said, these studies, by omission and by not identifying any mappings that conform to the object-oriented paradigm, suggest that object-oriented thinking is not representative of the way in which novices describe problem solving in English.

Programming languages have not been designed to accommodate the natural problem-solving characteristics exhibited by novice learners and are often in direct opposition to them (Pane, Ratanamahatana, and Myers 2001). There is a mismatch between the way novice programmers think about problem solving and the way a solution must be expressed to a machine. In terms of this research, it is important to have some understanding of what resources and capabilities learners already possess. It is encouraging if learners possess some ability to express problem solutions, even if this expression is in natural language.

### 3.6 Conclusion

Both industry and government contribute to form the current research environment. Leaders of UK industry have identified an urgent need for employees with skills in the STEM subjects to contribute to their economic output. Government policy makers are responding with calls for universities, colleges, and schools to better prepare learners to fill these positions. The teaching of programming and computational thinking skills contributes to the
development of skills applicable, not only to STEM focused careers, but also to the ability to function and contribute to a 21st century society. The use of specific terminology evidenced in these published works leads directly to the conceptual framework, previously presented.

This framework provides the basis for exploring the separate concepts of problem solving, computational thinking, and the teaching of programming. In order to establish the meaning of computational thinking, published literature has been presented that defines the term, its individual components, the controversy surrounding it, and specifically how it is to be interpreted in the education environment of the UK. In order to inform the classroom practice of teaching programming, the factors that make programming difficult to learn have been discussed. The characteristics of an effective novice programmer have been identified. Examples of effective teaching strategies and methods have also been presented. In order to establish a description of the skills and capabilities that learners possess when embarking on learning to program, the literature has been presented that identifies the way in which learners inherently think about solving problems. Effective strategies and methods to promote problem-solving skills in the context of programming have also been presented. Each category in which literature has been presented has direct applicability to this study.

By reviewing and identifying the voids in the literature, justification for this study can be delineated. The results of this study fill the research gaps by helping to determine a definition of computational thinking, by exploring the relationship between the teaching of programming, the teaching of problem solving, and the teaching of computational thinking. The study makes original contributions to the body of knowledge that may be used to inform the issue of effective teaching strategies, not just for programming, but for computational thinking as well. This study responds directly to Guzdial’s (2008) call for more research into how to teach computer science in a way that enforces computational thinking.
Chapter 4. Method

4.1 Approach

In order to respond to the previously defined research questions, this study uses a grounded theory approach employing qualitative data collection methods and qualitative data analysis techniques. Cohen, Manion, and Morrison (2007) describe some of the characteristics of grounded theory. Their description includes the fact that categories and concepts do not have to be identified before data collection commences, but are allowed to emerge on the way. This approach provides a mechanism for incorporating new ideas from participant responses without being constrained by predefinition. It allows these new ideas and concepts to be explored, even when they fall outside expected responses. In this way, it is easier to focus on ideas and themes in the data. In addition, grounded theory studies can be well supported by the use of data collection through interviews (Cohen, Manion, and Morrison 2007). The different flavours of grounded theory and this researcher’s own interpretation of the method are discussed later in the “Grounded theory” section.

At the highest level, the approach taken here involves two main components. One of these is associated with data collection via instrument administration and the second is the analysis. There is a necessary amount of iterative processing required between the main components. This is to be expected in grounded theory and will be discussed further in a following section. The first activity is the administration of an Internet-based questionnaire instrument designed to identify a subset of respondents suitable for continuing in the research process. The second activity is the administration of a face-to-face, audio recorded, semi-structured interview schedule to respondents identified by an analysis of the questionnaire results. There will be a single interviewer for all respondents. The third activity is the collection of data from an Internet-based community of practice forum. The fourth activity is the analysis and theory generation. The questionnaire data and the community of practice forum data can be coded directly from the collected format. The recorded interview data requires transcription. All data is analysed in-line with Strauss and Corbin’s
(1998) grounded theory procedures and techniques, including open coding, axial coding, and selective coding until theoretical saturation. There is anticipated to be an iterative process whereby each activity may be revisited by the researcher, especially as emerging concepts require further explanation. Individual instrument’s design, reliability, validity, and sampling strategies are discussed in the “Sampling and participants” section.

4.2 Ethical issues
This section considers the ethical issues that have been addressed before embarking upon the implementation of the study. High-level ethical concerns are discussed below. Further particular issues of ethics, associated with individual instruments, are presented in the “Instruments” section. At this stage, the issues of paramount importance concern informed consent, anonymity, confidentiality, and data protection.

Where appropriate, the participants will be informed of the purpose of the research and can give their own consent. All participants are adults, aged 18 years or older. Participants engaging with the on-line questionnaire and the interview process will be informed in writing of the purpose of the research, who the research may help, and how the results of the research may be distributed. This information will be stated in text on the first page of the on-line questionnaire and in text on paper for interviewees. In this way, all participants will have sufficient information on which to base an informed decision concerning consent to participate.

Where appropriate, participants will be required to give consent for the use of their data. Participants in the on-line questionnaire will be asked to tick a box giving this consent. Those not giving consent will be denied entry to the questionnaire. Participants in interviews will be asked to sign a form giving consent for the use of their data. They will also be asked to sign a form, giving consent for the audio recording of the interview. Should a participant not give consent for the audio recording, then the researcher may only take field notes. Interviewees will be sent both consent forms and text describing the purpose of the study prior to the interview so that they may have time to read and understand them. Because the contributions from the community of practice
are in the public eye, the consent of the members will not be sought. In these ways, it can be assured that all data collected can actually be utilised in the study.

The data collected, either by the questionnaires, by the community of practice observations, or by the interviews, will be kept confidential. Some personal information will be collected, including, but not limited to name, institution, email address, and postcode. All personal and identifiable information will only be accessible by the researcher and the research team directly responsible for the conduct of the interviews or analysis of the data. Because of the use of face-to-face interviews and the proposed purposive sampling, anonymity cannot be guaranteed; the researcher will be able to associate responses with individual respondents. However, the researcher can assure participants that no names, institutions, or other text will be published which could lead to the identification of any individual respondent. While anonymity cannot be guaranteed, due to the nature of the data collection instruments, confidentiality can be assured.

Data protection is an issue of concern, especially as the contents may be used to identify individual participants. All electronic data will be kept on computers that are password protected. On-line questionnaire data will be kept on the university’s own computers. All data stored on physical media, i.e. audio tapes, paper responses, or field notes, will be kept in securely locked locations. All data, both electronic and physical, will be kept for no longer than needed. All data, both electronic and physical, will be destroyed after the publication of the results of the study. With these measures in place, the collected data should be secure.

In looking forward to identify potential instances in this particular research that may challenge the established ethical assurances, several concerns should be addressed in detail. These include the use of an Internet-based community of practice forum, the identification of learners or institutions in participants’ responses, the expression by participants of inappropriate views, and concerns for the safeguarding of children. Of particular interest with the community of practice are the concerns for privacy and informed consent. The Economic and Social Research Council (2010) suggests that Internet forums may be
considered in the public domain, if they are purposely open to the public. The Association of Internet Researchers (Ess 2002) indicates that the more public the nature of the forum, the less the concern for privacy and informed consent. While members of the community of interest in this research are required to register as members, the community itself consists of over 1200 individuals. No individual members of the community are being studied. No relationships between individuals in the community are being studied. The organisation, structure, and function of the community, as a whole, are not being studied. The "Instruments" section acknowledges the likelihood of changing the behaviour of the participants by seeking consent. In light of these facts, a decision has been made not to seek informed consent from the community of practice. Members of the community whose discussions are used as data are afforded the same level of assurance of confidentiality in processing and anonymity in publishing as the participants giving fully informed consent. Another area for concern is the possibility that a participant’s response may include inappropriate information. For example, a learner, colleague, or institution may be referred to by name; an inappropriate racist or sexist view may be expressed; or a pejorative comment about learners, colleagues, or institutions may be made. While unpleasant or inappropriate responses may be incorporated into the originally collected data, a decision has been made to disregard them for the purposes of coding. Any responses falling into this category will be consciously ignored by the researcher and will not be included in the dataset. Even though the participants in this research are all adults, the participants themselves may be involved in and report on interactions with younger learners. Although remote, this raises the possibility that a participant’s response may include revelations regarding issues of child safeguarding. If such an unfortunate event occurs, the researcher is obligated by professional duties to inform an appropriate authority. The ethics associated with the areas of child safeguarding, inappropriate content in responses, and the absence of informed consent are of particular interest in this research. Acknowledging and anticipating how these issues will be addressed during this study contributes to maintaining high ethical standards.
This section has addressed several issues of ethical concern, both at a high level and in areas of particular interest in this research. Methods for obtaining consent from the participants, for protecting the confidentiality of the participants, and for protecting the data have been presented. Several concerns of particular interest in this study have been addressed, including the use of a community of practice, inappropriate content in responses, and issues regarding child safeguarding. Consciously exploring the ethical issues associated with this study, understanding how they will be addressed, and adherence to standard university ethical procedures, should ensure that the research is conducted with the highest of ethical standards.

4.3 Grounded theory
As stated above, this research is based on the grounded theory approach. There are several different styles of grounded theory in use in social science research. Three of these styles are presented below. The discussion provides the basis for the researcher’s own choice of style that will govern the implementation of this research.

In its most basic form, grounded theory is an inductive approach whereby theory is developed from or “grounded in” the data. It does not follow other positivist approaches where data supports or denies a pre-existing theory (Cohen, Manion, and Morrison 2007). No theory is proposed before beginning the data collection and analysis processes. There are many styles of grounded theory, but the three most prominent are those advocated by Glaser and Strauss, Strauss and Corbin, and Charmaz (McCallin 2009). The original interpretation of grounded theory is attributed to Glaser (McCallin 2009), although it was the result of collaboration between Glaser and Strauss (Strauss and Corbin 1998). The work of Strauss and Corbin (1998) reflects an evolved grounded theory method, maintaining many features of the original. A more recent interpretation, constructivist grounded theory, has been advocated by Charmaz (Mills, Bonner, and Francis 2006). The three approaches do not necessarily sit well together as a group.

The original grounded theory, in the words of Glaser (2009), “…is just a simple, straightforward procedural method to induct theory from any type of data…”. 
Chapter 4: Method

While observations and interviews may well support grounded theory, Glaser (2009) goes on to include other data sources such as conversations, newspapers, books, videos, etc. Grounded theory is also identifiable by its supposition that the researcher is professionally naïve, that the researcher suspends his or her own beliefs, and that the researcher trusts in the emergence of concepts from the data (Christiansen 2008). While it may seem improbable that a researcher can enter into data collection without introducing bias, Glaser affirms that with constant comparison, multiple collections, and continuous conceptualisation, any bias is corrected and therefore the data may be used objectively (Glaser 2002). Theory will emerge directly from the data based on a core variable, whose existence is endorsed by Glaser (Christiansen 2008). Another characteristic in which grounded theory differs from other positivists approaches is in its consideration of literature. Much research begins with a review of appropriate literature. Grounded theory requires no such endeavour prior to data collection (Mills, Bonner, and Francis 2006). Further, highly focused reading may actually hinder the ability of the researcher to remain objective and view all data objectively (Cohen, Manion, and Morrison 2007). Grounded theory is not based on description but is based on categorical conceptualisation of the data. Constant comparison leads to the discovery of a core variable, from which theory can be derived. Glaser’s original grounded theory represents a way to move from the specific to the general, from data to theory. It provides the foundation for the expanded work of Strauss and Corbin, as discussed in the following section.

Strauss and Corbin (1998), while not departing from the philosophy of the original grounded theory, focused their attention on the use of structured processes and techniques for promoting the emergence of theory. They contend that “Theorizing is the act of constructing (we emphasize this verb as well) from data an explanatory scheme that systematically integrates various concepts through statements of relationships.” (Straus and Corbin 1998, p. 25, emphasis in original text). Their systematic approach includes open coding, axial coding, and selective coding until theoretical saturation. In this way, the researcher is directed in the use of strategies and analytical tools that may efficaciously lead to theory emergence. Along with this guidance, of course,
comes the possibility that the data may be forced to fit an emerging theory. Indeed, Urquhart (2007) criticises Strauss and Corbin’s guidelines because they can be interpreted as very prescriptive. When considering how the researcher’s previous knowledge or experience may influence the study, they advocate that the researcher acknowledge the influence, but use it to promote sensitivity to the data. This is not to be interpreted as implying that the researcher in any way is creating data. The influence of published literature is viewed in the same way. Literature in the study area can be used to enhance sensitivity to the collected data. However, the literature is not to be used as the data (Strauss and Corbin 1998). One aspect of this style of grounded theory is its flexibility. There is an open acceptance of mixing qualitative and quantitative methods, while recognising that there must be an interaction between them, one feeding back into the other. Strauss and Corbin’s grounded theory, while adhering to the ethos of Glaser’s original, provides a guided, but flexible, approach which researchers may find reassuring. While Strauss and Corbin allow consideration for restrictive researcher influence, the approach of Charmaz (2003), discussed below, embraces the researcher’s contribution.

Constructivist grounded theory is the approach espoused by Kathy Charmaz (2003). She advocates that consideration must be given to interpreting the participants’ realities (Charmaz 2003). This is in direct opposition to Glaser’s original idea of the supremacy of the data. Further, her constructivist approach leads to the possibility that the researcher may contribute to the construction of the data with the participant, especially to provide a context of time and place (Charmaz 2003). This is a far step from the detached researcher envisioned by Glaser. The desire to be faithful to the participants leads to the inclusion of raw data throughout analysis and presentation, even using individual quotations in the results presentation (Mills, Bonner, and Francis 2006). Glaser (2002) himself criticises Charmaz for her focus on storytelling at the expense of real conceptualisation. While she, in turn, criticises him for the assumption that the researcher does not have to evidence any concern for the quality, amount, or accuracy of the data (Bryant and Charmaz 2007). Constructivist grounded theory, by design, shows great concern for the individual contributions of
participants while allowing for a larger researcher influence than either of the previously described approaches.

With the three different approaches in mind, justification for the use of a specific method in the proposed research can be given. Glaser's original approach, while analytical and empirical, is very scientifically detached, allowing no provision for existing knowledge either in the form of literature or researcher. While identifying with the intent of this detached and uncontaminated approach, the ability to view data with a knowledgeable and experienced eye is viewed, by this researcher, as an advantage for conceptualisation. This researcher may also be hesitant to take Glaser’s assurance that a core variable will be uncovered which will lead to emergence of a theory. Less hesitancy would be exhibited if Glaser did provide more in-depth information about a toolbox of procedures that may help in the revelation of a core variable. Of course, it could be argued, that this is what Strauss and Corbin have attempted to achieve. Charmaz’s constructivist approach is founded on determining the underlying reasoning behind the participants’ behaviours and responses. Although this approach could be applied in this research, the main objective is not to determine reasoning, but to determine effect. In this research, it may be advantageous to understand why a strategy works, but the objective is not to understand why the participant chose that strategy. At the current time, the desire to narrate a story from the participants' perspectives is not viewed as contributing to the research questions. In line with Glaser, this researcher views the concepts derived from aggregation of the data to be more important than the individual contributions. Strauss and Corbin’s approach, although criticised for being prescriptive in some aspects, provides the flexibility to deviate from that prescription. By encouraging the mixing of methods and techniques, their grounded theory approach supports researcher creativity and freedom. Their focus on the data and procedures for encouraging the identification of concepts and promoting the emergence of a core variable will provide support and assurance for the researcher. Their allowance for external influences such as researcher knowledge and literature provide a flexible framework in which research can be set. As stated above, Charmaz’s constructivist grounded theory, with its emphasis on reflecting individual views is not appropriate for the
proposed research. A more appropriate choice is that of Strauss and Corbin’s (1998) grounded theory. It is true, in most respects, to Glaser’s original grounded theory, but provides structure and support in the form of procedures and techniques, which are appropriate for a new researcher.

4.4 Alternative research design

Many different approaches to research exist which could have formed the basis for responding to the original research questions. Previous sections have detailed the nature of the research, its design, and associated ethical concerns. In this section, alternative approaches to the research design are considered along with their appropriateness to the proposed research. These approaches include case studies, questionnaires, experimental, action research, ethnography, phenomenology, and phenomenography.

Case studies are suitable for identifying descriptive characteristics of what works or does not work. This approach would be suitable for identifying specific instances where the teaching of programming either positively or negatively influences the development of computational thinking skills. The usefulness of case study results would need to be determined by an individual practitioner based on the similarities between their own environment and that of the case studies. This approach is not appropriate for this study because the objective is to develop a conceptual theory of the relationship between the teaching of programming and computational thinking skills.

Questionnaires, written in a way that affords quantitative analysis, are especially useful for fact-finding (Bell 2005). Although it might be a simple matter to construct questions asking where, when, why, or what, the responses may not provide the depth of information required by this research. In addition, questionnaires still have the same issues with sampling as discussed in “Sampling and participants”, such as defining the entire population and then refining a subset for sampling. For these reasons, an approach based solely on questionnaires is not appropriate for this research.

An experimental style is suitable when effects of single variables are easily measured (Bell 2005). In the case of education, however, there is rarely an effect attributable to a single variable. In addition, the need for an experimental
Chapter 4: Method

group to be measured against a control group introduces issues of ethics concerning the different treatment of learners. An experimental approach would be suitable if there were large numbers of learners available, if the research was testing a single teaching strategy, and it was considered ethically acceptable to treat the two groups differently. None of these conditions applies in this study, which makes an experimental approach unsuitable.

Action research is suitable where some specific knowledge is sought for a specific problem in a specific situation (Bell 2005). This approach is especially appropriate in education where a researcher practitioner may need to understand how to improve his or her own teaching practice (McNiff and Whitehead 2006). This approach, however, is not appropriate for this study because the researcher, although a practitioner, teaching programming and computational thinking skills, aims for the research results to inform more than her own practice or institution.

Ethnography, as naturalistic enquiry, refers to the study of social and cultural phenomena (Cohen, Manion, and Morrison 2007). There is emphasis on the portrayal of these phenomena from an insider’s perspective (Fetterman 2008). The participants, whose views are of interest to this research, although forming a group with common characteristics, are not a social or cultural group whose interactions can be studied as they live their lives. In this instance, an ethnographic study is not suitable for this research.

Phenomenology, another naturalistic enquiry method, refers to the study of an individual’s perception of a direct experience, in other words, the meaning of the experience (Cohen, Manion, and Morrison 2007). This approach has its foundations in philosophy and emphasises an individual’s conscious awareness of the world (Gibbs 2010). Although understanding participants’ perceptions of their experiences could be enlightening, it would not lead to a formation of a theory that could inform teaching strategies. It is not appropriate for this research.

Phenomenography refers to the study of the ways in which people experience and perceive a phenomenon (Gibbs 2010). It is subjective, qualitative, and focuses on an individual’s own way of experiencing a phenomenon (Limberg
While this approach would be suited to describing different ways of experiencing or thinking about the learning of computational thinking skills, this approach is not suitable for answering the proposed research questions.

This section has considered several different approaches, which may have been utilised in the proposed research, including case studies, questionnaires, experimental, action research, ethnography, phenomenology, and phenomenography. Although each has merits, it is the decision of this researcher to decline their use in favour of grounded theory, as justified in the “Grounded theory” section.

4.5 Design reliability and validity

As previously indicated, the proposed research is set clearly in the qualitative genre. The choice of grounded theory supported by the use of qualitative data collection instruments reinforces this placement. The topics of reliability and validity, although easier to apply in quantitative research, nevertheless, must be addressed when using qualitative methods. In dealing with the question of reliability in qualitative research, the overriding idea is not to match the reliability achieved in quantitative research but to achieve a fitness for purpose. In other words, would different researchers studying the same participants generate the same findings (Prosser 2006)? Cohen, Manion, and Morrison describe reliability in qualitative research as including “… fidelity to real life, context- and situation-specificity, authenticity, comprehensiveness, detail, honesty, depth of response and meaningfulness to the respondents.” (2007, p. 149). In dealing with the question of validity in qualitative research, the overriding idea is to determine if it measures what it is purported to measure (Prosser 2006). In other words, do the instruments and the research as a whole appear to measure what they claim to measure? One way to achieve this goal is to minimise the amount of bias introduced by interviewer and instrument design. The results of each instrument could be checked against another known valid instrument. Much of the reliability and validity of this qualitative research will be a reflection of the reliability and validity of the individual data collection instruments, sampling strategies, and the rigour with which the chosen analysis is applied. The reliability and validity of individual data collection instruments and sampling strategies will be addressed in the “Instruments” section.
This researcher interprets the topics of reliability and validity in line with Morse et al., who state, “We argue that strategies for ensuring rigor must be built into the qualitative research process per se. ... These strategies, when used appropriately, force the researcher to correct both the direction of the analysis and the development of the study as necessary, thus ensuring reliability and validity of the completed project” (2002, p. 9). This researcher agrees that the reliability and validity of research must be continually maintained. It is not a discussion or decision that takes place either before the research begins or after the research ends. This researcher identifies with Morse et al. (2002) in recognising that reliability and validity must be addressed and ensured throughout the entire research process. This researcher is also cognisant of the fact that it is skill in applying verification mechanisms to ensure reliability and validity that will ultimately determine the overall reliability and validity of the research (Morse et al. 2002).

The five verification mechanisms for qualitative research identified by Morse et al. (2002) are methodological congruence, sample appropriateness, concurrent collection and analysis, thinking theoretically, and theory development. These relative measures of reliability and validity appear to conform to the philosophy of grounded theory. Although, the indicated mechanisms should fit any method, the following only addresses them in the context of grounded theory. The first, methodological congruence, attempts to verify the consistency between the design and execution of the research and the indicated research method. Is there a fit between the problem and the methods? Is the research planned and carried out in line with the chosen method? Does the type of data collected match that expected in grounded theory? Are appropriate grounded theory analysis techniques used? This mechanism can be satisfied by careful planning to meet grounded theory requirements and verifying that any deviation from that plan, which is acceptable in grounded theory implementation, is still in accordance with the spirit of the method. The second verification mechanism, sample appropriateness, attempts to verify that the sample selection is representative of both the problem and those with knowledge of the problem. Do the participants represent those with knowledge of the topic? Is there representation of all facets of the problem? The anticipated use of purposive
sampling provides the opportunity to identify those with appropriate knowledge and to identify dissenting cases. In grounded theory, this mechanism can be satisfied by the occurrence of theoretical saturation. If no new concepts or no new dissenting cases are emerging from the data, then it is assumed that the participants are responding in similar ways. This implies that the chosen sample is representative of both the problem and those with knowledge of the problem. The third verification mechanism, concurrent collection and analysis, attempts to maintain an updated view of research progress. In grounded theory, it is a requirement that there is coordination between data collection and data analysis. The next step in grounded theory should always be based on the current state of the data analysis. It is dependent upon the researcher to enforce this requirement by controlling the pace of the research, by resisting temptation to remain in a comfortable stage, and by recognising when it is time to move to another step. The fourth verification mechanism, thinking theoretically, attempts to address the need for confirming new ideas with both new and older data. Does the data support the conclusions? In grounded theory, it is important to verify continuously that the data fits the developing theory and that the theory fits the data previously collected. The researcher must ensure that no ideas slip into the research results without foundation. This mechanism can be satisfied by the continual rechecking of aggregated data and results. The fifth verification mechanism, theory development, attempts to verify a movement from the specificity of data to a conceptual understanding. Providing that rigour has been practised throughout the grounded theory research, Glaser (2002) assures that a theory will emerge. This statement is not arguing that by generating a theory alone, research can be considered reliable or valid. It is the process of maintaining rigour throughout the research process that leads to confidence in the development of theory. This mechanism can be satisfied with, not only a developed theory, but also a theory that is based on adherence to a rigorous methodological process. All five of these verification mechanisms, if applied with rigour by a conscientious researcher, contribute to the reliability and validity of the entire research project. This study should be judged in light of these mechanisms.
As part of justifying the reliability and validity of a project, it is inherent upon the researcher to anticipate threats to that reliability and validity. Specific threats to the proposed research fall into three categories: the researcher’s own performance, the issues associated with the chosen sampling method, and issues associated with the design and execution of the individual data collection instruments. Lack of responsiveness on the part of the researcher could compromise the reliability and validity of the project (Morse et al. 2002). The researcher may neglect or choose not to respond to indications that the design of the research or the design of the sampling techniques requires modification. They may choose to ignore indications that concepts or categories may need modification. These threats may be amplified when acknowledgement of them is in opposition to the researcher’s own beliefs. Although grounded theory research should not be time constrained, it is a fact of reality that only a finite amount of time can be devoted to data collection and analysis. A real threat, in this instance, is that the researcher may force the data to fit a preconceived theory, rather than wait for a theory to emerge from the data. Issues relating to sampling could also compromise the reliability and validity of the project. The size of the population to sample, in other words, the number of post-16 teachers of programming or computational thinking skills in the UK, is not known to this researcher. In light of this, it is not possible to determine a probability sample, representative of the wider population. Although this may raise further issues with generalising the results, a conscious decision has been made by this researcher to implement purposive sampling, which should lead to the selection of respondents with appropriate knowledge in the problem domain. It should also allow the researcher to select purposively for dissenting cases. Using a purposive sampling strategy may be a threat in itself, if the researcher is not able to control any tendency, even an unintentional one, toward choosing participants to force a theory from the data. Issues relating to the design and execution of the individual data collection instruments could also compromise the reliability and validity of the project. Individual instruments are discussed in the “Instruments” section. For now, in a broad sense, reliability and validity hinge upon how well the design of the instrument queries relate to the original research questions and whether the instrument queries provide scope for the depth of response required from the participants. In order to ensure reliability
and validity, it is necessary that all instruments be executed in the same way for each respondent. Without this assurance, there is a threat that the results will not be comparable between respondents. This section has identified several different categories of threat to the reliability and validity of this research project. Although threats are anticipated that are associated with the chosen sampling method and that are associated with the design and execution of the individual data collection instruments, the greatest threat to reliability and validity is the performance of the researcher. Specifically, for this researcher, the threats include acknowledging and controlling, as opposed to eliminating, researcher bias, maintaining sensitivity and responding to the need for design, sampling, concept, or category modifications, and maintaining faith that a theory will emerge in the allocated time without forcing the data to fit a preconceived theory.

The issues of reliability and validity in the design of this grounded theory study are addressed with reference to the five verification mechanisms defined by Morse et al. (2002). The mechanisms can be addressed by careful forward planning, monitoring, and justification of any deviation from the original plan, sample appropriateness as indicated by theoretical saturation, concurrent data collection and analysis, continuous reaffirmation of conformity of prior data, and rigor of process, which will lead to theory generation. Specifics of participant selection and the reliability and validity of individual instruments are discussed in a later section.

### 4.6 Sampling and participants

Although previous sections have revealed some information about the participants and the selected sampling method employed in this research, this section unites the information and provides further detail.

The participants in this research all have some interest in the teaching of programming, computational thinking, problem solving, or any combination of the three. This does not mean that all participants are teaching a programming class, a computing class, or a computer science class. Some participants may be teaching programming as part of a mathematics class. Some participants may be employed in industries where computational thinking skills and
programming skills are useful. Other participants may be members of professional communities of practice, representing industry, academia, or education. They are still perceived, by the researcher, as having an interest in and appropriate knowledge of the research context. Learners, students, and pupils are not direct participants in this research. As has been stated in the “Literature review”, the questioning of learners, of any age group, about their own learning is fraught with difficulties. These include a lack of reflective ability, the inability to recognize their own lack of knowledge, and their inability to articulate their own thinking. Focusing on the practitioners and teachers of these areas defined in the conceptual framework, however, does assume that they have the depth of knowledge and reflective ability to provide insightful information.

As stated above, the size of the population to sample, the number of people with an interest in or knowledge of problem solving, computational thinking, or the teaching of programming, is not known to this researcher. In light of this, it is not possible to determine a probability sample, representative of the wider population. Although bringing into question the ability of the results to be generalised to that population, this researcher has made a conscious decision to implement purposive sampling. This type of selection is biased toward selection of participants who meet some criteria. Cohen, Manion, and Morrison succinctly reason that, “There is little benefit in seeking a random sample when most of the random sample may be largely ignorant of particular issues and unable to comment on matters of interest to the researcher, in which case a purposive sample is vital” (2007, p. 115). In the case of this research, the sample will be selected purposively to consist of those who are perceived, by the researcher, to have some knowledge and interest in the teaching of computational thinking or programming. In the case of the first instrument, an online questionnaire, the targeted sample will consist of members of organisations, both national and local to the researcher, whose ideologies promote the teaching of programming or computational thinking skills. Of course, there is no compulsion to respond to the questionnaire, but it is anticipated that some number will respond. In addition to the online questionnaire, there is an opportunity to include conversational threads from a
community of practice online forum. Questions and responses on this forum are in the public eye. Not every thread will be applicable to the research questions. However, the forum will be monitored methodically, for the same duration of time as the online questionnaire remains open, for applicable threads. Once purposively chosen for their applicability to the research questions, the contents of these threads will contribute to the dataset. From the questionnaire responses and the community of practice conversations, a further purposive selection will be made to identify targets for face-to-face interviews, the second instrument. This selection will be made, by the researcher, on the perceived ability of the respondents to provide in-depth knowledge about the original research questions. This refinement is envisioned to take place in an iterative process, whereby, the requirement for new data means revisiting the questionnaire responses and the community of practice conversations to identify those with appropriate knowledge. Although this combination of volunteer and purposive sampling may jeopardise claims to generalizability, it does offer an opportunity for achieving comprehensive responses from knowledgeable participants. As Strauss and Corbin affirm, theoretical sampling is a foundation stone of grounded theory that, “… enables the researcher to choose those avenues of sampling that can bring about the greatest theoretical return” (1998, p. 202).

4.7 Instruments

The individual data collection instruments are presented in this section. Each instrument, the on-line questionnaire, the community of practice conversation threads, and the semi-structured interview schedule, are discussed in terms of justification, reliability, and validity, threats to reliability and validity, and design. Each instrument is also described in terms of the original research questions. The sampling strategies employed for each instrument have been discussed in the “Sampling and participants” section. Although alluded to here, they will not be discussed again in detail.

4.7.1 Basic question contexts

Although the grounded theory method does not require a hypothesis to test, bounding the context of the data collection should result in more applicable
responses and fewer non-applicable responses. To that end, the following set of questions is developed. These questions act as a guide for developing the data collection instruments.

1. How can the teaching of programming be used to enhance computational thinking skills?
2. What is the connection between problem solving, programming, and computational thinking?
3. Is there a taxonomy of computational thinking skills and activities?
4. What is the set of problem-solving and programming skills that underpin computational thinking?
5. Can computational thinking be taught without teaching programming?
6. What specific programming activities contribute to computational thinking skills?
7. Are there other contributors to computational thinking skills, regardless of discipline?
8. What are the implications of this work for the teaching, in schools, of programming and computational thinking skills in the current context of computer science education?

4.7.2 On-line questionnaire

An Internet based questionnaire is a way, theoretically, in which a large amount of data can be garnered from a diverse and potentially geographically disparate group of respondents. The on-line questionnaire shares many of the same characteristics and question design challenges of paper-based questionnaires.

On the other hand, simply because it is based online, it presents additional unique challenges (Cohen, Manion, and Morrison 2007). The discussion presented here addresses justification for this choice, some issues of design, issues of reliability and validity, and threats to reliability and validity.

Consideration is also given to the relationship between the questions in the on-line instrument and the previously defined research questions.

As indicated above, the participants of interest in this research are only those who are perceived to have an interest in and knowledge of the teaching of programming, problem solving, or computational thinking. The on-line
questionnaire, much like a paper questionnaire, can be targeted to this group. Although not all people meeting the criteria will be found, it is anticipated that some number will be. Most importantly, because participation is on a volunteer basis, responses may be assumed authentic (Cohen, Manion, and Morrison 2007). Those responding to such direct targeting are most likely to have some interest in and knowledge of the topic and should provide credible responses. The responses will be used as the basis for selection to participate in the interview process. These responses could also be used for theoretical sampling to identify conforming or dissenting cases. These responses could also be analysed to identify concepts or topics to explore further in the interview process. In addition, some keywords, concepts, or ideas may be revealed which could be useful in the data analysis phase. The researcher has chosen an on-line questionnaire as a data collection instrument because of its ease of access by the targeted group and because of the propensity for responses to be of the required depth for use in this research.

In common with paper-based questionnaires, an on-line questionnaire must be designed with some requirements in mind. These common issues include the use of open or closed questions, the ordering of the questions, and bias in the questions (Woollard 2006). In addition, some issues of presentation and guiding or controlling navigation are unique to on-line presentation. While the responses to closed questions may be easier to analyse, they may not provide depth. The questionnaire makes use of some closed questions but the majority of questions are open-ended to allow participants to respond as they wish. The ordering of the questions is from general to specific, divided into major sections. Results are submitted one screen or page at a time. In this way, even the results of abandoned questionnaires have the potential to be used. Personal information is requested early in the response process to identify participants. This provides a mechanism for following up a participant. In addition, each question is designed to elicit a response from the participant that can be used to answer one or more of the original research questions. Each question is constructed in such a way to control researcher bias and to avoid leading the participant (Woollard 2006). Cohen, Manion, and Morrison (2007) provide a comprehensive list of practical implications for the design of questionnaires.
Although they present an equality of implications, in the view of this researcher, the most outstanding is the requirement that the data generated will answer the original research questions. The design of the resulting questionnaire aims to address each of these issues while controlling for researcher bias.

When discussing the reliability and validity of questionnaires, there are two issues to determine. Firstly, to establish validity, the instrument must appear to measure what it is designed to measure and the conclusions should follow directly from the data collected by the instruments (Bell 2005). A well-designed, piloted questionnaire satisfies the appropriateness of measurement requirement. The occurrence of theoretical saturation, whereby categories are identified by the data, indicates that the conclusions are derived from the data. Secondly, to establish reliability of the questionnaire, the respondents’ responses must be accurate, honest, and correct (Cohen, Manion, and Morrison 2007). This can only be encouraged, but not guaranteed, by appropriate sampling and unambiguous question design. While anonymity may lead to more candid and reliable responses, it cannot be assured in this research. The incorporation of questions to act as crosschecks, whereby the same information is asked for in different ways, can also help ensure reliability of responses. The issues of reliability and validity, in questionnaires as with other qualitative instruments, are addressed in terms of perception and rigour of implementation, rather than by a statistical measure.

Threats to the reliability and validity of questionnaires include poor quality of response because of inadequate question design, conclusions not following from the collected data, no verification of either respondent or accuracy of their responses, and an uncontrollable sample. There is a threat that questions do not actually elicit responses that can be interpreted in the context of the research questions or that can be used to filter for appropriate participants for the interview process. Results not concurrent with the data generated by the questionnaire are another threat. These threats may be attributed to either poor quality of questionnaire design, application of an inappropriate sampling strategy, or inappropriate analysis. In addition, badly designed questions may not provide enough scope for respondents to elaborate, giving enough information from which to identify abstract concepts. Suggestions for mitigating
these threats point back to the requirement of Morse and her colleagues (2002) for rigour to be consciously and conscientiously maintained through each step in the research process. Additional threats include those arising from the inability to verify the accuracy of responses, from the inability to verify who the respondent really is, and from the limited statistical analysis that may result from a low response rate. While not being able to eliminate all threats to the reliability and validity of questionnaires, strict adherence to rules governing question design, justification of the designed questions against the original research objectives, and enforcement of strict sampling can go some way toward mitigating them.

Table 2 reproduces the questions found in the online questionnaire. The second column indicates the research question or questions tied to that questionnaire question. The questions themselves are open and not leading. The ability to provide free format text allows the respondent to supply any information they determine to be appropriate. There are some questions, included in the first section of the questionnaire, that generate demographic information or simple statistical data. The context column refers to the set of broad questions guiding the development of the questionnaire.
### Online questionnaire

<table>
<thead>
<tr>
<th>Section 1. Information about you</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>In order to invite your participation in the next phase of the research, please provide your email address. This information will be kept secure and is only available to the researcher and the project supervisor. You may, of course, at any time withdraw from this research by contacting the researcher at <a href="mailto:C.Selby@soton.ac.uk">C.Selby@soton.ac.uk</a></td>
<td>Info only</td>
</tr>
<tr>
<td>Which country do you live, work, or study in?</td>
<td>Demographic</td>
</tr>
<tr>
<td>Which county, state, or province do you live, work, or study in?</td>
<td>Demographic</td>
</tr>
<tr>
<td>Which of the following describes your areas of interest or ages you teach? (This is not your qualification level). Tick all that apply.</td>
<td>Demographic</td>
</tr>
<tr>
<td>Key Stage 1 and younger (Age up to 7), Key Stage 2 (Age 8 to 11)</td>
<td></td>
</tr>
<tr>
<td>Key Stage 3 (Age 12 to 14), Key Stage 4 (Age 15 to 16)</td>
<td></td>
</tr>
<tr>
<td>Post-16 (Age 17 to 18), Higher Education</td>
<td></td>
</tr>
<tr>
<td>Post Graduate Education, Professional Body</td>
<td></td>
</tr>
<tr>
<td>Qualification Awarding, Industry, Other</td>
<td></td>
</tr>
<tr>
<td>Question 1.4b Please provide any additional details here.</td>
<td>Demographic</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section 2. Programming</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Question 2.1 Which programming language(s) and/or environments do you use to teach or should be used to teach programming? Please elaborate on this choice.</td>
<td>6, 2</td>
</tr>
<tr>
<td>Question 2.2 Please describe the techniques you use when teaching programming? For example, do you use pseudocode tracing exercises?</td>
<td>6, 2, 1</td>
</tr>
<tr>
<td>Question 2.3 Are there other techniques that you might consider using to teach programming? Please elaborate.</td>
<td>6, 2</td>
</tr>
</tbody>
</table>
Table 2: Questionnaire mapped to question context

An on-line questionnaire has been chosen because it is easily accessible by the target group who are perceived to have some interest and knowledge of the research topics and it provides a mechanism for collecting in-depth information. Although its purpose in this study is to collect data for analysis, the questionnaire also serves to collect data on which purposive sampling decisions can be made to determine whom to interview in the next phase of the research.

Many of the reliability and validity issues posed by questionnaires can be mitigated by strict attention to details at the design stage and application of appropriate sampling strategies. The questionnaire has been designed with consideration for the original research questions. Each question has been shown to have relevance to the original research question. Obviously irrelevant information, if supplied by the respondent, will not be included in the dataset. An on-line questionnaire meets the requirements of the research for this phase.
4.7.3 Community of practice

Members of a community of practice are bound together by what they do together (Wenger 1998). They are informal and organic entities in which members engage in activities and learning based on a common interest. These groups are often based in the workplace, but this is not a requirement. The common interests, mutual engagement, and discussions result in learning for both new and old members. The discussion presented here addresses justification for the choice of a community of practice in this research, a description of how it will be used, issues of reliability and validity, and threats to reliability and validity. There is also consideration for how the member opinions contribute to the previously defined research questions.

According to Etienne Wenger (1998), a community of practice defines itself along three lines, its subject, its functionality, and its products. The members, themselves, decide and agree the subject or context on which the community is based. Communities of practice often evolve because of a common interest in and engagement with a subject. It functions by sharing information and experiences, which leads to learning from each other. Participating members develop both personally and professionally. A community’s products need not consist of tangible artefacts. The outcomes of the shared experiences of the community may be concepts, ideas, or techniques to influence practice. Communities of practice recognise no experts. Membership is gained by contributing and all are free to contribute in whatever way they consider appropriate. The community of practice, whose discussions and opinions are of interest in this study, is computer-mediated. Simply by contributing, the members signify some interest in topics that overlap with this study. However, some individuals share collaborative practices in the classroom. There are also face-to-face meetings, of varying scale, held throughout the year at both the national and local level.

An appropriate analogy to the use of web-based forums, such as this community of practice, in qualitative research is that of observation. When employing observation in research, the use of either overt or covert methods must be justified. While informed consent is appropriate for the online questionnaire, above, and the interview, below, in the case of a web-based
forum the act of seeking consent could influence the behaviour of the members’ interactions (Cohen, Manion, and Morrison 2007). By conducting covert observations the data collected will be uncontaminated by the presence of the researcher. Participants should not feel that their privacy has been invaded because the conceptualisation and categorisation of data required by grounded theory means that members’ exact words would not be used. Although email addresses will be part of the collected data, none will be coded as part of the dataset. To ensure further confidentiality, the name of the actual computer-mediated community of practice forum will not be published (Sixsmith and Murray 2001). These two efforts will, in no way, cause a loss in the quality of the data collected.

Although the community of practice discussion archives are available in their entirety, not every discussion, either historical or current, will be associated with a subject appropriate for this study. In order to identify the most appropriate threads for inclusion in the dataset, older archived discussions will be keyword searched. The keywords have been chosen to correspond to the terminology used in the literature review, such as computational thinking, abstraction, decomposition, algorithm, and problem solving. More recent additions, those appended conversations since the beginning of this study, have been read as they arrived. They are then purposively chosen for inclusion or discarded. If included, the discussions are coded and processed in accordance with grounded theory, in the same way as the questionnaire and interview data. Discussions are composed of individual and related messages, which should be considered as a whole (Sixsmith and Murray 2001). Regardless of the age of a discussion, once it has been identified as pertinent, every individual message in that discussion is read and coded.

Three issues relating to reliability and validity must be addressed when using observational data. The first is that of the participant observer. The participant observer could, even unintentionally, affect the behaviours of the members. Reciprocally, the observer could become so involved in the community of practice, that objectivity is affected. The second issue is that of observer bias. The usage of observational materials incorporates the act of inference (Cohen, Manion, and Morrison 2007). The community of practice discussions are not
Chapter 4: Method

initiated in response to specific researcher questions, as in the questionnaire or interview. In these instances, great care should be taken to ensure that responses are not presumed to apply to situations not indicated by the respondent. The third issue is that of representation. The use of unsolicited data may allow an over representation of the views of a vocal minority of the members. Although it would be possible to follow individual contributors and their respective views on individual topics across discussion threads, such a microanalysis will not be provided in this study. The commitment of resources necessary to provide this analysis is not viewed as commensurate with the contribution it would make to the study. This type of analysis would also violate the assumptions set out in the “Ethical issues” section, which asserts that the behaviour of individuals is not being studied. The community of practice consists of a broad variety of members, a great many of whom contribute. The community is not being employed as an outlet for a vocal minority.

Care should be taken to mitigate the possibility of researcher bias, of impairment of observer objectivity, and of observer effect on members. When coding the community of practice discussions, the same criteria should be applied as is used to code the questionnaire data. Any reference to material other than the topic of the study should be disregarded. The researcher should ensure that objectivity is maintained and that dissenting cases, if found, are objectively represented. The observer, who in this case is also a member of the community of practice, must endeavour to minimise any contamination of the member responses. By contributing to the community of practice, the researcher opens up the possibility of affecting the views of the members. This involvement of the researcher in generating or changing data is acceptable in constructivist views of grounded theory (Charmaz 2003), but is not in accordance with the grounded theory approach of Strauss and Corbin (1998), which forms the basis for this study.

Although the previous and the following sections include a direct link between the questions posed in the research instrument and the original research questions, the data gleaned from the community of practice discussions is not so easily assigned. Because the discussions are not in response to direct questions, it is not possible to assign discussions to specific research questions.
The researcher must read a discussion and determine which concepts can be coded without guidance from a direct instrument question. As indicated above, this requires the researcher to make some inferences about the intent of the discussion, which can be informed by an understanding of the discussion thread as a whole.

A computer-mediated community of practice discussion forum has been chosen to provide data for this study, due to the synergy between the community’s interest and that of this study. Discussions are either keyword searched or read as they arrive. If a discussion is identified as pertinent to the topic, each individual message is read and coded in accordance with the grounded theory approach. The threats to reliability and validity, when admitting observational data, can be mitigated by maintaining observer objectivity, minimising observer bias, and rigorous control of inappropriate attribution when coding the data. Although the discussions are not in response to direct research questions, it is possible to code the discussions to concepts in the dataset.

4.7.4 Interviews

Interviews are a method of information transfer, usually taking place in a face-to-face environment between interviewer and interviewee. Because it is based on one-sided questioning and answering, it is not an ordinary, everyday conversation (Cohen, Manion, and Morrison 2007). Interviews have a purpose, which may be either to gather information with relevance to the research questions or to delve deeper into the responses made in previous instruments. There is a range of different classifications of interviews, each exerting either more or less control on the process. Each is discussed below and justification is given for the choice deemed appropriate for this research. Issues of reliability and validity surrounding the use of interviews are discussed, along with identified threats to the reliability and validity. Design issues for an interview schedule are also presented.

Five different types of interview are identified by Heck (2006) and Cohen, Manion, and Morrison (2007). Two types of interview immediately discounted as inappropriate for this research are the non-directive and the focused interview. These types of interview allow the respondent to freely express ideas
and concepts with no or very little direction from the interviewer, similar to that used in psychiatric therapy (Cohen, Manion, and Morrison 2007). The remaining three types, structured, unstructured, and semi-structured are considered candidates for this research. The unstructured, or open-ended interview, is very flexible, allowing the interviewer discretion in asking any question deemed appropriate (Heck 2006). This flexibility extends to the wording of the questions and the sequence of the questions (Cohen, Manion, and Morrison 2007). This type of interview could lead to the omission of questions covering important concepts and lead to incomparability of participants’ responses. The structured interview, where questions and sequence are specified before the interview, offers the interviewer no unplanned flexibility (Cohen, Manion, and Morrison 2007). This may simply degrade to an answer aloud version of a questionnaire with no opportunity to elicit deeper responses. The semi-structured interview, where question wording and sequencing are determined prior to the interview, but all interviewer initiative is not removed, provides a balance. There is sufficient control to allow for comparability of results, while allowing for respondent freedom and interviewer probes. Of the original five different types of interview presented here, only the semi-structured approach provides some confidence in the ability to compare results while allowing the interviewer to probe for more depth in responses.

The design of the interviews used in this research is based on the semi-structured approach described above and the interview guide approach and standardised open-ended interview approach defined by Cohen, Manion, Morrison (2007). In particular, the question wording and sequences are specified in advance of the interview. The interviewer is not allowed to omit or reorder the questions. This should ensure that respondents interpret the questions in the context of previous questions and responses. The interviewer is granted the flexibility to provide additional questions in order to elicit greater depth in the responses. The interviewer is also granted the flexibility to record non-verbal indicators, such as body language or gestures. In the event that a response is provided to a question appearing later in the sequence, the interviewer should still present the question in sequence. This semi-structured
approach provides sufficient control to ensure comparability of results, sufficient flexibility to ensure depth of responses, and sufficient consistency to support the simultaneous collection and analysis of data indicated by the grounded theory method.

When discussing the reliability and validity of interviews, there are two issues to determine. Firstly, to establish validity, the instrument must be suitable for collecting the data necessary to answer the research questions, the instrument must appear to measure what it is designed to measure, and the conclusions should follow directly from the data collected by the instruments (Bell 2005). A well-designed, piloted interview satisfies the appropriateness of measurement requirement. The occurrence of theoretical saturation, whereby categories are identified by the data, indicates that the conclusions are derived from the data. While the results could be validated by returning them to the interviewee for comments, Morse et al. (2002) suggest that this could actually be a threat. In the case of grounded theory, the original responses may have been conceptualised and abstracted as an aggregate. Individual respondents may no longer actually recognise their individual contributions. Secondly, to establish the reliability of the interview instrument, the possibility of duplication must be addressed. A different researcher, under the same conditions, may not necessarily produce the same results. Once the respondents, especially the interviewees, interact with the posed questions, their thinking about problem solving, teaching programming, and computational thinking may actually change. In responding a second time to the same questions, this change may be evidenced, thereby indicating a reduction in the reliability of the instrument rather than a change in the participant. In addition, the responses to the opinion questions may give different results at different times, depending on the individual interviewee’s current emotional or education state. The most effective way to promote reliability in the interview instrument is to provide structure during the interview process. This interview control is discussed above. In addition, if theoretical saturation does occur, then the instrument is assumed to be reliable because the same instrument and the same researcher will have achieved similar responses from different respondents. The issues of reliability and validity, in interviews as with other qualitative instruments, are
addressed by rigour in design and implementation, rather than by a statistical measure.

Threats to the reliability and validity of interview instruments include dishonesty, human error, lack of nuance, and the inclusion of a limited number of respondents. The interviewee and/or the interviewer may not divulge all that they know about the questions. Of course, this may be due to ethical circumstances, personal beliefs, or even lack of rapport between the interviewer and the interviewee. If audio recording is employed, then the nuances of body language may be lost in the transcription. On the other hand, if video recording is used, then the additional coding of body language may lead to large amounts of data. If the interviewer is limited to taking field notes, then the respondent’s own words may not be faithfully transcribed. In the event that an interviewee declines to be recorded, then this may be the only option. If time is constrained, the intensive nature of interviews may result in the processing of a smaller number of respondents than, for example, a written questionnaire. While not being able to control for all reliability and validity issues, a crosscheck with the on-line questionnaire responses can provide a consistency check. The more highly structured an interview, the more reliable it is (Cohen, Manion, and Morrison 2007). As mentioned above, a semi-structured interview where questions and sequences are predetermined provides some control for reliability. The most appropriate way to enhance validity is to control for bias in the design and content of the questions and the attitudes and behaviours of the interviewer (Cohen, Manion, and Morrison 2007). Suggestions for mitigating these threats point back to the requirement of Morse and her colleagues (2002) for rigour to be consciously and conscientiously maintained through each step in the research process.

Table 3 reproduces the questions found in the interview instrument. The second column indicates the research question or questions tied to that interview question. The third column contains additional questions that may be used during the interview to prompt the interviewee for further clarification. The questions themselves are open and not leading. The interviewee may respond in any manner they deem fit. There is one question, included in the first section that simply serves as an icebreaker. However, it may generate data applicable
to any of the research questions. The research question contexts were presented in the “Research questions” chapter.
### Opening

<table>
<thead>
<tr>
<th>Question</th>
<th>Prompts</th>
</tr>
</thead>
</table>
| What prompted you to agree to this interview? | Any  
What about the topic is of interest to you?  
Do you view computer science as being influenced by problem solving, computational thinking, and programming? |

### Programming

<table>
<thead>
<tr>
<th>Question</th>
<th>Prompts</th>
</tr>
</thead>
</table>
| What is most challenging about teaching programming? | 2  
For example, is a while loop more difficult to explain than a for loop?  
Do students focus on just getting an answer and assume that they’ve then mastered the art? |
| What is difficult to learn about programming? | 3  
Do students ever want to put in blank else conditions, which means just keep going or do nothing? |
| What is easiest to learn about programming? | 3  
Is there an easy concept to learn if students join the course with no previous experience? |
| Is there a logical sequence to teaching programming concepts? | 4,6  
Should language constructs be taught first then put together?  
Should students master pseudocode before attempting real code?  
Is something like Scratch really pseudocode?  
Do you use a personally non- |
<table>
<thead>
<tr>
<th>Question</th>
<th>Page Numbers</th>
<th>Answer 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>How do you teach the use of functions and procedures?</td>
<td>4, 6</td>
<td>Do you use the notion of black boxes?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Do you use the term abstraction?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Do you have an analogy to maths?</td>
</tr>
<tr>
<td>How do you teach the way the machine works (notional machine)?</td>
<td>4, 6</td>
<td>How do students cope with the idea of each instruction being executed in the context or state of what has gone before?</td>
</tr>
<tr>
<td>How do you teach data representation and organisation?</td>
<td>4, 6</td>
<td>How do students respond to data structures of more than 2 dimensions?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>How do students respond to data structures of fixed length? Do they assume that size is automatically dynamic?</td>
</tr>
<tr>
<td>How would you describe the process of learning to program to someone who doesn't program?</td>
<td>2</td>
<td>How do you relate the necessity to be precise in giving instructions (i.e. what they're told and only what they're told)?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Is programming skill an innate ability or can anyone learn how to do it?</td>
</tr>
<tr>
<td>Computational Thinking Prompts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>What is the meaning of the term “computational thinking” in your work?</td>
<td>3</td>
<td>Do the terms decomposition, abstraction, and generalisation have an application in your work?</td>
</tr>
<tr>
<td>What activities contribute to development of</td>
<td>4, 5, 6, 7</td>
<td>Do students have opportunities to produce models, visualisations, or</td>
</tr>
<tr>
<td>computational thinking skills?</td>
<td>other representations of problem contexts or solutions? How do they normally go about this process? Do mathematical concepts play an important part in CT?</td>
<td></td>
</tr>
<tr>
<td>Do you think there is a relationship between programming and computational thinking?</td>
<td>2, 7</td>
<td>When students program, do they exhibit skills in decomposition, abstraction, or generalisation? Is computational thinking skill an innate ability or can anyone learn how to do it?</td>
</tr>
</tbody>
</table>

### Problem Solving

<table>
<thead>
<tr>
<th>Prompts</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>What does the term “Problem Solving” mean in your work?</td>
<td>3</td>
</tr>
<tr>
<td>What other subjects does the term bring to mind (maths, science)? Did you mention computer science?</td>
<td></td>
</tr>
<tr>
<td>What activities contribute to the development of problem solving skills?</td>
<td>4, 6, 7</td>
</tr>
<tr>
<td>Do practical activities help develop thinking? How do you make the connection explicit? Is there a step-by-step problem solving methodology that you find useful? Does the word “problems” immediately bring to your mind, the context of mathematics?</td>
<td></td>
</tr>
<tr>
<td>Do you think there is a</td>
<td>2</td>
</tr>
</tbody>
</table>
Does one of them involve a particular kind of constraints?
Is problem-solving an innate ability or can anyone learn how to do it?

Wrap Up

Recall the relationship you identified between programming and computational thinking.
Recall the relationship you identified between problem solving and computational thinking.

Does that imply a relationship between all three (problem solving, computational thinking, and programming)?

Could we draw a visual representation of that relationship?

<table>
<thead>
<tr>
<th>Does that imply a relationship between all three (problem solving, computational thinking, and programming)?</th>
<th>Is there a hierarchy in this relationship? Are all of the three items of equal standing? How might this relationship be reflected specifically in computer science?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3: Interview schedule mapped to question contexts

A semi-structured interview instrument has been chosen because it provides strictness in question wording and sequencing while not sacrificing flexibility for the interviewer to seek more in-depth responses or clarifications. In addition, it provides the freedom for respondents to express themselves in terms of their
own choosing without being bound by predefined response ranges. It also provides access to knowledgeable respondents who, simply by choosing to participate, have already demonstrated an interest in the research topic. Many of the reliability and validity issues posed by interviews can be mitigated by strict attention to details at the design stage and application of strict interview behaviour on the part of the interviewer. The interview schedule has been designed with consideration for the original research questions. Each question has been shown to have relevance to the original research question. Obviously irrelevant information, if supplied by the respondent, will not be included in the dataset. An interview meets the requirements of the research for this phase.

In summary, this research employs two individual instruments, incorporating directed questions, and discussions from an Internet based community of practice forum. The first instrument is an online questionnaire, where the participants are given the opportunity to respond using free format text. The second instrument is a semi-structured interview. Both the online questionnaire and the interview schedule incorporate open-ended questions to which the participant may respond in any way they deem appropriate. The questions for each instrument have been designed to provide data specifically for the stated research questions. The discussion threads from the community of practice are chosen for their direct applicability to the research questions. Each applicable discussion is coded in the same way as the questionnaire and interview data. All instruments are suitable for collecting data for grounded theory research. The instruments for data collection have been presented and their reliability and validity have been discussed in this section. The following section discusses the data collected via these instruments and its processing in accordance with the grounded theory approach.

4.8 Data collection, analysis, and model creation

The following sections, although presented in chronological divisions, are not distinct steps in the data collection and analysis process. Each milestone represents a momentary pause in the iterative process of data collection, codification, and theory generation followed by further data collection. Each pause provides an opportunity to reflect on progress, to identify concepts and
categories in the data, and to plan the next steps. The latter milestones include descriptions of the analysis that supports generation of a model.

4.8.1 Milestone 1: Tentative beginnings

During the first stage of data collection, two participants were invited to undertake the online questionnaire with a view to providing feedback concerning its overall structure, the completion time requirements, and the quality of the questions. Although positive overall, the respondents’ feedback highlighted a potential problem. One participant felt that he could not answer the questions appropriately because a definition of computational thinking had not been supplied. On further analysis of this participant’s responses, it is clear that he does possess a viable definition of computational thinking, even if he does not know the phrase.

The possibility of including a definition for computational thinking arose earlier in the questionnaire design phase. Introducing such a definition was discounted due to the possibility of distorting the results due to leading questions. Therefore, before modifying the questionnaire design to include a definition, a decision was made to invite another small group of participants to attempt the questionnaire.

The second group of participants consisted of members of the county’s computing curriculum development group, a group of teachers and administrators involved in secondary and post-16 education. Thirteen of the members volunteered to participate. Sample bias has been discussed above and will also be addressed in a following section. An analysis of the responses, specifically concerning the understanding of the term computational thinking, indicates that the respondents possess an understanding of the term aligned with the literature (Wing 2006). It is therefore decided not to amend the online questionnaire to include a definition of the term.

The next action involves coding the first 15 questionnaire responses as nodes using qualitative data analysis software. The first attempt at coding the responses is via hierarchical nodes. This is appropriate for data such as programming language and demographics. These data lend themselves to simple statistical analysis and fit discreet categories. The prose responses
Chapter 4: Method

facilitated the development of a range of free nodes. If a response indicated the concept of abstraction in the definition, it was coded to the concept of abstraction. If a prose response indicated the use of puzzles, it was coded to the puzzles node.

4.8.1.1 Next actions
At this very early phase of the investigation, the focus is on the continued collection and coding of data. This early coding and analysis will dictate the activities undertaken in successive phases. The following items are to be undertaken at this stage.

- Continued coding of on-line questionnaire data
- Continued collection and coding of community of practice conversations
- Conduct, collect, and code interview data
- Amend interview questions as required

4.8.1.2 Conclusion
These first steps in the data collection and analysis cycles have resulted in a small set of data with which the analysis may begin. It also affirms that the online questionnaire is fit for purpose and does result in data that can contribute to answering the original research questions. Further data will be collected via the questionnaire while the more targeted interviews, described in the “Interviews” section, are conducted.
4.8.2 Milestone 2: First concepts and categories

The next milestone in the data collection process occurred at the point when 12 conversations from the community of practice forum were transcribed and 18 online questionnaire results had been coded, mainly as free node concepts. At this point, the number of free node concepts, nearing 90, became unwieldy. Grounded theory (Strauss and Corbin 1998) requires that the concepts be analysed further to identify similarities, differences, and categorisations. In order to deal with the large number of concepts, many were merged based on their similarity. For example, all concepts referring to the use of templates, whether they were written in structured English, pseudocode, or real code were grouped together as “faded worked examples”. Application of this technique across all the free node concepts reduced their number by half.

As the concepts were revisited, they were also organised into broad categories. For example, the aforementioned “faded worked examples” concept, along with the “deconstruction” concept and the “puzzles, games, and role play” concept, is assigned to the “activities promoting computational thinking” category. The main categories, to which the concepts were assigned, identified during this exercise are teaching programming, activities promoting computational thinking, taxonomy of problem solving skills, and computational thinking skills.
Chapter 4: Method

Further analysis revealed relationships between the concepts in the main categories. For example, the concept of puzzles, although identified as an activity contributing to the development of computational thinking skills, does not in itself identify the contributed component. However, consideration for the properties of puzzles, such as dead ends and false trails, leads to the idea of persistence. The concept of persistence can then be incorporated into the “computational thinking skills” category.

The analysis, at this stage, has revealed four main categories, listed below. Following each category is a representative sample of the type of concept that has been assigned to that category. These categories and concepts may change, as more data is added and processed.

Teaching programming

- Collaboration is identified as an effective teaching strategy. This is usually described as paired or group work, most commonly involving discussion of analysis or design. There are no reports providing opportunities for group implementation or paired programming.
- Programming is perceived as an innate ability. Several respondents report that, regardless of the lower barriers presented by simplified visual environments (Scratch and Alice), some students hit a brick wall when asked to move on to more demanding problems and less forgiving development environments.

Activities promoting computational thinking

- Deconstruction or reverse engineering is identified as contributing to the development of computational thinking. This exercise starts with a working program and performs the analysis backwards. This is similar to the black box logic problems where only inputs and outputs are defined and the viewer has to surmise how the box works.
- Decomposition involves breaking the problem down, usually in a forward direction, identifying major tasks that move the solution closer to the objective. This is analogous to identifying the order of the black boxes above.
• Faded worked examples are implemented by giving students partially completed code and asking them to complete it correctly. This could be done in a development environment to provide feedback. This is analogous to fill in the blank worksheets.

• Puzzles, games, and role play, all provide opportunities for sustained and lengthy problem solving where there may be false trails, dead ends, but ultimately, an “ah ha” moment. Kinaesthetic activities fit into this category.

Taxonomy of problem solving skills

• Understanding the problem and its constraints appears to be a necessary first step to learning to program. There is also a link here with the ability to recognise what a solution may look like. In other words, it is necessary to know that the problem is actually solved.

• Students with a proficiency in mathematics are perceived to perform better in computer science. However, it is not necessarily mathematical ability that underpins this connection. It is the ability to think in a logical purposeful way, selecting data to identify a progression path to the problem solution.

• Persistence is also evident in the data. This is often linked with the idea of “not giving up” and puzzles or games, which provide sustained and lengthy problem solving with discrimination of useful data, back tracking, and constant evaluation.

Computational thinking skills

• Decomposition, the skill to break problems down, is often taken for granted. However, for some students, this is very difficult.

• Modelling is identified in the sense of high-level systems that are decomposed into smaller parts, with each individual part modelling behaviour of a subsystem.

• The concept of programming as a vehicle to teach computational thinking crosses many boundaries: academics, teachers, and industry;

• Algorithm design is tied heavily to problem solving. It is defining the steps, using some accepted convention, necessary to solve a problem.
Chapter 4: Method

This is different to program design, which is the translation of an algorithm into automation understandable by a computing device. For example, writing pseudocode to solve a problem is a computational thinking skill; translating that pseudocode to Java is a programming skill.

- There is a theme involving the use of the words "analytical" and "logical". Very few responses expound on these terms to clarify their meaning. The analytical thinking term appears to involve comparing alternatives, precisely describing, explaining how, criticising weaknesses, all in terms of computer science, but could be applicable to any context. This could be akin to critical thinking skills in any domain. One respondent makes a tie with "expository skills". This goes back to the "if you can't talk it, you don't understand it" idea. The logical term appears to be associated with programming constructs such as sequence, selection, and iteration or with mathematics. It is sometimes associated with the term procedural.

4.8.2.1 Next actions

At this early phase of the investigation, the focus is on the continued collection and coding of data. This early coding and analysis will dictate the activities undertaken in successive phases. The following items are to be undertaken at this time.

- Continued coding of on-line questionnaire data
- Continued collection and coding of community of practice conversations
- Conduct, collect, and code interview data
- Amend interview questions as required

4.8.2.2 Conclusion

The initial categorisation of concepts has led to the identification of gaps. From the literature review, it was anticipated that the concepts of generalisation, visualisation, and abstraction would be found in the participants’ responses. However, at this stage, none of these concepts has been addressed directly by the participants. Having identified group work, for analysis and design, as an effective teaching strategy, why is not more group implementation (paired programming) facilitated? Is the perception of programming ability as innate an indication of learners not having been exposed to enough opportunity to
practice appropriate computational and problem solving skills? These shortcomings may be addressed by new data or be pursued in the interview process. New participants may also reveal concepts and categories that are not yet represented in the dataset.

![Word cloud](image)

Figure 9: Words used in community of practice responses

### 4.8.3 Milestone 3: Additional concepts

At this point, more data has been prepared and added to the existing dataset. Fourteen conversations from the community of practice forum and an additional interview have been coded for concepts and incorporated. Much of this data affirms existing concepts. However, some new concepts have been identified as follows:

- Planning is important because it represents higher-level computational thinking skills. Planning is part of systems analysis in the real world. Other real world endeavours requiring computational thinking skills are software engineering, systems analysis, requirements gathering, and project management.
- Planning, as part of teaching at Key Stage 3, should be minimal. For example, stating the rules of a game, indicating what happens when the sprite touches the boundary, or indicating how the game ends are at an appropriately high level for younger learners.
Chapter 4: Method

- Planning at Key Stage 3 should use generic tools, not industry standard ones. Appropriate tools include the drawing of boxes with lines, sketches, storyboards, or user scenarios. Young learners should not be exposed to state diagrams, flow charts, or Unified Modelling Language diagrams.

- Moving on from visual languages, like Scratch, may be difficult due to its rich development environment. In other environments, the development is slower and results not achieved quickly enough for young learners. However, this movement has to be made otherwise new possibilities are not uncovered and the tool becomes the limiting factor.

- Problem solving is a real world, everyday activity. It should be explicitly identified as such. Normal problems and approaches to solve them should be explicitly pointed out and modelled.

- There is some question about what teachers are trying to achieve by using so many different tools, such as Scratch, Alice, Logo, and Greenfoot.

- Start early with problem solving in every subject, building complexity and difficulty up over time.

- Students must be given the opportunity to fail and helped with strategies to learn from their mistakes.

- Algorithmic thinking is a transferable skill, applicable in many contexts.

- Theory should be introduced when there is a defined need for it. Creativity comes first and often leads to pupils’ questions. This is the opportunity to introduce the theory. Use the coding hook to teach the theory covertly.

- Teaching of theory topics tends to move from abstract theory to concrete example. However, learning moves from seeing examples to generating the theory. Teaching is by deduction; learning is by induction.

- Computational thinking means learning to ask questions about alternatives, trade-offs, justifying decisions, identifying limitations, refining solutions, and evaluating results.
4.8.3.1 Next actions

At this phase of the research, the focus is on collecting not only a reasonable quantity of data, but also a significantly representative set of respondents. This means ensuring that teachers from several key stages, academics, and industry are all represented. To that end, the following tasks should now be undertaken.

- Collect more data from the community of practice conversations to add to the dataset
- Add additional interview data, as it is collected
- Continue coding data against the existing node structure

4.8.3.2 Conclusion

Although the previous section, "Milestone 2: First concepts and categories", identified that some specific terminology associated with computational thinking, for example abstraction, had not been addressed by the participants, the inclusion of these new data has begun to remedy that omission. In addition, some relevant concepts concerning the order in which learners should be presented information and the representation of that information have been highlighted. The theme of programming consisting of multiple steps, including analysis and design, and the concept of planning imply that even young learners need to be given opportunities to develop those capabilities, prior to engaging with code creation. These two concepts, process of programming and order of delivery, may make a significant contribution to the results of this research.

Figure 10: Words used in interview responses
Chapter 4: Method

4.8.4 Milestone 4: Initial queries
At this milestone, the data set is sufficient to allow queries. In this phase, two types of queries can be performed, simple text queries and matrix queries for identifying which groups of respondents already use the vocabulary associated with computational thinking. This phase of analysis also reveals two interpretations of the term analysis, and that age is not indicative of required scaffolding when learning to program. Questions are also raised during interpretation of the data. These questions concern identifying which computational thinking skills are more difficult to learn and identifying stakeholder expectations.

The current data set consists of 33 on-line questionnaires, 47 community of practice threads, 2 interviews. The number of unique respondents represented is 166. The existing data has been coded into 48 hierarchical nodes and 45 free nodes. Data has been also been segmented into 6 major tree nodes with a further division into 38 sub-nodes. These six major nodes reflect the original conceptual model in the literature review. The six major nodes are:

- Methods of teaching programming
- Bloom’s taxonomy
- Computational thinking skills
- Problem solving skills
- Programming techniques
- Teaching techniques

At this point, it is possible to use QSR NVivo™ (2007) to interrogate the data set in a more effective manner. Queries were constructed to perform simple text searches for the terminology associated with computational thinking, including abstraction, decomposition, algorithm design, and evaluation. Using a matrix query, it is possible to establish which group of respondents already use this vocabulary. The following figure indicates that the phrase “computational thinking” is slightly more familiar to post-16 teachers than those in institutions that cater for secondary as well as post-16. There is a large step down to the levels of use represented by infant and primary teachers. Along these same
lines is a significant association between the terms computational thinking and problem solving. This relationship will be explored further in the discussion of the results.

![Teachers' Use of Computational Thinking Terminology](image)

Figure 11: Teachers' use of computational thinking terminology

Coding the data during this step has highlighted several new concepts that may influence the results of this study. The following are areas deserving of further exploration:

- The coded data indicates two different interpretations of the term analysis. One definition involves studying for understanding of functionality. The other involves studying for an understanding of efficiency and effectiveness.

- Several respondents indicate that, regardless of the starting age, learners must all go through the same stages when learning to program. Older learners may move at a quicker pace, but still need to be provided with the same scaffolding.
Given the use of the terminology, shown above, the question arises concerning which skills are more difficult to teach or more difficult to learn. For example, is it more difficult to identify and create abstractions than to design an algorithm?

Analysis of the data raises questions about the expectations that stakeholders, students, parents, and government, have concerning the content of Post-16 or secondary programming courses. For example, questions are asked about the teaching of bubble sort when it is not the most efficient method of sorting and about the use of procedural programming languages when object-oriented paradigms are prevalent in industry.

4.8.4.1 Next actions
In the next phase of the data collection and analysis, the following items should be initiated.

- Investigate further the two different interpretations of the term analysis.
- Explore further the need to scaffold learning in the same way, regardless of the age of the beginning learner.
- Develop questions and queries to investigate the different levels of difficulty associated with computational thinking skills.

4.8.4.2 Conclusion
Each of these areas, exposed in this phase of analysis, can be explored during the next phase. Specifically, a better understanding of the term analysis could have a direct effect upon the building of a model demonstrating a relationship between the computational thinking terms. The necessity to scaffold learning in the same way, regardless of the age of the learner, could have an effect on the order in which topics, skills, or content are presented. This leads directly to a consideration of difficulty level. Knowing which computational thinking skills are more difficult to teach or to learn could affect classroom practices such as scheduling, differentiation, or interventions. Lastly, considering whether the focus of teaching programming should be on modern day techniques rather than historical foundations is of concern for those designing curricula or courses.
4.8.5 **Milestone 5: Defining a lexicon, hierarchies, and models**  
This phase of the analysis will focus on creating a lexicon of frequently used words in the data set. From these, terms of significant interest will be selected. These terms will form the basis for the creation of queries to support identification of hierarchies of programming terms and computational thinking terms. The results of these queries and original coding will be used to generate models and relationships found in the data.

In an attempt to supplement the choice of terminology of interest originally selected from the literature, an analysis of the text data was undertaken to identify relevant words based on their frequency of use by respondents. The prompt for this approach is that many responses use the words “break problems down” instead of the computational thinking term “decompose”. A complete analysis of the text generated 9078 individual words, not stemmed and capitalisation not ignored. Next, the entire list was reviewed to remove items not of interest, such as which, will, there, and their. Stemmed words and capitalised words were then grouped together, for example Teach, teaches, and teachers. Sampling across the selection revealed that words with frequency counts less than 8 did not result in additional coding of data. Therefore, these were dropped. The resulting lexicon of relevant words included items such as Teacher, skills, code, languages, and solving.

At this point, NVivo™ can be used to query the data set to elicit information about the relationships between the ways the words in the lexicon are used. The following table describes some of the queries created at this time.

<table>
<thead>
<tr>
<th>Proposed Query</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix query different combinations (harder near algorithm) across key stages</td>
<td>Answer questions like “which group of teachers think that algorithm design is harder than …”</td>
</tr>
<tr>
<td>Use of “because” or “cause”</td>
<td>Show cause and effect</td>
</tr>
<tr>
<td>“my concern”, “interest”, or a combination such as “concern” and “literacy”</td>
<td>Show high level of engagement or interest</td>
</tr>
</tbody>
</table>
### Table 4: Proposed queries and justification

<table>
<thead>
<tr>
<th>Query</th>
<th>Justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Words expressing emotion, such as “boring”, “exciting”, “fun”</td>
<td>Indicates motivation</td>
</tr>
<tr>
<td>Bloom’s language and synonyms</td>
<td>Begin to show relationship to other models</td>
</tr>
<tr>
<td>Bloom</td>
<td>Indicates an awareness of the cognitive domain</td>
</tr>
<tr>
<td>computational thinking</td>
<td>Indicates awareness of term</td>
</tr>
<tr>
<td>Using computational thinking language and synonyms</td>
<td>Indicates awareness of associated skills, even if the word computational thinking is not used.</td>
</tr>
<tr>
<td>Using “example”</td>
<td>To find where someone is giving a concrete example. This might work in with the idea of analogy.</td>
</tr>
<tr>
<td>Using “always”, “never”, “sometimes”. Perhaps in combination with Bloom or computational thinking terminology</td>
<td>Frequency</td>
</tr>
<tr>
<td>Query using comparatives or superlatives: harder, hardest, easier, easiest, higher, lower, highest, lowest, deep, deeper, long, longer, longest</td>
<td>Comparatives might lead to development of a hierarchy. Can cross these with other queries to produce hierarchy.</td>
</tr>
<tr>
<td>Using “harder” near “create or apply” or other combinations</td>
<td>To find complex ideas of hierarchy tied to Bloom or computational thinking terminology</td>
</tr>
<tr>
<td>Look for “computationally”, as in “think computationally”.</td>
<td>To cover all the possible options, in the case of uncertainty regarding stemming rules.</td>
</tr>
</tbody>
</table>

In order to identify any hierarchies in the data, which might be mapped to computational thinking, or to Bloom’s cognitive taxonomy, two different queries were created, each involving the use of comparatives such as easy, deep, high, and hard. One matrix query crossed these comparatives with Bloom’s cognitive taxonomy terminology. The other crossed the same comparatives with terms.
associated with computational thinking, as derived from the literature review. The figures below illustrate the results of these queries.
Chapter 4: Method

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>bloomAnalyse</td>
<td>2</td>
<td>6</td>
<td>9</td>
<td>13</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>bloomApply</td>
<td>5</td>
<td>8</td>
<td>10</td>
<td>43</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>bloomBloom</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>bloomCreate</td>
<td>6</td>
<td>14</td>
<td>12</td>
<td>17</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>bloomEvaluate</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>bloomRecall</td>
<td>1</td>
<td>6</td>
<td>9</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>bloomUnderstand</td>
<td>6</td>
<td>16</td>
<td>18</td>
<td>21</td>
<td>16</td>
</tr>
</tbody>
</table>

Figure 12: Bloom's cognitive taxonomy and comparatives
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ctAbstraction</td>
<td>✔️</td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>ctAlgorithmicThinking</td>
<td>✔️</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>ctAnalyse</td>
<td>✔️</td>
<td>2</td>
<td>6</td>
<td>9</td>
<td>13</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>ctAutomation</td>
<td>✔️</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>ctComputationalThinking</td>
<td>✔️</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td>37</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>ctDecomposition</td>
<td>✔️</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>ctEvaluation</td>
<td>✔️</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>ctGeneralisation</td>
<td>✔️</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>ctLogicalThinking</td>
<td>✔️</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>ctModellingSimulation...</td>
<td>✔️</td>
<td>4</td>
<td>11</td>
<td>9</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>ctProblemSolving</td>
<td>✔️</td>
<td>4</td>
<td>11</td>
<td>7</td>
<td>37</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>ctSystemsDesign</td>
<td>✔️</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>ctThoughtProcess</td>
<td>✔️</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 13: Computational thinking terminology and comparatives
Chapter 4: Method

Each reference represented in these queries was read and recoded specifically with the objective of identifying hierarchies of terms. This analysis resulted in 26 nodes representing relationships between terms that can serve as a basis for initial model construction. The nodes are illustrated in the following figure.

<table>
<thead>
<tr>
<th>Name</th>
<th>Sources</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstraction at secondary</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Abstraction before create</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Abstraction harder than decomposition</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Abstraction is difficult - Funcs Procs</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Analyse a problem</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Analysis at varying ages and levels of matur</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Analysis before Creation</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Analysis for more able</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Analysis is lower than design</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Apply knowledge below create knowledge</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Create algorithm before create program</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Decompose is part of analysis</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Decomposition before creation</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Decomposition before evaluation of solution</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Decomposition harder than algorithm design</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Decomposition is most difficult</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Evaluation for more able</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Evaluation higher than creation</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Knowledge Before Analysis</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Knowledge before Apply</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Knowledge before creation</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Knowledge before understanding</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Map Computing To Bloom</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Recall is low</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Sequencing before creation of solution</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Sequencing before evaluation</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 14: Nodes representing relationships
4.8.5.1 Model generation and discussion

From this initial analysis, two areas of interest have arisen, one focusing on the order in which programming is normally taught and the other focusing on the difficulty of some computational thinking skills.

The following illustration sets out, based on the data analysis, the respondents' views of a relationship between the skills associated with programming and computational thinking. All terminology associated with computational thinking is not represented in this figure. The terms in circles represent the views of the respondents. Where orders of teaching were indicated, they have been faithfully reflected in this distribution. The terms in rectangles represent the levels of Bloom’s Taxonomy (Cognitive Domain). Unsurprisingly, the order in which programming is taught by the respondents clearly corresponds to the cognitive domain categories.
Chapter 4: Method

Figure 15: (a) Teaching programming corresponds to Bloom’s cognitive domain

118
Figure 16: (b) Teaching programming corresponds to Bloom's cognitive domain
Chapter 4: Method

Further analysis of the data, especially with regard to the use of comparatives and superlatives in relationship to programming and computational thinking terminology, has resulted in the beginnings of a model illustrating the levels of difficulty associated with computational thinking skills. This is, of course, only the beginning of a model. There are highlights here that will need further investigation. For example, some respondents identify that abstraction is more difficult than decomposition, which in turn is more difficult than algorithm design. There is a dissenting case where one respondent describes abstraction as more difficult than decomposition and another reverses that opinion. When mapped to Bloom’s Cognitive Domain, this order is not consistent. Bloom’s taxonomy implies that algorithm design should be at a higher level of cognitive difficulty than decomposition. This discrepancy will, of course, need to be addressed in future interviews. Exploring the difference between an order of difficulty to learn and the order of complexity as a thought process will be the focus of the next phase of data collection and analysis.

![Diagram of computational thinking terminology hierarchy of difficulty](image)

Figure 17: Computational thinking terminology hierarchy of difficulty
4.8.5.2 Next actions

In order to move forward from this position and better inform the building of a model of the relationships between problem solving, computational thinking, and teaching programming, some further avenues need to be explored. These are detailed below, along with how the exploration might be achieved in the next phase of data collection and analysis.

- Although at least one higher education and at least one further education teacher have been interviewed, there should be broader representation. In order to achieve this, additional interview participants should be pursued. These additional interviewees should include:
  - a primary teacher,
  - a secondary teacher, and
  - a representative from industry.

- The term “generalisation”, although found in the text, is not used in the context of computational thinking. This omission should be explored in future interviews.

- The terms “modelling”, “model”, “simulate”, “simulation”, “visualise” and “visualisation”, again appear in the text, but not in the context of computational thinking. This omission should be explored in future interviews.

- Having identified an inconsistency in the interpretation of the term “analysis”, future interviews should attempt to distinguish between the terms “analysis” and “evaluation”.

- Having also identified, at this point, that some computational thinking terms are under-represented in the data set, an attempt should be made to identify if respondents are familiar with and have an understanding of the terms “abstraction”, “decomposition”, “algorithm design”, “analysis”, and “evaluation”, as identified in the literature review.

- In general, questions that reflect comparatives need to be pursued. In other words, “Which is more difficult: decomposing a problem or designing an algorithmic solution?”
Chapter 4: Method

- Update interview questions to reflect new findings and requirements as indicated above. New interview schedule can be found in “Appendix 2 Interview revision (af)”.

4.8.5.3 Conclusion
As presented above, a candidate lexicon has been created consisting of 9078 individual words. Words with frequencies of 8 or more were chosen to form the lexicon. Again, a set of queries has been created which focus on these words and the use of comparatives. From these queries, a hierarchy of programming skills and computational thinking skills began to emerge. These relationships are illustrated in the first attempts at creating models. The models presented in this section include a model of how the teaching of programming maps to the Cognitive Domain of Bloom's Taxonomy and a hierarchy of difficulty associated with computational thinking.

4.8.6 Milestone 6: Interviews and additional models
At this point, 3 more interviews are added to the data. These represent a Secondary+Post-16 teacher, a STEM Ambassador from industry, and a specialist ICT primary teacher. These will be coded into the existing set of nodes with new nodes being added as required. The following describes several of the new and interesting insights revealed during this analysis.

Formalisation of thinking is difficult. A query was created to identify words from the lexicon that might indicate moving between notations. This query is defined as "translation" OR "transition" OR "movement" OR "formalisation". Twenty-six responses, illustrated below, were returned. Note, some source names obscured for anonymity. This set crosses with the 3 sources coded to the “formalisation of thinking – Most Difficult” node. This set may also have connections with the “If you can’t talk about it, you don’t understand it” free node.
These responses have been reread and, where applicable, recoded to the “formalisation of thinking is difficult” node. This now stands at 5 sources with 11 references.

A number of responses identify a common teaching approach, that of beginning with a problem that the learners already understand and for which they have some concept of what a correct solution should look like. The next step varies. Some respondents introduce a solution for simple problems such as “guess the number” and “make a square”. They then go on to deconstruct the solution to
Chapter 4: Method

deduce the algorithm for the solution to the problem. Others begin with the problem and decompose it to identify a viable solution. This observation leads to the question about the difference between these two types of skills. Is deconstruction of a solution different from decomposition of a problem? This could be pursued further in additional interviews.

One respondent indicates that the ability to abstract can be evidenced by careful observation of learners’ questions. For example, at KS3 with Scratch, pupils might evidence abstraction by asking, “How do I make the sprite jump?” The pupils already use words that hide the complexity of the underlying implementation. They then go on to use the associated Scratch puzzle pieces as a group to implement jumping. Even at primary school, pupils may already be evidencing abstraction. Having once written a Logo procedure to draw the letter “E”, pupils systematically use the “E” on the keyboard to activate that procedure, without consideration for how it has been implemented. Further investigation may indicate whether the ability to abstract may be lower on the spectrum than some of the other computational thinking capabilities.

Again, in the area of abstraction, respondents struggle to evidence the ability to abstract. With reference back to the previous section, this would appear to depend on what evidence is acceptable. Is it sufficient to name an abstract thought as evidence of the ability to abstract? In a broader sense, this question crosses all key stages. In primary, perhaps the ability to create a procedure in Logo called “E” and use it to spell “HELLO” is evidence of abstraction. At post-16 this example might include the necessity to pass parameters into a procedure or function and return a result. Is this just an increase in level of complexity of the same level of a single computational thinking skill?

Understanding how different age and capability of learners evidence computational thinking skills will inform assessment across all key stages.

A few respondents have suggested that all learners must transcend the same steps when encountering computational thinking through programming. A primary teacher indicates that young learners go from concrete, kinaesthetic activities to the ability to develop algorithms. This may be because younger learners need the practice of giving precise instructions for physical activities as
a visualisation before moving to the abstraction of the machine. Another respondent identified a need for this same progression with some much older, post-16 and adult, learners. If this kind of scaffolding is always needed, then perhaps it is attainable in other subjects at other ages (mathematics, DT)? If so, then it may be transferable to programming. If not, then it must be taught as part of teaching programming.

Consideration for the coded data leads to questions about other skills that might limit computational thinking ability or the ability to evidence computational thinking. This is manifested by the coded responses that indicate a high level of concern with differentiation, as indicated in the figure below.

<table>
<thead>
<tr>
<th>Name</th>
<th>Sources</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collaboration</td>
<td>37</td>
<td>69</td>
</tr>
<tr>
<td>Differentiation</td>
<td>40</td>
<td>84</td>
</tr>
<tr>
<td>Exemplar</td>
<td>28</td>
<td>60</td>
</tr>
<tr>
<td>Live Correction</td>
<td>47</td>
<td>77</td>
</tr>
</tbody>
</table>

Figure 19: Differentiation is a concern for many respondents

This leads further to the need to identify why differentiation is needed. Is there a lack of some skill that limits the accessibility of computational thinking or the ability to evidence computational thinking? With reference to the above section indicating a varied range of capabilities possessed by those being introduced to computational thinking through programming, this differentiation may be needed to cater for the disparate range of existing knowledge at each learner level. If that is the case, then the higher up the age ladder, the more disparate the range of possible skills may become. A primary teacher identifies that for primary, the abilities to read and write often limit the ability to evidence computational thinking. Instead of writing comments in code, learners may need to talk to express understanding. A secondary and post-16 teacher also refers to this ability to “talk a solution”. However, in this context, it is not purely literacy that limits the ability to write a solution, but the inability to translate from an internal representation to an external one. The concept of computational thinking being limited by capabilities or even tools is beginning to emerge from the data. Currently 5 sources, illustrated below, have identified this as a concern.
### 4.8.6.1 Model generation and discussion

Interpretation of the data from this section reveals four distinct evolving models. Each of these is described below.

This figure illustrates computational thinking and programming skills mapped to the Cognitive Domain of Bloom’s Taxonomy. As with the previous models, it is not surprising that computational thinking skills and programming activities map onto the Cognitive Domain. The conflation of analysis, abstractions, and decomposition into the analyse level requires further exploration.
Figure 21: (a) Teaching programming corresponds to Bloom's cognitive domain
Chapter 4: Method

Figure 22: (b) Teaching programming corresponds to Bloom’s cognitive domain
The following figure illustrates an alternative distribution of programming skills and computational thinking skills. Here, the level of create has been subdivided into designing an algorithm and creating a program. In addition, the level of analysis has been subdivided into abstraction and decomposition. This helps to clarify a possible level of difficulty of computational thinking skills, not necessarily the order in which these skills are taught.
Chapter 4: Method

![Diagram showing subdivision of programming skills and computational thinking skills across the cognitive domain.]

Figure 23: (a) Subdivision of programming skills and computational thinking skills across the cognitive domain.
Figure 24: (b) Subdivision of programming skills and computational thinking skills across the cognitive domain
Chapter 4: Method

The following figure reflects the reported levels of difficulty associated with specific computational thinking skills. Additional data during this phase has not specifically affected this model.

Figure 25: Computational thinking terminology hierarchy of difficulty

The new addition, at this phase, is the revelation that formalisation of logic may be quite difficult for learners. The formalisations that learners are expected to master are the transformation of thinking logic to a simple formal notation such as flowcharts or pseudocode. From either of these notations, learners are expected to write working program code. The connecting paths between these notations represent different levels of difficulty. The following figure attempts to express this relationship between notations and the difficulty of moving between them. Respondents indicate that moving from thinking to flowcharts is relatively easy compared to moving from thinking to pseudocode. It is easier to produce pseudocode from flowcharts. However, it is easier to write code from pseudocode than from flowcharts. In all cases, the introduction of nested instructions makes the tasks much more difficult than simple sequencing of
instructions. The model is directly supported by the coded nodes indicated at the bottom of the model. This model further contributes to understanding how learners might evidence computational thinking by producing flowcharts or pseudocode.
Figure 26: Formal notations and difficulty of moving between them

A fourth set of diagrams reveals respondents’ interpretations of the relationship between problem solving, computational thinking, and programming. When
viewed separately, each of these models is understandable. However when compared, there are definite differences. The further education teacher’s model places computational thinking as a superset of programming and problem solving. On the other hand, the primary teacher’s model separates computational thinking and programming, but both become subsets of problem solving. The secondary teacher’s model views all 3 entities as distinct, but with an area that intersects them all. These models describe an interface relationship between problem solving, computational thinking, and programming, not necessarily a hierarchical relationship.

<table>
<thead>
<tr>
<th>Teacher – Further Education</th>
<th><img src="image1" alt="Diagram" /></th>
</tr>
</thead>
<tbody>
<tr>
<td>Teacher - Secondary</td>
<td><img src="image2" alt="Diagram" /></td>
</tr>
</tbody>
</table>
4.8.6.2 Next actions

Based on the additional concepts and questions resulting from this phase of the analysis, several new leads need to be explored in the next phase to better inform identifying the connections between problem solving, computational thinking, and the teaching of programming. These next actions are detailed below.

- Attempt to identify where abstraction lies on the spectrum of computational thinking skills by asking specifically how functions and procedures are taught and learned. This can be done by amending the interview question schedule to include appropriate prompts.
- Explore the concept of the first programming tasks involving simple problems for which learners may already know a solution. Ask respondents specifically how the first programming lessons are taught. Does it involve a problem with a well-understood solution? Again, this can be done by further probing during the interview procedure.
- Explore idea of scaffolding with concrete or physical examples, especially at Key Stage 4 and Post-16. This can be done by asking
questions concerning kinaesthetic activities, especially of teachers at the higher key stages.

- Attempt to identify examples of abstraction in use by Key Stage 3, Key Stage 4, and Post-16 teachers in their teaching as well as examples evidenced by the learners. This can be done by asking specific questions about the use of abstractions by teachers and by learners.

- Explore further, with new participants, these new ideas about factors that may limit computational thinking ability or evidencing computational thinking ability. Be careful to identify whether it is computational thinking ability or something like literacy, numeracy, or mathematical ability that is the source of the limitation. Quite specific questions may be needed to generate enough data to address this issue. However, the interview questions can be amended to prompt for these type responses.

- Update interview questions to reflect new findings and requirements as indicated above. New interview schedule can be found in “Appendix 1 Interview revision (ag)”. Although each of the preceding items may be addressed with specific prompting of new interview participants, the existing data set can also be re-explored to identify contributing responses. This will be done in the next phase of analysis.

4.8.6.3 Conclusion

Analysis of the data during this phase has revealed several new observations. One observation is that learners find the formalisation of thinking quite difficult. This alludes to the ability to express their thinking using a formal notation, such as flowcharts or pseudocode. Another observation is that teachers often begin the teaching of programming with simple problems, having known solutions, which the learners can deconstruct. Another common approach, involving kinaesthetic activities, is identified as being of benefit to all learners, regardless of age. One issue deserving of further investigation revolves around the factors that might limit the acquisition of computational thinking skills. Currently, some of these issues are identified as literacy, specification design, and the programming tools themselves. The existing models have been revised to include subdivisions of programming skills and computational thinking skills.
4.8.7 **Milestone 7: Additional data and refinement of models**

In this phase, 4 more interviews are added to the data. They are all post-16 teachers at further education colleges. These will be coded into the existing data set free nodes, the second pass data hierarchical nodes, and the terminology hierarchy nodes. Areas to be explored in this phase include the difficulty in understanding experienced by learners when combining and nesting programming constructs, the benefits of introducing an interactive debugger early, and recognition of drivers of the pedagogy of computational thinking and programming.

Several interviewees have identified that combining and nesting simple constructs lead to problems in understanding. These sources are illustrated in the following figure. This suggestion can be explored further by reinvestigating the existing data set. A query is constructed to verify if additional references can be found.

<table>
<thead>
<tr>
<th>Name</th>
<th>Sources</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>Knowledge before creation</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Knowledge before understanding</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Map Computing To Bloom</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Nested instructions more difficult than sequence</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Pseudocode easier than coding</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pseudocode harder than flowcharts</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Recall is low</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Sequencing before creation of solution</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Sequencing before evaluation</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Sequencing easier than repetition</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Thinking to Flowchart is easy</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Thinking to pseudocode is difficult</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 28: Sources indicating nested instructions are difficult to understand

Revisiting the lexicon to find words associated with the concept of nesting instructions found the following frequencies: nested 2, Nested 1, Inside 1, inside 12, repetition 18, selection 22, loop 21, loops 21. Each of the associated
sources was revisited, particularly for references to the concept of nesting introducing difficulty. One source has been obscured for anonymity. This examination did not reveal any additional instances where the context referred to the difficulty of following logic. However, this is a valid point and can be added to the next model.

<table>
<thead>
<tr>
<th>Name</th>
<th>Created</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Programming Curriculum 08</td>
<td>12/04/2012 17:29</td>
</tr>
<tr>
<td>New Draft PoS for KS1 and KS2 05</td>
<td>22/03/2013 08:32</td>
</tr>
<tr>
<td>Procedures Functions in OCR 14</td>
<td>23/03/2013 14:23</td>
</tr>
<tr>
<td>Procedures Functions in OCR 16a</td>
<td>23/03/2013 14:23</td>
</tr>
<tr>
<td></td>
<td>05/05/2013 16:20</td>
</tr>
</tbody>
</table>

Figure 29: Sources using words associated with “nesting”

It seems perfectly reasonable to describe the process of understanding nested code, described above, as debugging. However, this assertion reveals more questions. Is debugging part of analysis or part of evaluation? Do you analyse a problem and evaluate a candidate solution? Do you analyse an algorithm to find errors?

In order to address these questions, another query could be constructed to find responses that refer to “analysis” and “debugging” or the different combinations thereof. Such a query is illustrated below.

Figure 30: Query to identify usage of analysis and debugging
After rereading this set of responses, there is still no identifiable indication of the placement of the term “debugging” in either the analysis or evaluation category. This outstanding issue will form a focus for further data collection and analysis.

One respondent suggests that it is important for learners to learn to read and understand code, using a debugger, before writing their own code. As this is a new concept emerging in the code, a review of the existing data set is required. To that end, all references already coded to the “debugging” node, illustrated below, are reviewed for the finer interpretation of an early introduction.

A new free node is created indicating early introduction of debugging techniques. Note that these techniques go beyond using a debugger in an integrated development environment to include physical activities for Key Stage 2. This generates a larger number of respondents who agree that an early
introduction to debugging is an effective teaching technique. This is illustrated in the following figure.

<table>
<thead>
<tr>
<th>Term</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>CT &amp; Problem Solving Relationship</td>
<td>31</td>
</tr>
<tr>
<td>CT and PS Must Begin Early and Cross-Curricular</td>
<td>21</td>
</tr>
<tr>
<td>CT Without Programming</td>
<td>3</td>
</tr>
<tr>
<td>Danger - Programming becomes an Objective</td>
<td>19</td>
</tr>
<tr>
<td>Debugging - Early Introduction</td>
<td>6</td>
</tr>
<tr>
<td>Difference - HE, FE, Secondary, Primary</td>
<td>13</td>
</tr>
<tr>
<td>Engaging First Language</td>
<td>22</td>
</tr>
</tbody>
</table>

Figure 33: Early introduction to debugging is an effective teaching technique

One respondent remarks that when tracking down errors in code, learners fail at being able to interpret and predict the behaviour of precisely what they have written. Using the debugger, as described above, can be beneficial, but its output is not always directly interpretable by students. This highlights additional terms that could be used to interrogate the data set.

Searching for “trace” or “dry-run” gives 4 additional responses, which can be further investigated for applicability. The results of this query are illustrated below. Two identifiers are obscured for anonymity. However, on revisiting, none of these responses refers to learners having particular difficulty in tracing or dry-running their logic.

**Table:**

<table>
<thead>
<tr>
<th>Name</th>
<th>In Folder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Documents\Interview Transcripts</td>
<td></td>
</tr>
<tr>
<td>Documents\Interview Transcripts</td>
<td></td>
</tr>
<tr>
<td>Teaching programming at school and</td>
<td>Teac</td>
</tr>
<tr>
<td>Documents\Interview Transcripts</td>
<td></td>
</tr>
<tr>
<td>Teaching programming at school and</td>
<td>Teac</td>
</tr>
</tbody>
</table>

Figure 34: Responses containing the terms trace or dry-run

A possible contributor to this inability to follow code could be learners’ propensity to copy and paste from exemplar materials. This concept could be investigated in the data set by creating a query using “copy/paste” combinations. The results of this query are shown below. This gave two
additional responses. Note identifiers are obscured for anonymity. One respondent, in further education, indicates that copy and paste is a hindrance to understanding. On the other hand, a Key Stage 2 teacher states that copy and paste is mitigation for some literacy issues. Pupils face fewer literacy challenges if they are simply allowed to change examples that they have been given.

Figure 35: Responses containing the terms copy and/or paste

Two respondents suggest that students cannot create flowcharts correctly until they have actually written some real code in a programming language. This is an unanticipated assertion. It may imply that an understanding of how the machine behaves, evidenced by writing some working code, is a prerequisite for analysis and design.

Additional education teachers employ what might be considered the classical introduction order of programming techniques. This order is sequence, selection, iteration, modularisation, and functions. Introduction of variables is often late due to language choice. Iteration usually presents an opportunity to introduce variables. Is this an instance of the tool limiting the ability to acquire or evidence computational thinking? This may be a valid and verifiable observation. Familiarity with the data leads to the conclusion that the teaching of programming and computational thinking relies heavily on the facilities and tools of the programming language chosen. This assertion may be verifiable in the dataset as it exists. However, it is not immediately apparent how this will help answer the original research questions. Therefore, it shall be omitted for the time being.

Two further education teachers indicate that issues with the way the course specifications are written and the ways in which evidence is accepted by the
exam boards are two of the driving forces behind the way they teach. One teacher admitted, “It drives our teaching.” A least one observer, affiliated with industry, has expressed similar ideas. This is an interesting observation, which may well be true, especially in the high stakes assessment environment at post-16. However, an associated question is even more applicable and interesting. Do exam board requirements drive the type of computational thinking skills that teachers choose to develop in learners?

4.8.7.1 Model generation and discussion
This phase of data collection and analysis has only affected the existing models in minor ways. The first model illustrates the addition of the respondents who indicated that the nesting of programming constructs introduces complexity and a higher level of difficulty in tracing or understanding for the learners.
Chapter 4: Method

Figure 36: Nested instructions are difficult for learners to follow

The following diagram illustrates consideration for those responses indicating that an early introduction to debugging is an advantage and that flowcharts must follow some attempt at coding. Remarks concerning the layout of the models have not changed from the previous sections.
Figure 37: (a) Introduction of debugging early and flowcharts after coding
Figure 38: (b) Introduction of debugging early and flowcharts after coding
4.8.7.2 Next actions
This phase of the analysis has highlighted some new issues, as discussed above. These issues can be addressed in the next phase of data collection and analysis. The interview questions can be amended to take account of these new lines of inquiry.

- Construct a question to investigate the reported inversely proportional relationship between complexity of logic with nesting and the ability to follow that logic.
- Pursue the possibility of a difference between analysis and evaluation. Do you analyse a problem and evaluate a solution? If so, then which should be ascribed to “debugging”?
- Update interview questions to reflect new findings and requirements as indicated above. New interview schedule can be found in “Appendix 4 Interview revision (ah)”.

4.8.7.3 Conclusion
Analysis of the data indicates that combining and nesting of programming constructs makes understanding logic much more difficult for learners. A further investigation into the concept of debugging, both as a term and a process, has revealed that it may be advantageous to learners if they are introduced to using a debugger early in their programming. Further investigation identified that copy/paste may be viewed as a contributor to the inability to follow code. However, there are opposing views. One teacher views copy/paste as a hindrance to understanding while another views it as an advantage in overcoming weak literacy skills. A surprising result, in this phase, is the suggestion that learners cannot create flowcharts until after they have some experience of actually writing programming code. Currently, it is unclear if it is the actual code writing that is the prerequisite or an understanding of the underlying behaviour of the machine. Several respondents have observed that there are external drivers for the teaching of programming and computational thinking. The most agreed upon drivers are the examination board specifications.
4.8.8 Milestone 8: Model clarification

In this phase of the analysis, several new ideas are explored and a previous idea is revisited. The new routes include the ideas that learners often need explicit explanation of programming techniques, that stakeholders, other than teachers or learners, actually drive the pedagogy, that inability to understand and use terminology limits the development of computational thinking skills, and that abstracting data structures is more difficult than writing code. The revisited idea involves reviewing the need for learners to have an understanding of how a notional machine works.

The first new idea arises when teachers fail to point out to the learners explicitly that writing functions is a mechanism they can use when creating their own code. By not addressing this explicitly in a group context, teachers often end up doing many one-to-one sessions with learners. By not identifying an abstraction (the function) and not applying it elsewhere (generalisation), an opportunity is missed in the pursuit of computational thinking skills. In this case, teachers often suggest that learners need “more practice” in the technique. Often the words “more practice” really mean ask the learner to keep doing the same thing until the idea “clicks” in his or her head. However, perhaps classroom practitioners should be using explicit explanation to ensure that the idea “clicks”.

This need for explicit statements is worth pursuing by creating a new query for “practice/practise” and by reviewing those responses already coded in the “practice problem solving” node, Figure 39.

The resulting query revealed 49 different respondents. Each will be reviewed and recoded if applicable. The concept of practice does seem to tie with an idea clicking in a learner’s head. It is often observed in response to questions about the types of activities that promote the development of computational thinking and problem solving skills. This new concept is represented by a new node “Practice == Ideas Clicking”, as illustrated below, Figure 40.
## Chapter 4: Method

<table>
<thead>
<tr>
<th>Name</th>
<th>Sources</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polyva Problem Solving</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Practical - HW Based</td>
<td>8</td>
<td>19</td>
</tr>
<tr>
<td>Practice == Ideas Clicking</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td><strong>Practice Problem Solving</strong></td>
<td><strong>26</strong></td>
<td><strong>37</strong></td>
</tr>
<tr>
<td>Problems - Solution Known By Teacher</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Problems - Solution Unknown By Teacher</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Real World Comparisons</td>
<td>22</td>
<td>66</td>
</tr>
</tbody>
</table>

Figure 39: Practice problem solving references

<table>
<thead>
<tr>
<th>Name</th>
<th>Sources</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Practical - HW Based</td>
<td>8</td>
<td>19</td>
</tr>
<tr>
<td>Practice == Ideas Clicking</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>Practice Problem Solving</td>
<td>26</td>
<td>37</td>
</tr>
<tr>
<td>Problems - Solution Known By Teacher</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Problems - Solution Unknown By Teacher</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 40: Practice promotes ideas clicking in learners’ heads
Chapter 4: Method

The second new idea to explore is that stakeholders, other than teachers or learners, actually drive the pedagogy. These stakeholders are identified as examination boards and government. One further education teacher asserts that these stakeholders “pressurise teachers”. The idea of external stakeholders influencing the teaching in the areas of programming, computational thinking, and problem solving has previously been identified in the data and coded, as illustrated in the following diagram, Figure 41.

The idea of external drivers needs to be explored further to refine the categories and identify specific drivers of pedagogy, as opposed to, for example, influences to specification content. This can be done by creating a new query using a stemmed search for “drive” or “pressure” and applying it across those sources already identified in the node above. This combination has resulted in a list of sources, as illustrated below in Figure 42. Some names have been obscured for anonymity. Each source has been reviewed and recoded, if applicable, to a node representing the drivers of pedagogy. These include classroom dynamics and maturity of students, league tables, time pressures, and the assessment requirements of examination specifications.
### Figure 41: External stakeholders drive pedagogy

<table>
<thead>
<tr>
<th></th>
<th>22</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exam Board Influence</strong></td>
<td>50</td>
<td>118</td>
</tr>
<tr>
<td>Explicit Explanation</td>
<td>11</td>
<td>22</td>
</tr>
<tr>
<td>Fashionable Language Choice</td>
<td>21</td>
<td>37</td>
</tr>
<tr>
<td>Finding a solution</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Formalisation of thinking - Most Difficult</td>
<td>8</td>
<td>28</td>
</tr>
<tr>
<td>If you can’t talk about it, you don’t understand it</td>
<td>23</td>
<td>57</td>
</tr>
<tr>
<td>Increasing Difficulty</td>
<td>24</td>
<td>68</td>
</tr>
<tr>
<td><strong>Industry Influence - Negative</strong></td>
<td>42</td>
<td>68</td>
</tr>
<tr>
<td>Language - Irrelevant</td>
<td>39</td>
<td>57</td>
</tr>
<tr>
<td>Language - Progression</td>
<td>44</td>
<td>70</td>
</tr>
</tbody>
</table>
### Chapter 4: Method

<table>
<thead>
<tr>
<th>Name</th>
<th>In Folder</th>
</tr>
</thead>
<tbody>
<tr>
<td>85</td>
<td>Documents\Computing Programme of Study</td>
</tr>
<tr>
<td>(16) Defining a Pedagogy of Programming</td>
<td>Documents\Defining a Pedagogy of Programming</td>
</tr>
<tr>
<td>Dijkstra's Views on Programming</td>
<td>Documents\Dijkstra's Views on Programming</td>
</tr>
<tr>
<td>Dijkstra's Views on Programming</td>
<td>Documents\Dijkstra's Views on Programming</td>
</tr>
<tr>
<td>Towards a Research Agenda 03</td>
<td>Documents\Towards a Research Agenda</td>
</tr>
<tr>
<td>Research Questions in Computing</td>
<td>Documents\Interview Transcripts</td>
</tr>
<tr>
<td></td>
<td>Documents\Interview Transcripts</td>
</tr>
<tr>
<td></td>
<td>Documents\Interview Transcripts</td>
</tr>
<tr>
<td></td>
<td>Documents\Interview Transcripts</td>
</tr>
<tr>
<td></td>
<td>Documents\Respected Sources</td>
</tr>
</tbody>
</table>

Figure 42: Drivers of pedagogy
The third new idea to explore is a search for those factors that limit the development of problem solving and computational thinking. One teacher identifies with the “if you can’t talk about it, you don’t understand it” node. However, he concedes that his students can “do” programming. So, what skills are limiting the students’ ability to express themselves? He suggests that it is an inability to use the terminology of programming, like “parameter”, correctly and to see the connections between building blocks taught, functions, and their applicability to the students’ own problem space, game development. On the other hand, another teacher reports that students can often explain, verbally, their logic and problem solving, but cannot make the jump to translating it to pseudocode or flowcharts.

The 23 responses currently classified under the “if you can’t talk about it” node will require some further investigation to see if reasons for this limitation are suggested. From this review, there appear to be four different categories of contributors, as illustrated in the following figure. The Cognitive Overload category somewhat overlaps the By Tools category. The cognitive overload is often identified as being associated with the use of an IDE or with the syntax of a language.

![Figure 43: Factors which may limit computational thinking skills acquisition](image)

The fourth new idea to explore is in response to a teacher who identified that learners find data structure abstractions more difficult to understand that writing code, even event-driven code. This mention of data structures as abstractions is in contrast to functions as abstractions, which is more frequently mentioned by respondents in the context of programming.
Chapter 4: Method

Data structures, as abstractions, being difficult to understand can be investigated in the data set by working with queries. Stemmed searches will not work with “data structures” because it is considered a phrase. Therefore, a simple, but long, query can be constructed to find all equivalents to the phrase “data structure”. This is illustrated below.

Figure 44: Searching for data structure abstractions

This query identifies 44 different responses, illustrated below. One item is obscured for anonymity. These can be further considered to identify if the context yields information to influence a hierarchy of complexity or difficulty. After further reading, 8 of these responses could be interpreted as suggesting that abstraction of data structures is more difficult than writing code or algorithm design.
Chapter 4: Method

The fourth idea reflected in the data is the many opportunities that support the idea that learners need an internal model of the machine before they completely understand the process of programming. This was reflected in the last milestone by the respondents who suggested that learners could not write flowcharts without having written some working programming code. This same idea of understanding the notional machine can be found in the work of du Boulay (1989). As interesting as this idea is, pursuit of it may not readily contribute to answering the original research questions.

4.8.8.1 Model generation and discussion

The node “abstraction harder than decomposition” has been revisited. Responses represented there have been reinterpreted and reallocated to more appropriate and expressive concepts.

The model representing the teaching programming terminology mapped to Bloom’s Cognitive Domain appears to be holding consistent. This model is shown below. Concepts coded to nodes indicating relationships of ordering fit the six levels well. The concepts are expressed in node names using a noun-adverb-noun tuple. The noun-noun pairs map to the familiar terms in Bloom’s levels. The adverb places the concept at a level relative to the other nodes, thus illuminating a form of hierarchy.

Figure 45: Results of searching for data structure or equivalents

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedures Functions in OCR 15</td>
</tr>
<tr>
<td>Teaching OOP vs Teaching Functional Programming</td>
</tr>
<tr>
<td>Computational Thinking is Informational Thinking 05</td>
</tr>
<tr>
<td>Dijkstra’s Views on Programming 14</td>
</tr>
<tr>
<td>Procedures Functions in OCR 09</td>
</tr>
<tr>
<td>Research Questions in Computing Education</td>
</tr>
<tr>
<td>Does the Programming Language Matter 24</td>
</tr>
<tr>
<td>Teaching programming at school and at university 02</td>
</tr>
<tr>
<td>New Programming Curriculum 08</td>
</tr>
<tr>
<td>44 Items</td>
</tr>
</tbody>
</table>
Figure 46: (a) Teaching programming corresponds to Bloom's cognitive domain
Figure 47: (b) Teaching programming corresponds to Bloom’s cognitive domain
Chapter 4: Method

The model representing divisions of teaching programming and computational thinking, illustrated below, also appears to be holding consistent. This model represents the order in which programming concepts are commonly taught across key stages. The noun-adverb-noun tuples used for naming the conceptual nodes is the terminology expressed by the participants. This terminology contains vocabulary familiar to teachers and learners, including sequence, debug, and flowcharts. There is overlap with terms found in the context of computational thinking, including decomposition, abstraction, and evaluation.
Figure 48: (a) Subdivision of programming skills and computational thinking skills across the cognitive domain
Figure 49: (b) Subdivision of programming skills and computational thinking skills across the cognitive domain
The formalisation and transformation of logic model has also not changed. The relationships illustrated below are not disputed in the data. The concept that learners find the “formalisation” of their thinking to be difficult is upheld by several respondents. The transition diagram, indicating difficulty of moving between notations could indicate a preferred order of learning, thereby influencing order of teaching.
Flowcharts and Pseudocode are evidence of decomposition and abstraction.

Figure 50: Formal notations and difficulty of moving between them
The model illustrating the use of computational thinking terminology has been reordered and labelled indicating increasing levels of difficulty. From the responses, it is possible to determine a relationship between abstraction and decomposition, and algorithm design and decomposition. Although decomposition appears lower in the teaching programming model than abstraction, it is perceived to be of a higher level of difficulty. This is the same for algorithm design and decomposition. This appears to be the case across all key stages explored. However, this does not indicate a relationship between algorithm design and abstraction. From the data, the ability to design an algorithm, given named pieces, is not identified as difficult. On the contrary, ordering pieces is often an intervention for lower ability pupils.
4.8.8.2 Next actions

At this stage, two outstanding issues require further investigation. These deal with specific terminology that is often used in defining computational thinking, such as logical thinking, algorithmic thinking, and generalisation. In order to resolve these issues, the following actions need to be undertaken.

- Review data for use of terms “logical thinking”, “algorithmic thinking”, etc. in pursuit of a definition of computational thinking.
- Generalisation is not identified in the responses very often or very well. Double check the data for instances of this concept.

As the models presented above are beginning to settle, now is also an opportune moment to step back from the micro-analysis of the data to look at
the overall state of the research project as a whole. The following items need to be addressed as next actions.

- Check to see if the existing data supports sensible answers to the original research questions.
- Determine if the evolving model(s) can be refined and that they make sense. Do they show something new about the relationship between problem solving, computational thinking, and the teaching of programming?
- Review previous milestones to confirm that all next actions have been thoroughly addressed.
- Determine if more data needs to be collected to help inform the models.

4.8.8.3 Conclusion
In this phase of the analysis, several new ideas have been explored. Analysis of the data supports the notion of practice being equated with ideas clicking in the heads of learners. This is in contrast to explicit identification provided by a more knowledgeable teacher or peer. In pursuing the concept of examination boards influencing pedagogy, further stakeholder influence has been identified to include government policy, as reflected in league tables, and industry, as reflected in programming language choice. Factors that limit the acquisition of computational thinking and programming skills are pointed out by some respondents. These factors include an inability of learners to use correctly the terminology of programming and an inability of learners to explain the logic they have in their heads using a suitable notation. The identification of abstraction being difficult for learners is explored further in this phase. Exploration of the data reveals that creating and using data structure abstractions are more difficult for learners than creating and using abstractions of functionality, such as subroutines.

4.8.9 Milestone 9: Taxonomy of terms
At this time, the raw data set consists of the responses from 174 individual respondents, including 10 one-hour interviews, 42 on-line questionnaires, and 122 individual participants from the community of practice. The outstanding issue to address, in this phase, is whether the evolving models can be refined to
show a relationship between problem solving, computational thinking, and the teaching of programming. The existing models can, indeed, be redrawn to reflect these relationships, while revealing some surprising details.

### 4.8.9.1 Model generation and discussion

In a first attempt to model a hierarchy of computational thinking skills, the information found in the data has been simply mapped to Bloom's Taxonomy, as illustrated below. The previous versions of the models have revealed relationships between some of the computational thinking terminology, the terminology used in the teaching of programming, and the Cognitive Domain of Bloom's Taxonomy.

<table>
<thead>
<tr>
<th>Bloom’s Taxonomy, Cognitive Domain</th>
<th>Computational thinking terms from data (best fit)</th>
<th>Teaching programming terms (teaching order)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evaluation</td>
<td>evaluation</td>
<td>test, evaluate</td>
</tr>
<tr>
<td>Synthesis</td>
<td>algorithm design</td>
<td>create programs, algorithm design</td>
</tr>
<tr>
<td>Analysis</td>
<td>abstraction of data, abstraction of functionality, decomposition</td>
<td>discriminate, decompose, abstract</td>
</tr>
<tr>
<td>Application</td>
<td>generalisation</td>
<td>use programming constructs</td>
</tr>
<tr>
<td>Comprehension</td>
<td>structures</td>
<td></td>
</tr>
<tr>
<td>Knowledge</td>
<td>constructs, facts, types</td>
<td></td>
</tr>
</tbody>
</table>

Table 5: First mapping of terminology against an existing hierarchy

From this interpretation, it is clear that the order in which programming is taught coincides with the levels of complexity in the cognitive domain. The computational thinking terms that have been included in this best fit mapping are those terms used most frequently by the respondents. At this point, the term “generalisation” is interpreted in the broad sense of “where have I seen this type of problem before?” It is presumed that the ability to decompose
problems, and abstract data and functionality requires some skills from the application, comprehension, and knowledge levels. The data does not distinguish specific computational thinking skills attributable to these levels, only relative levels between the computational thinking skills. The respondents tend to conflate comprehension, application, and analysis into the equivalent computational thinking term, analysis. There is anticipated to be some movement in this mapping as further analysis is done on the data set.

It is possible to distinguish, in the data, relative relationships between the computational thinking terms. These have been expressed in Figure 51: Relationships between computational thinking terms. A logical next step is to attempt to represent the level of difficulty in the computational thinking terms in a linear fashion. This is done in the following illustration. Included here are also examples of types of programming activities with which different learners might engage.
<table>
<thead>
<tr>
<th>Perceived Difficulty</th>
<th>Computational Thinking</th>
<th>Year 6 (age 11)</th>
<th>Year 10 (age 15)</th>
<th>Year 13 (age 18)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decomposition</td>
<td>Know pieces</td>
<td>Know pieces</td>
<td>Know pieces</td>
<td></td>
</tr>
<tr>
<td>Abstraction – data</td>
<td>List</td>
<td>1 Dimensional arrays</td>
<td>Linked lists, multi-dimensional arrays, objects</td>
<td></td>
</tr>
<tr>
<td>Abstraction – functionality</td>
<td>Given to pupils, as “jump” code</td>
<td>Functions, procedures, subroutines</td>
<td>Methods</td>
<td></td>
</tr>
<tr>
<td>Generalisation</td>
<td>Ball → sprite</td>
<td>Sort numbers → sort strings</td>
<td>1 record → record set</td>
<td></td>
</tr>
<tr>
<td>Algorithm design</td>
<td>Simple copy/paste or ordering</td>
<td>Sequencing, simple flow control, library function usage</td>
<td>Unrestricted</td>
<td></td>
</tr>
<tr>
<td>Evaluation</td>
<td>Does it work?</td>
<td>Plan tests and simple test data</td>
<td>Design test data, consider original objectives</td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Perceived level of difficulty of computational thinking skills
4.8.9.2 Next actions

There are still 5 issues that are not yet resolved. These are set out in the following list and will be explored for the next milestone.

- Where does the term debugging fit into the computational thinking model?
- The term generalisation is not often mentioned in the data in any form, except for reference to “Where have I seen a problem like this before?” Perhaps some judicious use of queries could track this down.
- Is there a reason for decomposition being so difficult? Is it because it’s taught first? Is it because there is a lack of experience in the process?
- The data has not yet revealed a relationship or a distinction between the terms analysis and evaluation. Do you analyse a problem and evaluate a solution? Does analysis imply studying for understanding of functionality? Does evaluation imply studying for understanding of efficiency and applicability?
- Review data for use of terms “logical thinking”, “algorithmic thinking”, etc. in pursuit of a definition of computational thinking.

4.8.9.3 Conclusion

In this phase of the analysis, the focus has been on the emerging models. Relationships between the Cognitive Domain of Bloom’s Taxonomy, various computational thinking terms, and the teaching order of programming skills are identified. The teaching order maps directly to Bloom’s levels. The computational thinking skills fit into the top three levels. The perceived level of difficulty of various computational thinking skills has also been modelled.

4.8.10 Milestone 10: Final issues

Several issues are still outstanding from the previous analysis phases. Each of those issues will be addressed in this section, in preparation for another attempt at model generation. These issues involve debugging, generalisation, analysis versus evaluation, types of thinking, and decomposition.

The first issue to resolve is the positioning of debugging within the computational thinking model. In order to address this, a new query was generated to perform a text search for any term relating to debugging, such as
mistake, bug, or debug. Execution of this query resulted in a set containing 36 individual responses, illustrated below. Several items are obscured for anonymity. After careful rereading, none of these responses helped to identify a position for debugging in the model.

The next step in pursuing the idea of debugging is to revisit those responses already coded to nodes associated with debugging. From reviewing the nodes already classified, the following relationships have been identified. It would appear that debugging is not equivalent to tinkering, that it is a backwards process from the bug to the source, that strategies are needed at a high-level, that a visual debugger is useful in aiding understanding, and that access to a representation of the logic is useful as a reference. This is illustrated in the following diagram.
Because respondents often refer to debugging as a type of analysis, revisiting Bloom’s Taxonomy may provide further enlightenment. The description of the process of analysis, found therein, is an analogous description for the process of debugging. Bloom (1956) divides analysis into three types: elements, relationships, and organisational principles. These three subdivisions map to the items under investigation during the debugging process. Elements map to the decomposed parts of the problem, data or functionality; relationships map to the interactions between data and or functionality, and organisational principles map to the flow of control and data structures. Often analysis is interpreted to mean developing a solution by using logic. An extension of this is the definition of debugging which means finding similar details in an existing solution. Therefore, debugging sits squarely on the same level as analysis, incorporating all three sublevels as ascribed by Bloom.

According to the data, debugging is taught or practised using dry runs, trace tables, and apprenticeship models. It can also be represented as a list of simple items to check as part of the process. Debugging does not invoke skills different from those employed during the analysis phase. The same skills are applied in a slightly different context.

Figure 53: Characteristics of debugging
Chapter 4: Method

All key stages, primary through higher education, exhibit the same lack of skills when it comes to debugging. Many respondents attribute this to a lack of problem-solving skills rather than specific debugging skills.

The second issue to resolve is the positioning of generalisation within the computational thinking model and why this term is not used frequently by the respondents. Generalisation is the ability to transfer solutions or parts of solutions to new or similar problems.

Slightly modifying the definition of generalisation to include the ability to transfer smaller fundamental parts or even techniques between contexts reveals additional relevant references in the data. Generalisation of strategies has been identified by some respondents, for example recognising that ordering is important in some solutions. This is exemplified when a pupil learns that mending a bicycle puncture requires ordering and recognising that same need in another context, such as installing new software that requires prerequisites. Another example indicates that once having identified that most information can be retrieved via Google, that strategy can be applied to most problems. Generalisation of concepts has also been identified. These examples extend to the ability to understand the fundamentals of one programming language being applied to another and to the behaviour of number systems, such as denary and binary. The key concept identified in the data that is associated with generalisation is the application of knowledge from one domain or context in another. From this, it would be logical to place generalisation on the same level as apply in Bloom’s model. In support of this, Bloom purports that “The effectiveness of a large part of the school program is therefore dependent upon how well the students carry over into situations applications which the students never faced in the learning process.” (Bloom 1956, p. 122). The latter part of this statement, as a definition of generalisation, is upheld by the views of the respondents. Although the term generalisation is not often used specifically, there are many examples of equivalencies to be found in the data set, especially in areas outside those of moving programming solutions from one context to another. These examples support the conclusion to place generalisation on the same level as application in Bloom’s Taxonomy: Cognitive Domain.
The third issue to address is the finding that decomposition is difficult for many learners. The data set may hold some indication of why this is true. Reviewing the data implies that decomposition is seen as a broader problem-solving skill with which students, coming to further education, should already have some experience. However, as reported by respondents, this appears not to be the case. In a broader sense, learners may exhibit the ability to decompose a known process or solution into its component parts, but not be able to decompose an unknown process into its component parts. This could be tied further to the idea of known problem/solutions being more easily understandable by learners. This could explain why problems similar to “guess the number” are less of a challenge to break down than “sorting a list”. It is one reason that teachers often introduce initial problems that are very simple with known solutions. Even known solutions cannot be written without decomposition having been performed. On the other hand, building up functionality incrementally is a concrete example of implementing decomposition. Ownership of the decomposition is in question. Leading the learner on a step-by-step journey through the teacher’s decomposition logic may not be the best approach. Recall that creating algorithm designs from component parts is not reported as difficult for learners. Identifying the component parts is reported as difficult. A lack of experience in the ability to decompose is identified as the problem in this case. Unfortunately, decomposition must be done before a problem solution can be created. There appear to be at least two contributing factors making decomposition difficult for learners. The first is the fact that learners do not have much practical experience in decomposing problems, of any type. The second is that decomposition necessarily precedes abstraction and algorithm design, except for very young learners.

The fourth issue to pursue is the relationship between the terms analysis and evaluation. Although this has been an outstanding issue through several analysis phases, it is now clearer how these terms can be distinguished. The use of the terms in the data set is often phrased in terms of a problem or a solution. A problem is analysed; a solution is evaluated. This use is consistent with the levels of complexity set out in Bloom’s Taxonomy. Evaluation is placed
at a higher level than analysis. One type of evaluation suggested by Bloom is “… concerned with tests of the accuracy of the work as judged by consistency, logical accuracy, and the absence of internal flaws” (Bloom 1956, p. 186). This same ordering, not of complexity, but of activity, is reflected in the computational thinking literature (National Research Council 2010, Wing 2006) where evaluation is discussed as an activity taking place at the end of a process or program development, where items such as correctness and efficiency are considered. These elaborations of the term evaluation provide some distinction between the terms. Based on these definitions and the way in which the terms are used by the respondents, it is consistent to assert that a problem is analysed while a solution may be evaluated.

The issue of whether the terms “logical thinking” and “algorithmic thinking” actually contribute to the definition of computational thinking requires further investigation. Queries have been constructed to search the data for these specific terms. Searches for the terms yield eight possible responses each, illustrated below. Several items are obscured for anonymity.
Unfortunately, after further reading, none of these responses actually supports the idea that logical thinking or algorithmic thinking actually defines computational thinking. They do indicate, as much of the literature does, that both types of thinking are characteristic of those who perform computational thinking.

Figure 54: Responses incorporating logical or algorithmic thinking
Chapter 4: Method

4.8.10.1 Model generation and discussion

Having addressed each of the outstanding issues from previous phases of the analysis, it is time to bring together appropriate individual models to create a more informative model of the relationships between computational thinking, the teaching of programming, and Bloom’s Cognitive Domain.

A first attempt at representing this relationship is presented in the rough sketch, directly below. An unsurprising result, as discussed above, is that the order in which programming skills are taught directly reflects the order of the levels in Bloom’s Cognitive Domain. However, the perceived levels of difficulty of the computational thinking skills when mapped to Bloom’s Cognitive Domain are a reversal of the expected order. This is more clearly demonstrated in the second figure.
Figure 55: Relationship between three concepts
### 4.8.10.2 Conclusion

In this final phase of the analysis, several issues still outstanding from previous phases have been resolved. These include debugging, generalisation, analysis versus evaluation, and decomposition. Debugging, based on the data available and the literature review, fits firmly into the analysis level of Bloom's Taxonomy, Cognitive Domain. It is most frequently taught using the techniques of tracing or dry-running. Although the term “generalisation” is not used frequently by the respondents, examples of it can be found in the data. These include the generalisation of strategies and generalisation of behaviours to alternative contexts, not just the generalisation of solutions. The identification of decomposition as being the most difficult computational thinking skill to master has been further explored. It is unclear why this task is so difficult for learners, although lack of previous experience and the fact that it is often the first task to
be taught may be contributors. In determining the relationship between analysis and evaluation, Bloom’s Taxonomy brings clarification to the respondents’ data and supports the assertion that a problem is analysed and a solution is evaluated. An attempt to determine if algorithmic thinking or logical thinking defines computational thinking was unproductive. These terms are only used to identify characteristics of computational thinkers, not to define computational thinking.

4.9 Final model

A more easily interpretable image to represent the model of the relationship between the Cognitive Domain of Bloom’s Taxonomy, the teaching of programming, and computational thinking skills is presented below. The levels of complexity of the Cognitive Domain are illustrated vertically, with the lower levels at the bottom of the pyramid. The order in which programming skills are taught is also illustrated vertically, with the earlier concepts appearing at the lower levels. A taxonomy of computational thinking skills is represented on the right of the diagram. This taxonomy is presented with the terms positioned in line with the levels of the Cognitive Domain. However, the order of perceived difficulty should be read in the opposite direction to the pyramid. In other words, learners exhibit fewer difficulties with evaluation than with decomposition.
Figure 56: Final relationship model
4.10 Final data set statistics

It is only at this point, in accordance with the grounded theory approach, that theoretical saturation of the data set has been achieved. Descriptions of the growing data set have been included in previous sections. This section presents a final view of the entire dataset.

![Total Respondents](image)

**Figure 57**: Total number of respondents
Chapter 4: Method

Figure 58: Respondents' reported occupations

Figure 59: Respondents' reported teaching level
Chapter 4: Method

Figure 60: Total number of words per data collection instrument

Figure 61: Average number of words per response
In conclusion, this section has described the overall method employed in this research. This study is based on grounded theory, as described by Strauss and Corbin (1998). Participants are purposively selected for their knowledge of and interest in the topics of computational thinking, problem solving, and programming. Participants are adults and can give their own consent for the use of their data. Although anonymity cannot be guaranteed, confidentiality can be assured. Proposals for the protection of participants’ data have been described and meet the ethics requirement of the university. The sources of data, identified in this study, include an online questionnaire, an online community of practice forum, and a semi-structured interview. Arguments for the reliability and validity of these instruments have been presented in this section. As evidenced in the previous sections, the instruments have generated data relevant to the research questions. The collected data has been coded for concepts and collected into appropriate categories, in accordance with the grounded theory approach. A model of the relationships between computational thinking and the teaching of programming has been derived.
based upon an analysis of the evolving data set. Further discussion of the findings and the resulting model are presented in the following sections.
Chapter 4: Method
Chapter 5. Discussion

This section presents an overall discussion of the research results. The Computational Thinking Taxonomy is detailed. Justification is provided for the ordering, from less to more complex, of the terms generalisation, decomposition, functional abstraction, data abstraction, algorithm design, and evaluation. The ordering parallels the top three levels of Bloom’s Taxonomy: Cognitive Domain (1956). A hierarchy of difficulty of computational thinking skills, as indicated by the participants, is also detailed. Justification, with reference to the supporting data, is provided for the ordering, from less to more difficult, of the terms evaluation, algorithm design, functional abstraction, data abstraction, and decomposition. This ordering does not parallel that of Bloom’s Taxonomy: Cognitive Domain (1956). Proposed justification for this observation is presented. In addition, the Computational Thinking Taxonomy is also discussed in terms of both Bloom’s Taxonomy (1956) and the revised Taxonomy (Anderson et al. 2001). Justification for the alignment with Bloom’s original in preference to the revised Taxonomy is presented. Similarly, the lack of observed specific computational thinking skills associated with the knowledge and comprehension levels of Bloom’s Taxonomy is discussed in this section. Justification for the gap is proposed. At the end of this section, the original research questions are revisited and addressed with individual responses.

5.1 Taxonomy of computational thinking skills

One of the objectives of this research is to define a taxonomy of computational thinking skills. Revisiting the results of the data analysis has afforded this development. The analysis, as defined in the methodology section, reveals the existence of the following computational thinking skills. The order in which they are presented indicates a best fit with Bloom’s original taxonomy. The data identified computational thinking skills aligned with the analysis, synthesis, and evaluation levels of the Cognitive Domain of Bloom’s Taxonomy.
Chapter 5: Discussion

<table>
<thead>
<tr>
<th>Bloom’s Taxonomy, Cognitive Domain</th>
<th>Computational thinking skills</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evaluation</td>
<td>evaluation</td>
</tr>
<tr>
<td>Synthesis</td>
<td>algorithm design</td>
</tr>
<tr>
<td>Analysis</td>
<td>abstraction of data, abstraction of functionality, decomposition</td>
</tr>
<tr>
<td>Application</td>
<td>generalisation</td>
</tr>
<tr>
<td>Comprehension</td>
<td></td>
</tr>
<tr>
<td>Knowledge</td>
<td></td>
</tr>
</tbody>
</table>

Table 8: Computational thinking skills to cognitive domain mapping

Bloom’s original taxonomy was ordered from simple to complex and from concrete to abstract (Krathwohl 2002). The same order can be observed in the table above. Cognitively, the computational thinking skills become more abstract as their position moves up the associated levels of Bloom’s Taxonomy. This fit is anticipated by Bloom, who asserts “… that essentially the same classes of behaviour may be observed in the usual range of subject-matter content, at different levels of education …” (1956, p. 12).

Although computational thinking is not defined by the activity of programming, a close association with progression in programming has been suggested by Lister (2000).

“I merely make the following broad observations, specific to the teaching of elementary programming: the lower two levels emphasise the skill of reading and comprehending code, the intermediate two levels emphasise the writing of fragments of code, but within a well defined context, and the upper two levels emphasise the writing of complete non-trivial programs … students should first be taught to read programs before they write programs.” (p. 159).
These observations confirm the existence of a taxonomy of computational thinking skills that aligns with an established educational model.

### 5.2 Hierarchy of difficulty of computational thinking skills

When these same six computational thinking skills, identified above, are further explored, the respondents reveal a divergence from the Bloom model. This occurs when considering the difficulty of mastering each of these skills. Respondents report that one of the easiest skills to master is that of evaluation, while the most difficult skill to master is that of decomposition. On close inspection, the order of difficulty is a reversal of the original levels of evaluation, synthesis, and analysis in the Cognitive Domain of Bloom’s Taxonomy. This is reflected in the illustration below. This reversal at first seems implausible. However, the respondents themselves reveal several possibilities for why this reversal has been observed. Each of the computational thinking skills presented in the second column are discussed further in the following sections, in terms of examples found in other subject areas, examples of difficulties evidenced by learners, and identification of possible contributing factors to this perceived difficulty.

<table>
<thead>
<tr>
<th>Bloom’s Cognitive Domain</th>
<th>Computational Thinking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evaluation</td>
<td>Evaluation</td>
</tr>
<tr>
<td>Synthesis</td>
<td>Algorithm design</td>
</tr>
<tr>
<td>Analysis</td>
<td>Abstraction functionality,</td>
</tr>
<tr>
<td></td>
<td>Abstraction data,</td>
</tr>
<tr>
<td></td>
<td>Decomposition</td>
</tr>
<tr>
<td>Application</td>
<td>Generalisation</td>
</tr>
<tr>
<td>Comprehension</td>
<td></td>
</tr>
<tr>
<td>Knowledge</td>
<td></td>
</tr>
</tbody>
</table>

Table 9: Computational thinking skills and hierarchy of difficulty
Chapter 5: Discussion

5.2.1 Evaluation
Evaluation is seen as a broad skill, practised and applied in many curriculum areas across all key stages. English pupils are asked to evaluate their essays; chemistry students are asked to evaluate their practical experiments; ICT pupils are asked to evaluate their digital artefacts. The concept is familiar to all learners.

The rigour with which evaluation is performed by these learners may not be consistent or deep. For example, this can be observed in mathematics classes when learners calculate clearly inaccurate answers, but do not recognise them as such. Imprecise evaluation, as a computational thinking skill, can lead to efficiency and performance issues. This has been identified by responses in the data indicating that the skill of evaluation as a component of computational thinking should include “…a criticism of weaknesses in the solution, and … why it is better or worse than some alternative approaches.” This is in addition to the original requirement by Wing (2006) for the computational thinker to make trade-offs, by evaluating the use of time and space, power and storage.

Therefore, evaluation as a broad skill may be familiar to and practised by learners. However, the shallowness to which evaluation is executed may lead both learners and respondents to the perception that evaluation is the least difficult of the computational thinking skills.

5.2.2 Algorithm design
Algorithm design is again practised in many areas, especially as an ordering exercise. This skill is also developed in all key stages. Science students are asked to design simple electric circuits for switching lights; English pupils are asked to write sensible play dialogue; ICT pupils are asked to create Scratch animations. The skill is practised by all learners, whether or not the actual task is described as algorithm design.

From the programming perspective, classroom tasks, reported by respondents, cover a broad range of activities. Younger learners use Bee-Bots in play and order instructions for making a jam sandwich. For slightly older learners, as reported by a respondent, “it’s possible to give them the pieces of the jigsaw and say, ‘now, let’s get these into the right sequence’.” For pupils in secondary
school, one respondent employs a profit and loss spreadsheet scenario where pupils “… have to logically sequence the calculations and evaluate whether the result is sensible.” Students in further education are still challenged by algorithm design. One response states that “… with most groups at least, they say we need this, we’re going to need that … these ingredients. They won’t necessarily get them in the right order to start with.”

These examples cover a broad range of tasks attributable to algorithm design. It may be that the cognitive process evidenced in the instances where the learners are given the steps of the algorithm to order should be assigned to a slightly lower level. In defining synthesis, Bloom (1956) acknowledges that the difference between synthesis and the categories of comprehension, application, and analysis allows “… the possibility that they involve working with a given set of materials or elements which constitutes a whole …” (p. 162). However, in instances similar to the last two examples, students must first have decomposed the problem, identified and created abstractions of function and data, and designed a solution that did not previously exist. This property of uniqueness is mandated by Bloom (1956) who includes in the definition of synthesis that “… the student must draw upon elements from many sources and put these together into a structure or pattern not clearly there before.” (p. 162).

Therefore, algorithm design as a skill may be familiar to and practised by learners, in several subject areas. From a programming perspective, its description as a less difficult skill is understandable, especially in instances where learners are given the parts of an algorithm to order. If respondents interpret algorithm design only as the ordering of the steps required by a solution, then it is plausible that they view it as less difficult than both abstraction and decomposition.

5.2.3 Generalisation

Generalisation, although rarely used as a distinct term in the data, is represented in several responses. These responses are in terms of using equivalent strategies in different contexts or, more commonly, in terms of solving problems based on previously encountered similar problems and their solutions. For example, this is demonstrated when the solution for the classic
Chapter 5: Discussion

calculating tiles problems, found in mathematics, is applied to calculating fabric requirements for a textile project. One response goes further to apply concepts understood about base 10 numbers to binary.

“It opens your mind. You realise that base 10 is just an infinite number of possibilities. You don’t realise that when you’re young. So, when you finally realise that you can have numbers in another form is enlightening. Then, applying what you know about base 10 to other systems is quite exciting. It’s a powerful concept.”

No responses indicated that, in its broadest sense, generalisation was a very difficult skill to master. However, responses do indicate that learners may need prompting to consider the strategy of generalisation when attempting to solve problems. These responses are similar to “If you’ve seen something similar it can give a hint at how you might start”. This is analogous to one of Pólya’s (1985) suggestions for strategies to use during the planning phase of problem solving. These include reminders to think “Have you seen it before? Or have you seen the same problem in a slightly different form?” (Pólya 1985, p. 110).

Although learners may be familiar with and practise generalisation in the classroom, the term is rarely used explicitly. Generalisation is most commonly used in situations linking similar problems across contexts. It is not perceived as a difficult skill to master, but is viewed as a strategic skill whose applicability is not always immediately identified.

5.2.4 Abstraction

In common with generalisation, the term abstraction is not reported as being explicitly used in classroom situations. However, abstraction is practised in many subject areas across key stages. In an art project, students’ early focus is on the design of the artefact, without regard for any implementation details. The design is an abstraction. In ICT, pupils may well create storyboards representing digital animations. The storyboards are abstractions. These abstractions are not exactly the same as an abstraction of functionality. In computer science, the concept of abstraction most commonly involves the use of programming subroutines. These abstractions are functional; their activation
results in some behaviour. In the same vein, data manipulation is commonly encountered in ICT, mathematics, and the sciences. However, opportunities for learners to create their own data abstractions may be limited and previous learner experiences, either positive or negative, of abstractions may influence the learner.

The perceived levels of difficulty associated with abstraction, both of functionality and data, are attributed to a lack of previous experience. While learners are familiar with the concept of using functions, such as square or cosine, in mathematics, the ability to create their own abstractions is less frequently encountered. A response in the data identifies this association with the use of functional abstractions “…since they have met the concept of function machines in earlier maths lessons; many start by using the in-built maths functions of a programming language…” According to other respondents, older students extend these applications of functional abstractions by “… passing values as parameters to functions” and “… passing not just one parameter but a whole series of parameters which could include arrays of data to subroutines or functions.”

This same idea applies when manipulating data. Although data analysis may be performed in ICT, mathematics, or the sciences, the concept of creating and manipulating an abstraction of data, such as a list, record, or an array, may be less commonly experienced by learners. Responses identifying this lack of familiarity indicate, “The jump from there [single variables] to arrays is really challenging.” Another observes the difficulty experienced by pupils expressed as behaviour in the classroom.

“At KS3 I watched some of our brightest pupils glaze over with confusion whilst being taught arrays using ‘programming languages, one of which is textual’ yesterday at computer club. Are we really going to get our weakest pupils to do and understand this?”

A possible reason for this difficulty is suggested by another response, which states,
“… software often uses patterns and conventions that, while having sound software engineering reasons, seem bizarre to humans. For example, array indexes begin at zero … This is because under the hood array accesses are done through base/offset style addressing … to a newbie this just seems crazy – who starts numbering a list at zero?!?”

The way in which abstraction is interpreted by teachers and is evidenced by learners varies across key stages. Often learners start at high-level abstractions and refine downwards. For example, learners can describe how to draw a house using turtle graphics as requiring a square, a triangle, and one or more rectangles, whether or not they understand how to draw the individual shapes. For example, in primary, one teacher acknowledges that pupils don’t fully understand the concept of abstraction. He notes, “If you say you 3 are making a function, it’s not really, it’s so very simple. They loved the fact that they typed E on the thing then it drew an E.” At the other end of the scale, in further education, abstraction may only be employed in programming when the need for duplicate code arises. Failure to point this need out explicitly is explained by one teacher.

*Arrays, for example, would be for breakout. That’s where you’ve got the same bit of code for 15 breaks. You could copy the code 15 times. I have done those examples with them, but I have failed to point out that they could use if for more than 1 monster or snooker ball. That’s where I’ve failed because now I have to go around individually and say do you remember this example? No. Then, let’s go back and have a look at it. I end up doing many 1:1 where I should have done it different.*

It may be that it is the implementation of abstractions that is the real source of difficulty experienced by the learners. It is the programming mechanics of creating subroutines, parameters, and return values, not necessarily the idea of hiding complexity behind a name, that learners fail to grasp. This insight is
provided by one teacher who relates, “… what less able pupils DO run into difficulties with is parameter passing mechanisms.”

Although learners may practise abstraction in the classroom, the term is not always used explicitly. The types of abstractions previously experienced by learners are not necessarily equivalent to the types of abstractions necessary in the computer science classroom. Primary pupils use functional abstractions in programming, without being aware of it. They can give names to program subroutines. Secondary pupils begin to grasp the use of simple functional abstractions but experience difficulties when moving to simple data abstractions. Further education students continue to experience difficulties with functional abstractions, exacerbated by the introduction of parameters, and with complex data structures such as arrays. The disparity between the types of abstractions previously experienced by learners and the types of abstractions required in programming may be one of the contributing factors leading participants to identify that abstraction, both of function and data, is one of the most difficult computational thinking skills to master.

5.2.5 Decomposition

In contrast to several of the other terms associated with computational thinking, decomposition does appear to be used explicitly in the classroom. In addition, many practitioners may also use the phrase “break down” as an equivalent to decomposition. This type of language is usually considered more pupil-friendly. Decomposition is practised in many subject areas across key stages. Of all the subjects in which decomposition is practised, mathematics is perhaps the most obvious. Pupils are instructed to break problems down from an early stage. They often tackle the pieces and join the intermediate results. In other subject areas, for example, composition, pupils are often given the decomposition, such as “PQE” for point, quotation, and explanation. In the sciences, again, pupils may use decompositions, such as an ordered list of steps to conduct a practical experiment.

When considering decomposition, participants indicate that “Breaking problems down is quite difficult.” and “That’s probably the hardest part for them, really. It’s that kind of initial analysis of breaking down.” Several possible contributors
for the difficulties experienced by learners may exist. Some are suggested by the participants themselves. These include a lack of experience, incomplete understanding of the problem to solve, and the order of teaching programming skills.

The first, again in common with other computational thinking skills, is a perceived lack of previous experience in constructing their own. Although learners understand the concept of breaking a problem down, perhaps from a mathematical context, teachers indicate that learners struggle with implementing the process of decomposition. One participant notes that an

“… area that doesn’t seem to get included in topic lists that I have seen is ‘How to decompose a problem and design a solution?’ We often take for granted that our students will know how to analyse a problem and break it up so they can develop a solution. For some, this is the most difficult part of the exercise.”

Another further education teacher has also identified that students have difficulty in decomposing problems associated with their own coursework.

“The most difficult is to have an idea for a program and breaking the idea down knowing which constructs to use. Not so much constructs, it’s breaking the complexity of a game down into its component parts and realizing that it may have to be broken down even further and even further until eventually, you can program one single small part of it.”

The second suggested cause for the perceived level of difficulty is associated with the understanding of a problem or familiarity with a possible solution. Students appear to be able to use the skill of decomposition more successfully in situations where they already know the solution or understand the problem very well. For example, in creating an algorithm for a “guess the number” game, students understand how to play the game and have an understanding of which conditions generate which responses. This is reflected by a teacher uses this same example. “A typical one I use early on in the course now would be a
little guess the number game. So, we get the computer to generate a random number between 1 and 100.” Another teacher reports on implementing an algorithm for the television show, *The Apprentice*. “It’s breaking something real world into steps, which is what we’re trying to get them to do.” When asked if students already know the solution, the response was “Yeah, someone in the team will have seen the programme and know what should happen.” When asked if the results would be different if no one had seen the program, the teacher’s response was “Yeah, because they wouldn’t understand the problem.”

The third suggested cause of this difficulty is attributed to the teaching order of problem solving with programming. After a very basic introduction to simple programming constructs, such as assignment, selection, and repetition, learners are usually presented with a problem that must be immediately decomposed into its component parts. Indeed, some teachers may even start with a problem, such as guess the number, which is to be decomposed. It may well be that any skill introduced first, when learners are still coping with introductory programming constructs, would reflect the same level difficulty. However, understanding decomposition, based on the Computational Thinking Taxonomy, is a prerequisite for abstraction, algorithm design, and evaluation. As such, it must be mastered, to some extent, before the complexity of the following levels can be accessed. One participant had the following comment about developing the skill of decomposition, “The main concept I keep having to come back to with students is the strategy of thinking through a problem to break it down into components that can be expressed in the programming language in use.”

Although learners may have had opportunities to use decompositions throughout the key stages, it is clear, from the examples presented above, that the use of decompositions is not necessarily the same as analysing a problem using the technique of decomposition. Introductory programming may provide the first opportunity for learners to decompose fully their own problems in developing solutions. Success in this may also be tied to how well the problem is understood by the learners and where in the teaching scheme that the skill is introduced. Whatever the cause or causes of the perceived level of difficulty, decomposition must be mastered, to some extent, in order to facilitate the more complex computational thinking skills of abstraction and evaluation.
Chapter 5: Discussion

5.2.6 Conclusion
The hierarchy of difficulty of the computational thinking skills is reflected as a reversal of the skills mapped to the Cognitive Domain of Bloom’s Taxonomy. Although this may at first seem implausible, it is supported by further inspection of the data. Contributing factors to this reversal may include misunderstanding in implementing the skill in the classroom, as seen in evaluation; misunderstanding the level to which the skill should be performed, as seen in algorithm design; repurposing the term in a broader context, as seen in generalisation; and lack of experience in employing the skill from scratch rather than using the products of another’s skill, as seen in abstraction and decomposition.

5.3 Placement of taxonomy into current theory
At this point, relationships, as illustrated below, between 3 different models must be considered. As demonstrated previously, the taxonomy of computational thinking skills indicated in the current model maps directly to the levels of the Cognitive Domain of Bloom’s Taxonomy (1956). However, the top two levels of evaluation and synthesis are reversed in the more recent revision of the original model by Anderson et al. (2001). This more recent model calls into question the ordering of the taxonomy of computational thinking skills previously presented.

<table>
<thead>
<tr>
<th>Bloom (1956)</th>
<th>Revised Taxonomy (Anderson et al. 2001)</th>
<th>Computational Thinking Taxonomy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evaluation</td>
<td>Create</td>
<td>Evaluation</td>
</tr>
<tr>
<td>Synthesis</td>
<td>Evaluate</td>
<td>Algorithm design</td>
</tr>
</tbody>
</table>

Table 10: Current model related to Bloom's Taxonomy and the revised Taxonomy

The initial computational thinking literature may provide clarity for the placement of evaluation above algorithm design. In her initial article, Wing (2006) expresses the need for a computational thinker to make trade-offs, by evaluating algorithmic processes in terms of time and space, power and storage. This evaluation of algorithmic processes, including their power and
limitations, is foreshadowed by Gal-Ezer et al. (1995). In their IT approach, L’Heureux et al. (2012) include the ability to evaluate processes, in terms of efficiency and resource utilisation, and the ability to recognize and evaluate outcomes. Without a clear understanding of the design and creation of an algorithmic process under consideration, an evaluation, in computational thinking terms, would not be possible. Therefore, mastering the computational thinking skill of evaluation requires some level of mastery of the prior level, algorithm design. This conforms to Bloom (1956) who defines the simpler behaviours in the taxonomy as components of the more complex behaviours.

Although the revised Taxonomy (Anderson et al. 2001) may aid classroom practitioners in developing learning objectives, the ordering of the top 2 levels as evaluate and create does not reflect the reality of the way in which those involved in computational thinking view the skills. In the view of those engaged with computational thinking, the more appropriate order is reflected in Bloom’s Taxonomy (1956). Because of this later fact, in the scope of this research, the revised Taxonomy (Anderson et al. 2001) is rejected in favour of Bloom’s Taxonomy: Cognitive Domain (1956).

5.4 Where are knowledge and comprehension?

From the taxonomy presented above, it is clear that this investigation has not identified distinct computational thinking skills that align to the levels of Bloom’s Taxonomy associated with comprehension and knowledge. It is proposed that this preclusion is not intentional, but an indication that computational thinking is a skill most closely associated with more complex cognitive processes than knowledge recall and comprehension. According to Bloom’s own rules, a particular behaviour should be placed into the most complex, appropriate, and relevant class (Bloom 1956). This may indicate why the identified computational thinking skills appear at the topmost levels of the Taxonomy.

When considering the knowledge category, even Bloom (1956) is careful to note “… the knowledge category differs from the others in that remembering is the major psychological process involved …” (p. 62). In addition, Clark and Boyle (1999), in their study focusing on the final year projects of computer school undergraduates, also identify that subject knowledge represents the
“tools of a discipline” (p. 205) and that it is not possible to understand fully a tool until it can actually be used. An analysis of the data revealed no indication that participants perceived specialised computational thinking skills that correspond specifically to the process of remembering. It may be that the subcategories employed in the description of the knowledge category of Bloom’s Taxonomy are sufficient to describe the process of remembering when it forms part of computational thinking. However, the data collection process did not specifically pursue the lack of detail for this level.

When considering the comprehension category, Bloom (1956) subdivides it into translation, interpretation, and extrapolation. Items at these levels represent processes used to interact with or evidence the possession of knowledge. An analysis of the data revealed no indication that participants perceived specialised computational thinking skills that correspond specifically to any of the subcategories of the comprehension process. However, respondents did identify learner behaviour corresponding to this level. A behaviour observed at this level is the tendency for learners to read algorithms in a line-by-line mode, describing in detail, but missing the overall meaning. This may be attributed to translation which involves “… the giving of meaning to the various parts of a communication, taken in isolation …” (Bloom 1956, p. 89). It may be that the subcategories employed in the description of the comprehension level of Bloom’s Taxonomy are sufficient to describe the processes of translation, interpretation, and extrapolation when they form part of computational thinking. However, the data collection process did not specifically pursue the lack of detail for this level.

The computational thinking taxonomy presented above does lack specific terms that map directly to Bloom’s categories of knowledge and comprehension. However, this does not imply that knowledge and comprehension are not evidenced by computational thinkers. Indeed, these skills are the foundations of the higher level skills evidenced by generalisation, decomposition, abstraction, and algorithm design. The data collection process did not specifically pursue the lack of detail for the levels of knowledge and comprehension. Absence of corresponding identifiable computational thinking skills at these levels is not implied by this omission.
5.5 Response to research questions

The overarching question that this research has been designed to answer concerns how the teaching of programming might be used to enhance computational thinking skills. The formal research question is stated below. In order to respond fully to this question, 3 subdivisions were established, under which more refined supporting questions have been proposed. These subdivisions are taxonomy and definition of computational thinking, pedagogy, and difficulties of learning. This structure and refined questions have been detailed below. Following sections will respond to individual research questions. In order to preclude confusion, consensus implies a human agreement; consonance implies an accord interpreted from the data.

- Initial research question
  - How can the teaching of programming be used to enhance computational thinking skills?

- Taxonomy and definition of computational thinking
  - Is there a taxonomy of computational thinking skills?
  - Is there a consonance in the terms used to define computational thinking?
  - What is the connection between problem solving, programming, and computational thinking?

- Pedagogy
  - What specific programming activities contribute to the development of computational thinking skills?
  - Can computational thinking be taught without teaching programming?
  - What are the implications of this work for the teaching, in schools, of programming and computational thinking skills in the current context of computer science education?

- Difficulties of learning
  - What beginning programming skills are most difficult for learners to master?
  - What is the role of debugging in learning to program?
Chapter 5: Discussion

- What computational thinking skills are most difficult for learners to master?
- What problem-solving skills are most difficult for learners to master?
- What factors may limit the acquisition of computational thinking skills?

5.5.1 Taxonomy and definition of computational thinking

5.5.1.1 Is there a taxonomy of computational thinking skills?
Interpretation of the collected data has revealed a taxonomy of computational thinking skills. The taxonomy is best illustrated by the parallel relationship, between the Cognitive Domain of Bloom's Taxonomy and the Computational Thinking Taxonomy, as illustrated in the visual model, “Figure 56: Final relationship model”. The taxonomy of computational thinking, presented in order of cognitive complexity from simple to complex is composed of the terms generalisation, decomposition, abstraction, algorithm design, and evaluation.

5.5.1.2 Is there a consensus in the terms used to define computational thinking?
Attempting to define computational thinking based on classroom practice has generated various candidate terms. These terms include abstraction, algorithm design, analysis, decomposition, discrimination, evaluation, generalisation, modelling, and sequencing. Sequencing is a term that is subsumed by algorithm design. The resulting list can then be scrutinised in relation to the proposed definition from the literature review. Recall that this definition included abstraction, decomposition, algorithmic thinking, evaluation, and generalisation. Intersection of the two possible definitions results in abstraction, algorithm design, decomposition, generalisation, and evaluation. This is identical to the definition proposed in the literature review. The terms analysis, discrimination, and modelling are not included. Justification for the exclusion of modelling as being evidence of the use of computational thinking rather than defining it has been included in the literature review. Analysis is a broad term often applied in many problem-solving domains. It is not as precise a term as algorithmic thinking, mathematical thinking, or even logical thinking. Because it can be
interpreted in many different ways, it is unsuitable for defining computational thinking. The last term, discrimination, also belongs in the broad domain of problem solving. The ability to distinguish what is important in solving a problem is not unique to computational thinking. Therefore, the collected data affirms the definition of computational proposed in the literature review.

5.5.1.3 What is the connection between problem solving, programming, and computational thinking?

There is consonance that problem solving is an overarching category. It is an exercise that has application across all curriculum subjects and domains. High-level strategies for solving problems in one domain may well be transferable to another. There is also consonance that computational thinking is a specialised type of problem solving. The use of computational thinking results in solutions that can be implemented on computational devices, such as computers. It is similar to solving problems with a specialised set of tools. Computational thinking is not limited to the creation of computer programs. There is consonance that programming is a further specialisation of computational thinking. In particular, an artefact created by programming provides evidence of the use of computational thinking skills. Further investigation of the artefact and discussion with the learner may elicit evidence of the application of specific computational thinking skills. Where dissent exists, the description of the relationship between problem solving, computational thinking, and programming is represented by an overlap of the different categories. Therefore, problem solving is a broad term covering different strategies in various domains. Computational thinking is a specialised subset of these problem-solving strategies resulting in solutions implementable on computing devices. Programming is even further specialised. It is employed as a tool for the creation of artefacts, based on algorithms generated by computational thinking.

5.5.2 Pedagogy

5.5.2.1 What specific programming activities contribute to the development of computational thinking skills?

There is consonance that learning to program contributes to the development of computational thinking skills. However, in the classroom environment, the
emphasis is often on the production of an artefact rather than on the development of computational thinking skills. There is a need to be explicit when using programming as a tool. Every opportunity should be taken to identify where the use of computational thinking skills has been evidenced by programming. For example, care should be taken to point out that decomposing a problem is a technique that can be used in many domains. Respondents have identified 4 specific categories of activities that may contribute to the development of computational thinking skills. The first category is the ability to discriminate the information needed to solve a problem from any superfluous information given in the problem description. The second category is the ability to understand the problem constraints, such as those related to resources, be they computational or otherwise. The third category is the ability to decompose a problem into its constituent components, with the objective of solving smaller parts of the overall problem. The final category is the ability to formalise logical ideas using some acceptable convention, even verbal descriptions. The ability to create a formalisation of logic, using notations such as flowcharts or pseudocode, and translating between formalisations is evidence of the use of computational thinking skills. Learners would benefit from frequent opportunities to engage with activities such as those identified here and from frequent explicit identification of the use of computational thinking skills when learning to program.

5.5.2.2 Can computational thinking be taught without teaching programming?

There is a consonance in the data that the skills of computational thinking can be taught without teaching programming. The proposed definition of computational thinking does not require that a programming artefact be the objective. This could be done in critical thinking or problem-solving situations, in mathematics or the sciences, in particular. Creating a program, using an algorithm and a programming language, could be viewed as analogous to creating an intruder alarm, using a diagram, electrical components, and a breadboard. When considering the cognitive complexity of these activities, it may be that the identified level is not synthesis. The first is easily identifiable as translating from one formal notation to another. The second is also
understandable as a form of translation, albeit approaching the real physical end objective. Bloom (1956) ascribes the ability to translate from one level of abstraction to another or from one symbolic form to another as comprehension. This is lower on the scale of cognitive complexity than might have first been assumed. Although it is certainly possible to develop computational thinking skills without the teaching of programming, those who can program or write code in a programming language possess a skill that will allow them to evidence and validate their use of computational thinking.

5.5.2.3 What are the implications of this work for the teaching, in schools, of programming and computational thinking skills in the current context of computer science education?

The computing section of the new national curriculum for England (Department for Education 2013) explicitly requires that learners be given opportunities for logical thinking and for designing, using, and evaluating computational abstractions. In order to support these high-level aspirations and skills, classroom practitioners may identify that, depending on the capabilities of the learners, significant levels of scaffolding may need to be supplied. The collected data indicates that this is the case for many current practitioners. Although the national curriculum (Department for Education 2013) classifies computing as a foundation subject, compulsory at Key Stages 1-4, the proliferation of schools not required to follow the national curriculum may still produce learners entering Key Stage 4 and Post-16 who have very little or no practical experience of relevant computational thinking skills. Therefore, for the near future, wherever possible, the teaching of programming should focus on the underlying computational thinking skills of decomposition, abstraction, and generalisation, which contribute to learning to program. This focus could be achieved by asking learners to devote more time to the analysis and design processes that should take place before writing programming code. Learners could also be presented with a wide range of problems and asked to design solutions using programming constraints, but never actually implementing the programs. In addition, learners should have opportunities to develop their broader general problem-solving skills in many subjects, such as mathematics, the sciences, and the creative subjects such as art and design technology. For
example, in mathematics, clearly setting out the subtasks separate from the main body of the solution reinforces the concepts of decomposition and abstraction. In Chemistry, designing an experiment as a process involving decision-making points is analogous to flow control. This study has identified that beginner programmers’ lack experience in problem solving. This affirms previous work by Robins, Rountree, and Rountree (2003). The national curriculum framework may well lead the reader to assume that learners at the higher key stages will have had some exposure to computational thinking, but this is not necessarily the case.

5.5.3 Difficulties of learning

5.5.3.1 What beginning programming skills are most difficult for learners to master?

Previous literature on the difficulties of learning to program has suggested several contributing factors to learners' difficulties. This investigation has identified additional possible contributors. Both sets of factors are described below.

An inaccurate understanding of how a computing device executes a program may cause learners particular difficulties (Ma et al. 2011; Milne and Rowe 2002). Learners do not always understand and do not programmatically correctly handle the fact that an instruction is executed in the machine state left by the previous instruction (du Boulay 1989; Lahtinen, Ala-Mutka, and Järvinen 2005). Jenkins (2002) asserts that students demonstrate an inability to cope with the precision necessary to instruct the computer to carry out an algorithm. This inability is exemplified by learners who can often read and interpret code, but have difficulties writing their own (Jenkins 2002). This concept of difficulty in producing code has been extended by the results of this research.

Perhaps more important, in computational thinking terms, than writing code is the development of an algorithm design. Respondents report that any type of formalisation of logic, whether in English, flowcharts, or pseudocode is difficult for learners, at all ages. Many beginners even have difficulties expressing their logic in verbal terms. When translating from one formalisation to another movement in some directions is reported as easier than other directions. For
example, translating from internal logic to flowcharts appears easier than translating from internal logic to pseudocode. Pseudocode, however, is reportedly easier to produce from flowcharts. The writing of program code is viewed as more easily done after writing pseudocode than when done directly from flowcharts. Based on these observations, the path of least difficulty is from logic to flowcharts to pseudocode to code.

Another area of difficulty identified in this research involves following logic for understanding or for debugging. This concept has been previously identified by Lister, Fidge, and Teague (2009) who found that effective programmers had developed good tracing skills prior to good writing skills and that good students could explain the purpose of code without stating what it might do line by line. What is new in this research is identification that this difficulty is exacerbated by the nesting of code logic. Simple sequencing is straightforward for learners to follow; introducing nesting increases the complexity and leads to difficulties in understanding.

Three difficulties of learning to program, revealed in this investigation, are the difficulty associated with creating formalisations of logic, the difficulty in translating from one formalisation to another, and the difficulty in tracing and understanding nested code.

5.5.3.2 What is the role of debugging in learning to program?

The process of debugging may serve two purposes when learning to program. As defined in a previous section, debugging fits the analysis level of the cognitive domain of Bloom’s Taxonomy. The findings in this research indicate that debugging is a problem-solving strategy that works back to front, from the symptom to the error. It may involve the use of many different techniques, such as dry running, trace tables, or visualisations. Each attempt to debug an error will result in a unique experience. No two bugs are the same; no two debugging sessions will be the same. Classroom practitioners report, in this research, that early introduction to the process of debugging, often using an integrated development environment with a visual debugger, benefits learners. Not only can they independently find bugs in their own logic, but they can also begin to understand the way in which the machine executes a program. The main
Chapter 5: Discussion

objective of debugging is usually accepted as finding errors in logic or code. However, debugging, especially with the aid of a visualisation of the machine, can promote and validate an understanding of how the machine behaves when executing instructions.

5.5.3.3 What computational thinking skills are most difficult for learners to master?

While some research focusses on programming by asking which particular concepts are difficult to learn (Lahtinen, Ala-Mutka, and Järvinen 2005; McCracken et al. 2001) and other research focuses on the contributors to why the process of programming is difficult to learn (du Boulay 1989; Lahtinen, Ala-Mutka, and Järvinen 2005), this research has focused on questions concerning computational thinking skills. When asked to rank which computational thinking skills are most difficult for learners, respondents indicated that both abstraction and decomposition were the two most difficult skills for learners to master.

Difficulties in using abstractions were expressed in two ways. Understanding and creating abstractions of functionality, such as “draw a square” were reported as being less difficult than understanding and creating abstractions of data, such as records, lists, or arrays. This appears a reasonable ordering because beginner programmers tend to think linearly, with scant regard to flow control, (Miller 1981; Pane, Ratanamahatana, and Myers 2001) as a default method. Using high-level names, a type of abstraction, for steps in problem solving is a technique employed at all key stages. In a similar vein, abstraction of data has also been identified as difficult for learners. Organising and manipulating even simple data abstractions may be a computational thinking activity that is not developed by learners, even at the higher key stages. Although some students may have had an opportunity to approach problem solving from a data perspective (Michaelson 2012), others may not. Arrays are reported to be difficult for learners to master. However, it is not clear if this difficulty may be caused by unfamiliarity with similar real life structures, an order of introduction, i.e. arrays may often be the first abstract data structure introduced, or the mechanisms of implementing arrays in programming languages.
The inclusion of decomposition as the most difficult computational thinking skill to master is unanticipated. As indicated in a previous section, decomposition fits the analysis level of the cognitive domain of Bloom’s Taxonomy. While abstraction sits at the same level, it should be a more complex skill than decomposition, due to the taxonomy’s rule of cumulative hierarchy (Bloom 1956). From a programming perspective, this is not instinctive. Designing an algorithmic solution to a problem normally requires breaking the problem down into parts (decomposition), naming those parts (functional abstractions), and designing the functionality of those subparts. However, this research has found that respondents view decomposition as the most difficult computational thinking skill for learners to master. As discussed above, this level of difficulty may be due to lack of experience with the process, the additional complexity of an unknown solution, or the teaching order, i.e. decomposition may be one of the first computational thinking activities introduced in the classroom. Regardless of the cause, decomposition is reportedly more difficult to master than either functional abstraction or data abstraction.

5.5.3.4 What problem-solving skills are most difficult for learners to master?

When respondents were questioned about their perceptions of any specific difficulties of learning problem solving, three areas were identified. The difficulties experienced by learners include understanding the problem and its constraints, decomposing the problem into smaller problems, which may be easier to solve, and generalising a solution know for one problem to another. The items identified here are reflections of the steps required in problem solving proposed by Pólya (1985), in his seminal work in the field of mathematics teaching. Indeed, understanding the problem and its constraints is step one, in his 4 step approach. In addition, decomposition is suggested by Pólya (1985) as a fundamental strategy to use during step 2, the analysis. The skill of generalisation also is encouraged by Pólya (1985), albeit in the reverse direction of attempting to recognise problems of similar types with known solutions. The inclusion of these specific three items may further suggest that deficiencies in learners’ problem solving need to be addressed prior to introducing them to the more specialised skills of computational thinking.
Chapter 5: Discussion

It is not intended that this conclusion undermine the computer science claim to make a unique contribution to the general education of pupils (The Royal Society 2012). As described in the conceptual framework, computational thinking is a specialisation of general problem solving, dealing with the generation of solutions that may be implemented in a computational device (Wing 2006). As with any specialisation, a strong foundation is beneficial. The conclusion here suggests that if learners do not already possess a strong foundation in problem solving, then classroom practitioners may need to consider that fact when designing computational thinking activities.

5.5.3.5 What factors may limit the acquisition of computational thinking skills?

While previous sections have detailed specific responses identifying which programming skills and which computational thinking skills are most difficult for learners to master, this section responds to the equally important question concerning which factors might limit the acquisition of computational thinking skills. Respondents have indicated that there is a range of factors perceived as limiting the mastering of computational thinking skills.

The first limiting factor identified is the cognitive overload generated by the limited time allocated to achieve results. This is especially apparent in examined courses, at Key Stage 4 and Post-16. Learners with little relevant prior exposure to computational thinking are expected to perform well in these high stakes testing environments to ensure their next move into the workforce or university. Learners, as reported by respondents, find the time constraints, under which the development of computational thinking skills required to meet these challenges must be developed, a limiting factor.

Several respondents identified issues with literacy, both verbal and written, as limiting factors. One recurring theme in the data is the “if you can’t talk about it, you don’t understand it” idea. This may seem plausible, if the assumption is that the literacy level of the learner will allow such expression. However, the ability to express the use of computational thinking for some learners may not be commensurate with their ability to think computationally.
Another factor identified as limiting the acquisition of computational thinking skills is learners’ lack of resilience and independence. Classroom practitioners do recognise that part of their remit is to help develop these qualities in learners. However, they often report that learners are simply focused on outcomes, getting the correct answer as soon as possible, without appreciating that they can learn from their mistakes. Both of these qualities, resilience and independence, take time to develop. One participant acknowledges this investment of time at the post-16 level, “I’m trying to get them to the point where self-learning is a possibility.”

On the other hand, one factor, identified as a potential limiter of the acquisition of computational thinking skills, which may be within the control of the respondents is the selection of tools with which to teach. In most contexts, these tools consist of a programming language and a development environment. Although research (McCracken et al. 2001; Jenkins 2002) indicates that the choice of programming language has no effect on learners’ problem solving abilities, the data here indicates that mastering a tool, such as an integrated development environment, does limit learners in some ways. For example, a post-16 course might include a syllabus that emphasises procedural programming, an object-oriented programming language such as Visual Basic (VB), and the creation of a graphical user interface, which is event-driven. In order to achieve the syllabus requirements, the object-oriented paradigm of VB may be ignored, except when used for the event-driven code behind GUI objects such as buttons and combo boxes. Juggling all of these concepts and tools may add to the cognitive load already being experienced by the learners. A participant comments that learners

“… have to work out the problem, break the problem down into small chunks, design it, and write the code and learn the IDE all at the same time. As a cognitive exercise, it’s probably one of the most difficult things these students will ever do.”

Several of the factors identified as limiting the acquisition of computational thinking skills may not be in the direct control of classroom practitioners. The design of course syllabi and timetables are controlled by other stakeholders.
Chapter 5: Discussion

Literacy, resilience, and independence are capabilities that may require considerable time to cultivate. However, improvements in these areas can be promoted along with the acquisition of computational thinking skills. The one limiting factor identified here that may be under the control of the classroom practitioner is the choice of tools to use in the classroom. Classroom practitioners may choose tools most suitable for their learners, even if these tools are not those preferred by industry, higher education, or examination boards.

5.5.4 Initial research question

5.5.4.1 How can the teaching of programming be used to enhance computational thinking skills?

Previous responses to each of the research sub-questions influence this response to the initial overarching research question. How can stakeholders, such as classroom practitioners, curriculum designers, syllabi designers, resource creators, industry professionals, higher education academics, and government policymakers influence the way in which the teaching of programming can be utilised to enhance the acquisition of computational thinking skills? Several potentially effective suggestions identified in an analysis of this investigation’s data are restated in this section.

Abstraction is the foundation of computational thinking (Denning 2011, National Research Council 2010). The ability to hide complexity in multiple layers and manipulate these layers is one of the computational skills that is effectively exercised, developed, and evidenced by designing algorithms and programming. The use of subroutines, be they functions, procedures, or methods, could be introduced early in the learning process. Some introductory programming tools, even for the lower key stages, have the ability to create functional abstractions. The other type of abstraction identified in this research, data structures, is reported to be more difficult for learners. Again, data structure abstractions could be introduced early in the learning process. Even introductory programming tools usually have the ability to create and manipulate simple structures such as lists. Instead of focusing almost exclusively on sequential programming when beginning to learn programming, early
Introduction of the creation and manipulation of abstractions could enhance the development of computational thinking skills.

Generalisation is the ability to recognise parts of solutions that have been used in previous situations or that might be used in future situations (National Research Council 2011, Computing at School Working Group 2012). It is the ability to show relationships between solutions and to show extrapolations of solutions. For example, understanding how to draw a square by defining the number of sides and applying that understanding to draw an object with any number of sides is an example of generalisation. This is one computational thinking skill that is often reported as being developed “with practice”. It is one computational thinking skill that can also be viewed as a more generalised problem solving skill. It is possible, in either instance, to supplement the idea of practice with explicit scaffolding. Systematic presentation of problem sets with common features that become progressively more difficult may help develop the skill of generalisation.

The definition of computational thinking, presented in a previous section, includes the ability to think algorithmically (Wing 2011). This type of thinking can be developed and evidenced by the use of programming. Although a program artefact is often the end objective of the programming process, evidence of algorithmic thinking can be provided by other types of formalisations of logic. For example, a flowchart or pseudocode is evidence of the formalisation of logic, using an acceptable notation. This formalisation of logic is one area that learners are reported to find very difficult. The progression from internal logic to flowchart, to pseudocode, to code is reported to minimise the cognitive load at each step. As with generalisation, systematic presentation of problems requiring formalisation at each step, may help develop the skill of algorithmic thinking.

While it was never the intent of this research to delve deeply into general problem-solving strategies, the lack of these strategies is identified by the participants in this study as a deficiency of beginner programmers. This gap is illustrated by responses similar to this, from a further education teacher,
Chapter 5: Discussion

“General problem solving skills are weak. Each year we’re seeing them come through weaker and weaker.” Another response also points out this deficiency

“I find that students who are stronger at maths are more able to break problems down and find solutions. Others will really struggle. The ones better at math have probably already developed some of the problem solving skills necessary.“

This lack of skill is not only exemplified at high levels when developing algorithmic solutions, but also manifests itself in the debugging process. For one participant, this is exhibited when students are required to “… work backwards to find the error. This is where they have problems with the logic, let’s see, now I have to work backwards. So, it is a syntax error, or is it a logic error? Those are the things they have to think about.” Another participant views difficulties with debugging differently. “I see it more that they don’t follow through what is actually happening. Has it gone into that bit of code? Why hasn’t it gone in there?”

Algorithmic design and debugging are two areas associated with computational thinking that rely heavily on the ability of learners to plan, execute, and evaluate in systematic steps. In order to aid development of problem solving skills, it may be beneficial to give learners problems that can be solved systematically. Then, explicitly point out the strategies used to solve these problems. Focus less on finding the solution to the problem and more on development of strategies which might lead to a solution for the problem. Although respondents in this research identify the lack of general problem-solving strategies in beginner programmers, this discrepancy is not one that may be solved in the short term. Focus on problem solving and the identification and use of explicit strategies may be required for a sustained length of time.

Decomposition is the computational thinking skill that is identified, by respondents, as being the most difficult for beginner programmers, at any level, to master. Previous sections have addressed this concept and why learners may find it so difficult. The teaching of programming may provide an appropriate context in which to develop this skill. All but the most simple of
algorithms will involve identification of distinct steps. In identifying these steps of a problem solution, learners are evidencing decomposition. Identification of these smaller steps is also applicable in general problem-solving contexts, such as mathematics and sciences. All learners, regardless of subject, may benefit from explicit formalisation of the decomposition process. This, in some high-level cases, could simply be a to-do list. In more complex analysis, the decompositions themselves may be decomposed further. In common with the skills of generalisation and abstraction, the computational thinking skill of decomposition should be taught and explained explicitly during the learning to program process.

Finally, with regard to each of the computational thinking skills discussed here, decomposition, abstraction, and generalisation, focus should be maintained on what is possible within the capabilities of the learner. Observed improvements in the development of these skills may be small. Some learners may need significantly more scaffolding than other learners may. Developing computational thinking skills through programming may take a significant amount of time, even years. This extended time frame for the teaching of computational thinking may now be possible with the new national curriculum framework (Department for Education 2013).

5.6 Conclusion

In this section, each of the research questions, identified in “1.2 Research questions”, has been addressed. These questions and an analysis of the data has contributed to the development of a Computational Thinking Taxonomy, defined as generalisation, decomposition, abstraction, algorithm design, and evaluation. When placed in an order of difficulty of learning, the sequence revealed is evaluation, algorithm design, functional abstraction, data abstraction, and decomposition. The plausibility of this observation has been explored in this section. In addition, the relationship between the Computational Thinking Taxonomy, Bloom’s Taxonomy: Cognitive Domain (1956) and the revised Taxonomy (Anderson et al. 2001) is discussed.
Chapter 5: Discussion
Chapter 6. Final remarks

6.1 Consolidation

The previous chapters have described the background to, the implementation of, and the results arising from the current research into how programming may be used to develop computational thinking skills. This section serves as a synopsis of this research, including a description of the dataset, the research method employed, and selected results.

The complete data set, on which the research is based, consists of 123,580 words collected via three different instruments: an on-line questionnaire; a community of practice on-line forum; and a semi-structured interview. The data set represents the view of one hundred forty-three individual respondents, purposively selected for their knowledge of or interest in the areas of programming, computational thinking, or problem solving. The participants include teachers, academics, and industry professionals. The data is collected and analysed in accordance with the grounded theory method, as proposed by Strauss and Corbin (1998).

Individual chapters, presented previously, have highlighted significant points in the research process. In chapter 3.2.1 “Defining computational thinking”, a definition of computational thinking is proposed. This definition is based on an interpretation of Wing (2006), Denning (2009), National Research Council (2010, 2011), and other researchers. For the purposes of this research, computational thinking is a focused approach to problem solving, incorporating thought processes that utilise abstraction, decomposition, algorithms, evaluation, and generalisations. In chapter 5.1 “Taxonomy of computational thinking skills”, based on an analysis of the terminology used by the research participants, a proposed taxonomy of computational thinking skills is presented. The placement of computational thinking into a relationship with problem solving and programming, as suggested in “2.7 Conceptual framework”, is corroborated by respondents, as described in section 5.5.1.3. Computational thinking is perceived to be a specialised subset of problem-solving strategies resulting in solutions implementable on computing devices. Programming is perceived to
Chapter 6: Final remarks

be an even further specialisation. The data set is further explored to establish a proposed pedagogy of teaching programming and computational thinking. This pedagogy and the computational thinking taxonomy are mapped onto existing theory, specifically Bloom’s Taxonomy (Bloom 1956), resulting in a model of their relationship. Chapter 4.8.6 “Milestone 6: Interviews and additional models”, Figure 21 and Figure 22 present visual representations of this relationship. The levels of difficulty experienced by learners for each of the computational thinking skills and programming skills are also established by an analysis of the responses. When these levels of difficulty of computational thinking skills are added to the existing model, they appear at levels reversed from those predicted by Bloom’s Taxonomy. For example, evaluation is reported by respondents to be less difficult than abstraction of data structures, which appears at the analysis level of Bloom’s Taxonomy. However, decomposition, also assigned to the analysis level of the Bloom model, is considered the most difficult computational thinking skill. This is illustrated in 5.2 “Hierarchy of difficulty of computational thinking skills”, Table 9. The plausibility of this reversal is explored in chapter 5.2.5 “Decomposition”.

6.2 Reflection on reliability and validity

This section serves as a reflection on the reported research that is undertaken using the grounded theory approach. Firstly, the concepts of reliability and validity as applied in qualitative research are defined. Secondly, the aspects of reliability and validity, as appropriate to grounded theory, are explored in the context of this research. These include the lack of prescriptive process, the lack of a pre-existing theory, the accommodation of researcher naivety, the need for procedural guidance, the generation and selection of categories, the justification for theoretical sampling, researcher bias, and fidelity to the spirit of grounded theory. Where appropriate, the researcher has included personal reflections of the experience of undertaking a grounded theory study. This section ends with a final reflection on the reliability and validity of this particular research study.

In dealing with the question of reliability in qualitative research, the overriding idea is to achieve a fitness for purpose (Cohen, Manion, and Morrison 2007). In other words, has the purpose of the research determined the design of the
research and the selection of methods? Duncan (2005) asserts that a method that has a high level of fitness for purpose is able to answer the posed questions in the research context. In the research presented here, the context and purpose were defined prior to the selection of a method. The research questions were answered in the defined research context. On reflection, both criteria, purpose determines design and methods, and questions answered in context, demonstrate a fitness for purpose.

In dealing with the question of validity in qualitative research, the overriding idea is to determine if it measures what it is purported to measure (Prosser 2006). In other words, do the instruments and the research as a whole appear to measure what they claim to measure? Usher, Bryant, and Johnston (1997) report that there are three aspects of validity, pre-validation, internal validation, and post-validation. The one most applicable to grounded theory is internal validation. “…this refers to the actual conduct of the research itself as following the precepts of appropriate practices with respect to devising indicators, data collection and analysis” (Usher, Bryant, and Johnston 1997, p. 215). By following the formal rules of enquiry, the research becomes self-validating. Validity can be achieved in processes of grounded theory by the constant comparisons of new data with collected data.

One precept of grounded theory, as set forth by Glaser (2009) is the idea that there is no prescribed process for conducting research using this method. According to Glaser, grounded theory is a straightforward procedure, inducting theory from data. The beginner researcher may, as this researcher did, find that there is a lack of details describing this straightforward process. This researcher desired more specific information about the actual process of conducting a grounded theory study. Strauss and Corbin (1998) suggest processes and procedures that might underpin a grounded theory approach. Their systematic approach includes open coding, axial coding, and selective coding until theoretical saturation. In this way, this researcher was directed in the use of strategies and analytical tools that lead to theory emergence. Along with this guidance comes the possibility that the data may have been forced to fit an emerging theory. Indeed, Urquhart (2007) criticises Strauss and Corbin’s guidelines because they can be interpreted as very prescriptive. This
researcher did not find the guidelines overly prescriptive. Grounded theory is sufficiently flexible to allow for researcher interpretation and freedom. A researcher is always free to adhere to or dismiss a process or procedure suggested by Strauss and Corbin’s guidelines.

Grounded theory does not follow other positivist approaches where data supports or denies a pre-existing theory (Cohen, Manion, and Morrison 2007). No theory is proposed before beginning the data collection and analysis processes. Only a broad context for the research needs to be defined. Theory emerges through the data analysis process. A researcher undertaking a grounded theory study may need strength of conviction and faith in the method. On reflection, this is one area that this researcher found challenging. It was difficult, especially in the early data collection and analysis phases, to envision where the analysis might lead. However, as Strauss and Corbin (1998) assert, as the data became saturated, the theory did emerge.

In grounded theory, the researcher should be professionally naïve, should suspend his or her own beliefs, and should trust in the emergence of concepts from the data (Christiansen 2008). This naivety extends to the lack of requirement for a literature review, prior to commencement of the research (Mills, Bonner, and Francis 2006). This researcher, while recognising the lack of requirement for a literature review, felt that background reading would create a solid foundation on which to base the research. Strauss and Corbin (1998) suggest that literature may be used to sensitise the researcher to possibilities in the data. While the notion that a theory might exist could be enough to seed research, this researcher found the background reading to be an important step in narrowing the research area.

Ensuring that the research process is followed precisely as set out could provide an assurance of validity. The rules of grounded theory (Strauss and Corbin 1998) indicate that the data be collected simultaneously with analysis, that constant comparisons are made to previous data, that the theory change as the data dictates, and that the theory is allowed to develop, unforced. This researcher has ensured that each of these rules has been followed. The “Method” chapter, which is described in chronological order, provides evidence
that the analysis was performed simultaneously with data collection. Development of theory is demonstrated by the changing models, Figure 48 and Figure 49. Revisiting existing data to perform constant comparisons is evidenced in “Milestone 2: First concepts and categories” and “Milestone 7: Additional data and refinement of models”, where new data identified concepts that may have been overlooked in previous analysis. On reflection, this researcher did not consciously force the theory. By following the process during the first two phases of data collection, this researcher became assured that a theory would emerge.

The finer points of Strauss and Corbin’s (1998) interpretation of grounded theory indicate the use of concepts, categorisations, and selective coding. Concepts were straightforward to code from the collected data. Many concepts could be taken directly from the vocabulary used in the responses. At one point, as indicated in the “Method” section, the sheer number of individual concepts appeared overwhelming. When considering merging concepts to categories, this researcher attempted, wherever possible, to use the respondents’ vocabulary. This goal is supported by Usher and Bryant (1989) who indicate

“Categories predetermine the form of research outcomes, and, while they may be logically coherent, they may not be instantially relevant. If, however, they are grounded in informal or common-sense understandings then they are better able to speak to these understandings in a practically relevant way.” (p. 114).

The judgements about which categories and concepts should move forward to generate theory did reflect the relative frequency of use of those concepts and categories by the respondents.

One of the reasons that grounded theory is appropriate for this research is the affirmation by Strauss and Corbin that purposive sampling is a foundation stone of grounded theory that, “… enables the researcher to choose those avenues of sampling that can bring about the greatest theoretical return” (1998, p. 202). Without the ability to select participants based upon their perceived ability to
add value to the research, this researcher believes a model may not have emerged. In addition, once gaps had been identified in the data, participants with knowledge to fill those gaps could be selected. Purposive sampling could also be used to actively seek out participants with dissenting views. Overall, the ability to be assured of the quality of responses may ensure the validity of the results.

Another consideration in reflecting on reliability and validity of this research is researcher bias. Would a different person with different experiences and values develop the same theory? This is a question that cannot be answered fully. However, Glaser affirms that with constant comparison, multiple data collections, and continuous conceptualisation, any bias is corrected and therefore the data may be used objectively (Glaser 2002). As indicated above, each of these activities has taken place throughout the time frame of the research. With this in mind, it may be that researcher bias has been controlled.

One final personal reflection concerns the question of having been true to the spirit of grounded theory. While Strauss and Corbin (1998) have provided methods and techniques to ensure that the process of grounded theory has been faithfully followed, Glaser’s original vision for grounded theory should not be ignored. This researcher originally found not knowing what the end point would or should be very challenging. The initial assumption that, given enough time and data, a theory would emerge in the domain of computational thinking required a high level of trust. However, that is exactly what this researcher experienced. After a sufficient amount of time, data collection, and analysis, a theory did emerge. To that end, this research has been true to the spirit of Glaser’s original grounded theory.

In summary, the reliability and validity of this grounded theory study rely on the interpretation of these attributes in terms of qualitative research and grounded theory. Cohen, Manion, and Morrison describe reliability in qualitative research as including “… fidelity to real life, context- and situation-specificity, authenticity, comprehensiveness, detail, honesty, depth of response and meaningfulness to the respondents.” (2007, p. 149). In response, this researcher can provide assurances that the study is true to life, is context specific, is authentic, is as
comprehensive as time allows, is detailed and honest, and the results may directly influence the classroom practices of the respondents. Usher, Bryant, and Johnston describe internal validation as referring to “…the actual conduct of the research itself as following the precepts of appropriate practices with respect to devising indicators, data collection and analysis” (1997, p. 215). In response, this researcher can provide assurances that the study does follow the accepted processes for sampling, data collection, and analysis associated with grounded theory.

6.3 Limitations

This section identifies and acknowledges some of the limitations of the current research study, including the sample size, the use of Bloom’s Taxonomy, and the dissonance between the requirements of the research and the needs of the respondents.

In the first instance, although the respondents were selected for their perceived knowledge relevant to the research topics, their number represents a relatively small sample size. The preponderance of teachers of pupils and students in the 14 to 19 years age range, Figure 58 and Figure 59, may also have heavily influenced the research outcomes. A more balanced quota sample may have produced different results.

According to Thompson et al. (2008), the cognitive processes associated with programming are not well defined. In addition, Fuller et al. (2007) point out the difficulty of applying Bloom’s Taxonomy to practical subjects such as programming. Although there may be no specific categorisation of programming activities, careful examination of Bloom’s Taxonomy sublevels and definitions, along with collected data, may reveal appropriate candidate levels for these skills. These levels form the basis for the final relational model, Figure 56. This use of Bloom’s Taxonomy in computer science education research is supported by other researchers (Fitzgerald, Simon, and Thomas 2005; Whalley et al. 2006).

As is often the case with small-scale research projects (Bell 2005), the time available to conduct the research may have influenced the results. For
example, the relationship between the perceived levels of difficulty of computational thinking skills and the anticipated cognitive complexity has been recognised, but the causes for this have not been identified. The limits of time do not afford an opportunity to explore why a lower-level process like decomposition is actually perceived to be the most difficult for learners.

An unanticipated issue with the data collection occurred during the interview process. Attempting to balance the respondent’s “need to talk” with the researcher’s “need to find out” may have relaxed the structure of the interviews. Some respondents are very passionate about their profession, practice, and ideas. At times, their eagerness to share afforded few opportunities to change the flow of conversation. The subject of this ‘volunteered information’ may be outside the scope of this research, for example a comment about a particular student or the working or political environment. Any comments, not relevant to the research, were handled in line with the guidance set out in the ethics section, chapter “4.2 Ethical issues”.

An unanticipated issue with the data analysis involved the coding context. By default, the coding context was individual sentences. This was found to be too limiting. Coding individual sentences was appropriate for frequency counting of simple concepts and individual words. However, when attempting to code higher-level concepts, such as “evidencing abstraction” (chapter 4.8.8), single sentences did not provide enough context to support the assignment of a code. Therefore, the coding of individual sentences was abandoned and replaced with unrestrained expressive units. Sometimes this was a single sentence. More often, it was a group of two or three sentences that expressed a single cohesive thought.

### 6.4 Contribution

The results of this study contribute to the broad areas of research incorporating computational thinking and programming, the more specific area of computer science education research, and the area of computer science pedagogy. It responds to the call for more research into how to teach computing in a way that enforces computational thinking (Guzdial 2008).
The area of computational thinking and its relationship to programming is an existing area of research (Isbell et al. 2010; National Research Council 2010; Denning 2009; Guzdial 2008; Bundy (2007); Wing 2006). The results described by this research can only make a small contribution to that broad research area. The contribution of this study is a proposed taxonomy of computational thinking skills. This taxonomy attempts to coalesce the initial contributions to the description of computational thinking (National Research Council 2010; Denning 2009; Wing 2006). The proposed taxonomy of computational thinking skills adds to these initial discussions attempting to define computational thinking.

The area of computer science education research is also well established (Isbell et al. 2010; Schulte and Bennedsen 2006; Lister 2000). In this area, this study augments the body of knowledge focused on computational thinking and programming, specifically in the context of the 14 - 19 year age group (Isbell et al. 2010; Sakhnini and Hazzan 2008; Schulte and Bennedsen 2006; Gal-Ezer et al. 1995). In addition, the results also contribute to the body of knowledge exploring the possibilities of applying general education theories, such as Bloom's Taxonomy: Cognitive Domain (1956), to the field of computer science education (Fitzgerald, Simon, and Thomas 2005, Lister 2000).

From a pedagogical perspective, this study makes contributions that may be used directly to affect the classroom learning. The results of this study suggest that decomposition, which forms the basis for problem-solving strategies (Pólya 1985), is perceived to be the most difficult computational thinking skill to learn. The results also suggest that abstraction of data is perceived to be more difficult to learn than abstraction of functionality. An understanding of these perceived difficulty levels may be taken into account when designing classroom tasks or activities. A model, Figure 56, illustrating the relationships between cognitive complexity, the pedagogy of programming, and computational thinking skills is also proposed. This model could be used to contribute to the design of curriculums, classroom schemes of learning, and individual learning objectives.

In summary, the results of this study contribute to the broad areas of research incorporating computational thinking and programming, the more specific area of computer science education research, and the area of computer science
pedagogy. In the first instance, a taxonomy of computational thinking skills is proposed to aid understanding of the term. In the second instance, applying Bloom’s Taxonomy to the context of programming for 14 – 19 year olds, aids efforts to explore using general education theories in the computer science classroom. In the third instance, the proposed relational model, between levels of cognitive complexity, the teaching of programming skills, and the perceived levels of difficulty of computational thinking skills may be used to influence effective classroom practices.

6.5 Future work

The research presented here has produced, along with the results, several related questions. These questions can be separated into three broad categories: questions that may be focused on cognitive processes, questions that may be focused on classroom practice and curriculum decisions, and questions that may be focused on the age range of learners. Each of these different categories is discussed below. Further research topics in each area are suggested.

Some questions posited by this research belong to the domain of cognitive processes. It might be that these questions could be best addressed by collaborative projects between cognitive scientists and computer science education researchers. For example, this research has identified that respondents perceive decomposition to be the most difficult computational thinking skill for learners to master. Although possible reasons for this status have been proposed, this research has not revealed why this is the case.

Another area of interest for this type of collaborative research might involve investigating how the transition from thinking to formalisation of thinking can be made easier for learners. Understanding how this transition is made may be a prerequisite for proposing measures to make it an easier task for learners. A further question of interest to both of these groups of researchers may be attempting to identify the real effect of problem-solving skills on learning to program. Do those learners with better problem-solving skills find learning to program easier than those without? Even though there is some existing research concerning the applicability of Bloom’s Taxonomy: Cognitive Domain
to computer science (Fitzgerald, Simon, and Thomas 2005; Thompson et al. 2008), this particular association deserves further study, this research continues the theme by suggesting that the upper levels of Bloom’s Taxonomy are applicable to computer science education. However, the levels of knowledge and application are yet to be explored.

Some questions posed by this research may be better placed in the field of classroom practitioners and curriculum decision makers. It may be that these questions could be best addressed by collaborative projects between classroom practitioners, especially at the pre-university level, and computer science education researchers. For example, this research has proposed a definition of computational thinking that may be of use to classroom practitioners, but it needs validation in practitioners’ settings. From a practical perspective, it may be fruitful to explore why teachers consider that learners find it more difficult to build and understand abstractions of data than to build and understand abstractions of functionality.

Another consideration for opportunities identified by this research is a focus on learners from younger age groups. While computer science education researchers may find opportunistic samples in their own, often university, classrooms (Butler and Morgan 2007; Jenkins 2002; McCracken et al. 2001), the results of that research may not transfer directly to younger learners. Classroom practitioners, on the other hand, may be eager to share best practice, but their studies may be small scale and applicable only in narrow contexts. This could be addressed, as suggested above, by collaboration between classroom practitioners who have experience of and access to younger learners and computer science education researchers who have the skills and expertise to ensure rigour, reliability, and validity in a broader study.

Several areas for new research have been highlighted by the current study. Regardless of the specific areas under investigation, the most useful results may be generated by collaborative studies with expertise from two or more fields. Computer science education researchers may need cognitive scientists to aid understanding of computational thinking. Classroom practitioners may need computer science education researchers to identify effective classroom
practices that are applicable to different ages of learners and that are effective in different contexts

6.6 High-level summary

This section serves as a high-level summary of this research study into how programming may be used to enhance computational thinking skills. Each of the topics discussed here is covered in more detail in previous sections.

6.6.1 Summary of findings

Presented below is a summary of the main findings of this research. A more in-depth discussion of each item can be found in a previous section.

- A definition of computational thinking has been proposed, based on the literature presented in section “3.2 Computational thinking”. Computational thinking is a focused approach to problem solving, incorporating thought processes that utilise abstraction, decomposition, algorithms, evaluation, and generalisations.
- A taxonomy of computational thinking skills has been proposed, based on the literature presented in section “3.2 Computational thinking” and an analysis of the data set presented in section “4.8.5 Milestone 5: Defining a lexicon, hierarchies, and models”. The data identified computational thinking skills aligned with the analysis, synthesis, and evaluation levels of the Cognitive Domain of Bloom’s Taxonomy. This is illustrated in Table 8.
- Computational thinking is perceived to be a specialised subset of problem-solving strategies resulting in solutions implementable on computing devices. Programming is perceived to be an even further specialisation. This is described in section 5.5.1.3 and supports the relationship expressed in “2.7 Conceptual framework”.
- The pedagogy of teaching programming, specifically ordering of skills has been discerned from an analysis of the participants’ responses. Although other orders of teaching may exist, a frequently occurring order for teaching programming is types, constructs, using constructs,
decomposition, create programs, and test. This ordering is also illustrated in Table 5.

- Participants’ responses, incorporating programming skills and comparatives, are used to map programming skills to the levels of Bloom’s Taxonomy. This parallel mapping is illustrated in Figure 46 and in Figure 47.

- A categorisation of the perceived difficulty of programming skills and computational thinking skills, based on participants’ responses, is illustrated in Table 7.

- Combining each of these categorisations together into a single model highlights that the perceived levels of difficulty of programming skills and computational thinking skills are reversed from those anticipated by Bloom’s Taxonomy. This is model is illustrated in “Figure 63: A relationship model”.

### 6.6.2 A relationship model

This model is an illustration of the relationships between Bloom’s Taxonomy: Cognitive Domain (1956), the taxonomy of computational thinking skills, and the perceived difficulty of computational thinking skills.
Chapter 6: Final remarks

Figure 63: A relationship model
6.6.3 Future research

A more detailed discussion of possible future work has been undertaken in “6.5 Future work”. The ideas presented here, if more fully understood through rigorous research, may have an impact on classroom practitioners.

- Why is decomposition perceived to be so difficult for learners?
- How might decomposition be taught to make it easier for learners to master?
- Why is the transformation of logic, between notations (head, flowchart, pseudocode), perceived to be so difficult for learners?
- How might the transformation of logic be taught to make it easier for learners to master?
- Why is abstraction of data structures perceived to be more difficult than abstraction of functionality for learners?
- How might both abstraction of data and abstraction of functionality be taught to make them easier for learners to master?

6.6.4 Advice for classroom practitioners

The results of this study may offer some practical assistance to classroom practitioners. The model, Figure 63, and the relationships represented may be of value in adjusting teaching times and anticipating challenging topics for learners.

- Practitioners may use the model to allocate teaching time. Since decomposition is perceived to be the most difficult skill to master, then perhaps it may need more time devoted to it than evaluation.
- Practitioners may use the model to ensure that all the levels in Bloom’s Taxonomy are addressed in their schemes of learning. For example, testing a program solution may evidence behaviours at lower levels, including an analysis of different required data types and designing a sequence of steps for each test.
- Practitioners may use the results to anticipate where learners will encounter difficulties. The respondents have indicated that they perceive
the most difficult computational thinking skills to be decomposition and abstraction (data and functionality) and one of the most difficult programming skills to be that of transformation of logic. If practitioners anticipate these trouble spots, they are in a position to plan for them.
### Appendix 1. Interview revision (original)

<table>
<thead>
<tr>
<th>Opening</th>
<th>Question Context</th>
<th>Prompts</th>
</tr>
</thead>
</table>
| What prompted you to agree to this interview? | Any | What about the topic is of interest to you?  
Do you view computer science as being influenced by problem solving, computational thinking, and programming? |

<table>
<thead>
<tr>
<th>Programming</th>
<th>Prompts</th>
</tr>
</thead>
</table>
| What is most challenging about teaching programming? | 2 | For example, is a while loop more difficult to explain than a for loop?  
Do students focus on just getting an answer and assume that they’ve then mastered the art? |
| What is difficult to learn about programming? | 3 | Do students ever want to put in blank else conditions, which means just keep going or do nothing? |
| What is easiest to learn about programming? | 3 | Is there an easy concept to learn if students join the course with no previous experience? |
| Is there a logical sequence to teaching programming concepts? | 4,6 | Should language constructs be taught first then put together?  
Should students master pseudocode before attempting real code?  
Is something like Scratch really |

233
<table>
<thead>
<tr>
<th>Question/Context</th>
<th>Prompt</th>
</tr>
</thead>
<tbody>
<tr>
<td>pseudocode?</td>
<td>Do you use a personally non-preferred language in your teaching?</td>
</tr>
</tbody>
</table>
| How do you teach the use of functions and procedures? | Do you use the notion of black boxes?  
Do you use the term abstraction?  
Do you have an analogy to maths? |
| How do you teach the way the machine works (notional machine)? | How do students cope with the idea of each instruction being executed in the context or state of what has gone before? |
| How do you teach data representation and organisation? | How do students respond to data structures of more than 2 dimensions?  
How do students respond to data structures of fixed length? Do they assume that size is automatically dynamic? |
| How would you describe the process of learning to program to someone who doesn’t program? | How do you relate the necessity to be precise in giving instructions (i.e. what they’re told and only what they’re told)?  
Is programming skill an innate ability or can anyone learn how to do it? |
<p>| Computational Thinking Prompts | |
| What is the meaning of the term “computational” | Do the terms decomposition, abstraction, and generalisation have |</p>
<table>
<thead>
<tr>
<th>thinking&quot; in your work?</th>
<th>an application in your work?</th>
</tr>
</thead>
<tbody>
<tr>
<td>What activities contribute to development of computational thinking skills?</td>
<td>Do students have opportunities to produce models, visualisations, or other representations of problem contexts or solutions?</td>
</tr>
<tr>
<td>4, 5, 6, 7</td>
<td>How do they normally go about this process?</td>
</tr>
<tr>
<td></td>
<td>Do mathematical concepts play an important part in CT?</td>
</tr>
<tr>
<td>Do you think there is a relationship between programming and computational thinking?</td>
<td>2, 7</td>
</tr>
<tr>
<td></td>
<td>When students program, do they exhibit skills in decomposition, abstraction, or generalisation?</td>
</tr>
<tr>
<td></td>
<td>Is computational thinking skill an innate ability or can anyone learn how to do it?</td>
</tr>
<tr>
<td>Problem Solving</td>
<td>Prompts</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>What does the term “Problem Solving” mean in your work?</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>What other subjects does the term bring to mind (maths, science)?</td>
</tr>
<tr>
<td></td>
<td>Did you mention computer science?</td>
</tr>
<tr>
<td>What activities contribute to the development of problem solving skills?</td>
<td>4, 6, 7</td>
</tr>
<tr>
<td></td>
<td>Do practical activities help develop thinking?</td>
</tr>
<tr>
<td></td>
<td>How do you make the connection explicit?</td>
</tr>
<tr>
<td></td>
<td>Is there a step-by-step problem solving methodology that you find useful?</td>
</tr>
</tbody>
</table>
| | Does the word “problems”
Appendix 1

<table>
<thead>
<tr>
<th>Prompt</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>immediately bring to your mind, the context of mathematics?</td>
<td>Do you think there is a relationship between problem solving and computational thinking?</td>
</tr>
<tr>
<td>Which category is broader?</td>
<td>Does one of them involve a particular kind of constraints?</td>
</tr>
<tr>
<td>Does one of them involve a particular kind of constraints?</td>
<td>Is problem-solving an innate ability or can anyone learn how to do it?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Wrap Up Prompts</th>
<th>Recall the relationship you identified between programming and computational thinking.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recall the relationship you identified between problem solving and computational thinking.</td>
<td>3</td>
</tr>
<tr>
<td>Does that imply a relationship between all three (problem solving, computational thinking, and programming)?</td>
<td>1</td>
</tr>
<tr>
<td>Is there a hierarchy in this relationship?</td>
<td>Is there a hierarchy in this relationship?</td>
</tr>
<tr>
<td>Are all of the three items of equal standing?</td>
<td>Are all of the three items of equal standing?</td>
</tr>
<tr>
<td>How might this relationship be reflected specifically in computer science?</td>
<td>How might this relationship be reflected specifically in computer science?</td>
</tr>
<tr>
<td>Could we draw a visual representation of that relationship?</td>
<td>Linear? Hierarchical? Consuming?</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Appendix 2. Interview revision (af)

Changes from previous version are in italics.

<table>
<thead>
<tr>
<th>Opening</th>
<th>Question Context</th>
<th>Prompts</th>
</tr>
</thead>
<tbody>
<tr>
<td>What prompted you to agree to this interview?</td>
<td></td>
<td>What about the topic is of interest to you?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Do you view computer science as being influenced by problem solving, computational thinking, and programming?</td>
</tr>
<tr>
<td>Programming</td>
<td></td>
<td></td>
</tr>
<tr>
<td>What is most challenging about teaching programming?</td>
<td>2</td>
<td>For example, is a while loop more difficult to explain than a for loop?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Do students focus on just getting an answer and assume that they’ve then mastered the art?</td>
</tr>
<tr>
<td>What is difficult to learn about programming?</td>
<td>3</td>
<td>Do students ever want to put in blank else conditions, which means just keep going or do nothing?</td>
</tr>
<tr>
<td>What is easiest to learn about programming?</td>
<td>3</td>
<td>Is there an easy concept to learn if students join the course with no previous experience?</td>
</tr>
<tr>
<td>Is there a logical sequence to teaching programming concepts?</td>
<td>4,5</td>
<td>Should language constructs be taught first then put together?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Should students master pseudocode before attempting real code?</td>
</tr>
<tr>
<td>Question</td>
<td>Score(s)</td>
<td>Additional Questions</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>----------</td>
<td>---------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Is something like Scratch really pseudocode?</td>
<td></td>
<td>Do you use a personally non-preferred language in your teaching?</td>
</tr>
<tr>
<td>How do you teach the use of functions and procedures?</td>
<td>4,5</td>
<td>Do you use the notion of black boxes?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Do you use the term abstraction?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Do you have an analogy to maths?</td>
</tr>
<tr>
<td>How do you teach the way the machine works (notional machine)?</td>
<td>4,5</td>
<td>How do students cope with the idea of each instruction being executed in the context or state of what has gone before?</td>
</tr>
<tr>
<td>How do you teach data representation and organisation?</td>
<td>4,5</td>
<td>How do students respond to data structures of more than 2 dimensions?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>How do students respond to data structures of fixed length? Do they assume that size is automatically dynamic?</td>
</tr>
<tr>
<td>How would you describe the process of learning to program to someone who doesn’t program?</td>
<td>2</td>
<td>How do you relate the necessity to be precise in giving instructions (i.e. what they’re told and only what they’re told)?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Is programming skill an innate ability or can anyone learn how to do it?</td>
</tr>
</tbody>
</table>

Computational Thinking Prompts
<table>
<thead>
<tr>
<th>Question</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is the meaning of the term “Computational Thinking” in your work?</td>
<td>3</td>
</tr>
<tr>
<td>Do the terms decomposition, abstraction, and generalisation have an application in your work?</td>
<td></td>
</tr>
<tr>
<td>What do the words “abstraction” and “decomposition” mean to you and your learners?</td>
<td></td>
</tr>
<tr>
<td>What is your definition of the words “analysis” and “evaluation”?</td>
<td></td>
</tr>
<tr>
<td>Have you encountered the term “generalisation” before? If so, where?</td>
<td></td>
</tr>
<tr>
<td>What activities contribute to development of computational thinking skills?</td>
<td>4,5</td>
</tr>
<tr>
<td>What is a visualisation? What is a model? What is a simulation?</td>
<td></td>
</tr>
<tr>
<td>Do students have opportunities to produce models, visualisations, or other representations of problem contexts or solutions?</td>
<td></td>
</tr>
<tr>
<td>How do they normally go about this process?</td>
<td></td>
</tr>
<tr>
<td>Do mathematical concepts play an important part in CT?</td>
<td></td>
</tr>
<tr>
<td>Are some activities easier/more difficult than others? For example, is it easier to design an algorithm or to decompose a problem?</td>
<td></td>
</tr>
<tr>
<td>Do you think there is a relationship between programming and</td>
<td>2</td>
</tr>
<tr>
<td>When students program, do they exhibit skills in decomposition, abstraction, or generalisation?</td>
<td></td>
</tr>
<tr>
<td>computational thinking?</td>
<td>Is computational thinking skill an innate ability or can anyone learn how to do it?</td>
</tr>
<tr>
<td>--------------------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Problem Solving</strong></td>
<td><strong>Prompts</strong></td>
</tr>
<tr>
<td><strong>What does the term “Problem Solving” mean in your work?</strong></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>What other subjects does the term bring to mind (maths, science)?</td>
</tr>
<tr>
<td></td>
<td>Did you mention computer science?</td>
</tr>
<tr>
<td><strong>What activities contribute to the development of problem solving skills?</strong></td>
<td>4,5</td>
</tr>
<tr>
<td></td>
<td>Do practical activities help develop thinking?</td>
</tr>
<tr>
<td></td>
<td>How do you make the connection explicit?</td>
</tr>
<tr>
<td></td>
<td>Is there a step-by-step problem solving methodology that you find useful?</td>
</tr>
<tr>
<td></td>
<td>Does the word “problems” immediately bring to your mind, the context of mathematics?</td>
</tr>
<tr>
<td><strong>Do you think there is a relationship between problem solving and computational thinking?</strong></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Which category is broader?</td>
</tr>
<tr>
<td></td>
<td>Does one of them involve a particular kind of constraints?</td>
</tr>
<tr>
<td></td>
<td>Is problem-solving an innate ability or can anyone learn how to do it?</td>
</tr>
<tr>
<td><strong>Wrap Up</strong></td>
<td><strong>Prompts</strong></td>
</tr>
<tr>
<td><strong>Recall the relationship you identified between programming and computational thinking.</strong></td>
<td>3</td>
</tr>
</tbody>
</table>
Recall the relationship you identified between problem solving and computational thinking.  

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Does that imply a relationship between all three (problem solving, computational thinking, and programming)?  

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Is there a hierarchy in this relationship?  
Are all of the three items of equal standing?  
How might this relationship be reflected specifically in computer science?  

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Could we draw a visual representation of that relationship?  

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Linear? Hierarchical? Consuming?
Appendix 2
## Appendix 3. Interview revision (ag)

Changes from previous version are in italics.

<table>
<thead>
<tr>
<th>Opening</th>
<th>Question Context</th>
<th>Prompts</th>
</tr>
</thead>
<tbody>
<tr>
<td>What prompted you to agree to this interview?</td>
<td></td>
<td>What about the topic is of interest to you?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Do you view computer science as being influenced by problem solving, computational thinking, and programming?</td>
</tr>
<tr>
<td>Programming</td>
<td></td>
<td></td>
</tr>
<tr>
<td>What is most challenging about teaching programming?</td>
<td>2</td>
<td>For example, is a while loop more difficult to explain than a for loop?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Do students focus on just getting an answer and assume that they’ve then mastered the art?</td>
</tr>
<tr>
<td>What is difficult to learn about programming?</td>
<td>3</td>
<td>Do students ever want to put in blank else conditions, which means just keep going or do nothing?</td>
</tr>
<tr>
<td>What is easiest to learn about programming?</td>
<td>3</td>
<td>Is there an easy concept to learn if students join the course with no previous experience?</td>
</tr>
<tr>
<td>Is there a logical sequence to teaching programming concepts?</td>
<td>4,5</td>
<td>Should language constructs be taught first then put together?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Should students master pseudocode before attempting real code?</td>
</tr>
<tr>
<td>Question</td>
<td>4,5</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Is something like Scratch really pseudocode?</td>
<td>Do you use a personally non-preferred language in your teaching?</td>
<td></td>
</tr>
<tr>
<td></td>
<td><em>Do you use kinaesthetic activities when teaching programming?</em></td>
<td></td>
</tr>
<tr>
<td>How do you teach the use of functions and procedures?</td>
<td>Do you use the notion of black boxes?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Do you use the term abstraction?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Do you have an analogy to maths?</td>
<td></td>
</tr>
<tr>
<td></td>
<td><em>How do learners evidence the use of abstractions such as functions and procedures?</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td><em>How do learners express their understanding of abstractions?</em></td>
<td></td>
</tr>
<tr>
<td>How do you teach the way the machine works (notional machine)?</td>
<td>How do students cope with the idea of each instruction being executed in the context or state of what has gone before?</td>
<td></td>
</tr>
<tr>
<td>How do you teach data representation and organisation?</td>
<td>How do students respond to data structures of more than 2 dimensions?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>How do students respond to data structures of fixed length? Do they assume that size is automatically dynamic?</td>
<td></td>
</tr>
<tr>
<td></td>
<td><em>Do learners have an understanding</em></td>
<td></td>
</tr>
</tbody>
</table>
| How would you describe the process of learning to program to someone who doesn’t program? | 2 | How do you relate the necessity to be precise in giving instructions (i.e., what they’re told and only what they’re told)?
Is programming skill an innate ability or can anyone learn how to do it? |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Computational Thinking</strong></td>
<td><strong>Prompts</strong></td>
<td></td>
</tr>
</tbody>
</table>
| What is the meaning of the term “Computational Thinking” in your work? | 3 | Do the terms decomposition, abstraction, and generalisation have an application in your work?
What do the words “abstraction” and “decomposition” mean to you and your learners?
What is your definition of the words “analysis” and “evaluation”?
Have you encountered the term “generalisation” before? If so, where? |
| What activities contribute to development of computational thinking skills? | 4,5 | What is a visualisation? What is a model? What is a simulation?
Do students have opportunities to produce models, visualisations, or other representations of problem contexts or solutions?
How do they normally go about this process? |
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do mathematical concepts play an important part in CT?</td>
<td>Do you think there is a relationship between programming and computational thinking?</td>
</tr>
<tr>
<td>Are some activities easier/more difficult than others? For example, is it easier to design an algorithm or to decompose a problem?</td>
<td>2 When students program, do they exhibit skills in decomposition, abstraction, or generalisation?</td>
</tr>
<tr>
<td>Can you identify any abstractions that you yourself use in your teaching?</td>
<td>Is computational thinking skill an innate ability or can anyone learn how to do it?</td>
</tr>
<tr>
<td>Can you identify any abstractions that the learners’ use or evidence?</td>
<td>3 What other subjects does the term bring to mind (maths, science)? Does computational thinking skill an innate ability or can anyone learn how to do it?</td>
</tr>
<tr>
<td>What factors may limit the acquisition or evidencing of computational thinking skills?</td>
<td>4,5 Do practical activities help develop thinking? Do you use kinaesthetic activities</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem Solving Prompts</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>What does the term “Problem Solving” mean in your work?</td>
<td>3 What other subjects does the term bring to mind (maths, science)? Did you mention computer science?</td>
</tr>
<tr>
<td>What activities contribute to the development of problem solving skills?</td>
<td>4,5 Do practical activities help develop thinking? Do you use kinaesthetic activities</td>
</tr>
<tr>
<td>Do you think there is a relationship between problem solving and computational thinking?</td>
<td>2</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Which category is broader?</td>
<td></td>
</tr>
<tr>
<td>Does one of them involve a particular kind of constraints?</td>
<td></td>
</tr>
<tr>
<td>Is problem-solving an innate ability or can anyone learn how to do it?</td>
<td></td>
</tr>
<tr>
<td><em>Do learners begin with problems for which they may already know an acceptable solution, such as guess the number or draw a square?</em></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Wrap Up</th>
<th>Prompts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recall the relationship you identified between programming and computational thinking.</td>
<td>3</td>
</tr>
<tr>
<td>Recall the relationship you identified between problem solving and computational thinking.</td>
<td>3</td>
</tr>
</tbody>
</table>
Appendix 3

| Does that imply a relationship between all three (problem solving, computational thinking, and programming)? | 1 | Is there a hierarchy in this relationship? Are all of the three items of equal standing? How might this relationship be reflected specifically in computer science? |
| Could we draw a visual representation of that relationship? | 1 | Linear? Hierarchical? Consuming? |
## Appendix 4. Interview revision (ah)

Changes from previous version are in italics.

<table>
<thead>
<tr>
<th>Opening</th>
<th>Question Context</th>
<th>Prompts</th>
</tr>
</thead>
<tbody>
<tr>
<td>What prompted you to agree to this interview?</td>
<td></td>
<td>What about the topic is of interest to you? Do you view computer science as being influenced by problem solving, computational thinking, and programming?</td>
</tr>
<tr>
<td>Programming</td>
<td></td>
<td>Prompts</td>
</tr>
<tr>
<td>What is most challenging about teaching programming?</td>
<td>2</td>
<td>For example, is a while loop more difficult to explain than a for loop? Do students focus on just getting an answer and assume that they’ve then mastered the art?</td>
</tr>
<tr>
<td>What is difficult to learn about programming?</td>
<td>3</td>
<td>Do students ever want to put in blank else conditions, which means just keep going or do nothing?</td>
</tr>
<tr>
<td>What is easiest to learn about programming?</td>
<td>3</td>
<td>Is there an easy concept to learn if students join the course with no previous experience?</td>
</tr>
<tr>
<td>Is there a logical sequence to teaching programming concepts?</td>
<td>4,5</td>
<td>Should language constructs be taught first then put together? Should students master pseudocode before attempting real code?</td>
</tr>
<tr>
<td>Question</td>
<td>4,5</td>
<td></td>
</tr>
<tr>
<td>-------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Is something like Scratch really pseudocode?</td>
<td>Do you use a personally non-preferred language in your teaching?</td>
<td></td>
</tr>
<tr>
<td>Do you use kinaesthetic activities when teaching programming?</td>
<td>How do learners cope with the nesting of programming constructs?</td>
<td></td>
</tr>
<tr>
<td>How do learners evidence the use of abstractions such as functions and procedures?</td>
<td>Do you use the notion of black boxes?</td>
<td></td>
</tr>
<tr>
<td>How do learners express their understanding of abstractions?</td>
<td>Do you use the term abstraction?</td>
<td></td>
</tr>
<tr>
<td>How do you teach the use of functions and procedures?</td>
<td>Do you have an analogy to maths?</td>
<td></td>
</tr>
<tr>
<td>How do you teach the way the machine works (notional machine)?</td>
<td>How do students cope with the idea of each instruction being executed in the context or state of what has gone before?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>What is debugging? Is it analysis or evaluation?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Does nesting of constructs affect the ability to understand logic?</td>
<td></td>
</tr>
</tbody>
</table>
| How do you teach data representation and organisation? | 4,5 | How do students respond to data structures of more than 2 dimensions?  
How do students respond to data structures of fixed length? Do they assume that size is automatically dynamic?  
Do learners have an understanding of data structures as abstractions? |
|---|---|---|
| How would you describe the process of learning to program to someone who doesn't program? | 2 | How do you relate the necessity to be precise in giving instructions (ie what they're told and only what they're told)?  
Is programming skill an innate ability or can anyone learn how to do it? |
| Computational Thinking | Prompts | What is the meaning of the term “Computational Thinking” in your work? | 3 | Do the terms decomposition, abstraction, and generalisation have an application in your work?  
What do the words “abstraction” and “decomposition” mean to you and your learners?  
What is your definition of the words “analysis” and “evaluation”?  
Have you encountered the term “generalisation” before? If so, where? |
<table>
<thead>
<tr>
<th>Question</th>
<th>Ref</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>What activities contribute to development of computational thinking skills?</td>
<td>4,5</td>
<td>What is a visualisation? What is a model? What is a simulation?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Do students have opportunities to produce models, visualisations, or other representations of problem contexts or solutions?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>How do they normally go about this process?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Do mathematical concepts play an important part in CT?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Are some activities easier/more difficult than others? For example, is it easier to design an algorithm or to decompose a problem?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Can you identify any abstractions that you yourself use in your teaching?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Can you identify any abstractions that the learners’ use or evidence?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>What factors may limit the acquisition or evidencing of computational thinking skills?</td>
</tr>
<tr>
<td>Do you think there is a relationship between programming and computational thinking?</td>
<td>2</td>
<td>When students program, do they exhibit skills in decomposition, abstraction, or generalisation?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Is computational thinking skill an innate ability or can anyone learn how to do it?</td>
</tr>
<tr>
<td>Problem Solving</td>
<td>Prompts</td>
<td></td>
</tr>
<tr>
<td>-------------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>What does the term “Problem Solving” mean in your work?</td>
<td>3</td>
<td>What other subjects does the term bring to mind (maths, science)? Did you mention computer science?</td>
</tr>
<tr>
<td>What activities contribute to the development of problem solving skills?</td>
<td>4,5</td>
<td>Do practical activities help develop thinking?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Do you use kinaesthetic activities when teaching problem solving?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>How do you make the connection explicit?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Is there a step-by-step problem solving methodology that you find useful?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Does the word “problems” immediately bring to your mind, the context of mathematics?</td>
</tr>
<tr>
<td>Do you think there is a relationship between problem solving and computational thinking?</td>
<td>2</td>
<td>Which category is broader?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Does one of them involve a particular kind of constraints?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Is problem-solving an innate ability or can anyone learn how to do it?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Do learners begin with problems for which they may already know an acceptable solution, such as guess the number or draw a square?</td>
</tr>
<tr>
<td>Wrap Up</td>
<td>Prompts</td>
<td></td>
</tr>
<tr>
<td>Question</td>
<td>Score</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>Recall the relationship you identified between programming and computational thinking.</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Recall the relationship you identified between problem solving and computational thinking.</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Does that imply a relationship between all three (problem solving, computational thinking, and programming)?</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Is there a hierarchy in this relationship?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Are all of the three items of equal standing?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>How might this relationship be reflected specifically in computer science?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Could we draw a visual representation of that relationship?</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Linear? Hierarchical? Consuming?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix 5. Four Approaches to Teaching Programming

Four Approaches to Teaching Programming

Cynthia C. Selby
University of Southampton
Southampton, England
C.Selby@soton.ac.uk

Abstract

In a survey of literature, four different approaches to teaching introductory programming are described. Examples of the practice of each approach are identified representing procedural, visual, and object-oriented programming language paradigms. Each approach is then further analysed, identifying the strengths and weaknesses for the student and the teacher. The first approach, code analysis, is performed on existing code prior to producing their own. An alternative is the building block approach, analogous to learning vocabulary, nouns and verbs, before constructing sentences. A third approach is identified as simple units in which learners master solutions to small problems before applying logic to more complex problems. The final approach, full systems, is analogous to learning language by immersion whereby learners design a solution to a non-trivial problem and the sub-concepts and language constructs are introduced only when the solution to the problem is applied. The conclusion asserts that competency in programming cannot be achieved by mastering each of the approaches, at least to some extent. Use of the approaches in combination will provide novice programmers with the opportunities to acquire a full range of knowledge, understanding, and skills. Several orders for presenting the approaches in the classroom are proposed reflecting the needs of the learners and teachers. Further research is needed to better understand other approaches to teaching programming, not in terms of learner outcomes, but in terms of actions and techniques employed to facilitate the construction of new knowledge by the learners. Classroom teaching practices could be informed by further investigations into the effect on learning of different toolset choices and combinations of teaching approaches.

Introduction

In the Academy of Engineering (2005, p. 17) report, “ICT for the UK’s Future” states “It is essential that a proportion of the 14-19 age group understands Computing concepts — programming, design, solving, usability, communications and hardware”. This, along with current policymaker’s focus on the National Curriculum (DfE 2011) and the report on vocational education (Wolf 2011), is the importance of providing opportunities for learners to acquire knowledge, understanding, and skills with programming. This setting provides the context for an investigation into different approaches to teaching programming.

In the literature concerned with teaching computer programming, varied approaches began to emerge. The characteristics of the different approaches were identified. By grouping the characteristics of these approaches coalesced into the four categories discussed below. It was then possible to identify published literature where the different approaches, defined by their characteristics, were employed in classroom settings to teach programming. They were chosen to represent a range of programming practices, so are not language specific. The classroom or research based findings presented below were reviewed media or in peer-reviewed conference proceedings.

Understanding programming requires that learners be able to build solutions both in their mind and on a computer. This interpretation of programming can be supported by both a learning theory and a practical method. The idea of knowledge not being passively absorbed, but being actively constructed based on their own experiences, fits into the learning theory known as Constructivism (Ben-Asher et al. 2000). This theory forms the foundation for Papert’s Constructionism, which advocates that learning is facilitated through the creation of artefacts (Ackermann 2001). In the situation of learning to program, the act of constructing code (Constructionism) facilitates the building of knowledge (Constructivism). Some constructivist activities for teaching programming are identified by Wulf (2005) as including interactive feedback, design based learning (Zubizarreta 2004).
building blocks, simple units, and full systems are each described and illustrated with examples from current practice. The presented literature allows for the identification of some advantages and disadvantages of each approach. Using the results reported in the literature, it is possible to extrapolate further and suggest how a combination of the approaches may be used effectively to teach programming.

Programming Terminology

For those readers not familiar with the technical terminology used in the teaching of programming, a small amount of background vocabulary may be beneficial. The high-level definitions provided below are derived from their common usage by educators and learners in classrooms.

<table>
<thead>
<tr>
<th>Code</th>
<th>Sequences of textual or symbolic instructions, composed by a learner, which are interpreted by a device to affect its behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code reading, walkthrough, debugging</td>
<td>Employing various techniques to identify overall logic or causes of incorrect behaviour</td>
</tr>
<tr>
<td>Pseudocode</td>
<td>A language used to plan logic before implementing that logic in a device</td>
</tr>
<tr>
<td>Structured English</td>
<td>A type of pseudocode that uses recognisable English words such as if, while, for, begin, end, input, and output</td>
</tr>
<tr>
<td>Constructs Building blocks</td>
<td>Programming language specific keywords or symbols, which can be combined to form sequences of instructions; basic constructs include variable (name a place in memory), assignment (set a variable to some value), conditional (test if a condition is true or false), and repetition (repeat instructions for a number of times)</td>
</tr>
<tr>
<td>Structured programming</td>
<td>Instructions are interpreted and executed one after another in a sequence; data and code are usually kept separate</td>
</tr>
<tr>
<td>Object-oriented programming</td>
<td>An object is a collection of both code and data; instructions are interpreted and executed based on the passing of messages from one object to another</td>
</tr>
</tbody>
</table>

Code Analysis

In the code analysis approach to teaching programming, learners read and understand programming logic before writing their own. This approach is based on the use of pseudocode so it is not programming language dependent. Three studies at university level have supported this approach and another study suggests that this approach is especially appropriate for weaker students. The ability to explain programming logic and code appears to be a prerequisite for, although it does not imply, the ability to write code. This approach allows problem-solving skills to be developed, which are applicable to many domains. This approach may be disappointing to learners who want to engage with the physical computers and may not be suitable for independent learning situations because of the lack of feedback.

The code analysis approach is analogous to learning to read before learning to write. Learning to read exposes the rules of grammar for the target language. Attempting a language in this way should allow the learner to become familiar with the way in which components and constructs of the language are combined to produce meaning before attempting the construction themselves.

An implementation of this approach may involve providing learners with practice exercises. These exercises could be prepared using some type of acceptable pseudocode or structured English. These exercises might exist only on paper or could be displayed in an appropriate development environment, provided one exists for the pseudocode implementation. However, as a lone approach, code analysis does not require learners to interact with the code in an environment on a computer.

At least four recent studies support the use of the code analysis approach. At Victoria University Melbourne, Miliszewska and Tan (2007) redesigned one of the first-year computer science courses to include the object-oriented paradigm, but teaching structured programming first. The new approach to teaching the course includes studying examples of well-written code. They propose that the students will learn by imitating the examples of good practice. This approach is also supported by Kolling and Rosenberg (2001) who are proponents of studying code to internalise styles and idioms usage. Campbell and Bolker (2002) express agreement by their own belief that students learn more by reading programs written by experienced programmers than by starting with writing their own code. Code walkthroughs and code reading, both
elements of constructivist computer programming instruction (Wulf 2005), were employed by Lui, et al (2004) in their study on weaker programming students. They suggest that the editing, compiling, and executing cycle can drain the patience and confidence of weaker students. They advocate that weaker students work with paper and pencil, which presents an initial low risk environment.

There may be potential problems using the code analysis approach if employed under false assumptions. Glaser, Hartel, and Gurratt (2000) point out that their students often have difficulty constructing a program, even if the same students can read and understand the solution when it is presented to them. However, a more recent study of introductory undergraduate programming students (Lister, Fidge, and Teague 2009) found that the ability to trace and explain code had a statistically significant relationship to the ability to write code. Of course, together, these findings could simply imply that the ability to write code requires some skill in tracing and explaining, but that skill in tracing and explaining may not beget the ability to write code.

Although from a learner's perspective learning to program without using a computer may seem perverse, there are advantages for the introduction of code analysis from the beginning. First, this approach means that there are no tools or environments to master in preparation for learning programming logic. Logic can be tackled straight away. Second, any implemented programming language may be used, but perhaps the most appropriate is no defined language at all. Pseudocode or structured English could be presented. Both are independent of any implemented language. Third, analysis first allows the development of skills involved in debugging and tracing to correct or ensure program behaviours. Fourth, it also provides the opportunity to expose learners to the logic behind simple foundation algorithms such as linear search and bubble sort. Early exposure to code analysis may help prepare learners for examinations that include questions involving tracing or dry running of code.

The code analysis approach may not be suitable under all circumstances. Learners may feel cheated by a technology class in which the tools are pencil and paper. In addition, depending on the age and capabilities of the learners, it may be challenging for an educator to develop a set of meaningful and understandable pseudocode instructions. For example, should words or symbols be used? Analysis first could be problematic for independent learning because of the lack of immediate feedback. There is also no verification that the pseudocode logic, identified by the learner, is a correct interpretation of a problem solution. In addition, there is the possibility that the understanding of the pseudocode may not result in skills that can be transferred to an understanding of a language defined by a strict syntax.

Although not suitable in every situation, the code analysis approach has been shown by recent studies to be beneficial to novice programmers. It has the advantage of not requiring a particular programming language or computer environment. It also promotes the development of problem-solving skills and logical thinking required in many domains. Its use can enhance the ability to understand program code, which forms part of the underlying foundation necessary to facilitate the writing of program code.

**Building Blocks**

In the building blocks approach to teaching programming, learners develop an understanding of individual pieces before combining the pieces to create meaning. This approach requires a set of development tools and a defined programming language. Language constructs are introduced and understood one at a time, in isolation, before combining them. This approach is not limited to single language paradigms and has been used in procedural, functional, and object-oriented languages environments. The building blocks approach has the advantage of the introduction of a precisely defined language syntax and the immediate feedback from a syntax checking tool. Unfortunately, learning how to use a code editor, a syntax checker, and perhaps a compiler can be challenging for beginners. In addition, they may be disappointed to find that syntactically correct code may not result in correct logic.

The building blocks approach is analogous to learning to speak individual parts of a language before combining them firstly in the verbal form and later in the written form. Young learners usually acquire verbal language skills before written skills. Foreign languages are usually taught using verbal skills first then written skills. When attempting to master a language, individual components (nouns, verbs, adjectives) are understood to some extent before putting them together to create meaning (sentences). Complex meaning is built up based on the understanding of the smaller pieces.

A specifically chosen set of tools, consisting of a programming language and development environment, is required for this approach. The choice of programming language is beyond the scope of this paper, but it will be assumed that the language and environment chosen is suitable for the capabilities of the learners. Specific language code is introduced one construct at a time. Understanding the behaviours of the constructs will be enhanced by writing the constructs in an appropriate development environment where syntax errors may be highlighted by the tool. In addition, execution of these single constructs may be possible in a development environment where the learners can comprehend, perhaps by visualisation, the
behaviours of each construct.

Research indicates that the building blocks approach is not limited to a particular programming language. It has been used with procedural, functional, and object-oriented languages. In an attempt to address high attention rates in the computer science courses at the University of Denver, educators made the decision to switch to a games first approach (Leutenegger and Edgington 2007). In their approach, they use Flash ActionScript and introduce individual control constructs in the context of moving visual objects around the screen. Matthew Flatt, an advocate of the Racket programming language, reports using this type of approach with liberal arts students (Roy, et al 2003). Although a customised programming environment is used, only six concepts have been identified as necessary. The same approach can be used with the object-oriented paradigm, as illustrated by Sajaniemi and Hu (2006). They choose to introduce the behaviours of variables and control structures prior to objects. Again, this represents the building up of small pieces into more complex logic.

This approach incorporates the advantages of early introduction of precisely defined language syntax and early introduction of tools to assist in the programming endeavour. The different constructs of the language, broadly speaking, variable, assignment, conditional, and repetition can be taught in isolation. Further exploration of the different constructs can be accomplished by demonstrating how to vary their behaviours by changing their syntax. This approach also ensures that developed algorithms, incorporating multiple blocks, should execute, providing they are syntactically correct. This, of course, does not imply that any particular combination of blocks will be logically correct. Further, mastering the tools needed to express and verify the behaviours of the blocks provides learners with debugging tools and strategies that will be needed to develop real solutions. The immediacy of the feedback when working in a development environment can be a benefit to learners. Errors in implementing the syntax of a programming language led to frustration for Al-Imamy, Alizadeh, and Nour (2006). To overcome this, they developed a customised environment that makes use of construct templates where learners fill in the blanks. For example, the template for a conditional block can be presented with the correct syntax structure laid out. The learner only needs to fill in a box representing the test.

The building blocks approach may present challenges for novice learners. Recall from above that interactive learning requires mastering some complex tools at the same time as mastering the behaviours of the blocks. The scale of this task should not be underestimated. As stated above, Lui, et al (2004) recognised that the failure to compile, i.e. construct syntactically correct code, can have a derogatory effect on novice programmers. In addition, individual block constructs may not execute or may not be meaningful to execute even if the syntax is correct. For example, an empty repetition or an empty conditional are syntactically allowed in some languages but are not logically meaningful. Mastering individual block behaviours may not transfer to the building activity required to produce an algorithm that performs a meaningful task. It is possible for a learner to explain and understand a concept, yet be unable to apply that understanding in the construction of an algorithm (Laitinen, Ala-Mutka, and Järvinen 2005). In addition, the relationships and interactions between blocks may be difficult for learners to grasp without the context of a problem to solve. However, the introduction of a simple problem context could be a follow-on activity.

Although the use of the building blocks approach may be challenging for learners and educators alike, it does provide for the early introduction of a precisely defined language syntax and the early introduction of a toolset to aid code writing. Introducing each language construct, one at a time, allows for natural exploration until its behaviour is understood. Once a basic behaviour is understood, experiments in varying the syntax of the constructs can be undertaken to deepen learning.

Simple Units

In the simple units approach to teaching programming, learners master set phrases using a limited vocabulary before combining the phrases to create meaning. Useful and reusable units of programming code are constructed in a development environment with a defined language syntax. Two studies exemplify this use of simple units that result from the solution of small set tasks. These reusable units can be held in a programmer's toolbox, for use in more complex solutions. Small, highly focused contexts can be used to combine constructs to create units useful in larger solutions. In common with building blocks, the simple units approach requires mastering tools and constructs at the same time. In addition, new problem solving skills can be mastered.

The simple units approach is analogous to learning to speak a language from a phrase book with a limited vocabulary. In this approach, the individual nouns, verbs, and adjectives are put together to form useful phrases. It is possible for the meaning of the phrase to be understood without complete understanding of the meaning of the individual components. However, decomposing the meaning of the phrase can lead directly to a better understanding of the individual components.
In this approach, constructs are put together to form useful and reusable units of code. It is a way, not necessarily the only way or even the best way, of building commonly occurring units of code. For example, a set of constructs could be put together in some way to sort a list or to find the maximum of a set of numbers. Small, well-specified problems, such as making a visual object bounce when it touches a maze wall, could be solved with the use of a small number of constructs. In themselves, these units may seem trivial. However, they become more powerful when joined together to effect solutions to larger problems. As with the building blocks approach, the learners will necessarily need a development environment and, unfortunately, as mentioned above, will be exposed to all the problems and issues associated with it.

This approach has been successfully employed in two studies whereby learners created reusable units of code to provide a basis for complex solutions. Regez (2006) at the University of Washington, Seattle uses the idea of small set tasks to increase both enrolments and student satisfaction. Although the course is taught in Java, the approach relies heavily on mastering the basics of problem solving. Small set tasks are designed to incorporate the use of constructs to create a solution. However, each small task, such as displaying an output, repetition, taking input, or reading a file, can be used as a simple unit. Once the units have been written and understood, they can be joined together to create the solutions to more complex assignments. This approach of unit building is extended by Lui, et al (2004) who suggest that creating a toolbox of key program segments is valuable for weaker students, who find programming from scratch challenging. This toolbox gives learners, especially weaker ones, a starting point from which to develop solutions.

The ability to introduce small, manageable, and highly focused contexts is an advantage of this approach. If the required focus is a single construct, then a context can be chosen which will best illustrate its behaviour. For example, when introducing the repetition construct, the concept of average could be employed. Constructing a simple unit that finds the average of a set of numbers can easily be written using a repetition construct but will also result in a useful code segment that can be incorporated into programs that are more complex. Combining individual constructs leads to simple units; combing simple units leads to more complex units. An example of this is a unit that finds the maximum of a set of numbers. To implement this, a repetition and a conditional are required. Again, this simple unit can be used in more complex logic. Using these simple, but meaningful contexts also helps develop problem solving and debugging strategies. It is easier to identify a logic error if the outcome is well known, i.e. the maximum number.

As with the building blocks approach, interactive learning requires mastering some complex tools at the same time as learning to construct the simple units. This is often a hindrance to beginner programmers. In addition, the introduction of a context does not remove the necessity of mastering the behaviours of the individual constructs, so some overlap with the building blocks and/or code analysis approaches may be unavoidable. Furthermore, mastering the problem solving skills necessary to resolve the set tasks must occur simultaneously with learning the tools and mastering the block behaviours. As the objectives of the approaches become more complex, so do the set of skills necessary to address the complexity. In a worst-case scenario, this approach may degrade to repeated cycles of differing combinations of simple units, code analysis, and building blocks.

By combining basic constructs into simple units that solve small well-defined problems, learners can create toolboxes of useful and reusable code fragments. Although a defined language and a development environment are required for this approach, it has the advantage that the resulting units provide a starting point for learners to begin writing code solutions. The useful skill of decomposition, breaking a problem down into small solvable parts, is also developed using this approach.

**Full Systems**

In the full systems approach to teaching programming, learners are immediately immersed in the full use of language constructs and tools. This approach involves a non-trivial problem solution. Programming concepts and language constructs are introduced as needed to solve the problem. Four studies report success when using a full systems, problem-solving approach in object-oriented and visual language environments. Although it might seem counterintuitive given their complexity, real problems and their full solutions motivate some learners. This approach, in common with some of the above approaches, requires mastering many skills and tools at the same time.

The full systems approach is analogous to learning a language by immersion. This type of learning is encountered by learners in foreign language classrooms where only the target language is used, in English speaking classrooms where learners understand limited English, and often when living in or visiting a country where the native language is not understood. Learning by immersion usually requires the learner to master reading, listening, speaking, and writing all simultaneously.

In the full systems approach, learners design or help design a solution to a non-trivial problem. The
programming concepts and language constructs are introduced only when the solution to the problem requires their application. For example, a 2-player game of tic-tac-toe could serve as an introductory problem. To produce this, learners would need to demonstrate skills in decomposition (understanding the problem), handling inputs from the keyboard, displaying outputs in some format, tracking turns (variables/assignments), determining availability of position (selection/conditional), and identifying a winning move (repetition). The language constructs, show in ( ), would be introduced only when that part of the problem needs to be solved.

Four different studies have indicated success in using a full systems approach, incorporating problem solving, to teaching programming. Duke, et al (2000) employ this approach in their beginner classes with Java. In order to hide some of the object-oriented complexities, they provide a set of customised classes for their students. In one assignment, the students were asked to implement the game of 4-in-a-row. The constructs the students were expected to master were the use of repetition and Boolean expressions to maintain a system's internal logic. The students reported that this approach allowed them to learn Java effectively. The objective for Nuttall, et al (2008) is acquiring system design skills. For their students, they set tasks such as a robot in a maze. Students are expected to decompose the problem, design classes, and develop algorithms to create an effective solution. Interestingly, the actual production of the solution is not always part of the task objective, although practice using real programming tools is also included in the course. Storytelling with Alice, a 3D visual programming world, is the context chosen by Sattar and Lorenzen (2009) for introducing their learners to programming using a full systems approach. Students are presented with a set storyboard scenario and expected to develop the code to animate the story. Concepts (object, world, and scene) and constructs are introduced when needed to progress the animation. Another example of a full systems approach is used by Campbell and Bolker (2002). They employ immersive techniques and ask their beginners to read and alter a bank ATM simulation. Their approach focuses immediately on interfaces, architecture, and design. The syntax of the chosen programming language is addressed only when needed. They admit that this approach is not easy, but it places greater emphasis on design and skills than on mastering syntax.

Although it may appear that this approach should be daunting for the learners, the advantage is that they can be presented with real problems for which they should already have conceptual models, for example, a vending machine or a telephone billing system. In addition, they may feel motivated and empowered that they are learning to program a system that is representative of real life. They can also attribute the mastering of small steps to the eventual solution of a real problem. For example, the vending machine code may need to be modified to give the correct change from a £2 coin. If the learner can make this change, then the full system has become more functional and effective.

By definition, the immersion or full system approach requires mastering many skills simultaneously. The programming environment tools, individual block behaviours, interactions between blocks, and debugging must be confronted together. This could be demoralising for some beginners, as Lui et al (2004) has identified. From a teacher's perspective, the problem selection must be well considered to fulfil the underlying requirements of mastering the necessary constructs and acquiring the skills to work in the programming environment.

Although it may seem counterintuitive, the full systems approach may motivate some learners. Being able to understand how changes to code make a solution more effective can quickly move some learners on to further investigation. Real problems, in an already known context, may help develop more refined problem-solving skills, such as decomposition. Although, in common with the simple units and building blocks approaches, a defined language syntax and development tools must be mastered, this approach has the advantage of working on non-trivial solutions right from the beginning.

Suggested Combinations of Approaches

Effective programmers have some level of skill in explaining, tracing, and writing programming code. Each of the four previously described approaches to teaching programming allow development of one or more of these skills. Combining different approaches provides for progression toward becoming an effective programmer. Some possible combinations of approaches are illustrated in Figure 1. Different toolsets lend themselves to different paths around the model, based on the capabilities and needs of the learner.

The ability to write code requires some skill in tracing and explaining code (Lister, Fidge, and Teague, 2009). Each of the four approaches identified above encompasses development of one or more of these skills. Effective programmers exhibit skills in code analysis (tracing and explaining), in understanding block behaviours (tracing and explaining), in constructing simple units (writing), and in combining simple units to create full systems (writing).

By combining the approaches to teaching programming, it is envisioned that learners will be able to evidence
progression toward becoming effective programmers, regardless of their age and capabilities. Figure 1 is an attempt to represent possible orderings of these approaches that could facilitate progression. The linear approach from building blocks to simple units to full systems is perhaps the most obvious line of progression as it represents stepped movement in complexity. However, the reverse path is also a type of progression and can represent the acquisition or deepening of knowledge by the learner.

Orders of approaches presented in this model are dependent only on the starting point. A final decision about the approaches used and their order of presentation should be based on the requirements of the course, the age appropriateness of the programming environment or language, the capabilities and prior experiences of the learners, and the confidence of the teacher.

Figure 1: Ordering of Approaches Showing Progression

The use of a visual programming environment such as Scratch can be used to illustrate the approach of building blocks, simple units, and full systems. In this environment, the learner gives instructions, represented by puzzle pieces, to cause a visual object to move around a stage and interact with its environment. The behaviours of a subset of the available puzzle pieces could be introduced and demonstrated to the learner. After mastering the behaviours of this subset, the learner could put the blocks together to control the actions of the object. Once a sufficient number of these simple units have been constructed, they could be assembled to create a full system. An objective for a full system using this type of environment could be a short animation or a simple interactive game.

Visual environments also lend themselves to the order: simple units, building blocks, full systems. In the Alice programming environment, learners instruct a visual object to move around and interact with other visual objects represented in the world. Introducing simple units first might entail the learner being supplied with several objects possessing predefined behaviours. The simple units, in this case methods belonging to the Alice objects, could be combined to create interactions between objects. Progression toward building blocks could be accomplished by being able to understand and being able to modify the code in a simple unit to change the interactive behaviours between the objects. Once units can be constructed and the blocks can be understood, learners could create a full system, such as a story animation or a simple interactive game.

Full systems, as described above, may be the most challenging approach for learners, but by giving them access to a fully or partially functional system from the beginning, the students may be more motivated to engage with the learning. A functional system, for which the logic is well known, such as the arcade game Asteroids, could be presented to learners directly in the programming environment. This could be achieved in an environment such as Greenfoot, which is based on the Java programming language. New or modified behaviours could be explored by changing the code in a simple unit or by introducing new simple units. For example, a simple modification might require five hits before the asteroid explodes or a more complex introduction could involve growing asteroids when they collide. As cautioned above, this approach entails simultaneous learning of problem decomposition, a programming environment, language syntax, and debugging strategies.

Another option incorporates code analysis, building blocks, simple units, and full systems. This order has the attraction of exposing the learners to programming concepts in the analysis phase. The code analysis could be based on the use of pseudocode similar to a text-based language. When attempting to master the building block constructs of the chosen language, the learners could be asked to use a programming environment appropriate to that language. Because the tools for developing in text languages are targeted
Appendix 5

toward professional programmers, learners can find the mechanics of program creation challenging (Kolling, et al 2001). One programming environment based on Java that may address this issue is BlueJ. The next step, simple units, can give learners the opportunity for constructing parts of their toolbox of reusable program segments. For example, useful segments might include sorting a list of numbers or validating an email address. A full system gives the learner the chance to incorporate real problem solving with the use of blocks and their simple units. At this level, a full system might consist of a logic game, modelling the behaviour of a vending machine, or a quote generation system.

Conclusion

Upon reading published literature devoted to the learning of computer programming, common teaching approaches began to emerge. The characteristics common to these approaches were analysed and refined to fit into one of four identified approaches to teaching programming: code analysis, building blocks, simple units, and full systems.

Some research (Lister, Fidge, and Teague 2009) indicates that that, in order to write code, the learner must possess some skill in tracing and explaining code. Learners’ development of these skills can be enhanced by engagement with any or all of the previously defined approaches to teaching programming. Progression toward becoming an effective programmer could be evidenced by movement between the approaches:

- Analysing code to divine logic,
- Understanding block behaviours of language constructs,
- Constructing simple units of useful and reusable code, and
- Combining simple units to create full systems.

A visual representation of possible progression through the different approaches is presented in Figure 1. The final choice of approaches and their order of presentation should be based on the requirements of the course, the age appropriateness of the programming environment or language, the capabilities and prior experiences of the learners, and the confidence of the teacher. Regardless of the starting point, movement through the approaches provides learners with opportunities to progress and construct new knowledge.

Further research is needed to better understand these and other approaches to teaching programming, not in terms of learner outcomes, but in terms of teachers’ actions and techniques employed to facilitate the construction of new knowledge by the learners. Effective classroom teaching practices could be informed by further investigations into the effect on progression of different toolset choices and combinations of teaching approaches.

References


Appendix 5

Proceedings of the 38th SIGCSE technical symposium on Computer science education. Covington, Kentucky, USA: ACM.


Appendix 6. Teaching Programming – A Theoretical Framework

Teaching Programming - A Theoretical Framework

Maria Saelli
University of Eindhoven
Campus TU/e
Eindhoven NL
31 (0) 40 24726
m.saelli@tue.nl

Cynthia C. Selby
University of Southampton
Highfield
Southampton UK
44 (0) 2380 593475
C.Selby@soton.ac.uk

John Woolard
University of Southampton
Highfield
Southampton UK
44 (0) 2380 592986
J.Woolard@soton.ac.uk

ABSTRACT
This paper explores aspects of the teaching of computer programming in secondary schools (Gr-10) focusing on the difficulties of teaching and learning associated with the topic of arrays. An emphasis is placed on teachers’ Pedagogical Content Knowledge (PCK) and how it may be utilized to mitigate the difficulties and facilitate the learning of computer science topics. Research is currently being conducted in the important effort to portray the PCK in the context of IT, ICT, computing and computer science. Initial results of a multi-national European study into the PCK of teachers and trainee teachers are presented.

Categories and Subject Descriptors
K.3.2 [Computer and Information Science Education]: Computer Science Education

General Terms
Human Factors, Languages, Theory

Keywords
Teaching Programming, Secondary Education, Pedagogical Content Knowledge, Arrays

1. INTRODUCTION
The research efforts undertaken in the Computer Science Education (CSE) environment lead to a better understanding of the teaching and learning taking place in computer science classrooms. Previous research has uncovered information concerning students’ misconceptions and difficulties, learning environments, choice of programming language or paradigms, and methods used to engage and retain students. However, these are not the only topics within CSE that would benefit from exploration.

Pedagogical Content Knowledge (PCK) is one topic that is under-represented in the CSE literature. PCK is an important construct that has been widely explored in other fields (e.g. science, chemistry, English) [4][6][9][15] and is related to the knowledge teachers have about the teaching of the most common topics in their subject.

2. PEDAGOGICAL CONTENT KNOWLEDGE
Teachers are the main actors in the teaching process, and there is no exception in CSE. While teaching, teachers gain experience and knowledge in a number of areas, for example, the most common misconceptions students have while learning a certain topic, the most powerful examples to use in order to facilitate students’ understanding, insight into the necessary prior knowledge students need to have in order to learn a new topic, or concepts that are at the heart of learning a certain topic.

Lee Shulman [4][6][9][15][22][23] focused his work on knowledge growth in teaching. When teaching, teachers aim to foster students’ learning by using representations of their own content knowledge. As a matter of fact, this knowledge grows through the teaching experience. The latter confirms the popular saying “if you want to learn something well, offer to teach it”. This was the starting point for Shulman’s research. It can be imagined that there is a difference between teachers as they finish their teacher training and teachers who have already secured years of teaching experience, in terms of content knowledge but also regarding other aspects. In fact, PCK is knowledge that incorporates experienced teachers’ reformulation of a topic’s content and is a knowledge that grows though teaching experience. Shulman, who defined PCK as a result of his study, argues that:

[…] there are no single most powerful forms of representation, the teacher must have at hand a veritable armamentarium of alternative forms of representation, some of which derive from research whereas others originate in the wisdom of practice. Pedagogical content knowledge also includes an understanding of what makes the learning of specific topics easy or difficult: the conceptions and preconceptions that students of different ages and backgrounds bring with them to the learning of those most frequently taught topics and lessons. If these preconceptions are misconceptions, which they so often are, teachers need knowledge of the strategies most likely to be fruitful in reorganizing the understanding of learners, because those learners are unlikely to appear before them as blank slates. [22]

Because of its importance, this construct has been largely adopted among scholars of different disciplines [4][9][15][23] and some scholars have reformulated it [9][11][24]. When analyzing the literature in the subject of CSE, it is discovered that surprisingly little is known about the specifics of the PCK of computer
Appendix 6

science. The only example available is by Woodland [25], who studied the role of metaphor in the teaching of computing. Metaphor can be viewed as a form of representation, referred to in Shulman’s definition of PCK.

In this paper, PCK is interpreted using Grossman’s reformulation [9][10]. PCK is defined as the answer to these four questions (Figure 1):

- Why to teach?
- What to teach?
- Learning difficulties...
- How to teach?

Figure 1 – Diagram based on Grossman’s reformulation of PCK [10]

It is important to study and portray the PCK of a discipline because this knowledge helps to better design teacher-training courses and helps novice teachers in their first years of teaching. As mentioned earlier, PCK is a construct that grows with the years of teaching experience. One of the characteristics of PCK is that teachers have at their disposal several ways to address the most common taught topics within a certain subject. The more examples teachers have at their disposal and the better they recognize learning difficulties, the more effectively they can deploy their PCK [24]. However, more teaching to test the only method to effect an improvement of a teacher’s PCK. Grossman [9] observed that teachers develop their PCK if they attend a teacher-training course or some form of in-service training [24].

3. TEACHING PROGRAMMING

When discussing the teaching of a subject, it is acceptable to refer to the PCK of that subject because PCK is specific knowledge relative to the teaching of a subject. The teaching of PCK is explored in this paper. Answers are sought specifically to the last two questions of Grossman’s reformulation of PCK, namely: “What are students’ difficulties?” and “How to teach?”

The results of a study are presented which aims to portray the PCK of programming and compare it with the literature available from CSE. The study [26] involved 31 participants—experienced teachers and teacher-trainers—drawn from different European countries. Participants of this study were invited to participate in a workshop. The four aspects of the PCK of different topics within programming were discussed in the format of a structured group interview. The teaching of two topics, arrays and variables, is further explored. Study results from Satch and colleagues [26] include the PCK relative to the topic of arrays. This topic is explored further in the following sections.

4. ARRAYS AND VARIABLES

Very few would dispute the fact that learning to program is a difficult task for novice programmers. Indeed, introductory classes must cover a variety of basic concepts and topics to facilitate students’ progression. Reported research exists which attempts to classify some of these programming topics according to their difficulty to learn. Much of this research reflects the opinions of students and teachers in entry level university classes, although some secondary and high schools are represented.

4.1 Difficulty of Learning

A variety of programming paradigms is represented in the research discussed below. The object-oriented paradigm is represented in four studies [7][14][16][19] by the students with experience of Java, C++, and Simulink programming languages. The imperative/procedural paradigm is represented in two studies [14][19]. The visual approach is represented in a single study [18], the Alice framework [1].

In order to make comparisons between the presented studies, numerical results, where applicable, have been converted to a standard 10-point scale and rounded to a single decimal place. The value 1 represents very easy, the value 10 represents very difficult.

In an international study [14], over 500 students and teachers were asked to rank topics according to their perceived difficulty of learning and teaching. Students assigned arrays a difficulty rank of 5.4, while teachers perceived them as more difficult at 6.2. Variables were also perceived to be less difficult by students (ranking 4.2) than by teachers (ranking 4.5).

An earlier, smaller, UK-based study [16], revealed similar rankings. Students ranked arrays at difficulty level 4.6, while teachers placed them higher at 6.2. Students perceived the learning of variables to be less difficult (3.7) than the teachers’ ranking of 6.2.

A 2005 study [21] included participants from German high schools and Danish colleges. Here, rather than how difficult the topic was perceived to be by the students, the rankings indicate how many of the students had difficulty with the topic. This is an adequate proxy for a level of difficulty. As before, a rank of 10 indicates that almost all the students had difficulty in learning the topic. In this study, respondents assigned a difficulty level of 5.4 to arrays and a difficulty level of 5.2 to variables. The authors reported no discrepancy between the figures reported by university, college, and high school teachers. For example, arrays ranged from 4.8 to 5.2 on the comparison scale. Variable types (e.g., scope) were ranked harder, but consistently across the group (5.8 – 5.9).

An Australian university study [2] surveyed introductory programming students about their perceived difficulty of several programming topics. The authors begin with a proposed level of difficulty for the topics. They then ask assign both arrays
Appendix 6

and variables to the mid-range of difficulty. Responses from students affirmed the assigned rating of arrays with a value of 6.0 on the comparison scale. However, they assigned a rating of 3.1 to variables, which places the topic in the range of less difficult concepts.

The difficulty of learning arrays was again uncovered by a study that was focused on the objects-first or objects-later debate [7]. Results were based on a series of tests given to the students throughout the duration of the class. Arrays were identified as the second most difficult topic to learn. In addition, arrays were identified as the third hardest topic by the Monash students [2]. In both studies, the difficulty of learning arrays was only exceeded by the difficulty of some specific object-oriented concepts.

The difficulty of understanding variables is further highlighted in a study using the microworld Alice [18]. The authors observed that when the concept of variables was completely removed, as can be accomplished by using the Alice objects native behaviors, the understanding of other control concepts such as loops, conditionals, and events were easily grasped.

There is evidence [2][1][4][16][18][21] as to the difficulty of learning the introductory topics of arrays and variables, as perceived by students and teachers. The level of reported difficulty of these topics is consistent across the studies [2][1][4][16][21]. The two topics of arrays and variables are reported as difficult to learn across a variety of paradigms and programming languages [7][14][16][18][19].

4.2 Reasons for Difficulty

Of course, arrays and variables are not the only challenging topics for students. The topics ranked higher in difficulty tend to include those involving higher levels of abstraction, for example, how to design a program to solve a problem, procedures, debugging, recursion, and pointers [14].

Some researchers have made suggestions as to why students find the introductory programming topics so difficult to learn. An often-cited work by du Boulay [3] proposes one reason for the misunderstanding of arrays. He suggests that students become confused between an array subscript and the value stored at that memory location. As for variables, his work indicates that the initialization of variables is better understood than assignments to variables. Furthermore, updating and testing of variables were treated as equivalent by the students and were understood better than initializations [3].

The authors of the UK-based study [16] conclude that the most difficult topics result from a lack of understanding of what happens in memory when a program executes. The topics indicated as difficult by the students in this study are identified as having to do with memory and pointer operations. The issue is that memory and pointer manipulations underpin large numbers of more sophisticated concepts such as parameter passing by reference, objects, pointers used with character arrays, and pointers used to create complex data structures like lists and trees.

In the 2009 study by Ehler and Schulte [7], the authors conjecture that arrays are more complex because of:

- the need to understand iteration in implementing or addressing the members of the array

This idea that learning arrays incorporates the understanding of iteration and the additional processing logic necessary to address array members is echoed in the previously presented Monash University study [2].

In their review of literature, Robbins, Routtree, and Routtree [19] agree that the inability of students to establish a working model of the machine or the establishment of an incorrect model of the machine may contribute to the difficulties exhibited by novice programmers. They go further to suggest that the difficulties are exacerbated by the students’ inability to reconcile the differences between the actual execution of the program and what the student intended the program to do.

Appreciating that the topics of arrays and variables are difficult to learn is a step in the right direction. Having some understanding of why students find the topics difficult is another step. The next step should be the identification of successful teaching strategies that can be employed by teachers to assist students in overcoming these difficulties. The fuller understanding of the PKC of computer science could aid in the identification of these successful teaching strategies.

5. CONCLUSIONS

The PKC of different topics within computing, including arrays, was explored in a study by Saatli, Perrenet, Zwaneveld and Jochems [20]. Adhering to the PKC definition, data was also collected about students’ difficulties in learning individual topics. Information provided by the respondents regarding their learning of the array topic evidenced difficulties in understanding:

- that there is one name and several places;
- the use of a variable as an index;
- the range check error (when the index is going out of the array);

The respondents indicated these difficulties by discussing programming independently from any specific programming language paradigm. Regardless of the language, students have difficulties learning how to use arrays. It is interesting to note that the respondents’ answers are in line with what is found in the literature [2][3][7].

The reported difficulties do not appear to have changed with the passing of time. Du Boulay’s work [3] of 1989 identified the confusion students experienced using an array subscript while a similar situation is reported in 2010 by Saati and colleagues [20]. This could be a sign that little progress has been made in practice despite research efforts to understand issues related to the teaching and learning of programming.

Having knowledge of the PKC of programming facilitates being able to better design teacher training courses, to better design textbooks and to guide teachers with a weak content knowledge background to teach a subject outside her/his specialty. The latter is a very common situation in computer science classes in different countries at secondary level, where often teachers with a degree in other subjects (e.g. mathematics, economics, art, etc.) teach computer science. Those teachers often have a weak content
knowledge of computer science and would benefit from the opportunity to access the knowledge of experienced teachers with a computer science background and a strong PCK.

This leads to a call for research in this field and to fully portray the PCK of computer science. With more knowledge available, it will be possible to write handbooks for novice teachers or teachers with backgrounds other than computer science, in order to help them better teach computer science and to answer questions such as: why to teach?, what to teach?, what are students’ difficulties?, and how to teach?

In other words, knowledge about the PCK of programming and the work that could derive from it would help to bridge the gap between research and practice.

6. REFERENCES


Appendix 7. Promoting Computational Thinking with Programming

Promoting Computational Thinking with Programming

Cynthia C. Selby
University of Southampton
Highfield
Southampton UK
44 (0) 2380 593475
C.Selby@soton.ac.uk

ABSTRACT
The term computational thinking has received some discussion in the field of computer science education research. The term is defined as the concept of thinking about problems in a way that can be implemented in a computing device. Of course, after having thought about a problem using computational thinking skills, the next step should be to use programming skills to implement a solution. This work in progress is exploring ways in which programming can be employed as a tool to teach computational thinking and problem solving. Data is collected from teachers, academics, and professionals from various industries. They are purposively selected because of their knowledge of or interest in the topics of problem solving, computational thinking, and the teaching of programming. This data is analyzed within the paradigm of the grounded theory approach. The results of an initial analysis imply an ordering of complexity associated with computational thinking skills, imply connections between computational thinking skills and programming activities, and imply a relationship between computational thinking skills and other taxonomies of learning.

Categories and Subject Descriptors
K.3.2 [Computers and Information Science Education]: Computer Science Education

General Terms
Theory

Keywords
Computational thinking, pedagogy of programming, problem solving

1. INTRODUCTION
Problem solving skills are used in developing or implementing strategies to solve problems in many domains. These skills are often expressed as heuristics [12], appropriate and plausible approaches to a problem. Effective problem solving has been promoted by the use of strategies including means-ends analysis, schema acquisition, algorithmic approaches, and targeted frameworks [19, 12, 9]. However, in the domain of computer science, some research [11] has found that learners do not naturally solve problems in ways that can be translated to computing devices by the use of programming. This disparity is highlighted in the Lister study [6], where it is suggested that ineffective problem solving skills, including the ability to work through lines of logic, may be the cause of ineffective programming skills. Additional studies [15, 13] suggest that problems in learning to program are exacerbated by a lack of strategic tools.

The strategic tools, identified as useful for those attempting to solve problems with the aid of computational devices in various domains, include, but are not limited to, decomposition, abstraction, simulation, and generalization [10]. The more given to these specialized mental skills, resulting in solutions to problems directly translatable to a computing device, is computational thinking [21]. Actually implementing these solutions requires a different set of skills.

Programming skills are the specific technical skills needed to produce specific solutions using a set of defined digital tools, often associated with a programming language [9]. Research often reports that learners struggle with programming skills such as tracing [3, 6] and understanding a model of the machine [2, 9]. Having recognized this issue, other researchers [7, 20] highlight the need for a defined hierarchy of programming skills. One study [16] attempted to provide such a hierarchy for object-oriented programming. The researchers found that teachers interpreted the hierarchy as a capability hierarchy.

Given that a hierarchy of generic programming skills could be developed and interpreted as capability, the levels could be aligned with existing hierarchies, such as the cognitive domain of Bloom’s Taxonomy. These same programming skills could be mapped to the higher-level computational thinking skills that they evidence, thereby defining a hierarchy of computational thinking skills. This setting provides the context for an ongoing investigation into the relationship between the teaching of programming and its effect on the acquisition of computational thinking skills by learners.

2. STUDY METHOD
This study is based on a grounded theory approach employing qualitative data collection methods and qualitative data analysis techniques. The first activity is the administration of an Internet based questionnaire. The second activity is the collection of data from an Internet based community of practice forum. The third activity is the administration of a face-to-face, audio recorded, semi-structured interview schedule for respondents previously identified by an analysis of the questionnaire results and community of practice discussions. All data is iteratively augmented and analyzed guided by Strauss and Corbin’s [18] grounded theory procedures and techniques until theoretical
Appendix 7

saturation. It is anticipated that a product of the theory generation may be a model of relationships between problem solving skills, computational thinking skills, and programming skills.

2.1 Participants and Sampling
The participants in this research all have some interest in the teaching of programming, computational thinking, problem solving, or any combination of the three. Not all participants are teachers. Participants may be employed in industries where computational thinking skills and programming skills are useful or required. Other participants may be members of professional communities of practice, representing industry, academia, or education. They are still perceived to have an interest in and appropriate knowledge of the research context.

An individual participant may not engage with every data collection instrument. Participants are matched to instruments. In the case of the first instrument, an online questionnaire, the targeted sample consists of members of organizations whose ideologies promote the teaching of programming or computational thinking skills. In the case of the second, the online community of practice, conversation threads are chosen purposefully for their applicability to the context of this research, without regard to the identity of the poster. From the questionnaire responses and the community of practice conversations, a further purposeful selection is made to identify participants for the interviews. This purposive sampling is supported by Strauss and Corbin who affirm that theoretical sampling is a foundation stone of grounded theory which, "... enables the researcher to choose those avenues of sampling that can bring about the greatest theoretical return" ([18], p. 202).

2.2 Data Collection
The questionnaire and interview schedule have been designed specifically to elicit responses applicable to the topics of problem solving, computational thinking, and the teaching of programming. To ensure the same level of appropriateness of response, a set of keyword criteria has been developed on which the community of practice messages are searched.

2.2.1 Online Questionnaires
The questionnaire makes use of some closed questions but the majority of questions are open-ended to allow participants to respond as they wish. The ordering of the questions is from general to specific, divided into major sections. Results are submitted one screen or page at a time. In this way, the results ofAbandoned questionnaires have the potential to be used. Personal information is requested early in the response process to identify participants. This provides a mechanism for contacting the participant, should he or she be selected for an interview. The design of the resulting questionnaire aims to be as open as possible to facilitate depth of response, while controlling for researcher and question bias.

2.2.2 Community of Practice
The community of practice, whose discussions and opinions are of interest in this study, is computer-mediated. Simply by contributing, the members signify some interest in the topics that overlap with this study. Although computer-mediated, some individuals share collaborative practices in the classroom. There are also face-to-face meetings, of varying scale, held throughout the year.

In order to identify the most appropriate threads for inclusion in the dataset, discussions are keyword searched. The keywords have been chosen to correspond to the terminology used in the initial research literature and early questionnaire responses. These terms include computational thinking, abstraction, decomposition, algorithm, and problem solving. Discussions, composed of individual and related messages, are considered as a whole. Regardless of the age of a discussion, once it has been identified as pertinent, every individual message in that discussion is read and coded, in line with the questionnaire and interview data.

2.2.3 Interviews
The design of the interview schedule used in this research is based on a semi-structured approach, as defined by Creswell, Miller, and Morrison [11]. In particular, the question wording and sequences are specified in advance of the interview. The interviewer grants the flexibility to provide additional questions in order to elicit greater depth in the responses. The interviewer is also granted the flexibility to record verbatim indicators, such as body language or gestures. This semi-structured approach should provide sufficient control to ensure comparability of results, sufficient flexibility to ensure depth of responses, and sufficient consistency to support the simultaneous collection and analysis of data indicated by the grounded theory paradigm.

3. FINDINGS
The current, non-saturated dataset is being analyzed in line with grounded theory, first as conceptual free nodes, then as categories. These categories and concepts may change, as more data is added and processed. Three of the categories presented here, problem solving skills, computational thinking skills, and programming skills have been introduced above.

3.1 Problem Solving Skills
In the context of this study, problem solving skills are not specific to programming, but are a wider skill set applicable in many domains. Participants have highlighted problem understanding and persistence as important concepts in this category. Analysis of the data indicates that a common key first step in both learning to solve problems and learning to program is being able to understand the problem and its constraints. This observation agrees with Polya's problem solving approach [12]. The theme of persistence is often linked with the idea of "not giving up". Puzzles and games are named as appropriate activities to promote persistence. They are identified as providing sustained and lengthy problem solving with discrimination of useful data, back tracking, and constant evaluation. Many participants recognize that the opportunity to problem solve is relevant in many different contexts.

3.2 Computational Thinking Skills
Participants in this study identified explicit examples of computational thinking skills, as defined by the National Research Council [10], and related them specifically to problem solving. Recognizable computational thinking skills such as decomposition, modeling, and algorithm design are found in the responses, along with other skills such as planning, justifying, and evaluating.

Decomposition, the skill to break problems down, is viewed as being taken for granted. However, for some students, this is reported as being very difficult and requiring explicit teaching. Modeling is identified in the sense of high-level systems that are decomposed into smaller parts, with each individual part modeling behavior of a subsystem. The act of planning an
algorithm or a product is viewed as a high-level computational thinking skill. Algorithm design is expressly tied to problem solving by the participants. It is defining the steps, using some accepted convention, to solve a problem. This is viewed differently to program design, which is seen as the translation of an algorithm into automation understandable by a computing device. Unexpectedly, participants also included learning to ask questions about alternatives, identifying trade-offs, justifying decisions, identifying limitations, refining solutions, and evaluating results. These are frequently complemented by the term analytical thinking, which is perceived to involve comparing alternatives, precisely describing, explaining how, and criticizing weaknesses.

In general, participants in this study agree with the National Research Council [10] and Wing [21] concerning the broad definition of computational thinking and none limited the use of the term or the skill set identified by the use of the term only to the domain of computer science.

### 3.3 Teaching Programming Skills

The concepts in this category represent high-level concerns for the participants. Included here are the concepts of logical thinking, programming as a tool, and collaboration as a pedagogic strategy. The term logical thinking occurs prolifically in the dataset and appears to be associated closely with programming constructs such as sequence, selection, and iteration. This association is anticipated and parallels that of Sahu [9] who reports that the most identified big idea of programming is control structures. The idea of programming as a vehicle for teaching computational thinking crosses boundaries between respondents, encompassing academics, teachers, and industry professionals. The components of computational thinking, such as decomposition and generalization, are also reported by Sahu [9] to be a big idea of programming. Collaboration is identified, by participants, as an effective strategy for teaching computational thinking. This is usually described as paired or group work, most commonly involving discussions at the analysis or design phases of software development. Notably, there are currently no responses indicating provision for group implementation or paired programming.

While it is not surprising that participants associate the teaching of programming constructs, decomposition, and generalization with computational thinking, it is surprising that an established pedagogic technique, collaboration, has not been extended to opportunities for paired programming.

### 4. CONCLUSION

#### 4.1 Preliminary Model

Although the dataset has not yet been shown to be saturated, as proscribed by grounded theory, it can form the basis for preliminary theory generation. As indicated in the introduction, participants' responses are used directly to derive a model of the relationships between computational thinking skills, programming skills, and the cognitive domain of Bloom's Taxonomy. Figure 1, a preliminary model, has been derived to illustrate some of these relationships.

![Diagram](image)

**Figure 1: Preliminary Model**

The computational thinking skills, reflecting the terminology [10, 21] introduced previously, are represented by an increasing level of complexity. This hierarchy can be discerned from the participants' responses and the reported order of introduction in the classroom. For example, breaking problems down, decomposition, is one technique introduced early in the teaching of both problem solving and programming. The programming activities column represents those activities that participants view as promoting computational thinking. For example, the collaborative work, reported by participants and described in 3.3, usually takes place during the analysis or design phase where a problem is broken down into subcomponents. Interestingly, the terminology used in Bloom’s Taxonomy, the last column, is represented directly in the participants' responses. The terms analyze and understand are also used to describe activities found in the initial stages phases of problem solving or programming task. Although the dataset on which the model is based will change and grow, possible relationships can already be discerned.

#### 4.2 Implications

This study assumes, in line with Isbell and colleagues [5], that computational thinking skills are a requirement of 21st century society and that these skills must be taught. This research contributes to the body of knowledge that may be used to inform the issue of effective teaching strategies for both programming and computational thinking. By more explicitly defining the relationship between computational thinking and programming, educators may be motivated to move the focus of activities from the production of an artifact to the acquisition of computational thinking skills. In the context of the current educational requirements to include more computer science at all key stages,
the results of this study could influence the design of curricula aiming to incorporate the development of computational thinking skills. In addition, this research responds directly to Grazioli's call [4] for more research into how to teach computing in a way that engages computational thinking.

4.3 Future Work

Although the current study has not yet reached its conclusion, areas for further study have already been explored by analysis of the data. These include:

- How do learners move from the specifics of programming, such as language constructs or blocks, to more abstract concepts, such as sorting an array or finding an average, which aid higher-level problem solving?
- How could the explicit teaching of general problem solving skills and high-level problem solving strategies influence the development of computational thinking skills?

More work into the relationships between problem solving, computational thinking, and programming could lead to improved classroom lessons, improved curricula, and an improved understanding of the skills required in 21st century society.

5. REFERENCES


Appendix 8. Computational Thinking: The Developing Definition

Cynthia C. Selby
University of Southampton
Highfield
Southampton UK
44 (0) 2380 593475
C.Selby@soton.ac.uk

John Woollard
University of Southampton
Highfield
Southampton UK
44 (0) 2380 592968
J.Woollard@soton.ac.uk

ABSTRACT
Since Jeannette Wing’s use of the term computational thinking in 2006, various discussions have arisen seeking a robust definition of the phrase. With little consensus having been found in the intervening years, there are even suggestions that a definition is not important. Perhaps focus should be on how computational thinking is taught and how its acquisition might be observed. However, in order to facilitate consistent curriculum design and appropriate assessment, it is argued that a definition should still be sought.

In order to contribute to the discussions surrounding a definition of computational thinking, this review of literature spans the years since 2006. The most frequently occurring terms, descriptions, and meanings are identified. Consideration is given to the motivation for inclusion or exclusion of a term by each individual author. Where possible, if a description has been given, an associated term is supplied.

Criteria are developed for the objectives of a computational thinking definition, in accordance with the needs identified in the literature. Using the criteria as a guide and the collected terms as the vocabulary, a definition of computational thinking is proposed which encompasses the thought processes of abstraction, decomposition, algorithmic design, evaluation, and generalization.

Categories and Subject Descriptors
K.3.2 [Computers and Education]: Computers and Education, Curriculum

General Terms
Standardization, Theory

Keywords
Computational thinking, definition, abstraction, decomposition, algorithmic thinking, algorithmic design, generalization, evaluation

1. INTRODUCTION
The term “computational thinking,” when used by Jeannette Wing [19] in her call to make thinking like a computer scientist a fundamental skill for everyone, excited educators (1, 2, 3, 4, 5, 8, 1, 11, 1, 14, 1, 15) and academics (6, 7, 9, 10, 12, 13, 16, 1, 20). This presented an opportunity to promote computer science to a wider audience, but it also introduced a challenge. Wing did not precisely define the term and state exactly what “computational thinking” is for everyone. Since then, there have been attempts by authoritative individuals and groups [1, 16, 9, 6] to derive a definition for computational thinking.

The aim of this investigation is to shed new light on the discussions that attempt to develop a definition of computational thinking with the objectives including: to define more narrowly, not more broadly; to bring an order to the criteria not necessarily to accommodate all viewpoints; to refine the definition to facilitate assessment; to retain the validity of work that has been done previously, such as the development of curriculums; to separate a definition from those activities that might promote acquisition of computational thinking skills; and to separate a definition from those artifacts and activities that evidence the use of those skills.

1.1 Method
A selection of literature relating to the topic of computational thinking was examined using the following literature analysis method. An Internet search engine query using the criteria “Jeanette Wing” AND “computational thinking” was initially executed. The entries of the first four pages were checked for applicability of title. All documents identified as having applicable titles, indicating a focus on computational thinking, were individually inspected. This resulted in six documents. The ACM Digital Library was searched using the term “Jeanette Wing”. The articles were filtered according to the abstract/introduction text and being dated post 2005. This led to the identification of thirteen items. In addition, articles describing proposed or current computer science curriculum designs (in Israel [8], Germany [3], New Zealand [2], India [4], England [5], and the USA [1]) were identified. This gave seven more documents. Because of repetition of comments by the same author, 4 of the original 26 articles were discarded.

In an attempt to contribute to the development of a definition, the publications were analyzed in chronological order to discern the development, over time, of the phrase computational thinking. Descriptions and suggested definitions of computational thinking were identified in each publication. The terminology, common across descriptions and definitions, was collated. Where equivalences allowed, similar terms were grouped together. The most frequently occurring individual terms and groups are presented in the following sections. From this basic collection of terms, a definition of computational thinking is formulated and proposed.

Justification for the inclusion or exclusion of terms is presented on a term-by-term basis. Justification is based on consistency of usage and consistency of interpretation across the literature. The
resulting definition reflects much of the consensus found in the literature while removing the less well-defined terms.

2. EVIDENCE FROM LITERATURE

Some authors/papers/communities may assert that a precise definition of computational thinking is not required [10, 13]. However, the discussion presented in this paper is driven by a perceived need to support professionals working in the field of computer science education and the developing computing curricula. This need for definition is supported in the literature [1, 10, 17, 16].

Guzdial [10] has suggested that a very broad definition is acceptable. Such acceptance could shift the focus away from what computational thinking is to how computational thinking should be taught and how evidence of its acquisition might be observed in learners. Professor of Computer Science, Chengjie Hu [13], supports this by citing that teachers are confident that the teaching of computer science does promote computational thinking. Even though they may not know exactly how this mechanism works, teachers recognize that the more learners practice computation, in terms of computer science, the better at computational thinking they become. This same argument is expressed by some of those who design or influence the design of computer science curricula. Several curricula [5, 4, 2, 3], while acknowledging the vagueness of a computational thinking definition, continue to include a focus on concepts and techniques from computer science. In presenting these concepts and techniques, the curricula include terminology often found in descriptions of computational thinking. Some of this terminology will be explored in more detail later in the process.

Jan Curry suggests that if computational thinking is included in a curriculum, it requires assessment. Without agreement on a common definition of computational thinking, it will be difficult, if not impossible, to develop appropriate assessment tools that actually measure the ability to think computationally [16]. So, a rigorous and agreed-upon definition might ensure that computational thinking in new curricula for the K-12 years will be more than, as Joyce Malyn-Smith argued, "...just a bunch of examples that are placed into the curriculum at the discretion of individual teachers" [17, p.33].

The balance of argument is still in favor of searching for a robust definition of computational thinking. Although it may be possible, without a robust definition, to identify examples of the practice of computational thinking, the ability to measure computational thinking may be hampered by that same lack.

3. CONSENSUS TERMS

Three terms appear consistently throughout the literature reviewed here. There appears to be a consensus that a definition of computational thinking should include the concept of a thought process, the concept of abstraction, and the concept of decomposition.

3.1 A Thought Process

When introducing the term, computational thinking, Wing [19] described it as a way that humans think about solving problems. It incorporates the set of mental tools used in computer science. These tools are used to transform a difficult problem into one that can be solved more easily. In adding his voice to Wing's, calling for the explicit teaching of computational thinking, Guzdial [9] refers to computational thinking as a way of thinking about computing. Participants in the workshop on the scope and nature of computational thinking [16], although not tasked with defining computational thinking, nevertheless agreed that science. This led to a range of mental tools and concepts from computer science. This idea is extended to represent problems as information processes and solutions as algorithms [7]. Al Aho [7] picks up the idea of problem transformation when he focuses on the thought processes in formulating problems and solutions that can be expressed as algorithms. These thought processes do have focus, frequently that focus is described as problem solving. Finally, Wing expresses this relation of computational thinking 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" (Curry, Section 4.2). Because of this consensus, a definition of computational thinking should include the concept of a thought process.

3.2 Abstraction

Although the idea of abstraction, hiding complexity, as being part of computational thinking is introduced by Wing in his original article [19], the definition develops over the subsequent years. She amends the definition to include simultaneous consideration for multiple layers of abstraction and consideration for defining the interface between the layers [20]. Even Peter Denning [18] acknowledges that abstraction plays an important part in computing, including programming. However, he points out that the act of abstracting is not universally agreed upon. The next year, Wing [21] defines abstraction as the cornerstone of computational thinking. Several participants in the workshop on the scope and nature of computational thinking (NRC) concur that computational thinking has a focus around the process of abstraction, creating them and defining the relationships between them [16]. More recently, in their report on workshops sponsored by the Computer Science Teachers Association (CSTA) and the International Society for Technology in Education (ISTE) to incorporate computational thinking into the K-12 curriculum, Barr and Stephenson [1] also include the ability to abstract in a definition of computational thinking. The concept of abstraction is explored by Ullman et al. [12] where it is one of aspects of their information technology approach to computational thinking. Because of this consensus, a definition of computational thinking should include the concept of abstraction.

3.3 Decomposition

Breaking problems down by functionality is identified by Wing [19, 20] as part of computational thinking. Decomposition is required when dealing with large problems, complex systems, or complex tasks. The participants in the first NRC workshop also identify the need for problem decomposition [16]. In the next workshop, focusing on pedagogy, participants extend this idea. Robert Tinker views the core of computational thinking as breaking down big problems [17]. Danny Edelson points out that the creation of solutions requires breaking problems down into chunks of particular functionality and sequencing the chunks [17]. Most recently, in refining his own definition of computational thinking, Guzdial [11] includes the use of tools including abstraction and decomposition. In light of this consensus, a definition of computational thinking should include the concept of decomposition.

Three terms are proposed for inclusion in the definition of computational thinking. Inclusion of a thought process, abstraction, and decomposition is supported by a consensus found in the reviewed literature. These terms are used consistently
Appendix 8

Submitted: SIGCSE 2014, 5-8 March, Atlanta GA

across the literature. Their use does not reflect any discrepancy in perceived meaning of the terms. Although consensus has been demonstrated for these terms, others receive less support and more varied interpretation. Some of these additional terms and their applicability for inclusion in a definition of computational thinking are discussed below.

4. POSSIBLE TERMS

Although less consistently than the terms above, several different terms and ideas do occur across the literature reviewed here. Even if a term or idea occurs, its interpretation is not always consistent across articles. Several ideas proposed as part of a definition for computational thinking are broad and high-level. A lack of specific interpretation may make inclusion of these terms in a definition difficult. The terms identified fall into these four areas: thinking, problem solving, computer science and imitation terms.

There are two descriptions of thinking, three general terms associated with problem solving, three terms associated with computer science concepts, and three terms associated with the concept of imitation or representation. The specific terms are: logical thinking and algorithmic thinking, problem solving, analysis, and generalization; systems design, automation, and more general computer science concepts; and modeling, simulation, and visualization.

Support for inclusion or exclusion of these terms in a definition of computational thinking is presented in this section. Justification is based on consistency of usage and consistency of interpretation across the literature.

4.1 Thinking Terms

Although the idea that computational thinking represents a cognitive process attracts consensus, there are suggestions that several specific types of thinking should also be included. These specific types of thinking are: logical thinking, algorithmic thinking, engineering thinking, and mathematical thinking. This section explores the viability of incorporating these types of thinking into the definition of computational thinking.

The concept of logical thinking, although not specifically defined, occurs several times in the literature spanning these years. Albert not perceived exactly as equivalent, terms to describe similar types of thinking are grouped into this category. These include mathematical thinking, engineering thinking, and heuristic thinking. In her original article, Wing [19] indicates that computational thinking incorporates heuristic reasoning to devise a solution. In addition to abstraction and decompositon, Gould [11] also includes heuristic reasoning as an appropriate tool to use when engaging in computational thinking. Computational thinking is equivalent to the logical reasoning used by people [12]. Logical reasoning is included by Iyer et al. [14] in their model computer science curriculum in order to promote high-level thinking skills that are not necessarily subject specific. L'Horeux et al. [15], in detailing an aspect of their information technology approach to computational thinking, define logical thinking as the ability to develop and test hypotheses.

Computational thinking also intersects with engineering because computer systems interact with the real world. However, computational thinkers can design and create virtual worlds, not limited by physical reality [20]. Although Wing [20] states that computer science relies on mathematics as a foundation, Gerald Stussman [16] affirms that mathematical thinking revolves around abstract structures while computational thinking revolves around abstract methodology. Computational thinking could be viewed as bringing science and engineering together. It could be viewed as a meta-science concerned with studying methods of thinking that are applicable to many different disciplines [16]. While the ability to think logically, analytically, and from an engineering perspective are certainly capabilities that a computational thinker may exhibit, references to these terms in this literature are not well expanded.

Although the term logical thinking, as described above, may not be suitable to include in a definition of computational thinking, the potentially analogous term, algorithmic thinking, requires further investigation. In her original article, Wing [19] does not use the term algorithmic thinking, preferring the word heuristic instead. However, by 2011, she extends her definition of computational thinking to include algorithmic and parallel thinking [22]. David Moorman [18] suggests that computational thinking is related to the idea of procedural thinking, as proposed by Seymour Papert in Mindstorms. He defines a procedure as a step-by-step set of instructions that can be carried out by a device.

The same theme is continued by Gerald Stussman [16], who defines computational thinking as a way of devising explicit instructions for accomplishing tasks. Inclusion of algorithmic thinking in a curriculum for high schools appears prior to Wing's contribution. In the Israeli computer science curriculum, Gal-Ezer et al. [8] placed an emphasis on inclusion of the study of algorithmic processes. There appears to be a consensus that computational thinking incorporates aspects of algorithmic thinking and algorithmic design. The term algorithm is interpreted as a step-by-step procedure for accomplishing tasks, not just in computer science, but in other disciplines. It is evidenced through the creation of algorithms - algorithmic design. Because of its wide acceptance and appropriate definition, algorithmic thinking may be applicable for inclusion in a definition of computational thinking.

Not all of the types of thinking proposed for inclusion in the definition of computational thinking bring further refinement to the term. Tying a definition of computational thinking to other terms such as logically or heuristically, with their open-ended interpretation, or to specific disciplines such as mathematics or engineering may not help advance the development of K-12 curricula and may not aid in the development of computational thinking assessment instruments. For these reasons, terms expressing the idea of logical thinking or equivalence may dilute a definition of computational thinking. On the other hand, algorithmic thinking is represented consistently in literature and its interpretation does not vary. Of all the potential terms associated with thinking, algorithmic thinking is the only possible term which may be suitable for inclusion in a definition for computational thinking.

4.2 Problem Solving Terms

The idea that computational thinking has some relationship to problem solving appears frequently in the cited literature. The specific terms problem solving, analysis, and generalization are most frequently employed in discussions of general problem-solving skills. This section explores the interpretation of these terms and the viability of incorporating them into the definition of computational thinking.

Problem solving, in one form or another, appears frequently in the literature presented here. There is agreement for describing computational thinking as a problem-solving activity. However, the literature does not illuminate problem solving in detail. Wing [19, 21], of course, incorporates solving problems using computer science concepts in her definition of computational thinking. The breadth of the problem-solving skills employed in
computational thinking, in opposition to specific technical skills, is pointed out by Larry Snyder [16]. A requirement for a computing device is introduced by Barr and Stephenson [1], who state that the essence of computational thinking is solving problems in a way that can be implemented with a computer. Peter Henderson [17] concisely describes computational thinking as a type of generalized problem solving with constraints. Problem solving is emphasized by Marcia Linn [16] who includes in the qualities of a successful computational thinker, the ability to engage in sustained investigative processes to generate problem solutions. Although there appears to be a consensus that computational thinking is a type of problem solving, the term may not be sufficiently specific to define it. Due to the breadth of the term, problem solving may not be suitable for inclusion in a definition of computational thinking.

The term analysis is included by some commentators in the definition of computational thinking. Interestingly, the term appears in relation to both problems and solutions, as in analyze a problem and analyze a solution. Analyze, in the context of problems, fits the category of problem solving, as defined above. However, analyze, in the context of solutions, could be interpreted as the comparable term evaluate. In her initial article, Wing [19] expresses the need for a computational thinker to make trade-offs, by evaluating the use of time and space, power and storage. This evaluation of algorithmic processes, including their power and limitations, is foreshadowed by Gol-Ezer et al. [8]. Application of the term to user interfaces is evidenced in the second objective of the New Zealand proposed curriculum, as part of designing programs [2]. In their IT approach, L’Heureux et al. [15] include the ability to evaluate processes, in terms of efficiency and resource utilization, and the ability to recognize and evaluate outcomes. Although the term analyze attracts some agreement for inclusion in a definition of computational thinking, descriptions of the term found in this literature imply an evaluative process. Therefore, because of interpretations described above, the term evaluate may be suitable for inclusion in a definition of computational thinking.

A specific term that appears sparingly in the literature definitions is generalization. It is the ability to move from specific to broader applicability, for example, understanding how to draw a square by defining internal angles, then applying the same algorithm to produce an approximation of a circle. The ability to recognize parts of solutions that have been used in previous situations or that might be used in future situations is evidenced by Kolenbrander in a definition of computational thinking [17]. These parts, or functional pieces, can be used to solve the current problem or combined in different ways to solve new problems [17]. The term generalization itself, is described in a proposed curriculum as recognizing common patterns and by sharing common features [5]. The idea moves forward from decomposition, described above. Generalization is the step of recognizing how small pieces may be reused and reconfigured to similar or unique problems. Although the exact term, generalization, is used sparingly in the literature, the idea of recognizing and reusing common parts of a solution is a possibility for inclusion in a definition of computational thinking.

Possible terms examined in this section include problem solving, analysis, and generalization. Problem solving is a broad term which, although used consistently throughout the literature, is not well defined. Analysis, used in the context of a problem, is also a broad term, often incorporating the ideas of abstraction and decomposition, as discussed above. Analysis, used in the context of a solution, is analogous to evaluation and is used consistently in the literature. Although the term generalization is used infrequently in the literature, there are descriptions of analogous processes. Therefore, from this set of possible terms, the ones used most consistently, with the least disparity of interpretation, and which may be suitable for inclusion in a definition of computational thinking are evaluation and generalization.

4.3 Computer Science Terms

The authors cited here concede that computational thinking has a deep relationship with computer science. Some suggest specific computer science terminology to be included in a definition of computer science. The specific terms include systems design, automation, and more general computer science concepts such as recursion and recovery through redundancy. This section explores the viability of incorporating these terms into the definition of computational thinking.

Systems design, although not mentioned frequently, is still used to describe computational thinking. Designing systems based on concepts used in computer science is mentioned by Wing [19]. Again, this inclusion is foreshadowed by Gol-Ezer et al. [8] who incorporates the study of the design and implementation of computing systems in their curriculum. One of Peter Denning’s Great Principles of Computing includes a category based on the design and building of software systems [8]. He goes further in describing systems as one of the four core practices, in which computing professionals engage, along with programming, modeling, and innovating [18]. The focus in each of these cases is systems design as a product oriented process. It is evidence of the ability to think computationally, not necessarily a definition of it. Therefore, the term systems design may not be suitable for inclusion in a definition of computational thinking.

Another term, popularized by Wing in defining computational thinking, is automation. She connects the term to that of abstraction when discussing the mechanization of abstraction layers and the relationships between them [20]. Even Denning acknowledges that this is what happens when programming [18]. Later, a stronger connection is made by Wing [21] when defining computing as the “automation of our abstractions” (p. 3718). This introduces the need for a computational device to interpret the abstractions, the need for a computer to execute a program. The process or processes required in the creation of these automations may be possible terms for defining computational thinking.

On the other hand, a program artifact, similar to systems design as discussed above, is only evidence that computational thinking has taken place. Previously, a consensus was presented that emphasized the thought process aspect of computational thinking. Based on this consensus, automation, interpreted as a program artifact, may not be a useful addition to the definition of computational thinking.

Throughout the literature, terms closely related to the general content of computer science studies appear in descriptions of computational thinking. Wing [20] herself introduces computer science concepts such as thinking recursively, interpreting code as data and data as code, type checking, prevention, detection, recovery through redundancy, damage containment, error correction, debugging, and caching. Additional concepts such as parallel processing, testing, debugging, search strategies, algorithmic complexity, and pattern matching are recognized in the NRC report [16]. Barr and Stephenson [1] include the abilities to think iteratively and recursively. Closer analysis reveals that not all of these concepts are unique to the field of computer science. For example, mathematicians think iteratively and engineers plan for recovery through redundancy. While each of
these concepts may be mastered by computational thinkers, none of them uniquely defines or helps narrow a definition of computational thinking. Therefore, terms interpretable as computer science content may not be helpful in defining computational thinking.

Possible terms examined in this section include systems design, automation, and more general computer science concepts such as recursion and recovery through redundancy. Systems design, resulting in a product, is evidence of the use of computational thinking skills, not a definition of it. Again, automation, as a product or program, evidences the use of computational thinking skills. Finally, those terms that are interpretable as computer science content do not bring focus to the definition of computational thinking. Therefore, none of the suggested terms discussed in this section appear suitable to be included in a definition of computational thinking.

4.4 Imitation Terms

Three additional terms, also used in discussions of computational thinking, are modeling, simulation, and visualization. These terms appear frequently in the cited literature. This section explores the viability of including these terms in a definition of computational thinking.

Wing [19] began by defining computational thinking as modeling the appropriate parts of a problem to facilitate a solution. Later, Brian Bliss [16] insists that the definition of computational thinking should include modeling and visualizations. Brinda, Pulham, and Schulte [3] have identified, as one achievable curriculum standard, the processes involved in modeling data. On the other hand, Edward Fox and Janet Kolodner [16] point out that it is the manipulation of abstractions (models, simulations, and visualizations) that contribute to the development of computational thinking skills. Observing the results of changing variable values, forming hypotheses, finding anomalies in data, and identifying invariants can all be achieved by interacting with models, simulations, and visualizations. The manipulation of these representations are agreed to enhance the development of computational thinking skills, but do not necessarily define it. Although these tools are effective aids in developing computational thinking skills, they may not be suitable for inclusion in a definition of computational thinking.

The following section, based on the term’s consistency of use and consistency of interpretation across the literature, summarizes the arguments presented above and suggests a definition of computational thinking.

5. PROPOSED DEFINITION

The intent of this investigation is to shed new light on the discussions that attempt to develop a definition of computational thinking. The objectives for such a definition, as stated above, are: to define more narrowly, not more broadly; to bring an order to the criteria not necessarily to accommodate all viewpoints; to refine the definition to facilitate assessment, to retain the validity of work that has been done previously, such as the development of curricula; to separate a definition from those activities that might promote acquisition of computational thinking skills; and to separate a definition from those artifacts and activities that evidence the use of computational thinking skills. Justification for inclusion or exclusion is based on consistency of usage and consistency of meaning across the literature. The resulting definition reflects much of the consensus found in the literature while removing the less well-defined terms.

Table 1 summarizes the justification for each prospective term’s inclusion in or exclusion from a proposed definition of computational thinking.

<table>
<thead>
<tr>
<th>Term</th>
<th>Status</th>
<th>Justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>A thought process</td>
<td>Include</td>
<td>Consensus found in the literature</td>
</tr>
<tr>
<td>Abstraction</td>
<td>Include</td>
<td>Consensus found in the literature</td>
</tr>
<tr>
<td>Decomposition</td>
<td>Include</td>
<td>Consensus found in the literature</td>
</tr>
<tr>
<td>Logical thinking</td>
<td>Exclude</td>
<td>Broad term, not well defined</td>
</tr>
<tr>
<td>Algorithmic thinking</td>
<td>Include</td>
<td>Well-defined across multiple disciplines</td>
</tr>
<tr>
<td>Problem solving</td>
<td>Exclude</td>
<td>Broad term, evidences the use of skills, develops acquisition of skills</td>
</tr>
<tr>
<td>Evaluation</td>
<td>Include</td>
<td>Well-defined across multiple disciplines</td>
</tr>
<tr>
<td>Generalization</td>
<td>Include</td>
<td>Well-defined concept, although the term may not be familiar</td>
</tr>
<tr>
<td>Systems design</td>
<td>Exclude</td>
<td>Evidences the use of skills</td>
</tr>
<tr>
<td>Automation</td>
<td>Exclude</td>
<td>Evidences the use of skills</td>
</tr>
<tr>
<td>Computer science content</td>
<td>Exclude</td>
<td>Evidences the use of skills</td>
</tr>
<tr>
<td>Modeling, simulation, and</td>
<td>Exclude</td>
<td>Evidences the use of skills in their creation; manipulation develops acquisition of skills</td>
</tr>
</tbody>
</table>

Table 1. Computational Thinking Definition Terminology

As supported by the preceding arguments, computational thinking is an activity, often product oriented, associated with, but not limited to, problem solving. It is a cognitive or thought process that reflects:

- the ability to think in abstractions,
- the ability to think in terms of decomposition,
- the ability to think algorithmically,
- the ability to think in terms of evaluations, and
- the ability to think in generalizations.

This proposed definition attempts to incorporate only those terms for which there is a consensus in the literature or those terms that are well defined across disciplines. The intent is to focus on the thinking aspect of the original phrase.

In other words, computational thinking is a focused approach to problem solving, incorporating thought processes that utilize abstraction, decomposition, algorithmic design, evaluation, and generalizations.

6. CONCLUSION

There is a genuine need for a robust and agreed definition of computational thinking. The definition can facilitate the development of computer science curriculums in line with Wing’s original vision to encourage computational thinking for all. The definition may also ensure that the K-12 curriculums will not become just a collection of interesting resources presented at teachers’ discretion. The definition may ensure that appropriate assessment tools can be developed which measure computational thinking skills. The description narrows the definition by
excluding some proposed terms. It separates the practice of skills and the results or evidence of the application of skills from the activity of thinking. However, it does not invalidate the curriculum designs, especially as they often focus on the doing or evidence of doing computational thinking. It leaves open the possibilities to develop assessment tools to measure the ability to think computationally. Of course, the discussions of a definition for computational thinking are not yet concluded. It may well be that the definition changes as understanding of computational thinking develops over the coming years. This is especially true as younger learners are exposed to the concepts in fulfillment of Wing’s original vision of computational thinking for all. This review of the literature simply attempts to inform these discussions.

7. REFERENCES
Appendix 9. Research into ICT to support computing in education

TDA TiLT RESEARCH GRANTS 2010-11
The Technology in Learning and Teaching Team

Research into ICT to support computing in education

10 projects – the online surveys

John Woollard and Cynthia Selby
Southampton Education School
University of Southampton

Abstract
This TDA funded report describes the challenges and affordances of introducing computing into the initial teacher training of ICT teachers. It reflects upon the experiences of 10 recipients of the Technology in Teaching and Learning (TiLT) research grants. The data is collected through online surveys at the start and near the completion of the activities. Five projects are identified that reflect 5 different models of approach to integrating computing activities into initial teacher training in ICT – these are reported separately. They enable institutions to consider the model of approach most appropriate for their context.

Key themes emerging from the survey analysis are:

- Programming is perceived to be a key element of computing provision.
- Practical computing activities can be embedded in classroom experiences.
- Without a defined Computing Curriculum, training providers are not confident of exactly what subject knowledge should be presented.
- A narrow focus on individual skills, such as programming or manipulation of databases, may result in the neglect of broader computing concepts, such as abstraction, decomposition, or generalisation.
- For those trainees with applicable degree or industry knowledge, there should be differentiation in delivery and material.
- Trainees reveal concerns about their lack of specific computing subject knowledge when beginning the initial teacher training and question their ability to master programming to a Key Stage 3 level.

Introduction
This report reflects upon the experiences of recipients of the TDA Technology in Teaching and Learning (TiLT) funding 2011. It is research orientated and designed to enable ITT providers to try a new strategy with a new aspect of the curriculum and evaluate their experiences. The project research processes include: contacting the 10 successful projects and collecting initial data using an online survey; providing an opportunity for trainee teachers and others receiving training to comment upon their experiences; collecting data on completion of the project from the administrators and trainers and identifying up to 5 projects to act as case studies. A measure of the success of this funding is the impact upon a provider’s capacity to repeat this process in the future and the impact of the project on colleagues in the ICT teacher training community as a whole. An important aspect of the work will be to bring a clearer understanding regarding the nature of computing in ICT and within initial teacher training of ICT trainees from the perspective of experienced ICT teacher trainers.
Appendix 9

The institutions awarded the research grants were: Anglia Ruskin University; Bromley Schools’ Collegiate; Keele University; King’s College London; Middlesex University; Newman University College; University of East London; University of Huddersfield; University of Reading; and University of Wolverhampton.

Literature Review

The last year has seen activities revolving around the importance of ICT and Computing in school, especially in secondary education. There are questions concerning the need to stem a decline in the study of computing, questions concerning the place of computing in a new national curriculum, and attempts to identify a model for a computing curriculum for secondary schools.

On 05 August 2010, the Royal Society launched a study of the issues associated with the decline of uptake in ICT and Computer Science in schools (The Royal Society 2011). They identified that this decline in uptake may have a negative impact on the UK’s future economy. The study is to be completed early 2012.

On 20 January 2011, the Secretary of State for Education called for a review of England’s National Curriculum (Department for Education 2011a). The resulting review recommendations will result in replacing the current National Curriculum with a complete new one for all subjects. The new curriculum specifications will be introduced for all subjects beginning in 2014 (Department for Education 2011b).

The BCS Computing Fact Sheet describes why computing education is important for the UK and was prepared by BCS Academy of the Chartered Institute for IT. “The UK needs school-leavers to be familiar with rigorous computing principles if the economy is to be competitive over the long term” (British Computer Society 2011, p. 1). This has significant implications for initial teacher training of ICT teachers.

As part of the review process, there was a call for evidence from 20 January to 14 April 2011. Any interested parties were invited to provide submissions (Department for Education 2011c). The BCS and Computing at School made a joint response to the call for evidence, establishing the need for including computer science as a subject in the new national curriculum (British Computer Society and Computing at School 2011).

The subject knowledge and concepts that make up a rigorous model of computing for Key Stage 3 and Key Stage 4 has been developed by the Computing at School organisation (Computing at School 2011). The model curriculum is available at http://www.computingatschool.org.uk/data/uploads/ComputingCurric.pdf

In the national curriculum review’s report, published at the end of 2011, the expert panel recommends a proper consideration for the calls for “… more widespread teaching of computer science in secondary schools” (Department for Education 2011d, p. 24).

The calls for investigation, the development of a computing curriculum, and the recommendations in the National Curriculum review highlight the importance of investigating how initial teacher training providers might incorporate computing provision into their training schemes.

Methodology

Three online surveys were developed and administered to determine the implementation and effectiveness of the computing provision, as proposed in the original submissions from the participating teacher training institutions. The first survey [https://www.isurvey.soton.ac.uk/2075] was completed by the training coordinators from each institution, prior to beginning the delivery of the computing provision. The results of the first survey serve to establish a baseline of the current ICT provision and any existing computing coverage. These results are also used to select institutions to
serve as the basis for the associated case studies. The second survey,
[https://www.isurvey.soton.ac.uk/2442], completed by the training coordinators and the training
providers, provides an opportunity for trainers and administrators to reflect upon the processes of
integrating elements of computing into the ICT initial teacher training. In some instances, the
coordinators and providers were the same person, so had to complete the survey from two different
perspectives. The results of this survey serve to evaluate the provision and to reflect on the lessons
learned from delivering the computing provision. The third survey
[https://www.isurvey.soton.ac.uk/2600] provides an opportunity for trainees to reflect upon and
evaluate the learning and the effectiveness of the computing provision, from a student perspective.

Analysis of the survey data was performed using simple statistics for quantitative data and thematic
coding for qualitative data. All text responses were read and assigned to one or more themes.
Simple statistics could then be performed identifying the most popular themes. For example, one
institution anticipates providing the provision for NQTs in their community and another institution plans
to introduce CPD sessions for in-service teachers. Both these map to an in-service theme. The
identified themes form the basis for the results presented below.

Results
The results from the participating institutions’ coordinators and trainers reflect both positive and
negative opinions relating to the efficacy of implementation and the appropriateness of the
implementation. They describe both the successes and challenges of delivering the computing
provision. Although further results are included within the discussion section, the highlights of the
analysis are:

- Some providers have chosen to use the latest introductory programming environments
  including Kodu (http://research.microsoft.com/en-us/projects/kodu/), Scratch
  (http://scratch.mit.edu/), and BYOB (http://byob.berkeley.edu/).
- For those trainees with applicable degree or industry knowledge, there should be
differentiation in delivery and material. These trainees may be able to move quickly from the
KS3 and KS4 requirements to those of Post-16
- Some providers are at a loss as to which topics should be covered. Without a Computing
  Curriculum, it is unclear exactly what subject knowledge should be taught to those planning to
  enter classrooms in schools and colleges.
- A narrow focus on individual skills, such as programming or manipulation of databases, may
  result in the neglect of broader computing concepts, such as abstraction, decomposition, or
generalisation.
- Participating providers report that they will endeavour to adapt the current ICT provision to
  include the new computing activities, provided there is access to facilities, finance, schedule,
  and expert teachers for delivery.
- Two of the participating institutions question the requirement to include a computing
  component in the PGCE ICT.
- Making computer science knowledge a prerequisite for the PGCE ICT would result in trainees
  with better subject knowledge. However, those candidates with Computer science or industry
  subject knowledge are always looked upon favourably.

Although the analysis of the trainee dataset is not yet complete, themes already emerging include the
following:

- Trainees reveal concerns about their lack of specific computing subject knowledge when
  beginning the PGCE ICT.
- Many trainees associate computing with programming, find it challenging, and do not perceive
  that they can master it to a Key Stage 3 level.
- Many more trainees make comments indicating enthusiasm and higher levels of confidence in
  their own abilities to master programming skills.
Appendix 9

Discussion
Four categories are suggested to set the context for a discussion of the results of the data analysis. The significant results, including those mentioned above, fit one of these four categories. The categories are: programming skills, teaching programming, teaching broader computing concepts, and specific initial teacher training considerations.

Programming skills
All of the responses indicate that the trainees achieved programming skills to the level required in the Key Stage 3 classroom. A level of programming skills commensurate with Key Stage 3 should be viewed as a minimum requirement for all trainees. The results indicate that this is an achievable target.

Four respondents report programming skills achievement to Key Stage 4 level and 1 reports achievement to a Post-16 level. Success at Key Stage 4 and Post-16 levels may be dependent upon the existing subject knowledge of the trainees, rather than on providers being able to deliver training to those levels.

Teaching programming
Eight of the responding institutions indicate the use of practical and hands-on exercises in teaching programming. The majority (5) make use of prepared resources to facilitate the training. These include paper-based and digital resources, demonstrations, and peer collaborations.

Teaching techniques, reported by 9 respondents encompass learning the building blocks of a language (2), constructing simple code fragments (1), and progression to more complex problems(2). Allowing trainees to collaborate in the exploration of the programming environment was mentioned, by one institution, as an effective learning technique.

The choice of visual programming environments by 5 institutions, such as Kodu (2), Scratch (4), GameMaker (1) (http://www.yoyogames.com/make), and BYOB (1) provides high levels of engagement for both the trainees and their intended pupils, especially at KS3. The choice of more traditional languages (N=4), such as Python (1), Logo (1), Java (2) and BASIC (2), provides exposure to the level of understanding required at Key Stage 4 and Post-16.

Responses indicating reticence suggest that the more technically challenging a topic, task, or concept became, the more the trainees expressed doubt in their own abilities (2). The trainees also showed some reticence when asked to move onto high-level languages (2). This is probably understandable given their lack of prior subject knowledge and short timeframe for the course.

Teaching broader computing concepts
Two of the responding institutions question the value of implementing a computing provision within the initial teacher education of ICT teachers. The broader question raised is whether subject knowledge, ICT or computing, should be included when teaching should be the focus. There is perhaps a perception that the teacher training programme should focus on the craft skills of teaching, classroom confidence and competence, and knowledge of education processes and important aspects. This reticence is offset by the enthusiasm shown by the remaining institutions to engage with the challenges presented by the study. Three respondents indicated that trainees developed higher levels of confidence in their own abilities.

Five responses indicated that trainees mastered an understanding of modelling and decomposition to Key Stage 3 level. Four of those further indicated understanding abstraction and generalisation to Key Stage 3 level. While an understanding of modelling to Key Stage 4 was high (4), decomposition and abstraction was lower (2), and generalisation was lower still (1). The lower levels at Key Stage 4 levels may be a reflection of limited subject knowledge combined with restricted time for provision.

Of 8 responses, 7 indicated that trainees mastered writing code and algorithms at least to Key Stage 3 level. All 8 reported that trainees mastered architecture to the same level. Four of 8 reported
mastering algorithms and 5 reported mastering writing code to Key Stage 4 level. This is a positive result indicating good skill acquisition applicable in the secondary classroom.

Five report a reliance on the teaching of programming to embed the broader computing concepts of abstraction, modelling, decomposition, and generalisation. Some concepts are further facilitated by the use of a language survey (1) which might give opportunities for generalisation. One response indicates trainees developed classroom teaching resources for these concepts. Only a single response indicated that direct discussions of these concepts took place during the provision. While the focus on gaining skills is admirable, it may lead to the neglect of these broader concepts.

**Initial teacher training considerations**

From the responses, it is evident that some providers (3) have trainees with good prior computer science knowledge or industry experience. Respondents (2) indicate that, for these trainees, there should be differentiation in delivery and resource material. Their prior knowledge may allow them to quickly from the Key Stage 3 and Key Stage 4 requirements to those of Post-16, thereby ensuring highly qualified trainees with both good subject knowledge and teaching skills.

Institutions experienced three main areas of challenge in delivering the computing provision, including the need for expert planning and delivery (3), need for specialised hardware/software facilities (1), and the need for specialised learning resources (3). The lack of a Computing Curriculum was identified by one respondent; this may be overcome with the Computing: a curriculum for schools document (Computing at School 2011).

Eight of the responding providers report that they will endeavour to adapt the current ICT provision to include the new computing activities. A dissenting respondent acknowledges the need for the provision, but questions how it should be delivered. Enhanced computing provision is dependent on access to resources (2), finance (4), scheduling (3), and expert teachers (3) to deliver the provision.

Prior computing subject knowledge is identified as being particularly valuable by 4 of the respondents. Two institutions report that ICT trainees come from diverse backgrounds and degree subjects. However, those candidates with computer science or industry subject knowledge are always looked upon favourably.

Identified changes to the delivered provision are all in terms relative to the individual providers. The most valuable responses include the need to differentiate the provision based on the needs of the trainees and their subject knowledge (2), the need to provide out-of-classroom exercises for independent learning (2), and a focus on concepts not just skills (1).

**Conclusion**

After completing the planned computing provision, the participating coordinators, trainers, and trainees responded to the online survey questions. Analysis of the responses has helped reveal the achievements of and challenges for all those involved.

The achievements of the trainees at mastering computing concepts and programming skills to levels immediately usable in the secondary classroom, in such a short time, are praiseworthy. This positive result should not be undervalued. The challenge for the trainees and for those delivering the computing provision is overcoming the lack of confidence that some trainees bring to the course.

The challenges for the initial teacher training coordinators and the trainers include finding the time and resources to continue the provision, modifying the provision to incorporate higher levels of focus on computing concepts, not just programming skills, and providing differentiation for those trainees with prior subject knowledge or industry experience.

The challenge for the coordinators, trainers, and the broader community interested in promoting computing is the lack of a specified curriculum.
Appendix 9

Once these challenges are overcome and with the continued achievements and enthusiasm of the trainees, providers, and trainers, more ICT teachers will be entering secondary classrooms with the ability to engage learners with computing topics and concepts.

References


Appendix 10. Postgraduate and Early Career Research Showcase 2013 – Poster

Can programming be used to enhance computational thinking?

Context
- Confederation of British Industry (2013)
- Royal Society (Put down or restart? 2012)
- Rt Honorable Michael Gove
  Secretary of State for Education (BETT 2013)
- The Wolf Report (Vocational Education 2011)

Methodology
- Grounded theory dictate an iterative approach of data collection and analysis until theoretical saturation.
- Participants = Those with knowledge of or interest in computational thinking, including teachers, academics, industry professionals, etc.
- Selection = Purposeful selection based on perceived level of expertise in any area under investigation. First two instruments act as filters for further selection.
- Instrument = Online questionnaire
- Instrument = Community of practice online discussions
- Instrument = Interviews for collecting qualitative data
- Analysis = Qualitative codification and categorization of responses until theoretical saturation
- Model = Generation of a model to demonstrate relationships between problem solving, computational thinking, and programming

Preliminary Findings
- Problem solving skills are reported to involve problem understanding, interpretation of everyday activities, persistence, and practice.
- Computational thinking skills are reported to include decompositions, modeling, and algorithm design. Some responses also include the skills of planning, justifying, and evaluating.
- Teaching programming skills is concerned with logical thinking, the concept of programming as a tool, the concept of collaboration as a pedagogic strategy, and the concept of programming ability as innate.
- Promoting computational thinking can be achieved by activities involving decomposition, working backwards from a known solution, and decomposition, breaking problems down into smaller parts.
- Pedagogic occurs specifically include references to the theory content of social, planning solutions as part of learning, and transition between programming tool complexity.

"Computational thinking is a fundamental skill for everyone, not just for computer scientists. To read, write, and arithmetic; we should add computational thinking to every child's analytical ability" (J. Wing, March 2006)

Preliminary Model

Research Questions
- What is the connection between problem solving, programming, and computational thinking?
- Is there a taxonomy of computational thinking skills and activities?
- What is the set of problem solving and programming skills that underpin computational thinking?
- Can computational thinking be taught without teaching programming?
- What specific programming activities contribute to computational thinking skills?
- Are there other contributors to computational thinking skills, regardless of discipline?
Appendix 11. Computing at School 2012 – Poster

Can programming be used to enhance computational thinking?

**Context**
- Confederation of British Industry (2011)
- Royal Society (What is it worth? 2011)
- Rt Honorable Michael Gove
- Secretary of State for Education (BETT 2012)
- The Wolf Report (Vocational Education 2011)

**Methodology**
- Grounded theory dictates an iterative approach of data collection and analysis until theoretical saturation
- Participants – Anyone with knowledge of or interest in computational thinking, including teachers, academics, industry professionals, etc.
- Selection – Purposive selection based on perceived level of expertise in any area under investigation
- Instrument – Online questionnaire
- Instrument – Community of practice on-line discussions
- Instrument – Interviews for collecting qualitative data
- Analysis – Qualitative codification and categorisation of responses until theoretical saturation
- Model – Generation of a model to demonstrate relationships between problem solving, computational thinking, and programming

**Preliminary Model**

**Conceptual Framework**

**Research Questions**
- What is the connection between problem solving, programming, and computational thinking?
- Is there a hierarchy of computational thinking skills and activities?
- What is the set of problem solving and programming skills that underpin computational thinking?
- Can computational thinking be taught without teaching programming?
- What specific programming activities contribute to computational thinking skills?
- Are there other contributors to computational thinking skills, regardless of discipline?

**Findings**
- Early exposure and continued opportunities to engage with programming, with increasing complexity over time
- Acknowledge requirements gathering, planning, and evaluation
- Design and creation of systems incorporating both software and hardware subsystems
- Decomposition, abstraction, algorithm design, and generalisation are viewed as problem solving tactics, not unique to programming, but applicable in all contexts
- An order of complexity of skills moves on a continuum from decomposition, to abstraction, to algorithm design, to generalisation, and to modelling
- A topic not receiving enough attention is understanding the original problem and its constraints
- Computational thinking is not limited to computer science. The skills are evidenced everyday by evaluating results and refining solutions.

**Marketing is trying to service what people *want* — and to raise the perceived value of those wants. Education is trying to provide people with what they *need*, and to find ways to do so even when people don’t want what they need. (Alan Kay, 22-08-13)**

Cynthia C. Sally
Bay House School and Sixth Form
Gosport, Hampshire
C.Sally@southampton.ac.uk
Canby@bayhouse.hants.sch.uk
http://www.personal.soton.ac.uk/ecs306
Appendix 12. Computing at School 2011 – Poster

A research proposal: How do post-16 teachers employ the pedagogy of programming to enforce computational thinking skills?

Programming
- Allows people to instruct devices how to behave
- Instructions include assignments and sequencing
- Control constructs include conditionals and repetition
- Is accessible to all ages and capabilities of learners
- Requires some persistence to master the logic
- Is perceived to be hard to learn
- Is fun and challenging

Computational Thinking
- Is composed of high level thought processes
- Is done by people, not computing devices
- Is exemplified in problem-solving
- Allows people to be creators, not consumers of technology
- Draws on disciplines of mathematics, engineering, and technology

Methodology
- Participants - Teachers of ICT or Computing selected for the potential quality of their contributions
- Instruments - Pre-interview questionnaire to establish quality of potential participant’s contributions
- Instruments - Interviews for collecting qualitative data
- Analysis - Inspection of interview transcripts to identify concepts and relationships
- Grounded theory dictates an iterative approach of data collection and analysis until saturation

Conceptual Framework
Teaching of Thinking
Computational Thinking
Computational Thinking Content
Pedagogy of Programming
Generalising
Abstraction
Algorithm
Programming
Computation
Machine
Logic
Thinking
Human
Modeling
Decomposition
Science

Next Steps
- Identify teachers to participate in case studies
- Interview teachers about how they teach programming
- Interview teachers about how they reinforce thinking

Programming Pedagogy
- Use of pseudocode to express algorithms
- Analysis of code or pseudocode for understanding
- Mastering behaviours of individual language constructs
- Constructing small units of reusuable code
- Combining small units of code to solve larger problems
- Amendments to larger units of code to change behaviours
- Use of visualisations to simulate structures or execution
- Use of strategies to identify errors in logic or code

Thinking Strategies
- Abstraction – Defining what properties of a set of objects are important and what properties can be ignored
- Decomposition – Breaking big problems down into smaller problems which are easier to solve
- Modelling – Simulating the real world by using computational devices
- Generalising – Recognising common features in objects which may be shared in order to control complexity

It has often been said that a person does not really understand something until after teaching it to someone else. Actually a person does not really understand something until after teaching it to a computer, i.e., expressing it as an algorithm. (Knuth, 1990)
References


References


CUKIERMAN, D. & THOMPSON, D. M. 2009. The academic enhancement program: encouraging students to learn about learning as part of their computing science courses. *Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education*. Paris, France: ACM.


References


References

and technology in computer science education. Darmstadt, Germany: ACM.


References


References


