The Role of Computational Thinking in Introductory Computer Science

Submitted in fulfilment
of the requirements of the degree of

Master of Science

of Rhodes University

Lindsey Ann Gouws

Grahamstown, South Africa
December 2013

Abstract

Computational thinking (CT) is gaining recognition as an important skill for students, both in computer science and other disciplines. Although there has been much focus on this field in recent years, it is rarely taught as a formal course, and there is little consensus on what exactly CT entails and how to teach and evaluate it. This research addresses the lack of resources for integrating CT into the introductory computer science curriculum. The question that we aim to answer is whether CT can be evaluated in a meaningful way.

A CT framework that outlines the skills and techniques comprising CT and describes the nature of student engagement was developed; this is used as the basis for this research. An assessment (CT test) was then created to gauge the ability of incoming students, and a CT-specific computer game was developed based on the analysis of an existing game. A set of problem solving strategies and practice activities were then recommended based on criteria defined in the framework.

The results revealed that the CT abilities of first year university students are relatively poor, but that the students' scores for the CT test could be used as a predictor for their future success in computer science courses. The framework developed for this research proved successful when applied to the test, computer game evaluation, and classification of strategies and activities. Through this research, we established that CT is a skill that first year computer science students are lacking, and that using CT exercises alongside traditional programming instruction can improve students' learning experiences.

ACM Computing Classification System Classification

Thesis classification under the ACM Computing Classification System (1998 version, valid through 2013) [1].

K.3.2 [Computers and Education]: Computer and Information Science Education
K.8.0 [Personal Computing]: General — *Games*

General-Terms: Human Factors, Measurement, Design

Acknowledgements

The completion of this thesis would not have been possible without the support and assistance of a number of people.

This work was undertaken in the Distributed Multimedia CoE at Rhodes University, with financial support from Telkom SA, Tellabs, Genband, Easttel, Bright Ideas 39, THRIP, and NRF SA (UID 75107). The authors acknowledge that opinions, findings and conclusions or recommendations expressed here are those of the author(s) and that none of the above mentioned sponsors accept liability whatsoever in this regard. The financial assistance of the Ernst and Ethel Eriksen Trust is acknowledged.

A huge thank you and acknowledgement goes to my supervisor, Dr. Karen Bradshaw, for all of her direction, encouragement, and hard work. She has been an excellent supervisor throughout my postgraduate studies, and her contribution to this thesis has been invaluable.

I would also like to thank and acknowledge my co-supervisor, Prof. Peter Wentworth, for the limitless insight, enthusiasm, and guidance that has shaped this research, and for the support both on an academic and a personal level.

Finally, thank you to my family and friends for your constant love and emotional support.

Rawr

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The field of computer science (CS) education is ever-changing, with pressure to keep up with the latest technology, educational philosophies, and societal needs. Throughout its existence, there have been a number of challenges in the CS field, including misconceptions about its nature, homogeneity of demographic groups, and retention of students. CS occupies a distinctive place in the education landscape, with roles both as a service course and a discrete science in its own right [24, 26]. Reliance on technology is pervasive in most aspects of life, increasing the relevance of CS for the broader community. As well as the benefits for personal use, computers change the face of research and enable researchers to ask different kinds of questions [17]. Traditionally, much of the contribution towards research and education has been in the form of technological resources; however, there is an increasing belief that CS has more to offer in different ways of thinking and approaching problem solving.

The mental skills arising from the practice of CS are given the name computational thinking (CT). Jeannette Wing, a key voice in the field, describes CT as the *mental tools* that allow us to make the best use of our *metal tools* [78]. Therefore, CT is not about thinking like a computer, but rather thinking about problems from a computational perspective, emphasising the ideas of what is computable and how computation works. This approach includes exploring all aspects of the problem, considering the complexity of the problem, and finding an optimal solution that can be achieved with the available resources. CT is not an alternative to learning to program; it is a way of reinforcing concepts and supplementing programming education. Furthermore, it may help to build abstract reasoning and problem-solving skills [76].

CT is strongly advocated as a skill that all children need to develop, as well as an important component of preparation for a programming course [50, 78]. However, authors caution against interpreting the world computationally in contexts where this may prove unsuitable [12], and advise against accepting the notion that CS = CT [28]. When used appropriately, CT has influenced fields like biology, engineering, and economics [79].

1.1 Problem Statement and Research Goals

The importance of CT has been well established, but there is still a lack of understanding around this subject. There is very little research into this field and the effects it may have within and beyond the CS classroom, particularly in South Africa. Numerous attempts have been made to assess CT and introduce it into the curriculum [15, 57, 59], but these have not been widely adopted or standardised, and tend to vary in nature from case to case.

The problem that we attempt to address in this thesis is the definition, assessment, and integration of CT in introductory CS at a university level. The overarching question that we aim to answer is whether CT can be evaluated in a meaningful way. To answer this, a number of sub-objectives for this research were identified, as given below:

1. The development of a framework for understanding, evaluating, and assessing CT in students and activities.
2. Gaining an understanding of the existing CT ability of new university students, and the extent to which this is improved through an introductory CS course.
3. Provision of an artefact in the form of a computer game that can be used to teach, learn, and improve CT.
4. Provision of reference materials for planning, structuring, assessing, and facilitating CT endeavours.

1.2 Thesis Organisation

This thesis is organised into the following chapters:

Chapter 2 discusses relevant work in the fields of CS education, theories of learning, and discipline-specific forms of thinking.

Chapter 3 introduces and provides a detailed description of the proposed CT framework, as well as some ideas of how to apply it, as further explored in the subsequent chapters.

Chapter 4 discusses the student assessment used to measure the abilities of students in an introductory CS course.

Chapter 5 describes the computer game that was designed and developed as a resource for practising CT.

Chapter 6 discusses several problem solving strategies and activities that may be used to augment education with CT structures.

Chapter 7 provides an overall discussion of the findings of this thesis.

Chapter 8 summarises this thesis and presents the final conclusions from the research.

Appendix A provides the test questions used in the CT test described in Chapter 4.

Appendix B provides the levels of the computer game Rubot as described in Chapter 5.

Appendix C provides the questionnaire used to obtain user feedback for Rubot.

Appendix D provides the set of puzzles classified by their CT attributes in Chapter 6.

Chapter 2

Related Work

CS education is a contentious issue, with differing philosophies on the effective means to teach and learn computing, and even what content should constitute a CS curriculum. This chapter introduces CT within the context of CS education, beginning with a look at the challenges in CS education and a broad definition of CT.

The benefits and criticisms of CT and its place in the CS curriculum are outlined in this chapter. Different learning theories and taxonomies are presented to give an understanding of the purpose and methods of educational endeavours. Finally, different types of discipline-specific thinking are discussed and their relationships with CT explored.

2.1 Computational Thinking and Computer Science Education

As a field that has been developing since the 1950s [26], CS has been gaining an increasingly large body of materials and principles. The ACM CS curriculum for undergraduates [45] currently contains as many as 18 knowledge areas, or topical areas of study in CS. Computers directly augment human thought [21], and the development of computers has been likened to the development of the printing press in the way that it facilitates a particular set of skills that were previously not feasible for all people to practise [77].

Over the course of its existence, CS education has faced many questions: Is CS a distinct field apart from mathematics? Does it qualify as a science discipline? What is the

appropriate relationship between CS and the information and communications technology (ICT) industry? What is the right balance between theory and practical application, and how integral should the computer itself be in the instruction process? More recently, CT and questions about how we think about computing are altering understanding about CS and influencing discussions around the future of CS education.

2.1.1 Challenges in Computer Science

The enthusiasm for CS has waned since the excitement fifty years ago at being able to make a computer perform any task [43]. There are several challenges in the current CS education landscape, some of which have been persistent over a number of years. These challenges affect the way in which CS is conceptualised and taught, and are a necessary consideration in CS education. Four of the most prevalent issues are discussed.

Misconceptions about the discipline: There are numerous misconceptions about the nature and purpose of CS courses. CS is often confused with computational literacy and Information Technology (IT) fluency [24], and therefore cast as a service course with the primary purpose of serving other disciplines. Although ICT is recognised as a desired skill, this broad definition may include such skills as working with spreadsheets and website development, and ICT studies can become about trivial tasks such as word processing and web browsing [11]. Moving past this initial perspective, there is a pervasive view that CS is all about programming [28]. This notion calls into question the validity of CS as an academic discipline and suggests that it is primarily a technical or rudimentary pursuit. Research indicates that the perception of CS involving sitting in front of a computer all day is a deterrent for potential CS majors [19]. Further misconceptions by prospective students include the belief that the subject is aimed at intelligent and gifted individuals only [43], leading students to prematurely preclude themselves from a future in the field.

For many years, there has been concern about improving the public understanding of CS [21]. The authenticity of the field of computing as a genuine science has been interrogated [26], and Denning provides clear arguments for its inclusion as a science. These misconceptions may leave students ill-equipped to deal with the realities of their studies in the discipline, as well as deterring students who may have a real aptitude for the subject. The actual concepts in CS include logical reasoning, algorithmic thinking, design, and structured problem solving [21], extending far

beyond the scope of IT fluency and programming. Curzon et al. [24] believe that CT is a basis for forming a better interpretation of computing that is more precise, deeper, and wider than previously understood.

Attraction and retention of students: The number of students who select and complete a degree in CS remains a ubiquitous concern worldwide. Although there is a demand from the ICT industry for graduates, there is a decline in the number of CS enrolments in many countries. This decline in graduates has a significant impact on the ICT industry; local studies indicate that students have little faith in the computing job market despite the availability of jobs in the ICT sector [11, 43]. This may be a simple misunderstanding, but further research indicates additional deterrents that affect students' choices in whether to major in a computing subject.

A 2010 study at Carnegie Mellon University revealed that over half of the students never go any further than an introductory CS course, which is identified as an implication when structuring courses, as this first year exposure will define their understanding of the CS field [16]. The trend of low CS enrolments is prevalent even in countries with well-established ICT infrastructure and resources — the number of incoming CS students dropped by 60% in the United States and by 43% in the United Kingdom between 2000 and 2004 [19, 43]. Some studies attribute the drop in CS enrolments to the dot-com crash [43].

A number of mitigating factors that affect students' decisions about CS have been discovered. Carter [19] reports on a study in the United States where 836 high school students were surveyed to discover their perceptions and influences about CS. This study reflected the most significant negative influences as already having chosen a different major, having to sit at a computer for prolonged periods, and the lack of people-oriented activities. These influences correlate with the misconceptions of CS described above. In a survey of high schools in Grahamstown[1], Jacobs found that the factors influencing learners in their decisions included school status, the digital divide, demographics (including racial grouping and gender), and their perceptions about the job market [43]. Conversely, Carter found that the most positive influences towards CS are usefulness in another field, enjoyment of computer gaming, and students' previous experience with computers [19].

Despite the problems given above, it appears that there are techniques that can be successful in retaining a higher number of CS students. A proper introduction to CS can sway first year students to continue in CS, potentially even changing their

[1]Grahamstown, Eastern Cape, South Africa

majors [19]. In a different vein, pair programming has been used successfully in CS classes, resulting in higher retention and future enrolment rates, as well as greater student success and happiness in the course [21].

Representation of demographic groups: Historically, CS has had an unbalanced demographic representation, which is particularly manifested in a lack of female or racially diverse students. As well as creating problems in a classroom environment, this phenomenon indicates that there is potentially a large audience of students who are not being reached. Much research has been done investigating this trend, focusing especially on gender dynamics but also considering ethnic backgrounds. To quantify these concerns, the Taulbee Survey has been created as a benchmark to measure diversity among students of computing [52].

There are a number of reasons why the demographic representation in CS ought to be addressed. The demographics of the student population impact the future of the field [52], and increased diversity is important for the work force [70]. It is unfortunate that under-represented groups of students may perceive more instances of racism and sexism [7], deterring them from future studies.

The field should aim to be inclusive to better represent society [70], particularly as CS is relevant to many different people owing to its applicability in various spheres of life [21]. There appears to be an interest in CS from under-represented groups; diverse students may be attracted to CS majors despite a lack of prior experience [20]. In providing more opportunities to minority groups, the field of CS stands to benefit from inclusion of these students. The homogeneity of computing deprives the field of the perspectives, insights, and needs that these people may contribute [20, 70].

Countless research studies have been done on the role of women in computing. Despite the conception of programming as a role primarily occupied by women, the number of women in CS is alarmingly low [38]. A survey carried out in the United States in 2008 indicates that women constitute only 38% of the science and engineering field, while data from a 2010-2011 survey indicates that 87.3% of CS bachelor's degrees were awarded to males [52]. Studies have found a positive correlation between the male gender and an intention to major in CS [7]. Interestingly, female students who discontinue CS may have higher grades than the male students who remain in the field.

A well-supported theory for this imbalance is the lack of previous exposure of women to computing; male students are more likely to have programming experience [7,

19]. Furthermore, female students are less likely to have completed a high school computing course [20]. It appears that the motivation for pursuing CS may differ depending on gender. Different genders and ethnic groups have exhibited different areas of interest within CS; for example, male students prefer applications of rocket payloads while female students prefer photo mosaics [20].

There is a pervasive stereotype that CS is primarily the domain of the white male student. The 2010-2011 Taulbee survey in the United States and Canada indicates that 65.8% of CS bachelor's degrees are awarded to white students. A lack of ethnic diversity is a problem within CS education as well as in the science and engineering workforce [20, 52]. In the United States, most minorities have a smaller representation in the science and engineering workforce than their numbers in the population [52], and minorities are less likely to have completed a high school CS course [20]. In South Africa, black high school students display a greater inclination to study CS than their white counterparts [43].

Factors that affect student persistence in the field include gender, ethnicity, and prior experience [7]. Naturally, under-represented students are less inclined to persist in a CS major when they perceive that they are receiving different treatment [7]. Perhaps more worryingly, when students are exposed to group stereotypes they can internalise these, which further perpetuates the imbalance between students.

Lack of resources: Despite the boom of the ICT industry, lack of access to resources remains a significant barrier within CS. In South Africa the apartheid system has a legacy effect on educational structures [34], manifested particularly in poorly distributed resources. Many schools are not equipped to provide CS courses, and at a university level there is a dilemma since students with different levels of experience must be taught in a common classroom. This issue is not solely a third-world problem; even in developed countries students who have home exposure to computers tend to have an advantage and be more likely to major in the field [35]. Although there has been much research on this issue, further dialogue is necessary to identify, address, and remedy these impediments to CS education.

The first major concern is a lack of technological resources, both at home and at schools. The access to computer facilities is limited in South African schools, and in provinces like the Eastern Cape students have the lowest percentage of computer access despite having the highest number of schools [43]. As a result many students complete school without any exposure to computers [34]. This has an adverse effect on students' decisions to pursue CS studies and careers; students are not

able to select IT as a Matriculation (Matric) subject in high school [34], and unequal accessibility of computers results in a difference in computer attitude among students [14].

The school resources and teachers available to students are an additional cause for concern. In South Africa there is a shortage of skilled science, mathematics, and technology educators [34, 43]. One result of this shortage is that students are not equipped for their Matric exams and thus do not attain university entrance [34]. Additionally, a lack of funds and training makes it infeasible to adopt a new curriculum, which may have stimulated students' interest in IT [43]. For rural schools and disadvantaged schools this is especially problematic as they are unable to garner fees to invest in resources like computer laboratories [34]. The lack of skilled educators is a problem at all education levels — primary, secondary, and tertiary [43]. The problems with CS in high school are present in developed as well as developing countries. In the United States, there are few CS standards at the secondary level, affecting more than two thirds of the country [38]. Furthermore, high school students lack experience with computing and have developed misconceptions about CS as a major [19].

On an individual level, the absence of resources has a varied effect on students. Some students are precluded from studying CS at university because of their shortcomings in science and mathematics [43]. There is an increasingly large gap caused by the access to ICT skills or lack thereof [14], and students who do not have experience exhibit discomfort when sharing a classroom with classmates who demonstrate an advanced knowledge [20]. However, despite the bleak impression that is created, research indicates that students may be less deterred from pursuing a CS degree than might be expected. In a survey of high school students in Grahamstown, Jacobs found that learners attending previously disadvantaged schools, as well as learners who have no Internet access at school, are more inclined to study CS than their counterparts at schools with better resources [43]. The inclination of students from township schools towards computer-related careers may be attributed to their perception of new ICT opportunities in the labour market [14].

The challenges in CS education indicate that current educational approaches present numerous shortcomings. CT, with its paradigmatic shift, offers a different means of operating that may address these problems. For example, CT may be used to teach foundational CS principles in the absence of technological resources, or may increase interest in CS by focusing on problem solving and logic rather than syntax and mechanics.

2.1.2 Overview of Computational Thinking

CT, a recent philosophy in CS education, has emerged from earlier discussions about the nature of computing, such as the argument by Denning in 2005 for CS to be recognised as a legitimate science discipline [4, 26]. A concrete, unanimous definition for this term remains elusive [9, 10, 42]; however, CT is characterised as a problem solving approach that builds on the skills and techniques gained in CS. The term "computational thinking" was popularised by Jeannette Wing in a seminal paper in 2008 [78]. Although Wing's work is primarily focused on children, it is important for all CT studies as it establishes the foundational concepts of CT. In a subsequent 2011 paper, Wing presents the following definition of CT:

> *Computational thinking is 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 [79].*

To frame further discussions, the distinction between computer programming, problem solving, and CT must be made. Computer programming, or coding, refers to the process of writing programs or formulating solutions to computer problems, usually with a defined programming language. Problem solving refers to the broader process of formulating a solution to any given problem, while the concept of CT provides a means to define the specific problem solving skills required by computing [4].

The relationship between CT and the discipline of CS is the subject of much discussion. It appears that CT is promoted when computing is taught as a subject [42], regardless of whether it is labeled as such. Some views describe CT as a means of framing the problem solving skills employed in CS [4]. Alternatively, CT may be a richer contributor in the sphere of computing; Curzon et al. describe CT as a basis for more precise, deeper, and wider interpretations of computing [24]. Lu and Fletcher [50] advocate the development of a CT language that would assist in describing computation and abstraction, as well as being a form of notation for computational processes.

The importance of CT as a modern proficiency is expressed in its description as a "digital age skill for everyone" and a 21st century literacy [9, 39]. CT describes the symbiotic relationship between human and machine intelligence by interrogating the respective strengths of humans and machines in performing certain tasks [77]. Lu and Fletcher [50] emphasise the role of CT in helping students who act as computing agents to gain more

knowledge, skills and effectiveness. Much of the research in CT focuses on the school level, yet it is recognised as an intellectual and reasoning skill that professionals likewise need to acquire [59].

Although the term "computational thinking" has experienced a resurgence due to the work by Jeannette Wing, it has been used as early as the 1980s by researchers like Seymour Papert [58]. His concept of CT was not the same as Wing's current definition; however, work by Papert lays the groundwork for much of our understanding today. Papert's early work described procedural thinking and programming as the primary elements of CT [38], focusing particularly on LOGO programming as a means for children to learn about procedural thinking, mathematics, complex ideas, and creative expression [38, 47]. In the 1980s, Papert advocated the use of LOGO for the education of children in computing; more recently, the development of the Scratch programming environment was influenced by Papert's work [62].

Wing's work has updated the notion of CT for a 21st century audience [38]. Valerie Barr [67] points to the difference between the unidirectional view presented by Wing, and the bidirectional view presented by Papert that deals with the intersection of disciplines. In an effort to combine the merits of the work by both Wing and Papert, an adapted definition is proposed by Kafai and Burke:

> *[Expanding on Wing's definition of CT], computational participation is the ability to solve problems with others, design systems for and with others, and understand the cultural and social nature of human behaviour, by drawing on concepts, practices, and perspectives fundamental to computer science [47].*

A recurring theme in the discussions around CT is its importance beyond CS, as a fundamental skill for all people [78]; it has been suggested that CT should be added to the *three Rs*[2]. Examples of the impact of CT in statistics, biology, economics, and the humanities are given in [78]. This goal of teaching CT across the entire campus may require a different approach than the approach for teaching students who are assumed to be in pursuit of a career as CS professionals [39].

In her seminal paper, Wing expresses two visions and two challenges that influence the understanding of the present and future of CT. These points, taken from [78], are given below:

[2]The three Rs are reading, writing, and arithmetic.

Vision no. 1. I envision that computational thinking will be instrumental to new discovery and innovation in all fields of endeavour.
Vision no. 2. I envision that computational thinking will be an integral part of childhood education.
Challenge no. 1. What are effective ways of learning (teaching) computational thinking by (to) children?
Challenge no. 2. How do we make our technology and the wealth of our applications accessible to all? How do we balance openness with privacy?

There have been several attempts to create materials that incorporate CT into existing curricula, including Google's *Exploring Computational Thinking*[3] project. Internationally, there is considerable focus on the integration of CT into the K-12 curriculum[4], as discussed in Section 2.1.5. More broadly, resources such as Kahn Academy[5], CS Unplugged[6], and the CS4FN magazine[7] provide alternative resources for CS education with an inherent CT flavour.

2.1.3 Benefits of Computational Thinking

Despite CT not being fully understood, the field has a number of supporters. There are a range of benefits that appear to emerge from the pursuit and practice of CT. These can be arranged into three spheres: benefits within computing, benefits beyond computing, and benefits for personal development.

As a school of thought emerging from CS, CT has clear benefits within the field of computing. In combination with domain expertise, it is the collective thinking ability that makes computing meaningful and fruitful [42]. CT has the potential to make computing more accessible, particularly by lowering the threshold at the pre-university schooling level [66].

Concepts in science and mathematics are constructed around intuitive computational mechanisms, increasing understanding in both domains [66]. A sustained exposure to CT during the formative education phase can render students better prepared to deal with

[3] `www.google.com/edu/computational-thinking`
[4] K-12 is the equivalent of Grade 0 to Grade 12 in South Africa.
[5] `www.khanacademy.org`
[6] `http://csunplugged.org`
[7] `http://www.cs4fn.org`

programming and the concepts within the CS curriculum; the use of contextualised representations makes it easier for students to learn programming [50, 66]. Finally, CT could serve as a motivating factor for students to select CS, as it elevates the subject in terms of its intellectual content beyond the appeal of simply offering career opportunities [50].

The value of computation in other disciplines has long been established, and CT provides an additional avenue where the virtues of CS may be broadly beneficial. CT is a means for computing to "transform whatever it touches" [24], and extends the concepts and tools from CS beyond the commonly acknowledged IT fluency [4].

There is an inherent need for problem solving in all scientific and engineering disciplines [61], and this approach provides a means of exposing students with different interests to computing and CT [24]. The field of CT embodies core scientific practices and aids in the development of pre-algebra concepts [66].

On an individual level, CT helps to develop important skills and understanding in students. Knuth eloquently expresses the sentiment that "you don't really understand something until you've taught it to a computer" [53]. Proficiency in CT builds a range of skills, including the ability to systematically and efficiently process information and tasks [50], a practice that is inextricably linked with CS. When CT and abstract thinking are lacking, it results in an inability to create designs and programs that are clear and elegant [42]. Barr et al. [9] identify a number of attitudes that are both essential to and a result of CT. These include confidence in dealing with complexity, persistence, tolerance for ambiguity and open-ended problems, as well as communication skills.

2.1.4 Criticisms of Computational Thinking

Despite the popularity that CT is gaining, there are some critical views and concerns about its value being aggrandised. Peter Denning is significant in the CS field and a notable sceptic of the CT movement. Some of the arguments made by Denning are presented and discussed below.

> *I am concerned that the computational thinking movement reinforces a narrow view of the field and will not sell well with the other sciences or with the people we want to attract [28].*

Despite Denning's concerns, there are numerous examples of the adoption of CT within other sciences and beyond the science faculty. CT and computational metaphors have been used in fields as diverse as proteomics and law, and enable researchers to ask different kinds of questions within their fields [17]. The contribution of this field to science applications is in the form of understanding the right methods, architectures, programming tools, and techniques [71]. Perhaps the most important idea to take from this concern is the importance of interaction with the wider science faculty to design effective CT courses [40].

> *Computation is present in nature even when scientists are not observing it or thinking about it. Computation is more fundamental than computational thinking [28].*

In order to respond to this concern, we must better understand the nature of the relationship between CT and computing. CT is about the set of mental tools that enable the use of computing for human problems [50]. An understanding of computation is essential to CT, as it often involves selecting or devising a model of computation that is appropriate for formulating and solving problems [2]. Therefore, CT involves an element of understanding existing models of computation and devising new models for novel problems [2]. The same argument that Denning makes could be made for mathematics — mathematics is inherent at all levels of the natural world, but this observation does not lead to an abandonment of mathematical thinking. If computing is present in nature, CT offers an avenue to better understand and reason about the natural world and its inherent processes.

> *Computational thinking is one of several key practices at which every computer scientist should be competent. It short-changes computer science to try to characterise the field by mentioning only one essential practice without mentioning the others or the principles of the field... Do we really want to replace that older notion with "CS = computational thinking"? [28]*

This misconception about what CS encompasses, particularly the view that CS is equated with programming, must be addressed for CT to have a constructive future [61]. CT has an interdependent relationship with computing and is particularly involved with understanding appropriate models of computation [50]. CT has the potential to define computing disciplines and to capture the intrinsic nature of computing [41]. Rather than being an alternative to computing or programming, it is a possible way of communicating the nature of computing to the general public [41].

> *We are most valued not for our computational thinking, but for our computational doing [28].*

A number of educators share a similar viewpoint to Denning in the reluctance to downplay the importance of programming in favour of CT. At Stanford University, a choice was made to not offer a CT course for non-majors. This decision was primarily made to avoid the view of programming as a low-level, mundane activity, and to dispel notions that programming can be unpleasant [24]. Another view is that CS should be about the correct practice of computational doing [71]. Perhaps the best approach to this concern is to reinforce the role of CT as a foundation from which programming skills can be developed, and to incorporate productive activities into the development of CT skills.

2.1.5 Computational Thinking in the Curriculum

In this section, the emergence of CT in proposed curricula is discussed. Two documents are referred to: the Computing At Schools (CAS) curriculum [22] intended for use at the pre-university schooling levels, and the ACM Strongman Computer Science curriculum[8], which prescribes curricular guidelines for an undergraduate level.

Computing At Schools (CAS) Curriculum

The CAS curriculum [22] is a document that aims to define the scope of what should be taught in computing[9] as a school subject. The focus of this document is a definition of what computing is about rather than a prescription of how it should be taught [22]. Although developed by the CAS association with the British Computing Society for use in Britain, this curriculum has broader relevance and could apply to many countries. This curriculum makes a number of statements about the importance of CT.

The CAS curriculum defines CT as the single theme representing all of the key processes in computing. It highlights the ways in which CT causes us to think differently about ourselves and the world, particularly in identifying computing aspects in the world. CT empowers students to reason about natural and artificial systems and processes through the use of tools and techniques from computing. CT is something that is done by machines

[8] Final Report 0.9 (pre-release version); October 2013.
[9] The term "computing" is used interchangeably with "computer science".

rather than computers, and the curriculum indicates that it assists in understanding the relative powers and limitations of human and machine intelligence. A number of skills and abilities that comprise CT are discussed, including the ability to solve problems, design systems, and to think logically, algorithmically, recursively, and abstractly.

A peculiar dichotomy exists in the way that CT makes computing distinct from other disciplines, but also harnesses the skills and benefits from CS to influence other fields like biology, chemistry, psychology, and economics [22]. The CAS curriculum asserts that the ability to combine CT with computing principles and a computational approach is essential for success in science, engineering, business, and commerce.

ACM Strongman Computer Science Curriculum

The ACM Computer Science Curriculum contains a set of curricular guidelines for undergraduate programs in CS. Numerous versions of this curriculum have been released to reflect the changes in CS, with major releases in 2001 and 2013. In this section, we focus on the interim curriculum released in 2008, and the most recent version of the curriculum released in 2013 [45].

The interim revision of the CS2001 curriculum [3], released in December 2008, contains three key references to CT. In this curriculum, CT is loosely defined as "observations about the wider applicability and the relevance of ideas from computing". An understanding of CT elements is given as an expected cognitive ability for CS graduates. This includes the ability to apply CT in everyday life as well as in other domains.

At the time of the CS2008 release, the notion of CT as a facet of CS education was still developing and gaining traction. There is recognition of the need to refine the understanding of what is meant by CT [3]. CT is recognised as one of the important ideas that has matured since 2001, and its potential to change perceptions about the discipline of CS is acknowledged.

While the Strawman Draft [46] released in February 2012 contains no references to CT, version 0.9 of the Ironman Curriculum [45], released in October 2013, has been amended to contain two references. In Chapter 6, which discusses institutional challenges, it is recognised that CT is a fundamental skill for all graduates. There is a belief that in the future every undergraduate student should undertake some level of CS study, with a recommendation made to provide courses to students from varying disciplines. This

Table 2.1: The three parts of the PACT CT Domain (taken from [57]). The parts of this domain are combined to inform computational thinking practice.

CS Concepts	**Inquiry Skills**	**Communication and Collaboration Skills**
Algorithms	Evaluate	Publish
Programming	Explore	Present
Recursion	Analyse	Build Consensus
Abstraction	Explain	Discuss
Debugging / Testing	Elaborate	Distribute Work
Variables	Model	Lead/Manage Teams

is seen as a means of promoting interdisciplinary work and acknowledging the role that computing plays in other disciplines [45].

The second reference to CT is made in the substantiation of the inclusion of Computational Science (CN) in this version of the curriculum. CN is one of the 18 given knowledge areas, and is defined as "The application of computer science to solve problems across a range of disciplines" [45]. CT is described as the core of CN, with specific references to the use of computational power to solve problems within and outside of traditional CS boundaries. This inclusion of CT focuses on the problem solving element and is progressively linked to the collaboration between CS and other disciplines.

2.1.6 Assessing and Integrating Computational Thinking

Numerous attempts have been made to categorise and assess CT, usually with a view of integrating it into the CS curriculum at either the secondary or tertiary level. In this section, we discuss some of the ongoing attempts that have been made.

Principled Assessment of Computational Thinking (PACT)

The PACT project is an initiative to develop high quality assessments of CT. The goal of this project is to create a conceptual assessment framework for CT practices, using an Evidence-Centred Design approach [57]. The PACT project is still developing and has yet to produce an actual framework for use. The PACT team has presented an overview of the CT domain as decomposed into three areas: CS concepts, inquiry skills,

and communication and collaboration skills. The PACT diagram giving examples of these areas is shown in Table 2.1. The following seven design patterns have been identified as a starting point for the CT assessment framework: analyse one's own computational work and the work of others, apply abstractions and models, design and implement creative solutions and artefacts, analyse effects of development in computing, connect computing with other disciplines, communicate thought processes and results in simple formats, and work effectively in teams [57].

Computational Thinking Across the Curriculum

At DePaul University, much work has been done on integrating CT across a broader curriculum. The viewpoint that CT is relevant across many different disciplines is well established [21, 78]. Perkovic et al. [59] present a framework for integrating CT into different courses. At present, this framework does not provide a means of assessing CT, but gives instructors an idea of how it may be integrated into the wider curriculum.

Perkovic et al. [59] define CT as a skill set that is used to apply computational techniques and applications to problems in any given field. Their framework is aimed at an undergraduate Liberal Studies programme which incorporates subjects outside of CS. In this framework, CT is organised into seven categories according to Denning's Great Principles of Computing: computation, communication, coordination, recollection, automation, evaluation, and design [25]. These principles are redefined by the authors for a larger context. Perkovic et al. identify 19 courses within the Liberal Studies programme at DePaul University that could be reworked to include a strong CT component. These courses are categorised according to the seven principles, and three examples of this application are given in [59]. The authors identify the need to evaluate their materials in order to further refine examples and assessments for this framework.

Concepts, Practices, and Perspectives Framework

At the MIT Media Lab, an attempt has been made by Brennan and Resnick to create a CT evaluation tool for use with school students who are learning programming using Scratch [15]. This work is based on the definition of CT by Wing et al. [79] as given in Section 2.1.2.

	Concepts	**Practices**	**Perspectives**
Approach #1: Project Analysis	presence of blocks indicates conceptual encounters	N/A	N/A (possibly by extending analysis to include other website data, like comments)
Approach #2: Artifact-Based Interviews	nuances of conceptual understanding, but with limited set of projects	yes, based on own authentic design experiences, but subject to limitations of memory	maybe, but hard to ask directly
Approach #3: Design Scenarios	nuances and range of conceptual understanding, but externally selected projects	yes, in real-time and in a novel situation, but externally selected projects	maybe, but hard to ask directly

Figure 2.1: Discussion of the proposed approaches for assessing CT (taken from [15]).

The evaluation framework is based on three dimensions: computational concepts, computational practices, and computational perspectives. Examples for each of these dimensions, taken from [15], are given below.

- **Concepts:** sequences, loops, parallelism, events, conditionals, operators, and data.
- **Practices:** being incremental and iterative, testing and debugging, reusing and remixing, and abstracting and modularising.
- **Perspectives:** expressing, connecting, and questioning.

Having defined these dimensions, Brennan and Resnick devised three possible approaches for assessing students, namely, a project portfolio analysis, artefact-based interviews, and design scenarios. These approaches were assessed in terms of how well they address each of the dimensions, as shown in Figure 2.1.

2.1.7 The Chicken-Egg Causality Problem

In CS, a classic chicken-or-egg causality dilemma emerges: whether CT or programming is more fundamental. This issue is salient as it informs our understanding of the discipline and approach to teaching. The question whether programming, computers, and CT can legitimately be separated has been raised at a National Research Council workshop [38]. Although programming is undoubtedly a core CS practice, concerns have been raised

about its appropriate introductory point in the curriculum. Lu and Fletcher [50] argue that programming should not be a student's first encounter in CS, but that it should be introduced in higher CS.

Some argue that programming, or "computational doing" is the true value of CS [28]. Programming is established as a fundamental skill and a means of demonstrating computational competencies [38].

CT is defined by Selby [65] as strategic tools that are useful for solving problems with computational devices, implying that CT is contingent on the use of a conventional device. However, other proponents of CT state that it goes beyond the notions of software and hardware and is based in a body of theoretical and practical knowledge [22].

It is suggested that for students, the ability to produce working code is less important than the ability to understand and explain programs [22]. Denning [27] states that computing pre-dates the invention of computers, and that information processes are in fact a natural phenomenon. The CAS proposed curriculum states that insight into computational systems may be gained regardless of whether computers are used [22]. This sentiment is echoed in the commonly used maxim "computing is no more about computers than astronomy is about telescopes" [22], and in the comparison that "programming is to computer science as playing an instrument is to music" [21].

If therefore, the field of CS is not contingent on the use of computing machines, the theories and modes of thinking underlying it cannot be inextricably linked to the practice of programming. Furthermore, the use of CT to further other disciplines indicates that it has value beyond the specific practices in CS. Although a mutually beneficial relationship exists between CT and programming, the comments above suggest that the former underpins the latter.

2.2 Theories of Education and Learning

There are numerous studies and models aimed at understanding the developmental process that students go through and the kinds of thinking they employ. An understanding of these models helps us to frame our understanding of the student perspective, and to foster realistic expectations of how their abilities should progress. Although many of these studies are based on a general understanding of knowledge and education, in some cases they have specific ramifications for CS.

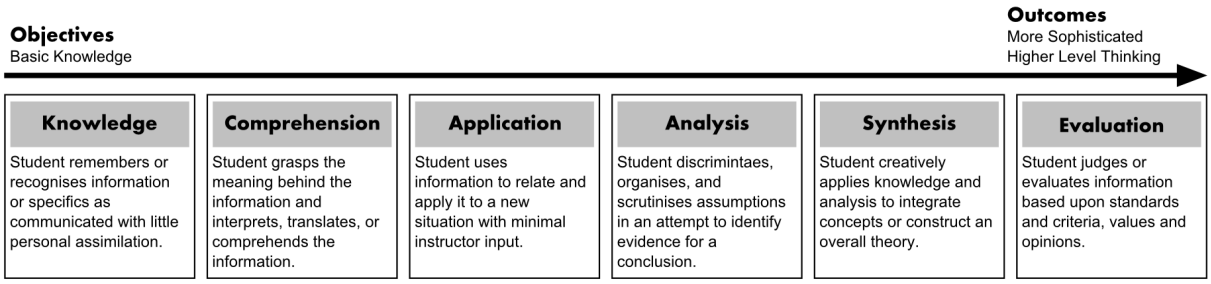


Figure 2.2: Bloom's taxonomy for the cognitive domain (reproduced from [13]).

2.2.1 Bloom's Taxonomy of Learning

Bloom's Taxonomy of Learning (BT) was created by Benjamin Bloom in the 1950s to describe the aspects of learning and remedy shortcomings with existing examination question types [44, 69]. This is one of the most widely used educational taxonomies [32]. BT has been associated with creative and critical thinking, problem solving, and technology integration [31]. Learning taxonomies are useful for describing learning outcomes and performance, and for describing the stages that an individual goes through in a learning process [32]. BT is widely used to ensure the correct balance between rote learning and high level skills, and forces educators to reconsider the nature of exam questions [44, 73]. Fuller et al. [32] identify a number of strengths of the taxonomy, including its simplicity and distinct aspects of the cognitive domain.

The taxonomy categorises human learning into three domains: cognitive, affective, and psycho-motor [64]. The cognitive domain encompasses mental and learning skills and is the basis for this discussion. The cognitive aspect has been divided into hierarchical levels [44]; there are six levels in the cognitive domain that should be mastered sequentially. An illustration of BT in the cognitive domain is given in Figure 2.2.

In a CS context, BT has been used to structure assessments, design and evaluate courses, and compare the difficulty of courses [73]. There are however, a number of shortcomings of BT when applied to CS. There is a lack of consensus about how BT applies to tasks in introductory programming courses, and it is difficult to apply consistently [73]. BT does not adequately represent key practices in CS, such as the development of artefacts [32]. Typical assessments in CS do not follow the BT levels sequentially. Suggested improvements to BT for CS include a focus on application as a higher-level aspect [44], or the adaptation into a new, two-dimensional taxonomy which places *producing* on one axis and *interpreting* on the other [32]. In this way, we see a focus being placed on the creative, tangible elements of CS as a distinguishing factor from other disciplines.

Below, we give a description (reproduced from [13]) and examples (taken from [64]) of CS questions for each of the levels in the cognitive domain of Bloom's taxonomy. An additional set of examples can be found in [69].

- **Knowledge:** Student recalls or recognises information, ideas, and principles in the approximate form in which they were learned.
  CS examples: Name three kinds of looping structures; list three methods of performing I/O on a computer.
- **Comprehension:** Student translates, comprehends, or interprets information based on prior learning.
  CS examples: Explain what happens in a given piece of code.
- **Application:** Student selects, transfers, and uses data and principles to complete a problem or task with a minimum of direction.
  CS examples: Combining two concepts such as arrays and records. For example, create a data structure that stores the name, job, and salary for 50 workers.
- **Analysis:** Student distinguishes, classifies, and relates the assumptions, hypotheses, evidence, or structure of a statement or question.
  CS examples: Compare the advantages and disadvantages of a given architecture.
- **Synthesis:** Student originates, integrates, and combines ideas into a product, plan or proposal that is new to him or her.
  CS examples: Give the interface for a class that can represent fractions; illustrate the inheritance of polymorphism for a class.
- **Evaluation:** Student appraises, assesses, or critiques based on specific standards and criteria.
  CS examples: Find logic errors in a given piece of code; comment on the negative and positive points of a code listing.

Revised Bloom's Taxonomy

A revised version of Bloom's taxonomy was created in the 1990s by Lorin Anderson, a former student of Bloom [31]. This revision was intended to update the taxonomy and make it more relevant for use in the 21st century. A major difference in terminology between the two versions is the adaptation of Bloom's six categories from a noun form to

Table 2.2: Mapping of the levels in the original and revised Bloom's taxonomy, adapted from [31].

Original Bloom's Taxonomy	**Revised Version**
Evaluation	Creating
Synthesis	Evaluating
Analysis	Analysing
Application	Applying
Comprehension	Understanding
Knowledge	Remembering

a verb form [31]. The mapping between the original nouns and the new verbs is given in Table 2.2. Additionally, a series of knowledge elements was added to the taxonomy, giving it a matrix structure [32]. The four elements are: factual knowledge, conceptual knowledge, procedural knowledge, and meta-cognitive knowledge. A description and definition of the six verbs in the revised Bloom's taxonomy are reproduced from [31] below.

- **Remembering:** retrieving, recognising, and recalling relevant knowledge from long-term memory.
- **Understanding:** constructing meaning from oral, written, and graphic messages through interpreting, exemplifying, classifying, summarising, inferring, comparing, and explaining.
- **Applying:** carrying out or using a procedure through execution or implementation.
- **Analysing:** breaking material into constituent parts, determining how the parts relate to one another and to an overall structure or purpose through differentiating, organising, and attributing.
- **Evaluating:** making judgements based on criteria and standards through checking and critiquing.
- **Creating:** putting elements together to form a coherent or functional whole; reorganising elements into a new pattern or structure through generating, planning, or producing.

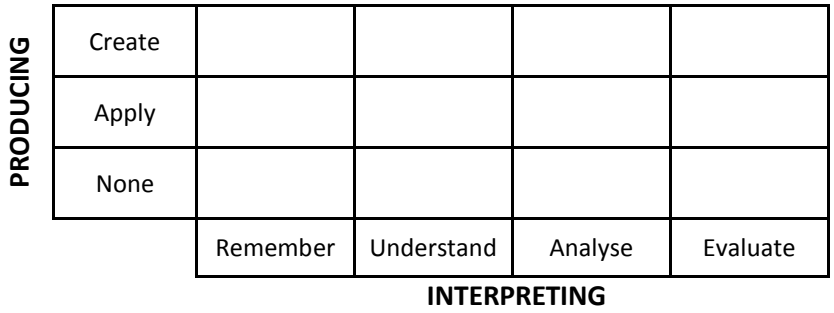


Figure 2.3: A graphical representation of the CS learning taxonomy (reproduced from [32]).

2.2.2 A Computer Science Specific Taxonomy

In recognition of the limitations of BT in a CS context, a new taxonomy [32] was developed to better fit the peculiarities of this discipline. These authors reviewed existing educational taxonomies including BT and the SOLO Taxonomy[10]. Ways in which CS may differ from other subject domains are identified, such as the ability to develop artefacts, learning through doing rather than through interpreting, and problem solving through models of the real world. A representation of this taxonomy is given in Figure 2.3.

The new taxonomy is based on the revised version of BT but attempts to move away from the linear approach by using a matrix structure [32]. This taxonomy incorporates two areas of competency: the ability to design and produce a new product, represented on the vertical axis, and the ability to understand and interpret an existing product, represented along the horizontal axis. Within these areas, names of levels from the revised BT are used. In the application of this matrix, students traverse each axis sequentially, beginning at the lower left corner. However, students may traverse one axis further than the other, leading to different "learning paths" that are mapped through the matrix. The learning path taken gives the educator an indication of the type of student, such as a theoretical student or practical student, as further described in [32].

2.2.3 The Felder-Silverman Learning Model

The Felder-Silverman Learning Model (FSLM) [29] provides a detailed description of different learning styles. This model is not entirely novel, with Felder and Silverman having based parts of it on the Myers-Briggs Type Indicator (MBTI)[11], Kolb's Learning

[10]Structure of the Observed Learning Outcome
[11]`http://www.myersbriggs.org`

Style Model, the Herrmann Brain Dominance Instrument, as well as work by Jung [29, 72]. The FSLM was developed as a reaction to the disconnect between professors' teaching and students' learning. In several ways, the teaching and learning styles are found to be incompatible. This poses a particular problem, as student success in a class is influenced by their compatibility with the instructor, as well as their own native ability and prior preparation.

The FSLM provides a means of classifying and understanding the different learning styles exhibited by students. This model is based on the understanding of learning as a two-step process consisting of the reception and processing of information; this reception and processing is what the learning-style model aims to classify [29]. In particular, the FSLM focuses on the preferred learning style of students rather than their abilities, and classifies students according to their position on a set of scales or dimensions [72].

The five learning style dimensions[12] are explained below. Note that unless otherwise referenced all information was taken from [29].

Sensing and Intuitive: The sensing and intuitive modes of perception were introduced by Carl Jung, and are also used in the MBTI. Sensing generally involves gathering information through the senses using observation, while intuition involves gathering data by means of the unconscious, such as speculation and imagination. In engineering, most professors appear to be intuitors, while most students are sensors, leading to a disconnect between preferred styles.

Sensing students are characterised by the following features: they prefer facts, data, and experimentation; they are more comfortable with standard problem solving methods; they like details but dislike complications, and they work carefully but can be slow.

Intuitive students are characterised as follows: they prefer principles, theories and innovation; they dislike repetition and are bored by details; they grasp new concepts easily; they work quickly but can be careless, and they are more comfortable with symbols than sensing students.

Visual and Auditory: The ways of receiving information are divided into three modalities: visual, auditory, and kinaesthetic (tasting, touching, and smelling). The first

[12] In a 2002 preface to the paper on learning styles, Felder makes changes to two of the dimensions. The inductive / deductive dimension is excluded as he believes that induction is the best method for teaching at a pre-graduate school level, and the visual / auditory dimension is renamed visual / verbal to accommodate written prose, which is perceived visually but processed by the brain in the same manner as the spoken word [29].

two modalities are included in this dimension. At a college level, most teaching is verbal but the preferred learning style for most students is visual. The best teaching methods should incorporate presentations with both visual and auditory modalities; resources such as flow charts and diagrams can be used to illustrate complex processes and reinforce visual learning.

Visual students experience the best recollection with items they have seen, such as pictures, diagrams, time lines, films, and demonstrations.

Auditory students prefer verbal explanations, and remember most of what they hear and say; they learn well by explaining to others.

Inductive and Deductive: At the time of writing for Felder and Silverman, most curricula in engineering were deductive whereas most students were identified as inductive learners. Inductive learning has been established as the more effective teaching approach. The danger with deductive teaching is that students develop an unrealistic concept of the instructor's ability to conceptualise a complex derivation and a negative view of their own abilities in relation.

Inductive learning is the most natural learning style and is characterised by a progression from particular data such as observations and measurements to generalities such as rules, laws, and theories.

Deduction is the most natural teaching style for technical subjects, and is characterised by a progress from governing principles down to applications, or the deduction of consequences.

Active and Reflective: The mental process for the conversion of information into knowledge takes two forms: active experimentation and reflective observation. Engineering students appear to have a greater tendency towards active learning. Felder and Silverman highlight an essential semantic point: the opposite of active is passive, rather than reflective, and passive class environments do not effectively service either type of student.

Active students are better at experimentation and learn by doing things in the external world. These students need activities beyond listening and working, they work well in groups and do not work well in passive situations.

Reflective students are more prone to introspection and thinking about information, they work best by themselves or with one partner. These students tend to be theoreticians or mathematical modellers, and are adept at defining problems and proposing solutions.

Sequential and Global: Materials in formal education are usually presented as a logically ordered progression. This system is the preferred method for sequential learners, who tend to grasp material in the order it is presented. In contrast, global thinkers need to see more of the big picture before they are able to grasp a concept, and often struggle with conventional teaching methods. Sequential learners are well catered for across all levels of education, but efforts need to be made to cater for global learners, such as providing the big picture of a lesson at the outset, and relating and contextualising information with students' experience.

Sequential students are strong in convergent thinking and analysis, they are able to follow linear reasoning processes and work with material that they only understand partially. These students favour a steady progression of complexity and difficulty.

Global students are better at divergent thinking, synthesis, and seeing connections, and tend to be multidisciplinary researchers and systems thinkers. These students are better at working directly with more complex material, but they do not learn in a predictable manner and have difficulty when they understand work only superficially.

Despite its conception as a model for understanding engineering students, the FSLM is equally applicable to CS students. Thomas et al. [72] surveyed a class of 107 introductory programming students and found Felder and Silverman's [29] observations about engineering education to apply to software engineering education. Some key findings from their study are given below [72].

- In the exam administered during the research, reflective learners performed better than active learners and verbal learners performed better than visual learners.
- The largest group, comprising 12% of the population, were the reflective, sensing, visual, sequential learners.
- The smallest group, who were also the best performers in both the course and the exam, were the reflective, sensing, verbal, sequential learners.
- The most disadvantaged group with current teaching methods were the active, sensing, visual learners.

2.2.4 The Five Strands of Proficiency Model

The five strands of proficiency model [51] is the result of a 2001 project by the National Research Council in the United States. Similarly to current sentiments about CT, the

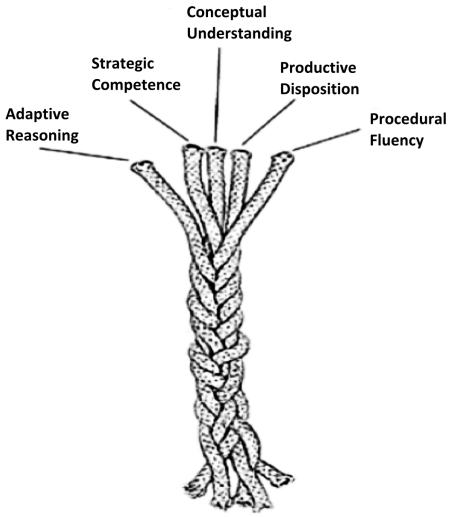


Figure 2.4: The five strands of mathematical proficiency (taken from [51]).

project stems from the view that all young students must learn to think mathematically. Mathematical proficiency is defined as a term attempting to capture the expertise, competence, knowledge, and facility in mathematics [51]. This investigation into proficiencies is important to create a foundation for mathematical learning [48]. Students' starting levels and proficiency must be specifically recognised to produce fruitful results. The five strands model is a framework that encompasses the knowledge, skills, abilities, and beliefs that are integral in mathematical thinking [51]. A visual representation of the model is given in Figure 2.4. The five stands of mathematical proficiency, taken from [51], are described as follows:

- **Conceptual Understanding:** comprehension of mathematical concepts, operations and relations;
- **Procedural Fluency:** skill in carrying out procedures flexibly, accurately, efficiently, and appropriately;
- **Strategic Competence:** ability to formulate, represent and solve mathematical problems;
- **Adaptive Reasoning:** capacity for logical thought, reflection, explanation, and justification;
- **Productive Disposition:** habitual inclination to see mathematics as sensible, useful and worthwhile, coupled with a belief in diligence and one's own efficacy.

This model is given as a set of twisted strands representing the interdependent relationship between the five items [48, 51]. Mathematical proficiency is a multi-dimensional trait [51], and cannot be achieved if any of the strands are neglected. A particular problem in school mathematics is the exclusion of some strands in favour of one or two strands that are deemed more important [51]. Acquiring proficiency is a process that progresses over time, with the development of each strand facilitating the development of others. Adequate time must be allowed for students to encounter activities for a specific topic in order to build proficiency. Despite the focus on mathematics, these strands of proficiency have numerous avenues where they are applicable, including CS. A more detailed description of each strand and its relevance in a CS context is given below.

Conceptual Understanding

This strand represents a grasp of mathematical ideas that is both integrated and functional [51]. Conceptual understanding refers to a sense of the context and importance of mathematical ideas that extends beyond the knowledge of isolated facts and ideas. This relates to students' mental models where facts and methods are interconnected, and results in a greater retention and ability to avoid critical errors. The organisation of understanding is often hierarchical with complex ideas that are representative of clusters of smaller ideas [51].

More broadly, conceptual understanding is about understanding why components fit together the way that they do. In CS, this is relevant to the smaller concepts and operations that students learn, and the ways in which different levels of abstraction work to combine and augment these concepts. Conceptual understanding can be observed in CS concepts such as system design and inheritance.

Procedural Fluency

Procedural fluency centres around student interaction with procedures. Specifically, it refers to the ability to use them, appropriately, flexibly, accurately, and efficiently. Students need to develop a certain level of ability independent of aids; for example, the ability to perform basic mathematical computations with whole numbers without a calculator. Algorithms, as well as mental and written computations, are essential for many mathematical tasks in everyday life. Additionally, procedural fluency should enable students to

estimate what the result of a procedure should be, and to see procedures as generalised solutions for entire classes of problems [51].

In terms of CS, procedural fluency requires students to learn the syntax of the language they are using, as well as developing skills of typing and using the necessary development tools. Without procedural fluency, students will be unable to solve problems in a computational way.

Strategic Competence

Strategic competence is similar to problem solving, and encompasses the ability to formulate, represent, and solve mathematical problems. This strand requires students to develop an understanding of a problem situation and to construct a mental model of the variables and relations, rather than using more primitive methods. A student with strategic competence should be able to choose from strategies such as reasoning, guess-and-check, algebraic, or other methods to suit the problem, and to devise novel methods as needed [51].

This strand encompasses the ability to formulate an appropriate strategy for a problem and to see where the approach is heading. Included in this is the ability to reformulate problems into an appropriate notation and to develop solutions that are contextually appropriate, both of which are skills that are needed to progress in CS.

Adaptive Reasoning

Adaptive reasoning is the connective strand between items, and represents the ability to think logically about concepts and situations and the relationships between them. This notion includes informal explanation and justification of an approach, as well as inductive reasoning based on items like patterns, analogies, and metaphors. Adaptive reasoning enables a student to see different facts, procedures, concepts, and patterns fitting together in a way that makes sense, and can manifest in the ability to justify their work [51].

Within this strand, students should develop the ability to adapt what they have already learned to apply to a new problem. In CS, solutions often consist of many different components and it is essential that students understand the purposes and interactions of these components.

Productive Disposition

A productive disposition is perhaps the most integral and essential strand. This strand develops when the other strands do and facilitates the development of the other strands [51]. Broadly, a productive disposition refers to the ability to see value, both in mathematics as a useful and worthwhile discipline, and in the student's own abilities and skills. Students must develop a view of themselves as capable of learning mathematics and using it for problem solving. To develop a productive disposition, students must have a chance to make sense of mathematics, see the benefits of their perseverance and enjoy the experience of making sense of a concept. If this fails to occur, it may have ramifications at a high school level, where students encounter the opportunity to avoid mathematics, perpetuating the shortage of skills [51].

The value of a productive disposition is transparent for all disciplines. Specifically, Barr et al. [9] refer to tolerance of ambiguity, confidence dealing with complexity, and persistence when working with difficult problems as essential to the development of CT. These attitudes are all hallmarks of the productive disposition students must develop.

2.3 Discipline-Specific Types of Thinking

CT is by no means the only form of discipline-specific thinking, and shares commonalities with other types of thinking. Mathematical and scientific thinking are well-established, while other forms are increasingly being explored. Computing is a subject without clear borders and has elements of engineering, science, and art, as well as scientific, mathematical, and practical dimensions [21, 24, 78]. Hu [42] defines CT as a hybrid thinking paradigm. Fisher [30] advocates for the importance of thinking skills and 'metacognition', or thinking about one's thinking. There is evidence of overlap in discipline-specific forms of thinking; these different forms of thinking exhibit both a complementary and supplementary relationship with many shared elements. There have been attempts to change the names given to these terms over time, as terms like algorithmic thinking and computational science are in some instances being replaced with titles like mathematical thinking, abstract thinking, and CT [53]. The ideas generated in CT inform work in other disciplines [22]. The relationships between CT and mathematical, scientific, analytical, and algorithmic thinking are described below.

2.3.1 Mathematical

The link between computation and mathematics has long been supported. Hu argues that programming is a mathematical activity, and that the practice of computing exercises the ability to think mathematically and follow mathematical thinking processes, and that CT necessarily contains a mathematical component [42]. His examples of mathematical thinking in computing include thinking recursively, abstractly, logically, and procedurally.

Moursund identifies computational mathematics as a recently emerged sub-discipline, alongside pure and applied mathematics [56]. He further states that CT is applicable in all of these sub-disciplines, and must be integrated into mathematical thinking. CT is an important part of mathematics owing to the overlap between CS and mathematics [56]. CT shares general approaches to solving a problem with mathematical thinking [78]. Although it began with foundations in mathematics, CS has developed principles that extend beyond this foundation [26].

2.3.2 Scientific

CT shares common areas with scientific thinking, such as the ways in which computability, intelligence, the mind, and human behaviour are understood [78]. Peter Denning states that "science, engineering, and mathematics combine into a unique and potent blend in our field" [26]. He identifies some activities that are primarily scientific, such as experimental algorithms, experimental CS, and computational science. CT must have a scientific element in order to understand and maximise use of models' capabilities, and to explore the effects that computation has in the problem domain [42].

2.3.3 Analytical

Analytical thinking is about decomposing a whole entity into its constituent parts to understand the features and relationships between these parts. CT is described as one kind of analytical thinking and has similarities with engineering thinking in the ways that large, complex systems are designed [78]. CS uses analytical concepts and tools [4, 24]. CT must include analytical thinking in order to create models that have purpose, evaluate and adjust these models, and study their implications and consequences [42].

2.3.4 Algorithmic

The concepts we now refer to as CT were known as algorithmic thinking in the 1950s and 1960s [28]. Algorithmic thinking is well established as a skill in CS [21]. It is defined as a pool of abilities relating to the construction and understanding of algorithms [33]. Algorithmic thinking can be learned independently from learning programming, but is an essential skill for programmers to develop. CT must have an algorithmic component, particularly to step-wise define or refine operational processes [42].

2.4 Summary

In this chapter, background information related to the integration of CT into introductory CS was provided. A number of ongoing challenges in the CS education landscape were identified. There is potential for CT to mitigate some of these challenges, by addressing misconceptions about the discipline and drawing interest from diverse students.

An overview of the existing definition of CT was given, which can be summarised as the thought processes and problem solving skills brought about by computing. The benefits of CT within computing, beyond computing, and for personal development were described, and criticisms of the field were discussed. The respective relationships between CS, CT, and programming were investigated to situate CT within the CS field. The importance of CT within the curriculum was established, and several existing attempts to assess and integrate CT were mentioned.

Different theories of learning, particularly relating to computing, engineering, and mathematical education, were discussed to understand the developmental process and perspective of students; Bloom's taxonomy and the CS-specific taxonomy are notably important for subsequent work in this thesis. Mathematical, scientific, analytical, and algorithmic thinking were discussed as forms of discipline-specific thinking, and their relationships with CT identified.

Throughout this chapter, various aspects of the teaching and learning of introductory CS have been discussed, which influence the subsequent work in this thesis. In response to the challenges in CS, the CT framework described in Chapter 3 has been designed with remedial use in mind, and a CT activity was used successfully with a remedial CS class, as described in Chapter 5. The results of the CT assessment in Chapter 4 show

both genders to be similarly matched in CT ability, partially addressing the problem of varied demographic group involvement in CS. The existing definitions of CT described in Section 2.1.2 have been proven to be too broad, which is part of the motivation for the CT framework. The different educational theories are contextually important as our CT work is intended to supplement traditional methods; the CS specific taxonomy of learning was particularly influential in the framework design described in Chapter 3. Finally, the different types of thinking were discussed to contextualise CT and identify appropriate areas of overlap. For example, algorithmic thinking forms an element of the CT framework in Chapter 3, while many of the CT strategies discussed in Chapter 6 have a grounding in mathematical thinking.

Chapter 3

Computational Thinking Framework

In Chapter 2, the definition of CT was discussed and views both supporting and questioning the existence of this field were presented. We further considered learning taxonomies and educational theory in order to contextualise this research.

Having identified the need for ways of incorporating CT into the education of novice students, we have designed a CT framework (CTF) to serve as a foundation for designing CT materials [36, 37]. This framework describes the skill sets that comprise CT as well as the different natures in which these skills may be practised. This chapter includes a description of the purpose, development process, and final form of the CTF. The primary uses of this framework and its relevance within a CS curriculum are also described.

3.1 Purpose of the Framework

The aim of developing a CT framework is to provide a theoretical grounding for future endeavours in this field. This framework is intended to address the problem of how to identify, evaluate, and incorporate CT into education. The possible uses for the CTF in five broad areas are discussed below.

Curriculum design: The framework should serve as a tool around which curricula can be designed. It should allow educators to consider the different facets of CT that may be beneficial to cover in a foundational CS module.

Classification: The framework can be used to evaluate and classify existing materials and exercises to better understand the CT implications of these. The kinds of materials that could be classified include games, logic exercises, and puzzles.

Diagnostic: The framework can be used to design an assessment for students, providing detailed feedback for individual and collective students. This allows specific problem areas for each student to be identified and remedied, as well as gauging the competencies of a class as a whole.

Remedial: Together with a diagnostic exercise, the framework can be used to design activities around the specific areas where students have weaknesses, providing targeted intervention, and ensuring that the necessary cognitive skills are in place when further studies are undertaken.

Supplementary: While upholding the integrity of CT as a distinct field, the framework has enough parallels with traditional programming practices and studies to allow CT to be presented in a supplementary form, thereby enhancing and augmenting traditional CS studies.

3.2 Framework Design

The design and development of this framework was a cyclic process involving research, construction, and application. Initially, a literature survey was completed to identify key ideas around CT, forming the theoretical basis for the framework. The key ideas emerging from this survey were organised into a provisional framework, as described in Section 3.3. This framework describes CT in terms of seven practices, with a description of the actions that might fall under each. Problems with this version were identified and influenced the design of a new framework.

The design of the new framework has been shaped by existing learning taxonomies, particularly the CS-specific taxonomy described in Section 2.2.2. The final version of the framework, described in Section 3.4, groups CT into six different categories that describe actions, skills, and ideas, and includes a dimension for different natures of engagement with the subject.

Computational Thinking In order to be proficient in computational thinking, one must possess the following abilities	
Think algorithmically	Specify (describe) a problem precisely; identify the basic actions needed to solve the problem; construct an algorithm using these actions; explain the reasoning when presented with an existing algorithm; evaluate and improve the efficiency of an algorithm.
See solutions as processes	Describe a solution as being composed of a number of discrete steps; identify the existence and relevance of sequencing in these steps; trace a flow of execution – the progression from one step to the next; distinguish the differences between the first and final state.
See patterns	Establish that the problem belongs to a general class of problems; determine that patterns occur within the problem itself (iteration); use smaller instances of the same problem to solve the larger problem (recursion); identify the boundary cases in a problem, for which a variation on the solution may be needed.
Think abstractly	Establish that a problem has different levels of complexity; differentiate between the micro level (inner workings of a class) and the macro level (how that class is used by other classes) of a problem; work simultaneously across multiple layers.
Design a system	Identify the problem domain; identify the solution space; determine the resources available (including an information-processing agent); prepare a solution that addresses the problem within the solution space.
Understand data	Describe data in terms of models and representations; explain data types and the organisation of data; see solutions as a process of transforming data; investigate how data types and representations correlate with certain techniques.
Use inference	Observe an error in the solution to a given problem; hypothesise about what and where the error may be; evaluate the hypothesis; amend the solution to fix the error; investigate a generalisation to get a result for a specific case; derive (infer) generalised knowledge from specific examples; apply heuristics to a problem to simplify or solve it.

Figure 3.1: The first draft of the computational thinking framework (CTF). This framework was derived from a review of the literature on CT, in which key concepts were identified, grouped into related categories, and defined as sets of practices.

3.3 First Draft of the Framework

At the start of this research, an initial framework was designed and developed to make sense of the concept of CT. This initial framework sought to describe the particular abilities required for CT in students, and defined CT according to seven abilities: thinking algorithmically, seeing solutions as processes, seeing patterns, thinking abstractly, designing a system, understanding data, and using inference. A series of practices was defined for each of these abilities, drawing on the practices that emerge while programming. The first draft of the framework is shown in Figure 3.1.

CT Categories	Nature of Engagement: Recognise	Understand	Apply	Assimilate
Process and Transformations				
Models and Abstractions				
Pattern and Algorithms				
Tools and Resources				
Inference and Logic				
Evaluations and Improvements				

Figure 3.2: The final version of the CTF.

With this initial version of the framework in place, a CT assessment was carried out on the game Light-Bot[1]. This assessment was used as means of evaluating the practical application of the framework and to identify potential problems for further use. This attempt at using this initial framework for an assessment revealed multiple avenues for improvement. The shortfalls of this version of the framework were: it was not broadly applicable, it did not encompass all concepts deemed important, it did not allow for the different types of engagement, it was too closely related to programming, and it did not lend itself to producing a quantified result.

3.4 Final Version of the Framework

The final CTF is illustrated in Figure 3.2. The CTF has two primary components: the CT categories, embodying different groupings of CT skills and concepts, and the nature of engagement that defines a progression in the acquisition and use of these skills. These components are arranged along two axes, with a matrix structure that is used for grading and producing results. The vertical axis with the CT categories is the primary axis, which is referred to in all applications of the framework. The horizontal axis with the nature of engagement is the secondary axis, which is used depending on the nature and requirements of the application.

[1]A detailed discussion of Light-Bot and an evaluation of the game using the final version of the framework is given in Chapter 5.

3.5 Vertical Axis of CTF

For the CTF, we have broken down the field of CT into six distinct areas, encompassing the problem solving skills that are learned through programming and CS. As a student becomes more experienced in CS, their ability in all of these areas should improve. This decomposition allows for a multi-pronged approach; students may be tested in all areas, and then receive further intervention in areas where they are found to perform poorly.

To classify the skill sets, we aggregated all the concepts that other authors believe to be part of CT. We then arranged these concepts into related groupings, and gave each grouping a suitable name. The resulting classifications are described below.

3.5.1 Processes and Transformations (P & T)

The ability to see solutions as processes is cited as a major focus in CT [42]. Many computational exercises revolve around the processing and transformation of some kind of data or input [42]. Furthermore, it is suggested that CT should include the process of defining solutions as a series of ordered steps [9]. Through appropriate activities and examples, students may also be introduced to input, output, and parallel processing. There are various artefacts that students can use to help them to visualise processes, including state transition diagrams and flow charts.

This category encompasses the idea of decomposing a problem into multiple steps, and the processes required to solve the problem for each of these steps. Furthermore, it includes the way that data is transformed throughout the process, and the recognition of different states. The take-away idea for this skill set is planning: chunking a problem into its constituent parts that need solving, and then organising the solution into a logical progression.

3.5.2 Models and Abstractions (M & A)

The ability to think abstractly has been cited as one of the most fundamental skills for CS students. Many would agree that abstraction forms the basis of all forms of CT [9, 78]; it is suggested that an inability to think abstractly is closely related to an inability to produce elegant, high-quality solutions [42].

Barr et al. [9] state that the representation of data through abstractions is a necessary skill. The notion of representations is prevalent in CS, particularly for data storage and manipulation. Anecdotal experience suggests that novice programming students may experience difficulty moving between different representations; for example, when given a mathematical formula, they exhibit an inability to express the formula programmatically in a language like Python. This ability to reformulate problems is a necessary skill for programming studies [77]. Models and simulations can reinforce the understanding of abstraction [9], and assist in comprehending the problem domain.

This CT category embodies the question of how to represent a problem and solution. It underscores the importance of abstract thinking, particularly the ability to simplify complex problems, getting to the core of the problem, and moving between higher and lower levels of detail.

3.5.3 Patterns and Algorithms (P & A)

Algorithmic thinking is used extensively in CS, and is, therefore, acknowledged as a component of CT [9]. Patterns in CS can take many forms, including the use of loops for iteration and recursion, and functions for widely used pieces of code. An understanding of these patterns leads students to solutions that are simpler and more elegant, or that would have been impossible without using the appropriate strategy. Being able to identify repetitive behaviour allows students to capitalise on this in the problem solving process.

This category encompasses the ability to recognise patterns and similarities in a problem or group of problems, as well as the use of an algorithmic approach to devise solutions. The primary focus is on generalisation and simplicity: by using patterns, students are able to create solutions that are simplified, and both adaptable and reusable for a wider range of problems.

3.5.4 Tools and Resources (T & R)

CT is more tool-oriented than other forms of thinking [9]. There is a dependence on machines as tools to facilitate computation, but we can also view programming constructs as tools to be used and manipulated. When novice students begin programming, there are a number of concepts that are conventionally learned, such as variables, conditionals,

and iteration. These concepts become building blocks or *tools* that can be used to solve more complex problems.

Resource dependence and allocation are useful concepts for students to grasp [63, 77]. The ability to select the most appropriate tool for the task signifies an accomplished programmer. The nature and limitations of the desired goal should be considered when making these design decisions.

This CT category encompasses the availability, strengths, and limitations of the resources that can be used to solve a problem. When approaching a problem, students must consider what tools are available. It may be necessary to create new tools or combine existing tools. Similarly, students need to consider what resources are available, what the limitations of these resources are, and which resource is best suited to achieve the requirements.

3.5.5 Inference and Logic (I & L)

There can be no doubt that logic is essential in CS. Barr et al. [9] list logically organising and analysing data, as well as the ability to generalise and transfer problems, as essential components of CT. They also discuss a tolerance for ambiguity as an important mindset. Following from this, we assert that CS students need strong comprehension skills to extract the relevant details from a problem, using techniques like induction and deduction. CT is a creative exercise, requiring students to develop their own heuristics for solving problems, and to develop new, inventive approaches to problems [42, 63].

This category encompasses the thinking skills and techniques used to formulate a solution to a problem; it includes how information is discovered and expanded, as well as how heuristics are developed to produce novel approaches to problem solving.

3.5.6 Evaluations and Improvements (E & I)

Identifying, analysing, and implementing possible solutions are significant steps in becoming an accomplished problem solver; this is done with the intention of finding the optimal combination of actions and resources [9]. There are two core ideas here, debugging and performance.

The approach to debugging should go beyond the simple task of finding flaws. Firstly, debugging should involve a *divide and conquer* approach to identify the specific location

of a problem. This can be done formally through the use of mechanisms like breakpoints, or more informally through methods like print statements. Secondly, debugging involves a process of hypothesising and experimenting to ascertain under which conditions the problem occurs, often done as an iterative process of narrowing down the search space for the error. Once the location of the error has been identified, attempts at correcting it can be made.

The performance of a solution is often overlooked by novice programming students, but can become a significant issue at a later stage. There are a number of factors students need to consider concerning the performance of their solutions, including scalability, redundancy, and generalisability.

This CT category encompasses the skills and techniques used to identify and correct errors in the proposed solution to a problem, as well as the ability to evaluate and select the best solution for a given problem. The take-away idea for this category is the ability of the student to critically evaluate their own work, for identifying flaws or possible improvements.

3.5.7 Illustrative Example

As an example of the relevance of these categories, consider the scenario where a student needs to design an application to assist in managing his music collection. He has developed a system that allows him to input the songs that he owns, each of which is represented as a *Song* object. The program contains a collection of *Song* objects in the order in which they were entered. The student would like to add a method that allows him to sort the list of songs. From his prior CS studies, he is familiar with the selection-sort, merge-sort, and quick-sort algorithms. Information about the relative merits of sorting algorithms is taken from [55].

Processes and Transformations

The sorting mechanism is a process that is applied to the collection of songs. The pre-condition is an unsorted collection and the post-condition is a sorted collection of items. The student must determine what the desired state after the transformation is, for example, whether he would prefer the sorted collection to be contained in the original structure or as a copy in a new data structure.

Models and Abstractions

The representation of the group of items in a data structure like a linked-list, dictionary, or array will affect the student's implementation of the sorting method. Furthermore, he must consider the representation of each *Song* object to determine which attribute should be used as the sorting criteria. Once the sorting criteria has been chosen, the student may employ abstract thinking to disregard the irrelevant details of each song and focus specifically on the sorting mechanism.

Patterns and Algorithms

Sorting is an inherently iterative process that is contingent on a number of actions being performed repeatedly until the desired criteria are met. The quick-sort and merge-sort algorithms are more elegant using recursion, whereas the bubble-sort uses a looping mechanism. The student in question may base his choice of algorithm either on the iterative technique that he is more familiar with, or the one that he wishes to improve through practice.

Tools and Resources

The tools at the student's disposal depend on his prior knowledge, and could consist of different programming constructs with which he is familiar. The choice of algorithm is dependent on constraints like the size of the input elements, the system's main memory, and the number and nature of the elements in the input list. Cognisance of the available resources may affect the student's design choice to ensure that the sorting operation is as memory and time-efficient as possible.

Inference and Logic

Heuristics may be developed to reduce the time spent on the problem. The student could take the initiative to learn a new sorting algorithm that may be better suited to his problem scenario. For example, the radix sort can be used to sort records that are keyed by multiple fields. This may be more appropriate for a collection of songs that contain many features such as genre, artist, and album, all of which could be integrated into the sorting criteria.

Evaluations and Improvements

As the student is currently familiar with three possible algorithms, it is necessary for him to select which one to use. It is prudent to consider the relative merits of each algorithm. For example, the selection-sort is slow for a large amount of data, while the merge-sort works well for larger lists. Alternatively, the quick-sort is available in many standard libraries, which may reduce the work required by the student to sort his collection.

3.6 Horizontal Axis of CTF

As a discipline, CT is involved with the development of mental abilities, and the progressive way in which students learn to think computationally. Research on Bloom's Taxonomy of Learning [5] suggests that students learn different kinds of skills in a hierarchical manner [32], as discussed in Section 2.2.1. To reflect the complexity of CT skills we adopt a similar approach, reflecting the different levels at which these skills can be practised. We have defined four nature of engagement levels in the CTF; the lower two levels reflect how CT is engaged to understand a given problem, while the higher two levels reflect how CT is actively used to devise a solution. Although these nature of engagement levels correspond loosely to the categories of the revised Bloom's Taxonomy [32], the order in which they are placed is more applicable to the domain of CT, rather than a broader educational domain. When designing assessments or curricula, cognisance should be given to the level at which the materials are aimed. Should an educator wish to place a greater focus on any of the CT categories, the requirements for those categories may be higher than for the rest. The four nature of engagement levels are discussed below.

3.6.1 Recognise

The *recognise* level correlates most closely with the *remember* level of the revised Bloom's Taxonomy. At the *remember* level, students are expected to recognise and recall knowledge [32]. At this level in the CTF we expect that students should be able to identify aspects of the problem that are computational in nature, but only exhibit a superficial understanding of the concept. The emphasis at this level is on observation.

3.6.2 Understand

The *understand* level in the CTF relates directly to the *understand* level in the revised Bloom's Taxonomy, or the *comprehension* level in the original Bloom's Taxonomy. At this level of the taxonomy, students are expected to interpret, compare, and explain the problem [32]. In the case of the CTF, we expect that students should exhibit an understanding of the computational nature of the problem, and use this insight to frame the way they view the problem.

3.6.3 Apply

At this level in the CTF, students should understand the nature of the problem, and should be actively using CT to find a solution to the given problem. By exercising CT, they should create solutions that are correct, elegant, and appropriate in the context. This level corresponds to the *apply* and *create* categories of the revised Bloom's Taxonomy.

3.6.4 Assimilate

At this final level in the CTF, students should be able to combine lower-level concepts into new, high-level tools and ideas that can be used to solve more complex problems. To do this, students must understand how different concepts complement each other, and must be able to critically analyse and decompose a problem. Although this level is significantly different to the categories of the revised Bloom's Taxonomy, it relates most closely to the *analyse* and *evaluate* levels.

3.6.5 Illustrative Example

To illustrate the progression through these levels, iteration is used as an example.

- At the *Recognise* level, a student is able to observe the fact that some repeated action is being performed.
- At the *Understand* level, the student would have a grasp of why the action is being repeated, and a clearer idea of the confines of the repetition.

- At the *Apply* level, the student is able to leverage the repetition to simplify the problem or work towards a solution. In a programming context, this would likely take the form of an iterative looping structure.
- At the *Assimilate* level, the concept of iteration is fully integrated into the student's understanding and can be used to solve higher-order problems, for example, using a loop within a sorting algorithm.

3.7 Computer Science Concepts in the CT Framework

The elements in this framework have been influenced by CS concepts and practices, and as such, the framework forms a means of classifying CS concepts. This will allow CT activities and CS concepts to be linked in an educational setting. Below, we give an indication of the types of CS concepts that could be categorised according to the framework. These examples provide a sample and do not form a comprehensive list. Items are given in alphabetical order.

- **Processes and Transformations:** decomposition, input, output, parallel processing, pre- and post-conditions, and states;
- **Models and Abstractions:** abstraction, modularisation, problem and solution space, and reformulating problems;
- **Patterns and Algorithms:** boundary cases, classes of problems, conditionals, iteration, and recursion;
- **Tools and Resources:** caching, restrictions and limitations, resource allocation, tools, and trade-offs;
- **Inference and Logic:** approximation, deduction, heuristics, induction, non-determinism, randomisation, requirements, and trial and error;
- **Evaluations and Improvements:** alternatives, debugging, efficiency, errors, generalisation, redundancy, scalability, testing, and verification.

3.8 Application of the CT Framework

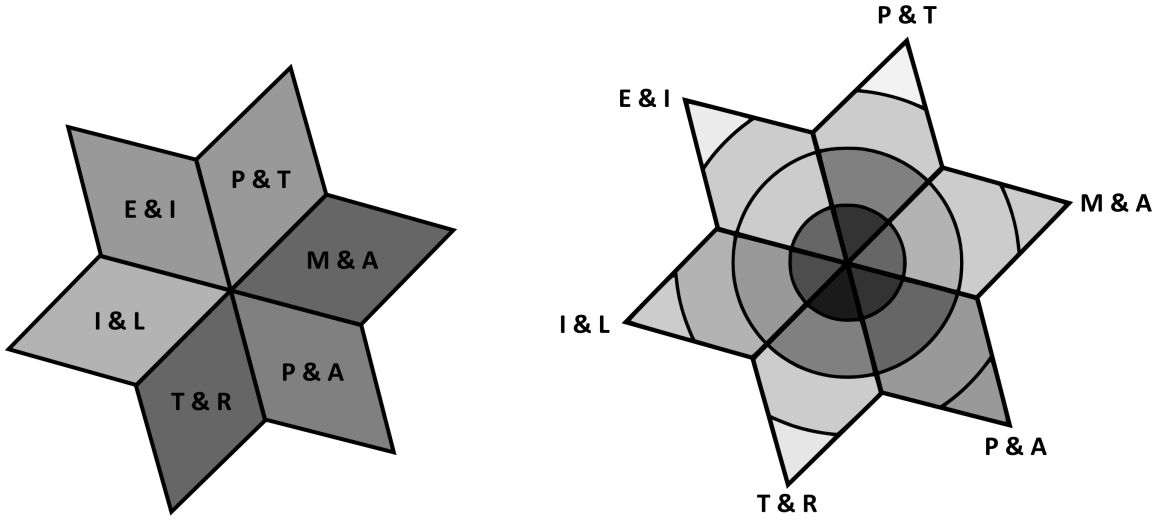
As previously stated, this framework is intended to be suitable for a range of CT exercises and forms the foundation for the work in this research study. In Chapter 4, the framework is used as a basis for creating a CT test to administer to students. In Chapter 5, the framework is applied to provide a CT score and identify areas of possible improvement in a computer game. In Chapters 6 and 7, the framework is used to identify smaller activities and sets of skills that serve to utilise and improve CT abilities.

3.9 Representing Framework Results Visually

A means of graphically representing results from the application of the CTF has been provided. The results are displayed as a six-sided star, with each point representing one of the CT categories. For each category, the degree of shading represents the score for that category, so that darker points represent stronger areas and lighter points represent weaker areas. This visualisation may be used to give each student an indication of their own "CT landscape", or to analyse results from an assessment. If the nature of engagement axis is not invoked, each point can be represented as a solid shaded colour; if the axis is invoked, each point is a staggered set of rings, where the inner ring represents the lowest level and the outer ring represented the highest level on the horizontal axis of the CTF. An example of this representation is given in Figure 3.3.

3.10 CT Framework and the Curriculum

Increasingly, CT is being discussed as a crucial element in the CS curriculum. In this section we examine the relevance of the CTF with reference to three curricula: the Computer Science Teachers Association (CSTA) Model Curriculum for Computer Science [21], the CAS Curriculum for schools [22], and the ACM Ironman Curriculum for undergraduates [45]. The curricula discussed in Section 3.10.1 and 3.10.2 are aimed at the junior and secondary schooling levels, while the research in this thesis is primarily aimed at an introductory university level. However, owing to the relatively privileged IT landscape in South Africa and the lack of learners experience with computers at school, the same concepts that are relevant at secondary schools elsewhere in first world countries are relevant at an introductory university level within the South African context.



(a) Results with the CT categories only.

(b) Results with the CT categories and the nature of engagement levels.

Figure 3.3: Visual representation of the CTF results.

3.10.1 CSTA Model Curriculum for Computer Science (2003)

In the 2003 CSTA Model Curriculum for Computer Science, concepts that would now be known as CT are mentioned. The curriculum discusses *IT fluency* as an advanced form of IT literacy that includes algorithmic thinking and problem solving [21]. IT fluency is decomposed into three orthogonal axes: concepts, capabilities, and skills. In this document, a four-level model curriculum for K-12 education is proposed. One of the overarching goals of this curriculum is to enable students to use algorithmic thinking in problem-solving endeavours within their other studies, echoing the vision of CT as a broadly applicable skill. The four recommended levels are discussed below; information about these levels is taken from [21].

Level 1 (grades K-8): At this level, students should be provided with foundational CS concepts, and receive an introduction to algorithmic thinking through hands on experiences.

Level 2 (grade 9 or 10): This level should provide students with a broad understanding of the principles, methodologies, and application of CS.

Level 3 (grade 10 or 11): This level should continue on from level 2, and provide students with a focus on the mathematical principles, algorithmic problem solving, programming, software and hardware design, networks, and social impact of CS.

Level 4 (grade 11 or 12): The final level should allow students to focus on one particular aspect of CS with the level 3 course as a pre-requisite.

In these levels, we see a progression with similarities to the horizontal axis of the CT framework. Beginning at **level 1**, foundational knowledge is acquired and learning is centred around concrete exercises. Moving to **level 2**, the focus shifts towards understanding, and in **level 3** there is an application of skills and concepts, particularly with regards to mathematical principles and algorithmic thinking. In the final **level 4**, prior knowledge and understanding is assumed and these skills are integrated into deeper, more focused studies. Similarly, the CT framework moves from a knowledge and understanding of concepts in the lower levels, to an application and synthesis in the higher levels.

3.10.2 CAS Computing Curriculum for Schools (2011)

As fully described in Section 2.1.5, the 2011 CAS Curriculum [22] has numerous references to the importance of CT. The curriculum document points particularly to the importance of CT as a unifying theme for the key processes that comprise CS. A key process is something that students should be able to do, while key concepts describe the different topics and themes within CS [22]. Below, we discuss the five key concepts outlined in the curriculum and their relation to and divergence from our CT framework.

Languages, machines, and computation: This concept includes the understanding and use of languages, algorithms, virtual and physical machines, and computational models [22]. In terms of our CT framework, this concept is represented by the *models and abstractions* category, and to a lesser extent by the *tools and resources* category.

Data and representation: This concept includes the representation, organisation, storage, and transmission of data [22]. Although the representation and organisation correspond with the *models and abstractions* category in our CTF, storage and transmission do not fit neatly into any of the CT categories.

Communication and coordination: This concept is about processes, actions, and events in a program, as well as the communication and cooperation of networked computers [22]. While some of the facets of these concepts fall into the *processes and transformations* CT category, the idea of communication is not explicitly represented in the proposed CTF.

Abstraction and design: This concept represents the layers of abstraction and interfacing in software and hardware, as well as simulation and modelling [22]. Within our CTF, this concept maps to the *models and abstractions* category.

Computers and a wider context: This concept encompasses societal themes, such as intelligence and consciousness, the natural world, creativity, privacy, and intellectual property [22]. This concept falls largely outside the scope of the CTF; however, intelligence and creativity could fall within the *inference and logic* CT category.

Although there is some divergence, a number of overlaps have been identified between the key concepts outlined in the curriculum and the CT categories outlined in our CTF. The least represented categories in terms of the concept definitions are *patterns and algorithms* and *evaluations and improvements*. Nevertheless, these categories are likely to form an underlying theme within many of the concepts when put into practice.

3.10.3 ACM Ironman Curriculum (2013)

In the ACM Ironman Curriculum [45], discussed in Section 2.1.5, the Computational Science knowledge area represents the need for CT within the curriculum and the applicability of these problem solving practices within and beyond computing. This is presented as a tier-1 skill, meaning that it should be required as a core subject for every CS curriculum [45]. The curriculum outlines a number of ideas and techniques for this area, which are grouped below according to our CTF.

- **Processes and Transformations**: processing, parallel systems, and software processes;
- **Models and Abstractions**: numerical representation, modelling and simulation, information visualisation, and data representations;
- **Patterns and Algorithms**: parallel algorithms, algorithm design, well-known algorithms, and pattern recognition;
- **Tools and Resources**: program construction, trade-offs (performance, accuracy, validity, and complexity), and computing cost;
- **Inference and Logic**: error analysis, information visualisation, and the broader concept of analysis;

- **Evaluations and Improvements**: error analysis, optimisation, and program testing.

3.11 Summary

CT is a multi-faceted field that needs to be better understood to harness its value for CS students. In this chapter, we presented a framework describing the skills, activities, and ideas in CT. This framework decomposes CT into six themes or categories, based on a literature survey, and four different levels of the nature of engagement to reflect the developing abilities of students. Examples of the relevance of our CTF to common CS concepts were given, and its relationship with the ACM computing curricula was discussed. Potential applications of the framework were identified; the subsequent chapters discuss our use of the framework to devise a student test, assess a computer game, and prescribe problem solving strategies and activities.

Chapter 4

Student Assessment

Chapter 3 presented our CT framework to describe CT skills and practices. One of the envisioned uses of this framework was the development of an assessment tool to measure student ability in CT. In this chapter, we describe our efforts to create a CT assessment that is suitable for introductory level university students, and present the results obtained from the administration of this test to a first year CS class. Numerous attempts at creating an aptitude test for CS have produced varied results [18, 49]; this assessment is not intended as a measure of aptitude, but rather a means to gain detailed insight into the needs and abilities of students.

4.1 Motivation

It is well established that students begin university level CS courses with different levels of experience in computing and programming. As discussed in Section 2.1.1, this discrepancy tends to make students uncomfortable in class and is a contributing factor in students choosing not to major in CS. It is imperative that educators acknowledge and address the differences between students to foster a productive environment for potential CS majors.

One of the primary benefits of CT is that it extends beyond CS and programming, and it is possible to develop CT skills without using a computer. Therefore, although students may arrive at university with no prior programming experience, it is likely that they would have developed some CT skills through mathematics, science, or other life experience. If the CT abilities of students can be quantified, educators would have a better picture

of the inherent abilities of students and an opportunity to assist individual students in addressing their weaker areas.

In light of this discussion, we believed that it would be beneficial to create an assessment aimed at incoming CS students. Assessing students at this level should afford the ability to create a strong foundational knowledge and build confidence that may empower students to continue their studies in CS. Because it is built around our CT framework, this assessment should give a detailed impression of the CT abilities of these students.

4.2 Experimental Method

The overall method for creating and utilising the CT assessment is discussed in this section. Our approach consists of three phases: assessment design, administration of the exercise, and analysis of the results.

Assessment Design: The goals for the assessment were identified before the design commenced. The format selected for the assessment was a test that could be administered in a single session with the students, and which was designed around the CT concepts described in our CT framework. An appropriate source for the test questions was identified. The design of the test is further discussed in Section 4.3.

Administration: The test was integrated into the course work for the introductory CS class and administered twice, once at the beginning of the CS course and again six months later, after the conclusion of the semester. The administration is further discussed in Section 4.4.

Analysis of Results: The results from both test iterations were collated and analysed using Microsoft Excel and R statistical software[1]. The CS course marks and some demographic information of the students were included in the analysis. The analysis of results is further discussed in Section 4.5.

4.2.1 Hypotheses and Queries

This assessment aims to gather information about the existing CT abilities of students and the relevance of this to their CS studies. A set of hypotheses has been developed

[1] `http://www.r-project.org`

to state the expected outcomes of this research. Furthermore, as this research is largely exploratory, a set of queries has been developed to guide the inquisition into the broader results. The hypotheses and queries are listed below.

Hypotheses

- **H1:** Students who selected information technology (IT) or computer application technology (CAT) as a Matric subject will have better CT scores.
- **H2:** There will be a large variance in CT scores across the class.
- **H3:** CT scores will improve after a semester of CS studies.
- **H4:** CT scores may be used as a predictor for success in the students' CS studies.

Queries

- **Q1:** What is the overall CT ability of the class?
- **Q2:** In which CT areas are students, respectively, the strongest and weakest?
- **Q3:** Are the results of students who discontinue CS after one semester distinctive, given their initial CT scores?
- **Q4:** Which CT areas are most affected (i.e., improved) by a semester of CS studies?
- **Q5:** Does gender cause a significant difference in CT scores and CS results?

4.3 Test Paper Design

The CT test was designed in the form of a multiple choice and single answer traditional test, with a number of overall goals and guidelines. This section describes the design of the CT test, including the test format, sourcing of test questions, and use of our CT framework to quantify the CT properties thereof.

4.3.1 Goals and Guidelines

To maximise the benefit derived from the test without compromising the experience of the incoming students, a set of goals and guidelines was developed to outline the approach for the design and application of the test. The goals for the test were as follows:

1. It should provide a picture of the CT ability of individual students as well as the group as a whole.
2. It should provide detailed insight into the specific facets of CT that may be particularly good or troublesome.
3. It should not discriminate against students who have not had the opportunity to study CS and programming previously.
4. It should not discriminate against students who are weaker in mathematics.
5. It should be suitable for students at a first-year university level.
6. It should not intimidate students or serve to reinforce prejudiced ideas about CS.
7. It should use questions with definite answers rather than open-ended questions.
8. It should be suitable for use in both a short term and longitudinal study.
9. It should build on an existing test or platform to validate the integrity and relevance of the test questions.

4.3.2 Test Format

The test is a pen-and-paper test, with spaces provided on the question paper for answers to be recorded. The recommended time frame for the test is 90 min. The test consists of thirty questions: the first five questions are *warm-up* questions, intended to build student confidence and self-efficacy, while the remaining 25 questions are used to determine the CT ability of students. The results for the first five questions were captured but not subsequently used in any data analysis. The scoring of the test was not disclosed to the test participants. This format supports the goal for an assessment that can be administered within a single session, and allows students with no experience with computers to participate equally. The final question set is given in Appendix A.

Table 4.1: Breakdown of test questions relative to each CT category.

CT Category	Number of Related Questions
Processes and Transformations	9
Models and Abstractions	9
Patterns and Algorithms	9
Tools and Resources	6
Inference and Logic	11
Evaluations and Improvements	9

4.3.3 Sourcing Test Questions

A number of possible question sources were considered for this assessment. It was decided to use an existing question bank rather than devising novel questions, as this would lend credibility to the questions and allow us to focus more deeply on the CT properties. Disregarded possible sources include the Standard Aptitude Tests (SATs) used in the United States and the National Benchmark Tests used in South Africa. The source ultimately selected was the National Computer Science Olympiad, particularly the questions from the "Talent Search" round of the competition. Information about the Olympiad is available from the Olympiad website[2]. The Talent Search round of the competition was introduced as an aptitude test, and has the benefit of comprising questions that do not rely on specific knowledge of any programming language or paradigm. This makes the questions suitable for students who may have CT abilities without having concrete experience in programming. Furthermore, the clear CS focus of the competition means that the nature of the questions was suitable for our test purposes. As the Olympiad is intended for high school students, the difficulty level was considered appropriate for incoming university students.

4.3.4 CT Properties of the Test

The test questions were selected based on our CT framework described in Chapter 3. The questions in the 2009-2012 Talent Search test papers were analysed and coded according to the CT skills that could be employed in solving them. From these coded questions, a representative set of questions was selected. The number of test questions related to each of the CT categories is given in Table 4.1.

[2] `http://www.olympiad.org.za/talent-search`

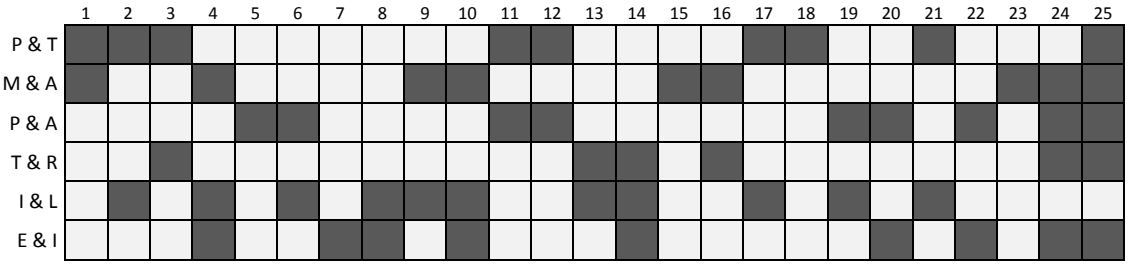


Figure 4.1: Classification of test questions using the CTF. A dark shaded block indicates that the question number given at the top has been classified into the CT category given to the left.

Discussion of Individual Questions

Each question in the test was selected for its CT merits. When classified using our CT framework, each question fell into one or more CT categories. An overview of the classification of all the questions is given in Figure 4.1. A short description of each question and the reasons for its inclusion are given below; the full questions are given in Appendix A. The warm-up questions are labelled A-E and the CT questions 1-25.

A-E. These are the relatively easier warm-up questions intended to put the students at ease while writing the test. No further explanation is given as they were not used in any analysis.

1. A 0.1 mm thick sheet of paper can be folded double ten times; how thick will the folded sheet be?
   P&T: In this question, the paper thickness is transformed through a process of folding; the goal is essentially to find the post-condition of the paper.
   M&A: The notation used to represent doubling a value ten times can be simplified to 2^{10}, making the question quicker to solve.

2. Jason is 5 years old. In three years' time Ramone will be twice as old as Jason. How old is Ramone now?
   P&T: A logical approach to this question is to decompose it into a sequence of steps: (1) find Jason's age in three years; (2) find Ramone's age in three years; and then (3) find Ramone's current age. Each step in the process has a defined outcome that feeds into the next step.
   I&L: Using heuristics, a mathematical formula can be derived to find the solution: Ramone's age $= ((5 + 3) \times 2) - 3$.

3. Arrange four sixes using the standard mathematical operators so the answer is 1.
   P&T: A sequential process is required to reach the solution, with the order of the operations being important.
   T&R: The tools in this question are the mathematical operators that must be placed correctly to reach a solution.

4. LEAD is to DEAL as 9514 is to ... (four options given).
   M&A: The letters in the words are represented as numbers.
   I&L: Heuristics may be used to find the answer more quickly; for example, observing that the outer characters are swapped while the inner ones remain unchanged.
   E&I: A faster approach to this problem may be to rule out options that are obviously incorrect, narrowing down the options that must be deeply considered; this could also be useful for checking the accuracy of the solution.

5. Work out how long it will take a frog to get out from the bottom of a well, given information about the frog's ability to jump.
   P&A: A pattern emerges of "jump-recuperate-slide"; students must trace through this pattern to find the amount of time lapsed when the frog reaches the top. A special condition must be considered: for the final jump the frog will not need the hour to recuperate and slide back.

6. Identify the next number in the sequence: 0, 3, 8, 15, 24, 35, 48, 63.
   P&A: To solve this problem, the student must identify the pattern used to progress to the next number.
   I&L: The difference between numbers in the sequence must be found (3, 5, 7, etc.), from which we can infer that the increment starts at three and increases by two for each iteration.

7. Given five statements, work out which one statement is true.
   E&I: Each of the five options must be considered; the correct solution can be found by ruling out incorrect options or by inverting the initial statement to find the correct option.

8. Which of the following cannot be paid using only 5c and 7c coins? (four options given).
   I&L: Heuristics, such as finding the closest multiple of five or seven, may be used to speed up the process of identifying which options do have solutions.
   E&I: In this question, each option must be evaluated. Option A may be selected fairly quickly, but this answer is best confirmed by solving the other options.

9. What is the smallest number consisting of only 1's and 0's that is divisible by 15?
   M&A: In this question, the possible permutations of numbers represented with 1's and 0's must be considered to find an option that is divisible by 15.
   I&L: Although the most obvious approach may be to start with 15, and continue counting upwards until a number with only 1's and 0's is found, this approach would take 74 iterations to find the correct solution. By starting with the number 10, and adding 1's or 0's to the right, the solution can be found in 11 iterations.

10. Given an encoding where A=1, B=2, C=3 ... K=10, L=11, etc., what common English word has the code 2 1 1 2 1 2?
    M&A: In this question, letters are represented as numbers; some ambiguity is introduced where combinations of numbers can represent multiple letters, which students need to consider to find the solution.
    I&L: Using their knowledge of the English language, students may develop heuristics to decide which letter combinations are most likely.
    E&I: When a number or sequence of numbers can represent different letters, the options must be considered and the correct option selected.

11. If you write down all the numbers from 1 to 100, how many digits have you written down?
    P&T: To solve this question, students may decompose the numbers into single, double, and triple digits, obtain a sub total for each group, and combine them at the end. An approach that does not use a process may be too time-consuming to be viable.
    P&A: The numbers in this sequence form a somewhat repetitive pattern; by noting the way in which numbers increment, students may find a simpler algorithm to reach the solution.

12. A man ate 100 bananas in five days, each day eating six more than the previous day. How many bananas did he eat on the first day?
    P&T: There are a number of approaches to solving this problem. A possible process could be: count the number of additional bananas eaten each day with the formula 6 + 12 + 18 + 24, subtract the result from 100, and then divide the answer by 5.
    P&A: Since this problem is iterative in nature, rather than using a trial and error method to find the solution, students should take a more sophisticated approach, particularly by paying attention to the way that the number of bananas increases each day.

13. Given a set of mass pieces in pounds and ounces, work out the largest mass that can be measured in ounces, and which mass pieces could be used to measure 11 oz.
T&R: The mass pieces in this question form tools that must be used in the correct configuration to reach the solution, using basic arithmetic. The solution is limited by the number and size of mass pieces available.
I&L: Logic must be used to find the best combination of mass pieces for part (b).

14. Use the mathematical symbols +, -, ×, ÷ once each to resolve a given equation.
T&R: Each mathematical symbol is a tool that can be used to reach the solution; the correct configuration must be chosen.
I&L: This question requires a good knowledge of how mathematical operators work; a potential approach is to solve part of the problem, and then determine whether the remaining operators can be used to solve the rest of the problem.
E&I: There are a number of approaches that can be taken to solve this problem; students must choose the best approach to solve the problem.

15. Work out how many rectangles there are in a given figure.
M&A: This question is represented visually and students must find a way of logically decomposing the shape to count all of the rectangles without double-counting any part.

16. Given a figure with eight squares, fill in the numbers 1-8 so that no squares with consecutive numbers touch each other.
M&A: This question highlights the importance of organizing data according to criteria. To solve this problem, students must think carefully about how to place numbers; if they realise that two and seven fall in the middle of the numbers that need placing, it makes sense to put them in the outer cells to leave more options available for placing the adjacent numbers.
T&R: In this question, there are limited resources in terms of the blocks available for inserting numbers into, and restrictions on how the resources may be used.

17. If a girl has an equal number of brothers and sisters, but each brother has twice as many sisters as brothers, work out how many daughters and sons are in the family.
P&T: This question can be decomposed into several steps: (1) find the ratio of boys to girls, (2) find the number of one gender, and then (3) deduce the number of the other gender.
I&L: Although this question can be answered using a guess-and-check method, a better option is to represent the number of girls and boys mathematically as a series of equations, and then solve the equations using simple algebra to find the answer.

18. Given an example of modular arithmetic, work out two values using the modulus operator.
P&T: This question has been decomposed for the student into a simple process: (1) divide x by y, (2) multiply the answer by y, (3) subtract the result of (2) from x, and the final result will give the remainder.

19. Identify the next item in the sequence: Z, Z, Y, Z, Y, X, Z, Y, X, W, Z, Y, X, W.
P&A: Repetitive sequences in the series must be found to deduce what the next letter should be.
I&L: The pattern is easier to spot if it is read backwards; it soon becomes clear that each chunk of the pattern represents the last few letters in the alphabet with a new letter added each time.

20. Given a set of squares with a letter filled in to each quarter, work out which square would complete the series (three options given).
P&A: To find the correct solution, the pattern in the series must be discovered: each square is the previous combination with the letters rotated clockwise.
E&I: There are three options that appear viable at a first glance. Two different criteria are needed to rule out the incorrect options: (1) the letters must appear in the correct order, and (2) the letters must be rotated into the correct position.

21. A volcano doubles in height every day. After 30 days, it is as high as Table Mountain. How many days does it take to grow to half the height of Table Mountain?
P&T: In this question the volcano transforms by growing every day; as the starting value is not given, it should be deduced that the only way to find the solution is to work backwards from the end value.
I&L: It should be logical that if the mountain doubles every day, it would take one day to go from half the height to the same height. Therefore, it was half the height on the second last day, which was the 29th day.

22. Given a map with a series of named nodes and weighted edges, find the quickest route between two given points.
P&A: This question can be solved using any one of the shortest-path algorithms.
E&I: There are a number of options for the ambulance to take. Some options are misleading, for example, G—X is only one step, but G—H—X takes less time.

23: Given a grid of symbols that represent numbers, work out the total for a given column in the grid.

M&A: This question contains numbers coded as pictures of fruit; to find the solution the values of the fruits must be decoded, after which the values in column one can be summed to find the solution.

24. Given a simple instruction set, write the commands to make a robot traverse a maze to a given point and back.
M&A: This question uses the model of a programmable robot, with a collection of possible instructions and a finite board on which it can move.
P&A: The solution to this problem can be seen as a "collect treasure" algorithm; students may either design this algorithm from scratch, or adapt the algorithm given to collect treasure X.
T&R: The tools in this question are the instructions that can be given to the robot and which must be selected and placed into the correct order.
E&I: There are different pathways through the maze; the shortest route to the treasure must be identified and programmed.

25. Write the instructions to make a robot draw a square and a pentagon, using only a forward command and a turn command.
P&T: In this question, the main chunk of work is drawing the required shape. The pre- and post-conditions must be considered to ensure that the robot draws the shape in the correct position, and that the final direction in which the robot is facing is correct.
M&A: The model for this question is a programmable robot with two instructions, each of which must be expressed using the given notation.
P&A: To simplify the solutions the repetitive sets of instructions given to the robot must be identified and placed within brackets.
T&R: The tool in this question is the robot that may be used in two ways, either to draw a line or to turn.
E&I: There are several ways of expressing the instructions to draw the required shapes; students may begin by writing out the instructions in full, but must evaluate and simplify the solution as much as possible.

4.4 Test Administration

The CT student assessment was administered to an introductory CS class in 2013. The testing took place at two different times: the initial test was conducted at the beginning

of the course to assess raw student ability, and a follow-up test was administered after the conclusion of the course to measure the differences in scores after one semester of CS instruction. The same question paper was used for both tests, although it was slightly adapted for the second testing, as described below. For both tests the venue was a computer laboratory on campus, although computer use was not allowed during the test.

4.4.1 Participants

The participants in this exercise were the students in the Computer Science 101 (CS 101) course at Rhodes University. CS 101 is the entry-level course for students majoring in CS, which runs in the first semester of the academic year. The modules in CS 101 for 2013 were programming (5 weeks), problem solving (4.5 weeks), computational thinking (2 weeks), and social issues (1 week). Participation in the test was mandatory.

The demographic composition of the students who took both CT tests is given below:

- **Degree**[3]**:** Bachelor of Arts (BA): 7, Bachelor of Science (BSc): 35, Bachelor of Science Foundation course (BScF): 10, Bachelor of Science Information Science (BScS): 9, Other: 9.
- **Gender:** Male: 55, Female: 15.
- **IT Related Matric Subject:** IT: 23, CAT: 8, Neither: 39.
- **Mathematics Related Matric Subject:** Yes: 69, No: 1.

4.4.2 Phase 1

The initial round of the test took place in February 2013, on the day of the first CS 101 practical. The test was integrated into the practical session as an introductory CS exercise. Of the 108 students registered for the CS 101 course at the time of the test, 104 students took the test. All the test question papers were collected at the conclusion of the test and answers were not discussed with the students to preserve the integrity of the follow-up test.

[3]The BScF is taken by students who have the ability to undertake tertiary education but whose schooling or other experiences have left them not adequately prepared for university. The BScS is a 3 year degree intended for students who wish to become computer specialists in a commercial environment.

4.4.3 Phase 2

The second test took place in August 2013 during a practical session for the CS 102 course. This test was presented as a follow-up CS exercise. A total of 75 students completed the second test. The question paper was amended to exclude the warm-up questions as they were no longer deemed necessary. The time limit was adjusted accordingly to 75 minutes, maintaining a consistent time ratio of three minutes per question.

4.5 Findings and Analysis

In this section, the results from the two phases of the CT test are discussed. The data set used in obtaining these results consists only of the students who completed both CT tests as well as the June exam for CS 101. This amounted to a total of 70 students, equalling 74% of the recorded registrations.

Our results included the following data for the 70 student participants:

- The scores for CT test one and two;
- The results for the four class tests in the CS 101 course;
- The results for the CS 101 June exam;
- The demographic data for the students.

4.5.1 Overall Computational Thinking Results

The average, minimum, and maximum results are given in Table 4.2. The average CT score for the first round of the test is 50.3%, which is lower than both the class test average and June exam average. The variance between the highest and lowest score is 84% for the first test, indicating a large disparity between the students, and 72% for the second test. There is also a substantial difference of 20% between the lowest score for the first test and that for the second test. The CT scores for both tests are considerably lower than the results for the class mark and June exam.

CT results according to the six categories in our CT framework were obtained for both tests. Figure 4.2 shows the CT results for the first phase of testing. In this phase,

the strongest CT category is *evaluations and improvements* with a score of 63%, and the weakest categories are *processes and transformations* with 42.9% and *models and abstractions* with 44.4%. The results for the second phase of testing are given in Figure 4.3. Again, the strongest CT category for this phase is the *evaluations and improvements* category with 77%. The weakest category is *models and abstractions* with 56%, followed by *processes and transformations* with 58.6%.

Table 4.2: Overall results for the CT tests and CS course work.

	Average	**Minimum**	**Maximum**
CT Test 1	50.3%	4.0%	88.0%
CT Test 2	62.6%	24.0%	96.0%
Class Test Average	69.5%	42.5%	97.5%
June Exam	75.0%	43.7%	98.7%

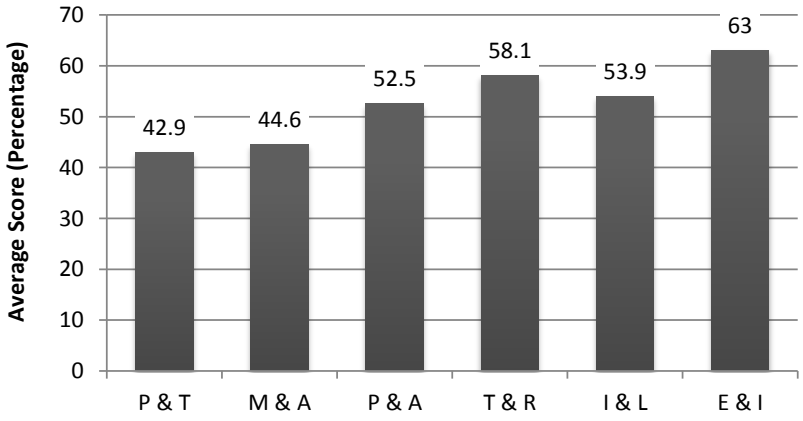


Figure 4.2: Detailed CT results for phase 1.

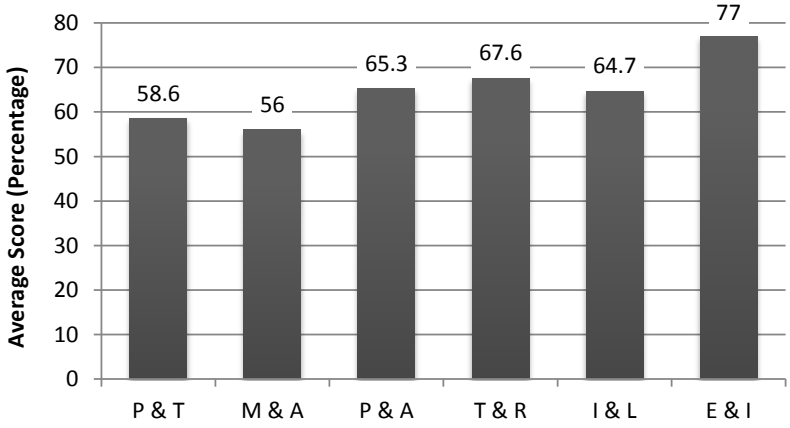


Figure 4.3: Detailed CT results for phase 2.

Ranked Results

There are substantial differences in the scores obtained by students for the CT test. To obtain a better view of the relative results of students, the data has been broken down into three classifications, top third (28 students), middle third (27 students), and bottom third (28 students). These rankings are assigned based on the overall CT score, which is the total score for the CT test, incorporating all of the CT categories. The CT rankings have been correlated with the number of class tests passed and failed by the students in each group to gain an initial view of the relationship between their CT performance and class test performance. The average pass rate for the class was 81%.

In Figure 4.4, the number of passes and fails are shown as ranked by the first CT test. The pass rate for the top CT performers is 11% higher than the average; conversely, the pass rate for the bottom performers is 9% below average. The ranked results based on the second CT test are shown in Figure 4.5. For phase two, the pass rate of the top CT performers is 14% higher than the average, while the rate of the bottom performers remains constant at 9% below average.

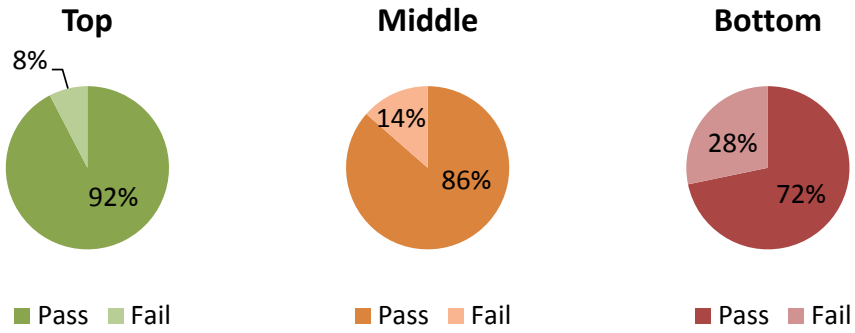


Figure 4.4: Student pass rate for the CS 101 tests, ranked by the results of CT test 1.

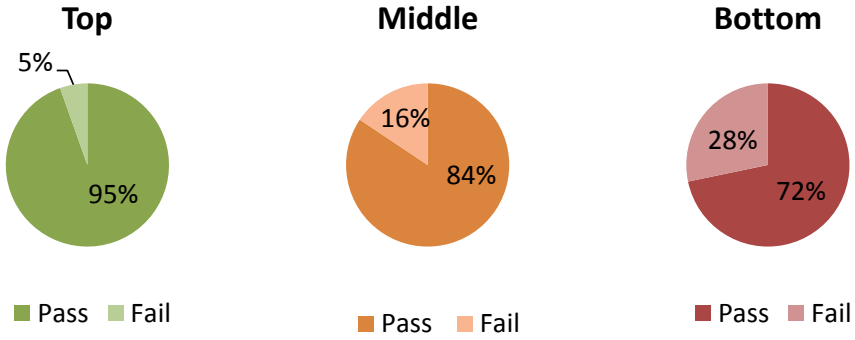


Figure 4.5: Student pass rate for the CS 101 tests, ranked by the results of CT test 2.

CT Results and Discontinued Students

Figure 4.6 gives a comparison of the CT test results for phases one and two, contrasted with the students who discontinued their CS studies after the CS 101 semester-long course. The discontinued students are not included in the results for either phase one or two. As the green line in the graph illustrates, the results for the discontinued students are lower overall than the results for the students who remained in the course. Interestingly, the shape of the graph for the discontinued and remaining students is similar, indicating a consistency in the distribution of skills within the CT categories. The largest variation between phase one (the blue line) and phase two (the red line) is in the *processes and transformations* category, and the smallest variation is in the *tools and resources* category.

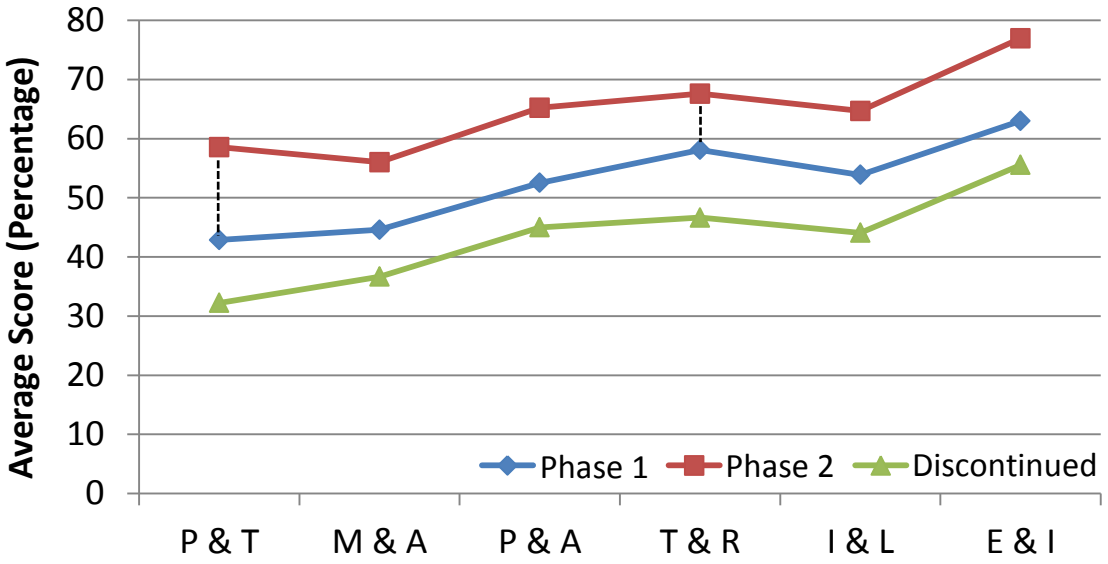


Figure 4.6: Overall CT results for both phases of the test, and for students who discontinued CS after the CS 101 course.

4.5.2 Dependency Results for the CT Tests and Course Work

As part of this investigation, we intended to discover whether there is a relationship between the CT score and class test scores. The results for the four class tests, the test average, and the June exam were correlated with the CT scores for each phase. Pearson's chi-square test was used to determine the dependency levels between these items. The resulting p-scores for phases one and two are shown in Tables 4.3 and 4.4, respectively. Areas of significance are annotated with asterisks.

Table 4.3: *P*-values for the relationship between class test performance and CT test 1. * Significant at the 0.05 confidence level. ** Significant at the 0.01 confidence level. *** Significant at the 0.001 confidence level.

	CS 101 Test 1	**CS 101 Test 2**	**CS 101 Test 3**	**CS 101 Test 4**	**Test Average**	**June Exam**
CT Score	0.235	0.0635	0.665	0.87	0.101	0.217
P&T	0.031 *	< 0.001 ***	0.017 *	0.297	0.687	0.333
M&A	0.002 **	0.073	0.1	0.782	0.267	0.285
P&A	0.594	0.143	0.004 **	0.421	0.205	0.315
T&R	0.06	< 0.001 ***	0.235	0.054	0.463	0.438
I&L	0.192	0.057	0.646	0.312	0.497	0.35
E&I	0.21	0.088	0.471	0.116	0.395	0.291

Table 4.4: *P*-values for the relationship between class test performance and CT test 2. * Significant at the 0.05 confidence level. ** Significant at the 0.01 confidence level. *** Significant at the 0.001 confidence level.

	CS 101 Test 1	**CS 101 Test 2**	**CS 101 Test 3**	**CS 101 Test 4**	**Test Average**	**June Exam**
CT Score	0.159	0.01 *	0.046 *	0.701	0.107	0.2135
P&T	< 0.001 ***	0.072	0.696	0.002 **	0.42	0.3094
M&A	0.004 **	< 0.001 ***	0.875	0.144	0.215	0.2707
P&A	0.181	0.265	0.028 *	0.312	0.327	0.7274
T&R	0.06	< 0.001 ***	0.235	0.0544	0.463	0.4384
I&L	0.571	0.122	0.013 *	0.258	0.16	0.227
E&I	0.002 **	< 0.001 ***	0.861	0.1	0.457	0.3398

These results indicate no significant relationships between the CT scores and the test average or June exam, but some varied relationships between the CT scores and the scores for the individual class tests. Class test two displays the most significant relationships with the CT tests, while class test four has the least relationships. The second test phase was found to have more significant relationships to the course work than the first phase.

4.5.3 Demographic Results

Although it may have proved interesting to consider the results based on whether students took mathematics at a high school level, the population of students who did not take mathematics was too small for a meaningful statistical analysis. The results for students who took IT, CAT, or neither, are given in Figure 4.7. As expected, students who took

a computer subject at school performed consistently better in their CT tests, as well as in their June exams. The results for students based on gender are given in Figure 4.8, reflecting 55 male and 15 female students. The results are very closely matched for both genders, with the male students performing slightly better in CT test 1, and the female students performing slightly better in CT test 2 and the June exam. However, as discussed later in this section, these differences are not significant.

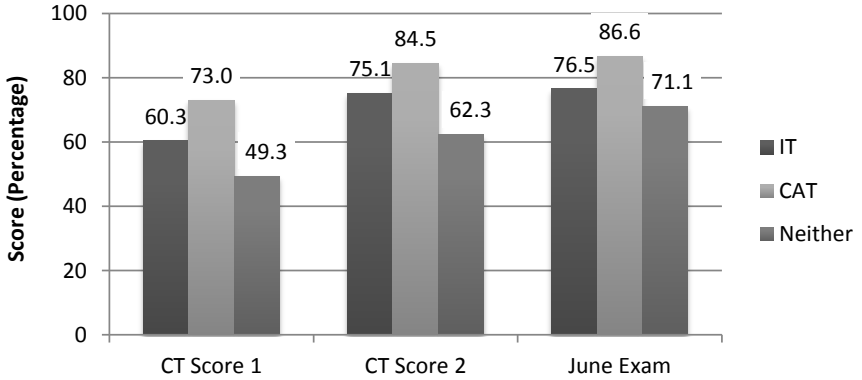


Figure 4.7: Scores for students according to choice of computer subjects in Matric.

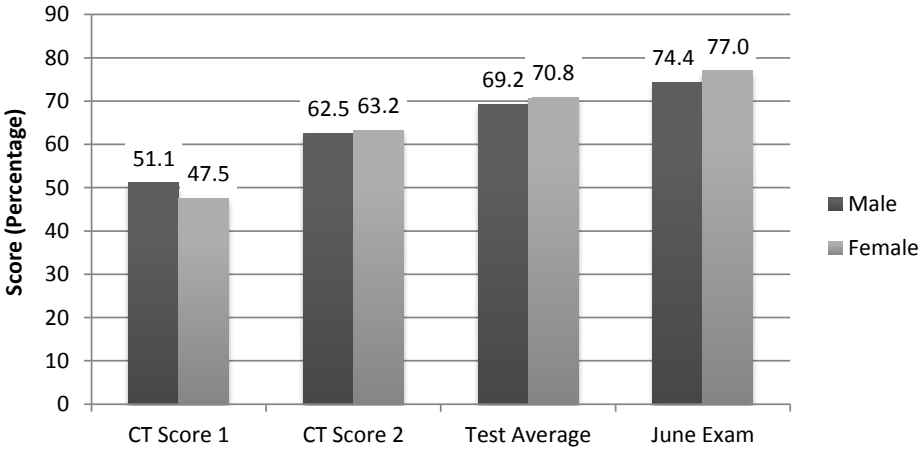


Figure 4.8: Scores for students according to gender.

4.5.4 Hypotheses and Query Results

H1: This hypothesis stated that students who selected a computer related subject in Matric should have better CT scores. The results in Figure 4.7 indicate that those students who took either IT or CAT performed consistently better in both CT tests as well as the June exam. ANOVA procedures were performed to test for computing subject choice

effects on test scores for CT test 1 and 2. No significance was found based on computing subject choice (ANOVA: *CT test 1*: F = 0.686, df = 2, 67, $p = 0.507$; *CT test 2*: F= 0.874, df = 2, 66, $p = 0.422$).

H2: This hypothesis expected there to be a variance in CT scores across the class. Using the numbers from Table 4.2, there is a variance of 84% between the minimum and maximum scores for CT test 1, and a variance of 72% in the scores for the second CT test, supporting the hypothesis.

H3: The CT scores were expected to improve after a semester of CS studies. The average CT score increased by 12.3% between the two tests, and the results for all of the CT categories were consistently higher for test 2 than for test 1. Part of this difference may be attributed to the fact that students had previously seen the question paper, although attempts were made to mitigate the effects of this, as described in Section 4.4.2. The differences between the two tests in the six CT categories are illustrated in Figure 4.6.

H4: Students who performed strongly in their initial CT scores were expected to perform better in their CS studies. When grouped according to their CT results for test 1, the top third of students had a test pass rate 11% higher than the class average, and when grouped according to their results for CT test 2, the students had a test pass rate 14% higher than the average, as shown in Figures 4.4 and 4.5. Regression analysis was performed using the CT test 1 percentage score as a predictor for both the CS 101 class test average and June exam mark. The regression analysis rejects both null hypotheses on the coefficients of the intercept and predictor (CS 101 test average: $R^2 = 0.392$, Intercept: $p < 0.001$, Predictor: $p < 0.001$, June exam: $R^2 = 0.339$, Intercept: $p < 0.001$, Predictor: $p < 0.001$), which shows that CT test scores can be used as a predictor for the CS 101 test average and June exam.

Q1: The first query was intended to give an overall indication of the CT ability of the class. The average CT score given for test 1 was 50.3%, which increased to 62.6% for test 2. There was a failure rate of 50% for the first test, which dropped to 31% for the second test. The scores for test 1 indicate that the CT ability of students is not at a desirable level when they enter university; however, the scores for test 2 indicate that these abilities may improve, which is encouraging. Further tests are needed to ensure that the improved test 2 scores were not as a result of using the same test questions as in test 1.

Q2: Given the detailed breakdown of CT abilities, this query asked in which area students are the strongest. For both tests, the strongest area was *evaluation and improvements*, while the weaker areas were *processes and transformations* and *models and abstractions*.

This difficulty with abstract thinking is of particular concern; if further tests yield similar results, abstract thinking would be an important focus for future research.

Q3: The next query asked whether there was any significance in the CT scores of the students who discontinued the CS 101 course. The scores for discontinued students were consistently lower than for the rest of the students in test 1, as shown in Figure 4.6. The average CT score for the discontinued students was only 42.2%. Once again, it appears that our CT scores might be useful for early identification of at-risk students in the course, although there may have been other factors (not related to their marks) which led to students dropping the course.

Q4: This query asked which CT categories would be affected the most by a semester of CS studies. In Figure 4.6, the greatest variation in scores is in the *processes and transformations* category, indicating the biggest improvement in this area. The smallest variation is in the *tools and resources* category.

Q5: The final query asked whether either gender group performed better in the CT tests, which is particularly relevant given the low numbers of female students in CS. However, the results display very little variation between the genders, with the male students performing better in test 1, and the female students performing better in test 2, as shown in Figure 4.8. This is consistent with the class test results and June exam results, which also showed little variation between the groups. Independent student t-tests were performed to test for gender effects on mean CT test scores, mean CS 101 class test scores, and mean June exam marks. No significance was found based on gender (*CT test 1*: Male: 51.1, Female: 47.5, $p = 0.557$; *CT test 2*: Male: 62.5, Female: 63.2, $p = 0.899$; *CS 101 test average*: Male: 69.2, Female: 70.9, $p = 0.734$; *June exam*: Male: 74.4, Female: 76.9, $p = 0.559$).

4.6 Examples of Student Responses

As we have seen, the results from the test provide us with an initial perspective on student abilities in CT. The results of the CT test give us a general picture of the abilities of each student, as well as an overall view of the whole class of students. These results allow us to draw broad conclusions on the skills and abilities employed across the class. However, by examining specific answers to questions, particularly when students showed their workings, we can make inferences about the techniques that appear to have been used, and gain further insight into students' thought processes. Below, we provide some

examples of student responses to questions in the test, and provide some observations of the CT methods that appear to have been utilised or found to be missing.

Sample 1: Round 1, Question 4
LEAD is to DEAL as 9514 is to ..... a) 9514 b) 9451 c) 4519 d) 4159

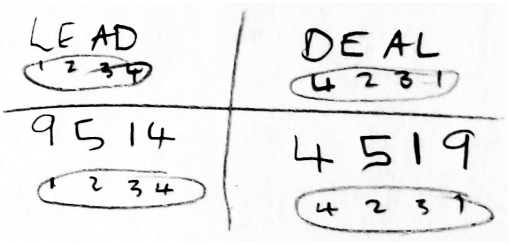


Figure 4.9: Student workings for sample question 1; the correct answer is 4519.

The question described in Sample 1 is classified under the following categories: *models and abstractions*, *inference and logic*, and *valuations and improvements*. In this question, students must look at a numeric representation of the word LEAD, and use this to find the representation of the word DEAL, which is an anagram of the original word.

The sample answer shows the process that the student used to correctly solve the question. There is evidence that the student engaged with the representation of the question by developing their own system of codifying symbols. By developing their own notation and using this to 'decode' the initial representation of the word, the student has devised an efficient way of reaching the correct answer. Their notation allows them to cross-check or evaluate the final answer by ensuring that the *1-2-3-4* values correlate correctly.

Sample 2: Round 1, Question 14
Use the mathematical symbols $+, -, \times, \div$ once each to resolve the equation. Write the symbols in the correct order: $(8 \_ 3 \_ 4) \_ 5 \_ 1 = 5$

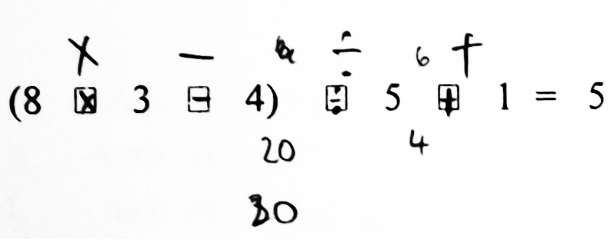


Figure 4.10: Correct student response to sample question 2.

The question described in Sample 2 is classified under the following categories: *tools and resources*, *inference and logic*, and *evaluations and improvements.* In this question, students are expected to insert mathematical operators into an equation in order to correctly balance the equation.

From the response, we can see that the student has attempted to decompose the parts of the equation by writing the in-between values that are obtained when each part of the equation is executed. The different values written above and below the equation seem to indicate that the student has considered different alternatives and selected the best solution for the question. The student has correctly adhered to the constraints of the question by using each symbol once only.

Sample 3: Round 1, Question 5
A frog is at the bottom of a well which is 19 m deep. The frog jumps 4 m up the side of the well, but then needs an hour to recuperate. During this hour the frog slides back 2 m. How many hours will it take the frog to get out of the well?

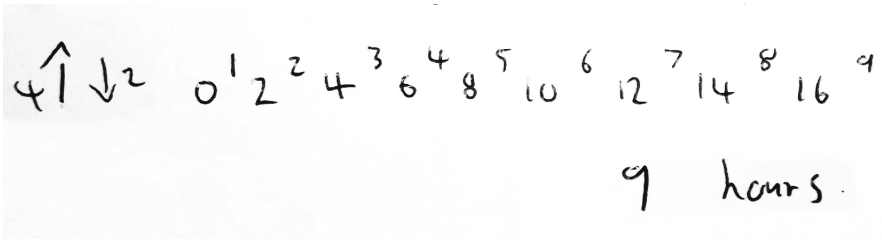


Figure 4.11: Incorrect student response to sample question 3; the correct answer is 8 hours.

The question described in Sample 3 is classified under the following category: *patterns and algorithms.* In this question, students are expected to extrapolate the pattern of jumping up by 4 m and sliding backwards by 2 m to identify the duration of time that it takes to progress forward by 19 m.

In this example, we see evidence of a pattern-based approach in the way that the student has laid out their response to the question. As indicated by the drawings to the left of the solution, the student has simplified the problem by deducing that within each hour, the frog has an effective progress of 2 m. However, by overlooking the boundary case, the student reaches an answer that is off by one: after eight hours the frog jumps up by 20 m, and since this jump reaches the top of the well, the final hour of recuperation is not

needed, meaning that the final answer is eight hours rather than nine hours. Although the general approach indicates a logical thought process, the student is unsuccessful owing to overlooking an important CT concept, namely, boundary cases.

Sample 4: Round 1, Question 10
A message can be sent as a string of digits using the standard alphanumeric code where A=1, B=2, C=3, etc. Thus 121 could be ABA or LA or AU. What common English word has the code 2 1 1 2 1 2?

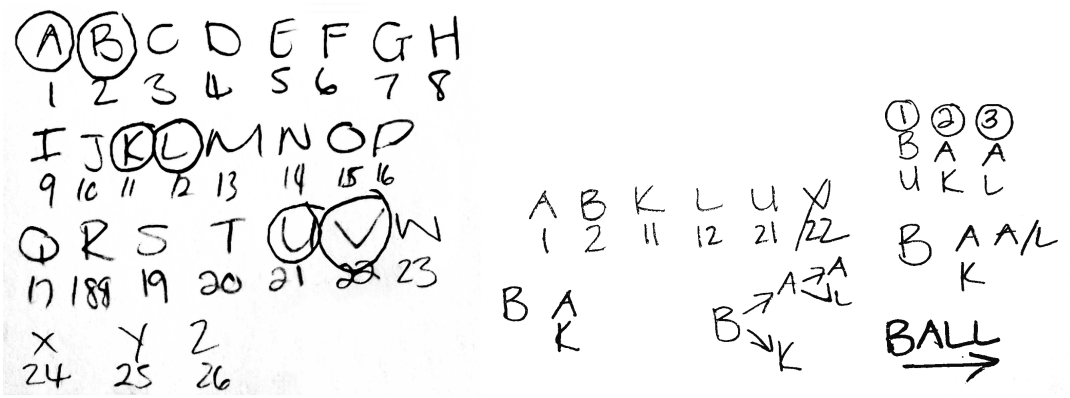


Figure 4.12: Correct student response to sample question 4.

The question described in Sample 4 is classified under the following categories: *models and abstractions*, *inference and logic*, and *evaluations and improvements*. In this question, students are expected to use a representation of the alphabet to decipher a numeric code, which could theoretically have multiple answers, but only decodes into one correct English word.

We can observe multiple aspects of CT in the solution to this question. Firstly, the student has explicitly represented the alphabet in the specified notation, and using a heuristic of circling all of the possible letters that may be in the encoded word, has simplified the problem. Below this, we can see the use of a diagram to evaluate the possible solutions to the problem. The student has taken an efficient approach by selecting a suitable starting point, the letter B, and only pursuing the options that are likely to lead to a solution.

4.7 Summary

In this chapter, the creation and use of a CT test for incoming university CS students was discussed. The development of the test paper using our CT framework was described,

including the motivation for using the particular format and the set of questions. The test was administered twice to a class of incoming CS students, the results of which were also discussed.

A number of particularly important findings were highlighted. The overall CT ability of students was found to be lacking, particularly in the area of *models and abstractions*. No statistically significant relationship was found between gender or selection of computing subjects in Matric and CT scores, although students who took a computer related subject obtained better CT scores that those who did not. Very little variation was found in the scores between the different gender groups. Regression analysis showed that scores for the first CT test can be used as a predictor for the CS class test and June exam results. These results give us an initial indication of student ability and the usefulness of testing CT ability; however, further research on a longitudinal level is needed to validate the specific findings.

Chapter 5

A Computational Thinking Game

The goal for this part of the research was to create a software item that can be used to practice and facilitate CT. In this chapter, we discuss our attempts to develop a game for these purposes. It begins with an introduction and evaluation of an existing computer game, Light-Bot[1]. We then discuss improvements to be made to the game, and our implementation thereof. Finally, the results of a user assessment of our enhanced game are presented, and future extensions discussed.

5.1 Development Process

The process of providing a CT game has been accomplished through a number of phases. The first phase involved identifying an existing game that appeared to have merit as a CT educational artefact, and verifying this value through the application of our CT framework. The next phase involved identifying possible improvements to this game, and then implementing an adapted version with CT principles in mind. Following this, the new game is evaluated by means of a user assessment carried out by students in a computer skills class. Finally, the user feedback and possible future extensions are explored.

[1] `http://armorgames.com/play/2205`

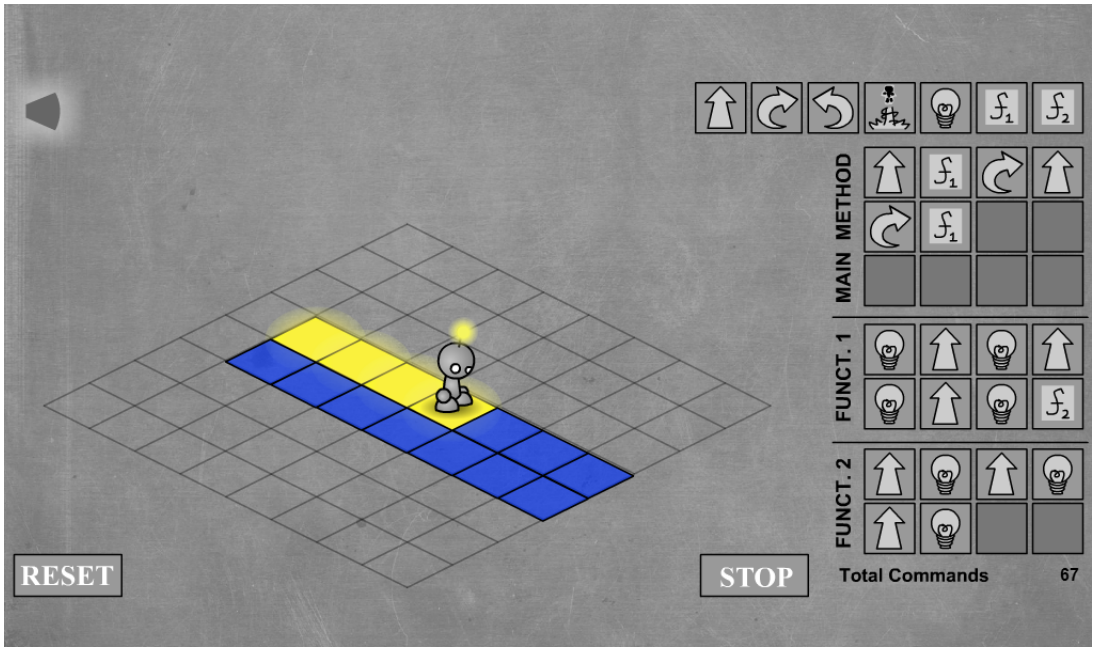


Figure 5.1: Light-Bot in progress on level 7.

5.2 Light-Bot

Light-Bot was selected as the original game on which an assessment should be done; other options considered were games like Jahoomas's Logic Box[2] or Sokoban[3]. The reasons for selecting Light-Bot are discussed below.

5.2.1 Basic Overview of Light-Bot

Light-Bot is an educational Flash game developed by *Armor Games*[4]. The objective of the game is to program a small robot to light up all the blue blocks on the board. This objective is achieved by giving the robot a series of instructions from a limited set of commands, within a finite instruction space. As the levels progress, the board becomes increasingly complex, and more sophisticated combinations of commands are needed to achieve the goal for the level. A screen capture of Light-Bot is shown in Figure 5.1.

When we considered Light-Bot, with its simplified programming environment, it seemed fairly transparent that interaction with the game would benefit novice CS students. The

[2] `http://www.kongregate.com/games/jahooma/jahoomas-logicbox`
[3] `http://www.game-sokoban.com`
[4] `http://armorgames.com`

structure and features of Light-Bot lend themselves to easy correlation with a programming language, providing educators with a lead-in to discuss topics such as objects, functions, and debugging. Light-Bot and its successor, Light-Bot 2.0, have been effectively used in some introductory CS courses [68, 74]. A quantified description of the educational benefits of Light-Bot would be of value to educators, allowing them to highlight particular concepts with analogies from Light-Bot, as well as discussing conceptual areas that Light-Bot fails to address. Here, we apply the proposed CTF (see Chapter 3) to obtain a quantified evaluation of the CT merits of Light-Bot.

5.2.2 CT Evaluation of Light-Bot

The following approach was taken to obtain a CT score (CTS) for Light-Bot. The researchers studied the game of Light-Bot, assessing all the levels to discover the skills required to solve each level. The overall aims and representation of the game were studied to observe the general trends that emerged. Using the CTF, a score was assigned for every block in the assessment matrix.

The scoring of the game was done using the following four-point Likert scale[5] with the values: *0 – Poor, 1 – Fair, 2 – Good, 3 – Excellent.*

The scores indicate the extent to which concepts are perceived as being present in the game. A score of zero indicates that the concept is completely absent, while scores between one and three indicate how integral the concept is to the game play. Once scoring was completed, the sub-scores for each of the six skill sets (CT categories) were aggregated and converted to a percentage. The percentages for all of the skill sets were then combined to form an overall CT score, represented as a single percentage.

The results of the Light-Bot evaluation are illustrated in Figure 5.2. This graph shows the overall scores for the different skill sets in the framework, represented as percentages. The results indicate that Light-Bot performs strongly overall, with strengths highlighted in particular areas. The final CTS for Light-Bot is 74%. Below, we discuss the observations that emerged from the analysis of Light-Bot, with reference to the scores it attained in the evaluation.

[5]The authors acknowledge that the Likert scale is by its nature a subjective system used to evaluate an individual's response to a situation or activity. Although attempts have been made to remain as objective as possible, it is acknowledged that different participants may approach the game with varying strategies.

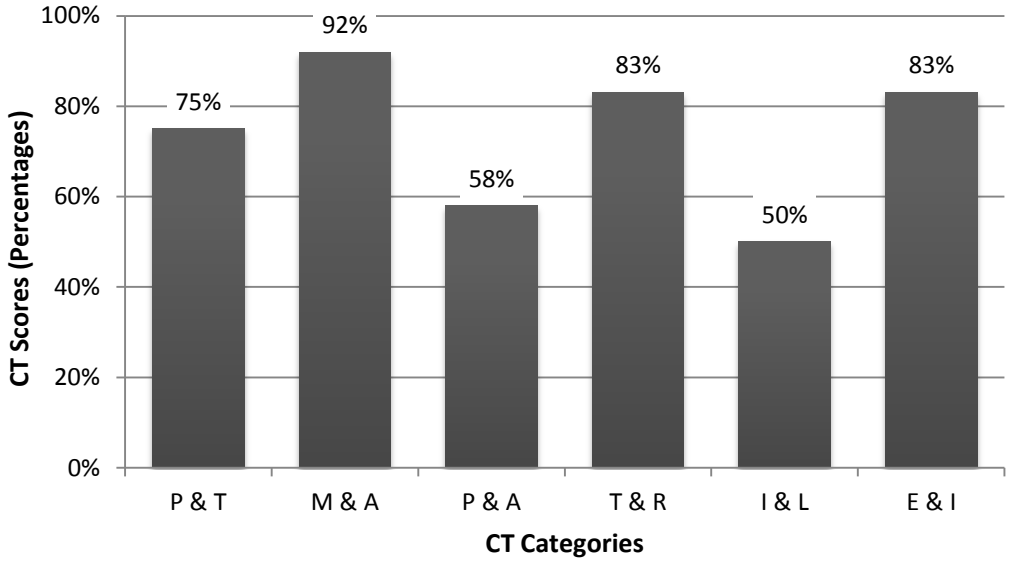


Figure 5.2: Results of the CT assessment of the computer game Light-Bot using the six categories in our CTF.

Processes and Transformations: 75%

The goal in Light-Bot is to transform blue blocks into illuminated yellow blocks. To achieve this transformation, the student must devise a plan to navigate the robot to each of the blue blocks, where a light-up command must be executed. When solving a level, the board can be decomposed into different sections that need solving, and the commands needed for each section can then be determined and arranged into a logical order. To reinforce the decomposition, functions may be used to group a sequence of frequently applied instructions. The game incorporates an aspect of working with states: at any one time it can be in the *design*, *running*, *stopped*, or *solved* state.

From these observations, it is clear that Light-Bot reinforces the concepts of processing and transformations. However, it lacks any form of data use and manipulation, as well as input and output.

Models and Abstractions: 92%

The model of Light-Bot, a visual representation of programming, makes it attractive. Students must understand how the model works and how to interact with it; for example, if the block in front of the robot is raised it must execute a *jump* command. However, they do not have any control over adapting the model or editing levels in the game.

In terms of abstraction, Light-Bot incorporates some abstraction through the use of functions. If a function is designed to carry out a particular activity, students need to think about the pre- and post-conditions, as well as which actions to include and which to exclude to make the function as general as possible.

Light-Bot could be used to teach students about notation, possibly by introducing a notation to represent solutions for the levels, but this is not an integrated game feature.

Patterns and Algorithms: 58%

The solution for each level of Light-Bot can be regarded as an algorithm solving that level. Each algorithm should have a specific intention. For instance, an algorithm could be designed to illuminate all the blue blocks in a checkered pattern, although this degree of intent is not apparent for all levels. Several levels of Light-Bot have repetitive sections; for example, level nine consists of the four sides of a square. In Light-Bot, repetition can be leveraged by placing repeated actions in functions, which can then be called repeatedly. However, the programming model for controlling the robot is not Turing-complete, meaning that it is limited in the computation that it can simulate. As a first step towards a more powerful programming model, it could benefit from a discrete mechanism for definite and conditional iteration. Although functions can call themselves, there are no conditionals, so it is not possible to provide a base case for a recursive call. Any recursive call is thus effectively a non-terminating loop.

Although Light-Bot has great potential for reinforcing the use of patterns and algorithms, we feel that in practice it fails to address these on a meaningful level. This could be remedied with thoughtfully designed levels, as well as the addition of new tools to achieve Turing-completeness.

Tools and Resources: 83%

The game has a defined set of tools in the form of the commands that the robot can execute. It could potentially benefit from an extension to allow students to create their own tools, or to combine tools to form new tools. This is somewhat possible using functions, but could be extended by allowing the student to name, save, and reuse functions.

The finite instruction space is a consumable resource, which forms an integral part of the game, forcing students to re-factor instructions or create functions, where they might otherwise have written the same set of instructions repeatedly to reach the solution.

Inference and Logic: 50%

Although it is necessary to think logically about solutions, Light-Bot does not provide much scope for practising inference, and comprehension of the premise of the game is relatively trivial. However, students may develop personal heuristics that help them to approach the game and progress through the different obstacles at each level. The addition of conceptual features like randomisation or conditionals could increase the logic skills needed to solve the levels in the game.

Evaluations and Improvements: 83%

Although it is possible to solve a number of the levels with more than one approach, Light-Bot does not actually reward students for elegant solutions. The number of commands used to solve each level is counted, but there is no reward for using fewer commands, except perhaps, personal satisfaction. In some cases, excessive commands may "undo" a solution by un-lighting a previously lit-up square, forcing students to look for simpler solutions. Similarly, some of the advanced levels can only be solved by making proper use of functions, forcing students to rethink the structure of their sequential solutions.

When debugging is required, Light-Bot performs well because the student can see what is happening as the robot moves on the board. The ability to watch Light-Bot move allows students to check the results of their solution and make adaptations as needed. Errors are clearly visible when the robot does not behave as expected. A possible improvement would be the ability to set breakpoints and single-step through instructions, which may assist students in logically thinking through their solutions.

5.3 Rubot

Rhodes University Bot (Rubot) is an adaptation of the original Light-Bot game that has been used at Rhodes University for a number of years. Rubot is a C# implementation of Light-Bot, with additional features incorporated to address the needs of the first year CS 101 class. A screen capture of the original version of Rubot is shown in Figure 5.3 while the final CT version is shown in Figure 5.4.

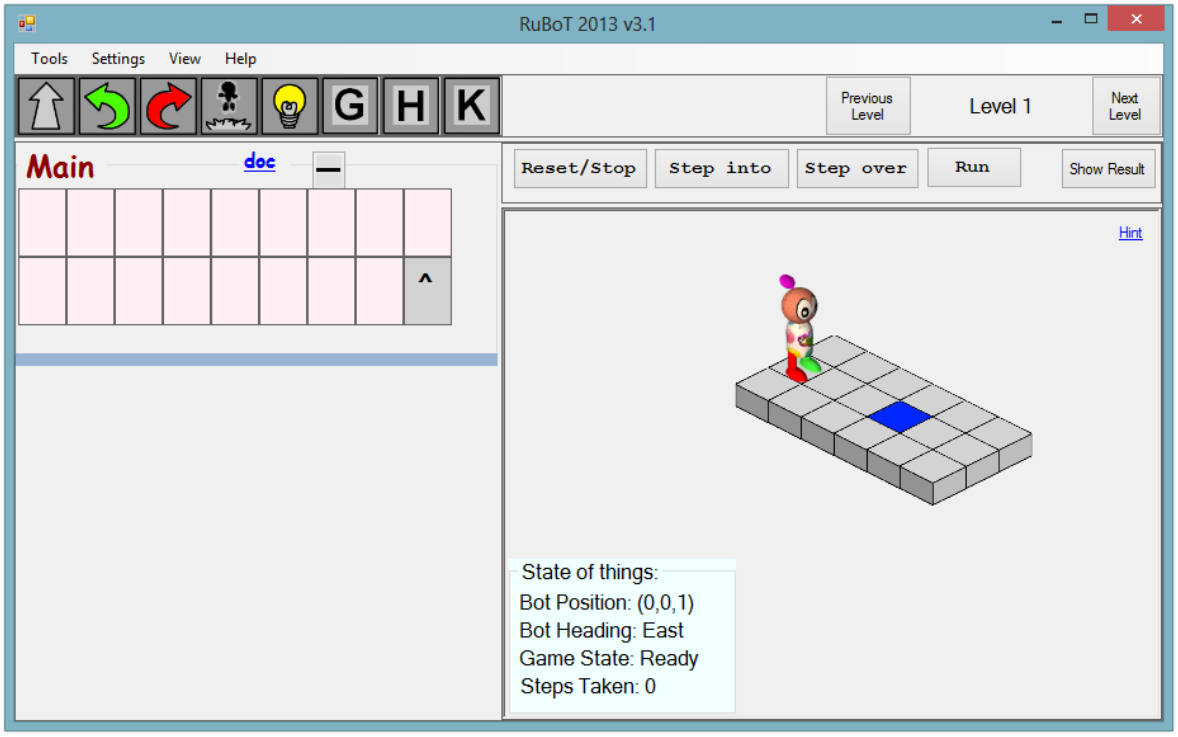


Figure 5.3: The original version of Rubot.

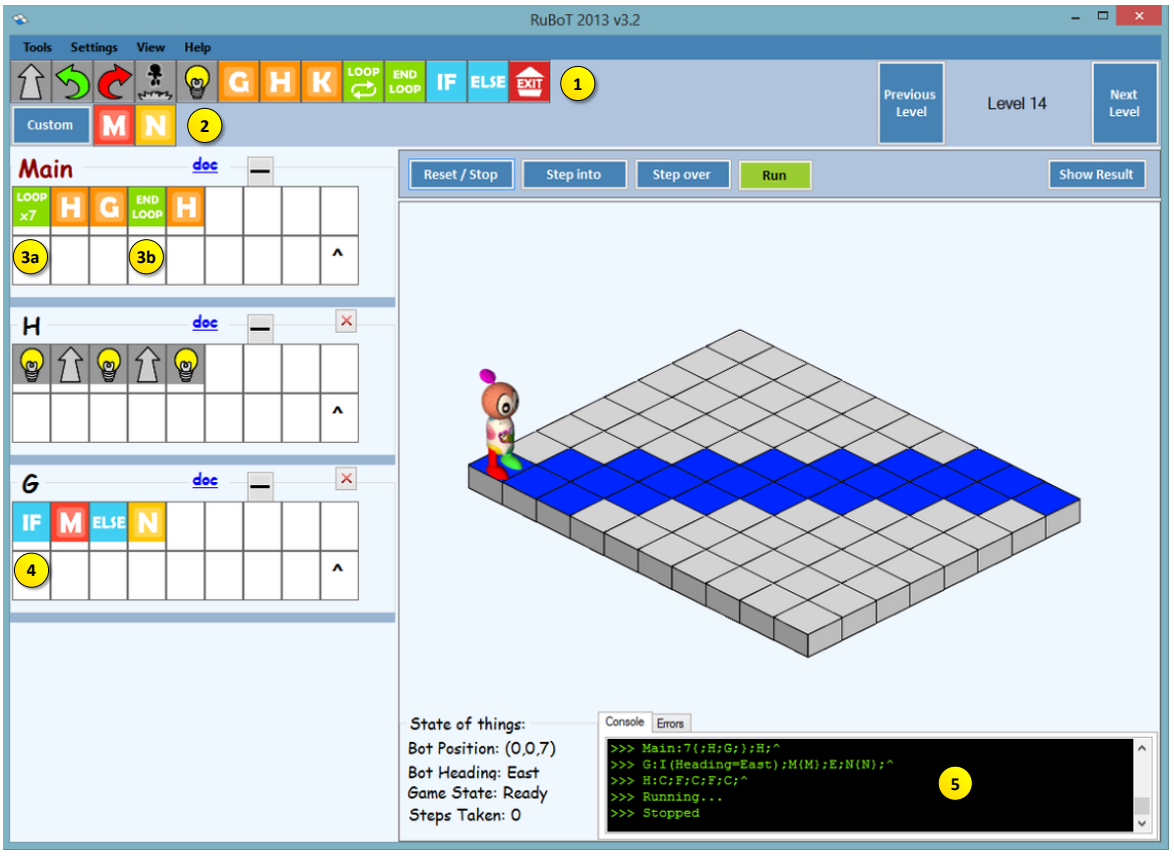


Figure 5.4: The version of Rubot produced for this research.

5.3.1 History of Rubot

Upon the commencement of this research, Rubot had already been established as a teaching tool within a large CS course, in which the majority of students were non-continuing students requiring some computer literacy and a light introduction to CT and programming. The version of Rubot used as the starting point for this research included the following features:

- An implementation of the original Light-Bot features;
- An integrated submission system for students to submit their solutions for marking;
- A settings dialogue allowing users to customise features like the colours used and bot speed; this feature was added to allow users to gain familiarity with property editing dialogues and to emphasise notions of state;
- Debugging tools, including *Step Into*, *Step Over*, and *Run* buttons, and the ability to set breakpoints, which were intended to mimic the features that students encounter in programming IDEs;
- An information box containing the following state information: bot position, bot heading, game state, and steps taken;
- Commenting abilities for functions, typically used to get students to express the high-level purpose of the function and any pre- or post- conditions;
- Options to see Rubot solutions represented in Python or C# style code.

5.3.2 Rubot Adaptations

This section describes the features that were added to Rubot, the motivation for adding these features, and a discussion of the implementation process.

Loops

The motivation for adding an iterative structure was to provide for repetition and patterns. This has been implemented as a for-loop mechanism with a pre-test, allowing for definite rather than indefinite iteration. Two additional commands were added to the standard

command set: *Start Loop* and *End Loop*, which are used to demarcate the looping section within a set of commands. When the user adds a *Start Loop* command, the number of repetitions must be specified. Loops may be used individually, adjacently, as nested loops, and within functions. A sample solution using a loop is given in Figure 5.4 at point (3a) and (3b).

Textual Notation

A textual notation is used for the XML serialisation of levels, to enable the submission and loading of solutions. Additionally, the notation provides a way to write a Rubot program using a set of text commands, which can be used to demonstrate the equivalence of two sets of notations, as well as providing a means for students to write Rubot programs on paper. This was originally implemented with single characters to represent commands, F = Forward, L=Left, etc. With the addition of new commands such as the looping and conditional mechanisms, a single character is insufficient to represent the additional information required, such as number of times to loop or Boolean expression for the conditionals. Therefore, the notation was adapted so that each distinct command is followed by a semicolon. This allows additional information to be appended to commands, such as **s4;** to indicate a start loop that repeats 4 times. An example of the textual notation generated for a solution can be seen in the console window in Figure 5.4 at point (5).

Splash Screen

Rubot contains a set of about thirty puzzles that are accessible for offline use; in a laboratory environment within a CS course, the levels for the current assignment are downloaded dynamically from a web service. The splash screen was added as a means of conveying specific information to students on an ongoing basis. The splash screen appears when the Rubot application is opened, and offers some information and instructions for the current assignment. Each splash screen is defined as an HTML document which is displayed in Rubot using the C# web browser control. This allows for the screen to be changed without interfering with the source files for Rubot. The splash screen for each assignment is loaded along with the levels when a student loads the game.

Conditionals

Conditional commands were added to the command set as a means of implementing conditional execution in Rubot solutions. The primary motivation for adding conditions was to allow for more logic-based exercises.

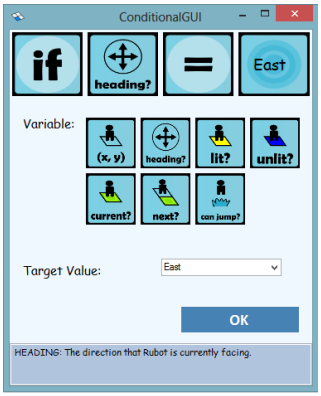


Figure 5.5: *If* command.

Conditions are implemented as an if-then-else mechanism. The commands added to the instruction set are the *If* and *Else*, represented in Figure 5.4 at point (4). When an *If* command is added, the test condition must be specified, as shown in Figure 5.5. *If* commands may be used independently, but *Else* commands must be paired with an *If* command. Each of these commands applies only to the command directly following it — in other words, a function must be used to allow a more advanced set of instructions to execute based on a condition.

Variables and the state of things: To use the conditional mechanism, some variable factor must be evaluated. As it seemed impractical to add a discrete variable mechanism to Rubot, these conditionals are based on information in the "State of Things" window. This state information is inherent in Rubot, and provides a set of values that change throughout game play. The conditional information that may be used includes the position and orientation of the bot, and information relative to the state of the board.

Custom Commands

The custom command feature allows users to extend the existing command set, with up to five user-defined commands. The primary motivation for including a user-defined command is to encourage abstract thinking, as the user must design the internal working of a custom command and thereafter use it as a single unit. Each custom command is given a name, description, instruction set, and dynamically assigned symbol. Custom commands differ from functions in that they are executed as a single command rather than stepping through the internal commands.

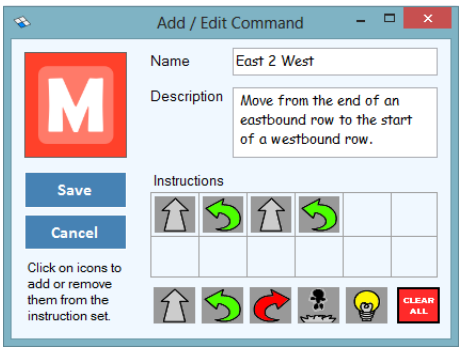


Figure 5.6: Defining custom commands.

Custom commands are persistent across levels for the duration of the Rubot session. The custom command section of the command set for a level is shown in Figure 5.4 at point (2), and the dialogue box used to define a custom command is shown in Figure 5.6. An additional benefit of this feature is that it encourages users to identify similarities across levels in order to reuse custom commands created previously.

Grouping of Commands and Levels

To prepare the Rubot environment for grouping levels, the full set of available commands is arranged into three clusters, referred to as libraries in the Rubot source code. In the XML specification for each level, indicators are given as to which command libraries should be made available to solve the level. The three command libraries are:

- **Core:** forward, left, right, jump, click, function G, function H, function K.
- **Complex:** start loop, terminate loop, if, else, exit.
- **Custom:** user defined commands, represented by the letters M - Q.

The reasoning for this approach is to allow simple levels to be defined using only the core command set, and more advanced levels to use the complex and custom command sets as well. This allows concepts to be introduced incrementally as users gain confidence with the initial levels and techniques, as well as encouraging the use of specific techniques based on the command set that is provided for each level. The full command set is shown in Figure 5.4, with the core and complex commands used at point (1) and the custom commands included at point (2).

Exit Command

The *Exit* command, depicted in Figure 5.4 at point (1), allows for recursion. For use in a constructive manner, an *Exit* command must be paired with a conditional command, usually within a function or a loop. This command executes a function return at the point it is reached, much like a *break* statement if in a loop and the *return* statement if in a function in iterative programming languages.

Console Window

The console feature was added to allow for textual feedback, providing an additional avenue for communicating with the user. The console window is shown in Figure 5.4 at point (5). This window is read-only. The information communicated in the console window includes the current level, the game state, textual representations of solutions, and feedback on the result of an execution attempt.

Error Reporting

Error reporting was added as a feature in Rubot to enhance students' ability to develop debugging skills. Each error that is generated falls into either the syntax or runtime category, and is reported as such. Syntax errors are identified when the solution is parsed in the *SyntaxTree* class, while runtime errors are identified when the solution is executed in the *RuBoTApp* class. Errors are reported in a special error feedback window, as shown in Figure 5.7. The possible error messages that students may encounter are described below:

- ***Syntax errors***
  - 'End-loop' command has no matching 'start-loop'
  - 'Else' command has no matching 'If' command
  - 'Else' command must directly follow 'If' action
  - Blanks are not allowed between commands
  - 'Cmd' is unrecognised
  - Start-loop command has no matching end-loop
- ***Runtime errors***
  - Cannot walk onto raised tiles
  - Cannot walk onto lower tiles
  - Invalid jump (cannot jump to block on same level)
  - Invalid jump (cannot jump up by more than one block)
  - Cannot light up floor tile
  - There is unreachable code after the 'Exit' command

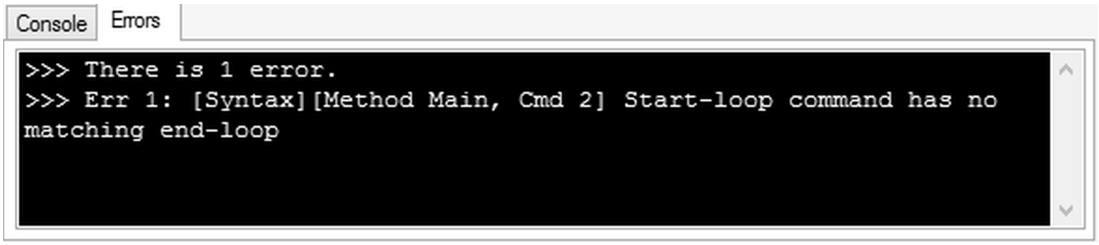


Figure 5.7: Rubot error feedback window.

5.4 User Study

A user study was carried out to assess the benefits to students of using the CT version of Rubot. The aim of this study was to determine whether it is worthwhile using Rubot in teaching programming, and to identify the specific strengths and weaknesses that students may respond to. For this assessment, Rubot was integrated into the course work for an introductory programming course, and at the end of the course a questionnaire was distributed to gather responses from students.

5.4.1 Research Participants

The class selected to participate in this research was the Computer Skills for Science (CS1S) class, which is a foundation course for teaching computer literacy and introductory CS. A module on basic programming in Python was presented in the final six weeks of the course. Rubot activities were integrated alongside the Python module, as further discussed in the next section.

5.4.2 Assessment Format

For this experiment, the students' use of Rubot was designed to coincide with their programming practical sessions[6]. For five consecutive weeks, the students were given some Rubot exercises to complete using roughly one third of their practical session time, amounting to about 30 minutes spent on Rubot each week. For each session, an information screen was designed and specific levels were set, as shown in Appendix B. In the final week of the course, the students were required to complete a questionnaire. A brief discussion of the content covered each week is given below:

[6]Each practical session consists of 90 minutes in the computer lab with assignments that count towards the final grade.

Week 1: In their course work for this week, the students completed an introduction to CS and programming. For the Rubot section of the week's practical, they were required to play four levels of Rubot, which gave an introduction to Rubot and introduced the basic command set.

Week 2: In their course work for this week, the students learned about variables and conditionals in Python. For the Rubot assignment, they were required to play two levels of Rubot, which were more advanced than those of the previous week and focused more specifically on the looping mechanism.

Week 3: The CS1S course work for this week introduced the Turtle 2-D drawing module and the use of functions in Python. For the Rubot section of the practical, the students were assigned two levels which focused on the use of functions in Rubot.

Week 4: The course work for this week focused on string handling in Python. Building on the assignments for the previous week, the two Rubot questions were based on the use of the custom commands feature.

Week 5: For this week, the CS1S course work focused on input validation, debugging, and trace tables. The Rubot section of the practical included two questions requiring students to make use of the skills they have learned thus far, and encouraging them to independently explore additional Rubot features.

5.4.3 Questionnaire

Feedback from the students who had used Rubot was obtained in the form of a questionnaire using a five-point Likert scale and divided into two sections: usability and usefulness. The usability section was intended as a general assessment of user experience and satisfaction. The usefulness section measured the effectiveness of the game in the given context: facilitating computer skills studies and encouraging CT. These questions primarily measured the students' perceptions of their own CT activities while using the game. The feedback questionnaire is given in Appendix C.

Usability

The usability section of the questionnaire was based on a System Usability Scale (SUS) test, with the word "game" substituted for the word "system" [6]. This section contained 10 Likert scale questions, as follows:

1. I think that I would like to use this game frequently
2. I found the game unnecessarily complex
3. I thought the game was easy to use
4. I think I would need the support of a (technical) person to be able to use this game
5. I found the various functions in this game were well integrated
6. I thought there was too much inconsistency in this game
7. I would imagine that most people would learn to use this game very quickly
8. I found the game very cumbersome to use
9. I felt very confident using the game
10. I needed to learn a lot of things before I could get going with this game

In addition to the Likert scale questions, two free response questions were included. These questions asked the respondent to identify three things that they enjoyed, and three that they did not enjoy about the game.

Usefulness

The Usefulness section was divided into two subsections: relevance to the CS1S course, establishing whether it would be fruitful to use the game in the future, and CT characteristics. An opportunity was given at the end of the questionnaire for any additional feedback. An outline of the questions is given below:

Relevance to the Computer Skills Course

1. This game helped me to understand computer skills concepts
2. The difficulty of this game was appropriate for students in this class
3. I understood the purpose of this game
4. I think that future computer skills classes would benefit from playing this game

CT Characteristics

- *Processes and Transformations*
  5. I put a lot of thought into how to approach each level
  6. I often broke problems down into smaller sections that needed solving
- *Models and Abstractions*
  7. I understood how to look at a level and represent the solution using commands
  8. I was able to work effectively with different levels of detail (particularly with functions)
- *Patterns and Algorithms*
  9. Once I learned how to do something I used this knowledge over again
  10. It was useful to look for patterns in the problem
- *Tools and Resources*
  11. I put thought into how to solve the levels efficiently
  12. I understood how to use and arrange groups of commands to perform a desired action
- *Inference and Logic*
  13. I had to be creative to play this game
  14. I developed my own heuristics to make the levels easier to solve
- *Evaluations and Improvements*
  15. It was easy to find and fix mistakes in my logic
  16. It was important to me to produce a good solution (rather than one that simply worked)

5.4.4 Results

The questionnaire was administered in the final week of the CS1S course. Of the 32 students in the class, there were 30 respondents. For referencing purposes and to preserve anonymity, respondents' questionnaires were labelled with the numbers 1-30. The results are presented according to the sections in the questionnaire; with usability feedback discussed first, followed by the usefulness feedback.

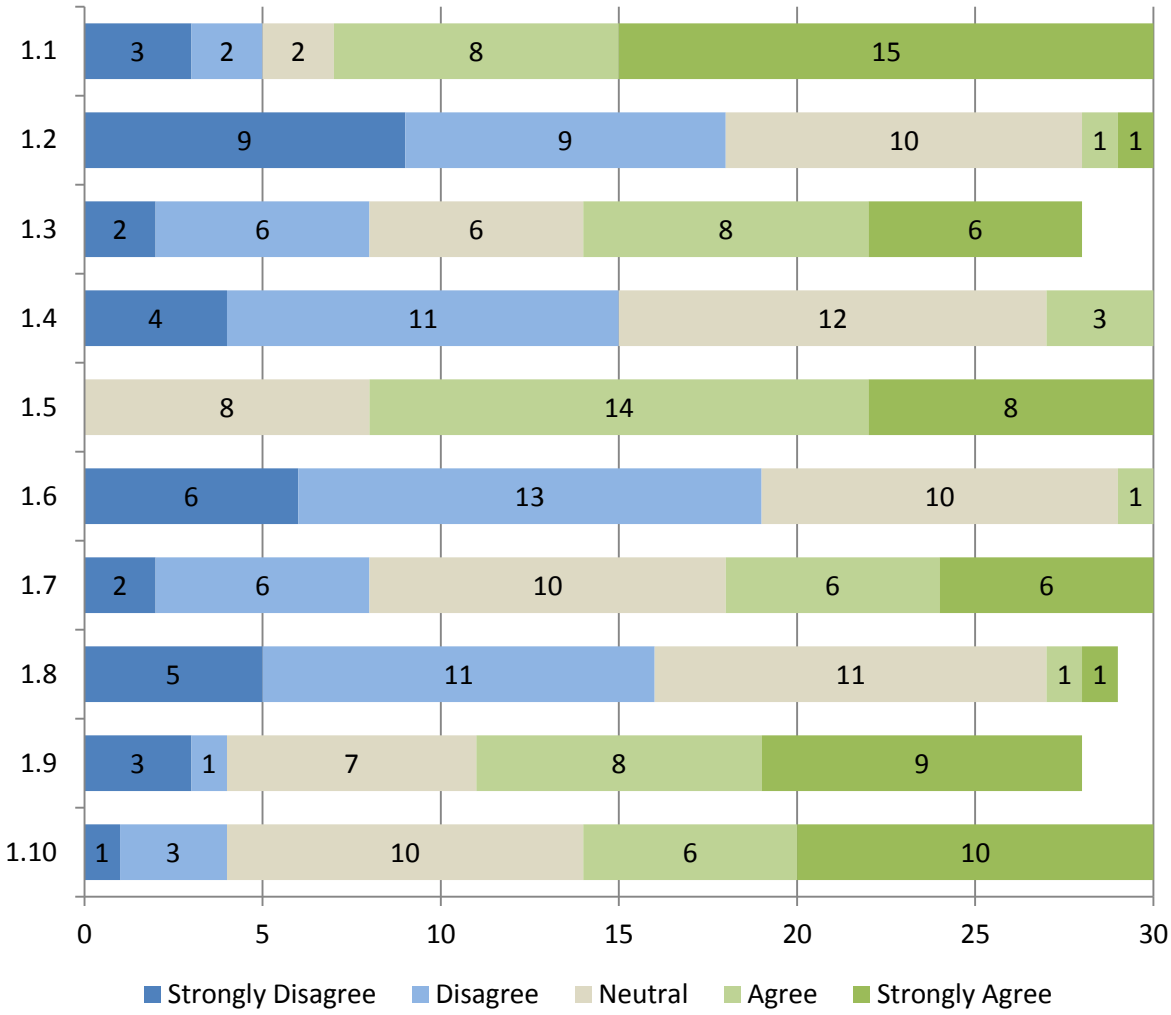


Figure 5.8: Likert scale responses for Section 1 Questions 1 - 10. Question numbers are given on the vertical axis, and responses on the horizontal axis. The number of respondents per answer is indicated using data labels.

Usability Feedback

The aggregated results for Questions 1.1 to 1.10 compute to a mean SUS score of 63.4[7]. According to [6], this corresponds to an adjective rating between OK and GOOD. The breakdown of responses is shown in Figure 5.8.

From these results, 77% of respondents expressed interest in using the game regularly. Although 73% of the respondents felt that the functions in the game were well integrated, only half of the respondents found the game easy to use and 53% felt that they needed to learn a great deal before they could get going. The responses indicate that the game

[7]https://www.measuringusability.com/sus.php

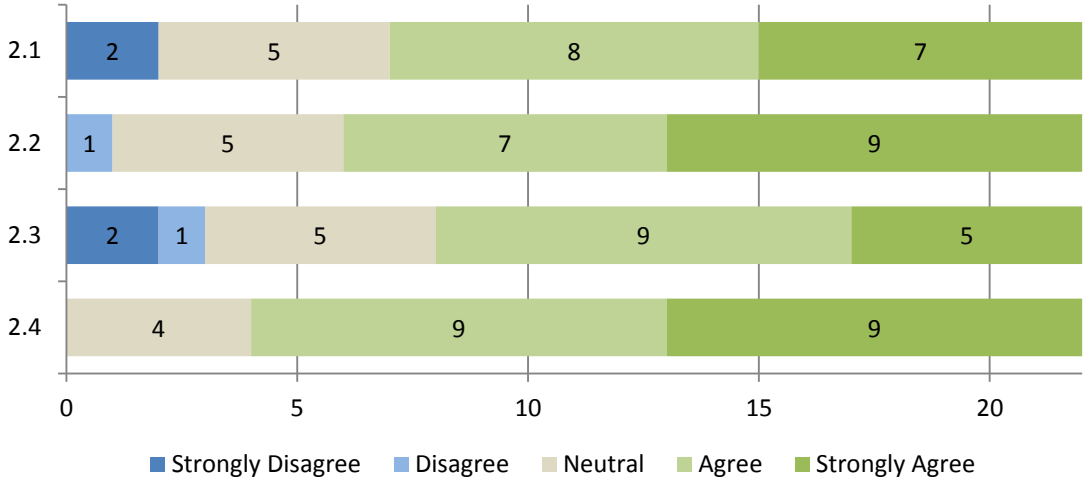


Figure 5.9: Likert scale responses for Section 2 Questions 1 - 4. Question numbers are given on the vertical axis, and responses on the horizontal axis. The number of respondents per answer is indicated using data labels.

is perceived to be complex, but that it functions well once the learning curve is mastered. This is in keeping with the opinions expressed in Section 5.4.5.

Usefulness Feedback

The results for Questions 2.1 to 2.4, illustrated in Figure 5.9, give an indication of the response to the game in the context of a computer skills class. These results are largely positive, 73% of respondents agreed that the difficulty level was suitable and 82% agreed that future students would benefit from playing Rubot. Although students perceived Rubot as valuable the results reflect a lesser understanding of why – 68% of respondents agreed that the game assisted in the understanding of computer skills concepts while 64% agreed that they understood the purpose of the game.

Questions 2.5 to 2.16 reflect the CT characteristics of Rubot, as defined by students' feelings about their own engagement. The pair of questions for each CT category are combined into a single Likert scale value, shown in Figure 5.10. Based on these values, the weakest area of engagement is models and abstractions, with a score of 3.52. The strongest areas are processes and transformations and patterns and algorithms, with scores of 4.0 and 4.1, respectively. This is consistent with the responses in Section 5.4.5 where students indicate that loops were a particular feature that they focused on. The concrete nature of the game may be a contributing factor to the lower scores for abstract thinking.

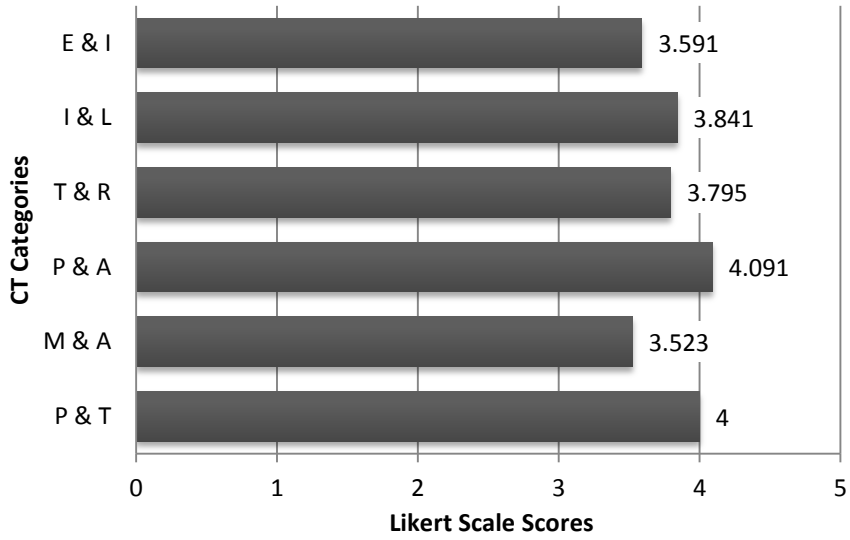


Figure 5.10: A summary of the student responses to the *CT Characteristics* section of the questionnaire.

5.4.5 Specific Feedback from Students

Some trends of common experience emerge in the student answers to the free-response items in the questionnaire. These findings are outlined below.

Aspects the students enjoyed: A large proportion of students indicated that they enjoyed the ways that Rubot made them think, particularly engaging their critical thinking abilities. Participant 9 stated:

> *It is challenging and fun at the same time. It shows you how quickly you can come up with a solution to a computational problem.*

Eleven students indicated that they appreciated the challenging nature of the game; it encouraged them to learn to solve problems and think out of the box. Six students indicated that Rubot was fun or enjoyable to play.

The responses contain many references to specific functionality available in Rubot, particularly the use of loops in solutions, mentioned by ten respondents. Further responses indicate that the students enjoyed the visual nature of the game, particularly watching Rubot move around the tile world, and observing where errors were made. One respondent indicated that Rubot gave him/her a visual picture of what loops do and therefore a

better understanding of how loops work. A number of respondents recognised the correlation between Rubot and their work with Python. The motivation for using a game like Rubot is to make the computational model more concrete for students who have difficulty with abstraction. It is encouraging to find that students are able to relate the concrete, visible loops in Rubot to more abstract Python loops, as indicated by the comments above.

The game encouraged persistence in working towards a solution, as respondent 28 indicated:

> *[It is] exciting when you "finally" get it right.*

Other respondents indicated that the game taught them to be patient and pay attention, and expressed satisfaction in watching the successful execution after the effort of designing a solution. A contributing factor could be that the students appreciated having unlimited attempts at solving a level.

The responses indicate that game play encouraged a level of creativity among the students, with respondent 11 indicating that he/she enjoyed:

> *Creating my own unique code of running Rubot.*

Additionally, the feedback reflects that the students appreciated creating their own commands and that they developed heuristics to approach the game.

Aspects the students did not enjoy: Many respondents commented on the amount of time required to play Rubot; some indicated that it took them too long to solve the levels, while others felt that the game was slow and that they would rather have spent the time on course work. Conversely, one student responded that there was not enough time to practice, while another indicated that s/he would prefer for there to be more than two levels a week. On a similar note, some students did not enjoy playing the game for marks and having it integrated into practical sessions.

A number of students experienced difficulty with the complexity of the game. The feedback indicates that the game could be difficult and frustrating on some occasions. Participant 1 highlights a potential problem:

> *When you run the code and an error occurs, and you can't figure out how to correct it.*

Responses indicate that some students found the game over-complicated with too many instructions, and that they tended to use features with which they were familiar. Although some views reflect different opinions, a perceived lack of continuity was cited as a problem, as participant 30 stated:

> *[I didn't enjoy] the fact that each level has its own instructions like using loops and customise.*

Other students found hindrances in having to reset the execution and the absence of a pause feature. Incidentally, the single-stepping feature in Rubot could serve as a pause feature but was not actively promoted as a feature within the given assignments. Several responses referred to the varied experiences across different levels. Some students felt that higher levels are too difficult:

> *As you approach higher levels it gets difficult (participant 4). The levels got more difficult every week, I got bored (participant 10).*

Some respondents felt that the levels did not connect with each other enough, and that solutions were not applicable across a range of levels. Two students indicated that they would have enjoyed having more levels available, which may have bridged the gaps between seemingly disparate levels.

A lack of guidance was a concern that emerged through the responses. Respondent 29 summarised the problem saying:

> *There is not much guidance in how to control your Rubot.*

One student felt that the instructions were not clear, while another stated that it was difficult to learn at the outset. Respondent 17 indicated that s/he did not enjoy reading the instructions and respondent 20 said there was too much instruction, which suggests that the provided instructions could have been presented in a more user-friendly way.

5.5 Improvements and Extensions to Rubot

Throughout the process of extending and assessing Rubot, a number of possible future extensions have been identified. These adaptations are based on the use of Rubot as

a teaching tool, and aim to enhance the CT flavour of Rubot, user satisfaction, and usefulness in an educational context.

Refining the conditional structure:

As part of the adaptations made for this project, a conditional structure was added to Rubot. Although this feature is programmatically accurate and can serve to reduce the number of required commands, we failed to identify a scenario where it would be imperative that the users must use a conditional command. This is largely attributed to the predetermined nature of each level — there is no situation where the user might not be able to foresee the state of Rubot or of the tile world. This situation may lead students to fail to perceive the usefulness of conditionals in programming.

The addition of a "trap-door" feature is proposed as a potential means to remedy this. A trap-door would be an item in the tile world where the state of the tile — open or closed — would not be known prior to runtime. This uncertainty would necessitate the use of a conditional to provide branched execution flows, and explore a set of skills that are currently underutilised in Rubot.

Planning and mapping tools:

It would be highly desirable for Rubot to be used as a tool for building strong mental models and visualisation of processes. In particular, the ability to do abstract thinking and "chunking" would be useful skills for students to develop. An avenue for this could be the use of discrete planning tools, encouraging students to create a mental picture of how the level should be solved rather than taking a primitive trial-and-error approach of incrementally adding commands until the solution succeeds.

A proposed solution is to have a drawing feature allowing the student to make drawings and annotations over the Rubot board. This would allow them to dissect levels and visualise an approach before writing concrete commands.

Rubot as a C# module:

A recurring cause for concern with educational games is the students' ability to draw parallels between their use of the game and their studies within the discipline. The best case would be to retain the assets of a game, such as the concreteness of the model and the inherent motivation to play, and integrate this into a programming environment. This could be accomplished by recreating Rubot as a module that could be imported into a programming language like C# (as has been done with

the turtle module familiar in Python[8]). The syntax of Rubot solutions could be adapted to look more similar to C# code, as in the example below:

```
1 Rubot.forward()
2 Rubot.forward()
3 Rubot.click()
```

Although the barrier to entry is greater using this approach, there would be additional benefits to the students, such as gaining a familiarity with an integrated development environment (IDE) such as Visual Studio[9] and working with real debugging tools.

Debugging:

Rubot currently contains several conventional debugging tools, such as the ability to set breakpoints, single step, and the printing of error messages. However, the practice of debugging is not heavily emphasised. The game could be adapted to promote sound debugging skills that would develop fruitful skills, possibly by making the tools more visible and through active encouragement.

Custom designed levels:

Currently, additional levels can be designed by creating an XML file representing a Rubot puzzle, and dragging-and-dropping the file into Rubot. The files are conventionally created using a text editor.

The ability for users to create and play their own levels adds an interesting layer to their experience of Rubot. In order to create a "good" level, students need to refine their understanding of the game model and consider the level and proposed solution concurrently. This may be an avenue for creativity and collaboration between students.

In a future version of Rubot, the custom level design could be integrated into the game itself, either as a text editor or with a graphical user interface (GUI) based world creator. This could overcome the preconception that Rubot is only useful for a defined set of problems, and encourage students to see the generalisability of the tool.

Skeleton solutions:

Both in academia and industry, it is rare for programmers to develop a system completely independently. Therefore, the ability to understand and work with partial

[8] docs.python.org/2/library/turtle.html
[9] http://www.visualstudio.com/

solutions is a necessary skill to develop. In Rubot, this could be represented effectively by allowing levels to come pre-set with some commands strategically placed in the solution space. This would force students to consider the intended approach, to understand the reasoning behind that approach, and to tailor their own work to fit in with an existing system. It is advised that this feature would only be used for advanced levels.

"Win-ability":

In its current state, Rubot offers no final end-point or reward for completing all of the levels. This may impact on user satisfaction and motivation when playing the game. Some kind of reward system could be integrated to encourage students to persist and to create better solutions to problems. This could allow moving away from assigning marks for playing the game, which students indicated that they did not enjoy.

Demo solutions:

Many games contain demonstrations or "demo" solutions, which show the user how to play the game successfully. In a game like Rubot, demos could take the place of a tutorial, allowing students to observe how key Rubot features should be used, and introducing more elegant means of solving levels. As the application already contains functionality for saving a solution to a level, demos could be created as solutions to specially designed levels. A list of demos could then be added to the Help menu option, which would be easily updated by an educator. This would address the student responses indicating a lack of instructions or apprehension at reading instructions, by moving from a textual explanation to a visual demonstration.

Variety and categorisation of levels:

The student responses indicate some dissatisfaction with the progression of levels in the game. A larger set of levels could be designed with the following goals in mind: there should be a smooth progression in the difficulty of levels; the variations in the levels should promote the use of all of the Rubot features over time; there should be a flow of thought processes and techniques used across levels, and there should be sufficient practice opportunities. If levels are categorised into groups according to out CT framework discussed in Chapter 3, each student could take a quiz when starting the game and then have a customised set of levels prescribed.

5.6 Summary

In this chapter, the efforts to provide a CT computer game were discussed. An initial evaluation of the existing game Light-Bot revealed it to be a useful game, with numerous avenues for improvement of its CT value. Light-Bot was weakest in the patterns and algorithms and inference and logic categories of our CTF.

Improvements that could be made were identified and implemented as an adapted version of Light-Bot, named Rubot. Rubot was incorporated into the introductory programming module for a computer skills class, and evaluated by the students in the class. Based on the user study results, Rubot was shown to have a fair usability score, and good responses for its CT value. Further extensions to Rubot were identified based on user feedback.

Chapter 6

Strategies and Deliverables

Chapter 5 gave an overview of a computer game that was developed to incorporate different areas of CT. In this chapter, we discuss specific resources for a range of CT practices. We begin with a discussion of problem solving strategies that may be applied to the different categories in our CTF, drawing on the existing work by George Pólya [60]. The applicability of puzzle based learning for CT is then discussed, and a number of puzzles are proposed to exercise the specific skills represented in the CTF. Finally, this chapter includes a description of some existing projects that have been used to facilitate and promote CT.

6.1 Problem Solving Strategies

As established in Chapter 2, CT is at its core a problem solving exercise, applying the particular skills gained from CS to improve human conditions or efficiency in some manner. Likewise, problem solving forms an integral role in many domains. Different sets of strategies have been developed to assist in the problem solving process; these range from critical thinking and lateral thinking techniques used in brainstorming, to specialised techniques developed to assist with examinations and student performance. A student may be instructed in the syntax of a programming language, or in various mathematical formulae, but without the proper problem solving skills they have limited ability in putting this knowledge to practical use.

Perhaps the best regarded problem solving techniques are those outlined by George Pólya in the book *How To Solve It* [60]. This book has been primarily influential in mathematics

Table 6.1: A comparison of Pólya's [60] problem solving process for mathematics and Barnes et al.'s [8] problem solving process for programming.

How To Solve It	**How To Program It**
Understand the problem	Understand the problem
Devise a plan	Design the program
Carry out the plan	Write the program
Look back	Look back / Review

education [8], although as Pólya himself states it is relevant to anyone with an interest in invention and discovery. In his book, Pólya identifies four steps that should be followed when solving a mathematical problem: understand the problem, devise a plan, carry out the plan, and look back on your work. These steps form the framework for a problem solving process that he illustrates with numerous examples of mathematical problems in the classroom. Barnes et al. [8] have recognised the usefulness of Pólya's strategy for CS students, and adapted his process into a new process they call *How To Program It*. They report great success in integrating a problem solving course at a remedial level for CS education, although they believe that such skills would be better taught at an introductory level. The correlation between the steps in the original and new frameworks is shown in Table 6.1.

Throughout *How To Solve It*, and particularly in step two and the discussion of heuristics, a set of strategies that may help in solving a problem are identified. The strategies outlined by Pólya have a remarkable similarity with many techniques used in CS. We have classified Pólya's strategies according to our CTF, and supplemented them with additional strategies that we believe to be beneficial in a CS context, as given below.

Processes and Transformations

- *Pólya:* work backwards; make an orderly list.
- *Additional:* define smaller goals; order sub-goals; identify dependencies between sub-goals; single-step through the solution.

Models and Abstractions

- *Pólya:* use a model; draw a figure; use symmetry (invariance).
- *Additional:* use a different notation or representation; use different levels of abstraction; change the perspective; use a "drill-down" approach.

Patterns and Algorithms

- *Pólya:* use a formula; solve a simpler problem; solve an equation; look for a pattern; consider special cases.
- *Additional:* find a similar problem; simplify the problem; find repeating actions.

Tools and Resources

- *Pólya:* use your head / "noggin".
- *Additional:* identify building blocks (tools); identify limiting factors; prioritise factors; combine tools into new tools; use caching.

Inference and Logic

- *Pólya:* be creative; use direct reasoning; use induction; guess and check.
- *Additional:* identify given information; gather details; develop personal heuristics; pay attention to detail.

Evaluations and Improvements

- *Pólya:* eliminate possibilities; check the result.
- *Additional:* consider different approaches; create a reusable solution; verify results; avoid excessive complexity.

6.2 Puzzles as Recommended Activities

In Chapter 5, we discussed our development of a computer game as an artefact for practising and developing CT skills. However, individual activities may be better suited than an integrated game for some purposes. Pen-and-paper activities would be especially useful in scenarios where there is no access to proper technological resources. In this section, efforts to identify CT puzzles and activities are discussed.

Puzzle based learning (PBL) offers a novel and attractive solution to stimulating student interest in critical thinking and mathematics, and aids the development of problem solving skills. Michalewicz et al. [54] are involved in promoting PBL as a course in Australia, with resources available on the Puzzle Based Learning website[1]. They describe a number of benefits that PBL has on the impressions of students, particularly in showing them that

[1] `http://www.puzzlebasedlearning.edu.au`

Table 6.2: Classification of puzzles using the CTF.

Puzzle	P & T	M & A	P & A	T & R	I & L	E & I
Bottle and Cork			✔		✔	
Bug Traffic	✔	✔	✔			
Get 4L	✔		✔	✔		✔
Kisses & Handshakes	✔		✔		✔	✔
Rock Climber Maze	✔		✔			✔
Square Size		✔			✔	✔
Square Window		✔		✔		
Truth Telling	✔		✔		✔	✔

the science and mathematics courses taken at university are useful, interesting, relevant, and not scary. A series of criteria for good educational puzzles are given as follows: good puzzles should explain universal problem solving principles, be easy to state and remember, be entertaining, and involve a Eureka factor where the result is counter-intuitive [54]. In many ways, PBL is compatible with the goals and ideals outlined for CT. To investigate the value of PBL for CT, we considered a series of puzzles from the website Puzzles.com[2], with most of the focus being on non-manipulative "math 'n logic" puzzles. Below, we discuss the CTF classification and problem solving strategies for eight puzzles from this website. The strategies suited to solving each puzzle are identified, and CT classifications are made accordingly. The full puzzle descriptions and solutions are given in Appendix D, while an overview of the puzzle classifications is given in Table 6.2.

Bottle and Cork: The goal of this puzzle is to work out the individual costs of a bottle and a cork, given the combined cost and the fact that the bottle costs one dollar more than the cork. The problem solving strategies recommended for this puzzle are solving an equation, simplifying the problem, and attention to detail.

The puzzle can be solved simply be creating and then solving a formula: $(bottle + 1) + cork = 1.1$. The most common mistake in solving this problem is to jump to the conclusion that the bottle costs \$1 and the cork costs 50c; if the student pays attention to detail they will realise that this answer does not fit the problem criteria.

Bugs Traffic: Given the starting location of four bugs on a square, as well as information about the way that they move, the goal of this puzzle is to find the distance that each bug travels before they meet. The strategies recommended for this puzzle are using a model, using symmetry, single-stepping through the problem, and looking for a pattern.

[2] `http://www.puzzles.com`

The biggest point of difficulty in this puzzle is the way that the bugs' trajectories change as each bug moves. Mapping a single move at a time allows the student to recalibrate the trajectory of each bug accordingly. By drawing a model of the square and measuring the distances accurately, the length that each bug has travelled can be obtained.

Get 4 Litres: The goal of this puzzle is to measure four litres of water given only a 3L and a 5L container. The recommended strategies for this problem are working backward, looking for a pattern, defining sub-goals, repetitive actions, identifying and combining tools, and creating a reusable solution.

In this puzzle, the given tools are the two containers and the objective is to discover how to correctly combine them to measure a different amount. The solution takes the form of a pattern with indefinite iteration, as water is measured in one jug and transferred across to the other jug. Once the solution is identified, a similar method could be used to measure different amounts. In certain variations the problem is not solvable; for example, given a 4L and a 10L container it would not be possible to measure seven litres. This could be used to illustrate the fact that some problems do not admit any solutions.

Kisses and Handshakes: Given instructions about the kisses and handshakes that occur between guests at a dinner party, the goal of this puzzle is to work out the number and composition of guests at the party. The recommended strategies for this puzzle are considering special cases, checking the result, defining sub-goals, finding repetitive actions, and guess and check.

Multiple pieces of information must be found for this problem. The first goal should be to find the number of men arriving, as this is the simpler case. Repetitive actions are performed as each guest performs the same set of actions. A guess and check approach can be useful to estimate the number of women arriving, narrowing the margin of possible alternatives, which can then be checked until the correct answer is found.

Rock Climber: This is a maze-based puzzle, with the goal of reaching the top of the stack of coloured blocks by traversing the coloured sides in a sequential fashion. The recommended strategies for this puzzle are looking for a pattern, working backwards, and considering alternative approaches.

A repeating pattern must be found to traverse the hill to the top. Beginning at the bottom of the hill there are a number of possible routes that must be considered

and evaluated, much like a maze. The problem is simpler to solve starting from the top and working backwards, as there are fewer alternatives to consider.

Square Size: The goal for this puzzle is to find the size of a square of numbered blocks, given only a partial view of the square. The recommended strategies for this puzzle are drawing a picture, induction, guess and check, checking the result, and simplifying the problem.

This puzzle is a good example where abstract thinking is required, as the bigger picture must be inferred from a smaller view. The problem can be simplified by focusing on solving only the rows that are partially shown in the sample, and then extrapolating this to find the bigger block. If a picture is drawn with the given tiles filled in, the rest of the square can be filled in and checked around them.

Square Window: Given a square window and an instruction to halve the light let in by the window, the goal is to find an appropriate method of doing so while retaining the height, width, and square shape of the window. The recommended strategies for this puzzle are drawing a picture, using symmetry, being creative, and identifying limiting factors.

This puzzle is a classic example of the "Eureka moment" required for stimulating PBL, as from a student's perspective it seems impossible to solve on first impressions. Students can find the answer more easily by drawing and manipulating a picture, and will need to use a creative approach within the given limitations.

Truth Telling: Given three characters who each make a statement, the goal for this puzzle is to discover which character is telling the truth. The recommended strategies for this puzzle are solving a simpler problem, using direct reasoning, induction, eliminating possibilities, verifying results, guess and check, defining sub-goals, and identifying initial information.

At the outset of this problem, there are three possible answers. The first goal may be to rule out one of the options, reducing the problem to the simpler case of two options. An alternative solution is to select an option randomly, and then evaluate the statements to decide whether that selection was correct.

6.3 Existing CT Resources

In addition to the options we have given, a number of resources exist for the practice and exploration of CT. These resources provide varied means of promoting CT that are

suitable for the classroom and personal use. Some of the more notable projects are briefly discussed below.

- **CS Unplugged**[3]**:** The CS Unplugged project uses different modalities to show children how computer scientists think, including activities, games, magic tricks, and competitions [11]. This website introduces children to the great ideas in CS, and to CT without using computers. These activities introduce a broad range of CS concepts such as algorithm complexity, interface design, and data compression, in ways that interest children and do not require technical experience. The CS Unplugged project is well adopted, having been translated into twelve languages and recommended in the ACM K-12 curriculum [11].

- **CS4FN Magazine**[4]**:** The *Computer Science for Fun* (cs4fn) project consists of a free print magazine, website, and shows for children [23]. This project exists to dispel misconceptions about CS and stimulate interest in young people. The materials in the cs4fn project are able to supplement other CT programs and illustrate the connections between CS and other disciplines. The website includes many resources and articles, including special editions like the "Alan Turing" edition and the "Women in Computing" edition.

- **Exploring Computational Thinking**[5]**:** Google's Exploring Computational Thinking (ECT) project provides a wealth of CT resources, including an operational definition, resources, and discussion among K-12 educators [75]. The ECT project defines CT in terms of four primary techniques: decomposition, pattern recognition, pattern generalisation and abstraction, and algorithm design. These resources are largely structured around teaching, and provide curriculum models, classroom-ready lessons, and resources for integrating CT in the classroom. ECT resources are given for a number of CS and CT topics, and are aimed at students across different grades at school.

- **Scratch**[6]**:** Scratch is an interactive programming environment designed to appeal to people who might not have imagined themselves as programmers [62]. This MIT Media Lab project aims to help young people to develop mathematical and computational skills, particularly the ability to think creatively, reason systematically, and work collaboratively. The basic premise of Scratch is inspired by the idea of Lego

[3] `http://csunplugged.org/`
[4] `http://www.cs4fn.org`
[5] `http://www.google.com/edu/computational-thinking/lessons.html`
[6] `http://scratch.mit.edu`

blocks and consists of a set of visual "programming blocks" that are connected to create programs. The Scratch project fosters an online community and promotes the accessibility of programming.

6.4 Summary

With the establishment of CT as an important area of focus, resources are needed to aid the practical aspects of developing CT skills. In this chapter we discussed the importance of problem solving techniques, and provided a series of techniques drawing on the work of Pólya that were classified according to our CTF. Following this, the merits of puzzle based learning were discussed and a set of puzzles was given as a means to practice different CT skills. Finally, existing projects for promoting and teaching CT were discussed.

Chapter 7

Discussion

The intention of this research was to explore the field of CT at an introductory level; this exploration has taken different forms. At the outset, a CT framework was developed, which has been systematically applied to the development of a student intervention, a CT specific computer game, and a series of strategies and recommended activities, as described in the preceding chapters. This application has revealed some strengths of the CTF, as well as possible future improvements. In this chapter, we discuss the success of the CTF as applied in these three areas, and suggest some improvements.

7.1 CTF and the Student Assessment

The first use of the CTF was the development of a CT test to assess incoming CS university students. The CTF was applied to classify the question set, and this classification guided the choice of questions to include in the test. For this test, the CT categories on the vertical axis were used, but the horizontal axis of the framework was not applied as all questions were of an introductory nature.

As seen in our breakdown of questions, it was difficult to find an equal number of questions related to each of the CT categories. In particular, the *tools and resources* category did not map well to the test format, resulting in only six questions representing this category. This imbalance was normalised in the comparison of categories by obtaining a total result as a percentage for each of the CT categories, and measuring these against each other.

The results of the student assessment provide some initial insights that are valuable in our understanding of CT. When dependency analysis was done with chi-square tests, the

results were varied and did not provide a clear impression of the relationship between CT and CS studies. However, the more detailed analysis including demographic information produced some interesting results. There were similarities in the CT scores and CS test scores when students were grouped by gender, selection of computer subjects at school, and CT test performance.

It was encouraging to find that our CTF was successful in classifying pen-and-paper questions with no actual programming, indicating that it serves as more than a programming skills framework and captures more fundamental ideas. The use of questions from the CS Olympiad validates the applicability of the CTF and test in the CS context.

As described in Chapter 4, the CT test produced different kinds of results. The use of the CTF provided a set of quantitative results, while an examination of the individual student responses provided qualitative results, which are better understood in the context of the framework.

The use of a two-phase test provided further insight into the CT abilities of students. The improvement of scores was an encouraging indication of students' abilities to improve these skills at a university level, although this improvement may have been partially caused by prior exposure to the questions before the administration of test two. As a future remedy for this, we propose the preparation of a second test paper. The class would then be divided into two groups, with each group receiving a different test paper for each phase of the test, ensuring that each student sees a different paper for test one and test two.

7.2 CTF and the Development of Rubot

Our second application of the CTF was in the scoring of a computer game, Light-Bot, and the subsequent development of our game, Rubot. The CTF was applied to Light-Bot resulting in an overall CT score of 74%. The assessment used a Likert scale to assign scores within the framework matrix, which were then summed to reach the final result.

The Light-Bot result given by the CTF was satisfactorily high; since the game is built on a programming model it was expected to obtain a high overall score. The results provided some insight into the improvements that could be made to Light-Bot, which were implemented in our design of Rubot. The user responses indicate that these additions were useful and well-received in the context of an introductory programming class.

Despite its successes, the Light-Bot assessment revealed some shortcomings of the CTF. The first of these was the ease with which different CT categories could be evaluated. Some categories, such as *patterns and algorithms* are clearly defined and easier to identify, while others, such as *inference and logic* are more loosely defined and open to interpretation. Throughout this process, cognisance should be given to the CS context when deciding how to interpret the framework.

An additional shortcoming was found in the subjectivity of the assessment. This is attributed to the possibility of different approaches to the same problem, requiring the evaluator to consider the most likely approach, rather than their own preferred approach. The use of the Likert scale is particularly prone to subjectivity; this could be remedied by using a different ranking scale or by collecting scores from a greater number of evaluators and using these scores to find an average ranking for the game.

The CTF was further applied to create a questionnaire to administer to our Rubot users, which was used to gain an understanding of the game's usefulness to them in CT terms. For this exercise, we aimed to avoid CS terminology and ask questions that would be better understood by the students. In the feedback questionnaire the *CT Characteristics* section contained two questions for each of the categories in our framework; however, the questions were not labelled so the respondents could not see the correlation.

The feedback from students reflected a positive impression of using a CT based computer game within a programming course. Most of the students found the game relevant to their studies. The identification of specific features, such as the looping structure, appears to indicate that the CTF can be successful in correlating CT concepts with CS practices.

7.3 CTF and the Strategies and Deliverables

In the final phase of this research, the CTF was used to classify a set of problem solving strategies and puzzle based activities.

The strategy classification began by organising Pólya's problem solving strategies into the six CT categories, which was followed by the inclusion of our own strategies supplementing these. Each of Pólya's strategies was only included in the most relevant category, although some of these strategies could arguably apply to multiple ones. The CTF was reasonably effective in classifying these strategies and allowed us to supplement areas that were not fully discussed by Pólya in more CS appropriate ways.

The classification of puzzles was done to provide an alternate means of practising CT skills, especially for students lacking the technological resources to use our computer game. To classify the puzzles, they were assessed according to which of the aforementioned problem solving skills would be useful in solving the puzzle. This analysis revealed similar results to those encountered in designing the CT test; it was more difficult to map certain CT categories to puzzles owing to an imprecise definition of these terms for the given modality.

7.4 Recommended Improvements

In our discussion of the CTF, shortcomings in a number of areas were identified. Below, a series of recommended improvements are given to address these areas for future use of our framework.

Communication: The framework fails to adequately address principles of communication, which are pervasive in fields like parallel processing and networking. We propose that a new category be added, or alternately the fifth framework category (inference and logic) be adapted to reflect the communication and transfer of information, as well as the existing aspects of information discovery.

Clearer definition of terms: In some contexts, the CTF proved difficult to apply consistently, owing to the ambiguity of terms. To remedy this, more specific terminology should be used to describe the CT categories, which may be facilitated through a re-examination of the literature.

Different applications: Currently, the horizontal axis of the CTF reflects the nature of engagement that students experience as they learn. This axis was not relevant to and therefore not utilised in some of the CTF applications. The CTF could be adapted to use a different series of items on the horizontal axis depending on the problem domain.

Nature of engagement: Following the point above, the focus on the nature of engagement axis in the CTF could be improved. Although this research focused more specifically on introductory CS, the framework could be used to identify appropriate activities over a range of levels.

Scoring method: The scoring method using the Likert scale hinges on the evaluator's perceptions of the activity. A more objective quantifier would be desirable when using the CTF for scoring.

Weighting of CT categories: Throughout this research, the six CT categories have been treated with equal importance. However, this may not hold for all scenarios, as educators or students may find it necessary for certain elements to be highlighted or excluded. In such a case, weightings should be assigned to the categories according to their importance so that the CT score is more reflective of the desired outcomes. The calibration of these weightings could be a basis for future research.

7.5 Summary

The primary underpinning of all of the work in this thesis is the CTF introduced in Chapter 3. In this chapter, we discussed the successes and limitations of the framework within the scope of this research. The framework was found to be suitable for the creation of a student assessment, design of a CT computer game, and classification of strategies and activities, as outlined in the intended applications. A number of difficulties in using the CTF were identified and discussed, and potential improvements to the framework were suggested.

Chapter 8

Conclusion and Future Work

The value of good educational programmes is inarguable, and CT has been identified as a means of improving the current CS education landscape. The pervasive use of computers in a personal and professional context has increased the need for students to develop CS related skills. CT was identified as a preparatory component of programming courses, and as a means to utilise CS skills and techniques in a broader context. Although a number of learning taxonomies exist, they have been found to have varying levels of relevance for CS. The importance of CT is referred to in the ACM and CSTA curricula for high school and undergraduate studies, but few guidelines are given for the practical application of this. Despite the existence of efforts to assess and integrate CT within the CS curriculum, there is a need for an organised approach and relevant, concrete CT materials for use at an introductory CS level.

At the outset of this research, we hypothesised that it would be possible to measure CT in a meaningful way within the context of introductory CS. We believe that this objective was successfully achieved, as discussed below. We have created a framework that functions as a basis for evaluating CT in students and activities and serves as a foundation for our subsequent work. The framework content is based on a literature survey of the field of CT, and the structure draws on existing learning taxonomies. The CTF comprises six skill categories and a progression in the nature of student engagement, arranged in a matrix structure. The CT categories defined are: processes and transformations, models and abstractions, patterns and algorithms, tools and resources, inference and logic, and evaluations and improvements. Following the design, the framework was applied to create CT resources.

The first application of the CTF was the development of a student assessment for incoming university CS students. This assessment provided insight into the CT abilities of our first year CS class when administered at the beginning and conclusion of a semester long CS course. Students who took a computing subject at high school performed better in the CT tests than those who did not, while no significance was found in the scores for different gender groups. The CT scores increased after the semester of CS studies, and the results of the first CT test were found to be a predictor for performance in the CS class tests and June exam. Using the CTF, an evaluation of a computer game, Light-Bot, resulted in a score of 74%, and areas for improvement within the game were identified. These improvements were incorporated into a CT version of the game that we developed, called Rubot. A user study of Rubot was conducted with a foundational computer skills class, which revealed positive responses to the game, particularly in relating the game features to the students' programming studies. Finally, a set of problem solving strategies and practice activities was outlined as a means to practice and increase CT performance.

This work is a general exploration of the field of CT within our introductory CS context, and several opportunities for future work have been identified. A number of limitations and potential improvements of the CTF have been found. Communication was identified as an area that could be integrated into a new framework definition. The need for a clearer definition of terms and a better scoring mechanism was identified. Adaptations of the framework for different applications could be made, and the CT categories could be assigned weightings to suit these applications. The CT student assessment should be conducted longitudinally over a number of years, as well as latitudinally across different institutions, to establish the relevance of these findings in a broader context. Although the use of our computer game, Rubot, was largely successful, a number of extensions have been identified based on user feedback, which could be added in further revisions. Furthermore, it would be interesting to combine the student assessment and the use of Rubot to address specific issues where individual students displayed weakness in CT, and to explore the results of this targeted approach. Although an initial user study with Rubot yielded positive results, it would be beneficial to conduct a formal investigation of the ability of students to transfer general CT skills to programming. Additionally, the work presented here could be used to structure a pilot course for CT within an introductory CS curriculum.

References

[1] ACM, 1998. The ACM Computing Classification System (1998 Version). Online: `http://www.acm.org/about/class/1998` [Accessed 10/12/13].

[2] Aho, A. V. Computation and Computational Thinking. *The Computer Journal 55* (2012), 832–835.

[3] Association for Computing Machinery; IEEE Computer Society, December 2008. Computer Science Curriculum 2008: An Interim Revision of CS 2001. Online: `http://www.acm.org/education/curricula/ComputerScience2008.pdf` [Accessed 13 Nov. 2013].

[4] Astrachan, O., Hambrusch, S., Peckham, J., and Settle, A. The present and future of computational thinking. *SIGCSE Bull. 41* (March 2009), 549–550.

[5] Atherton, J. S., 2011. Learning and Teaching; Bloom's Taxonomy. Online: `www.learningandteaching.info/learning/bloomtax.htm\#Cognitive` [Accessed 15 Nov. 2012].

[6] Bangor, A., Kortum, P., and Miller, J. Determining what individual SUS scores mean: Adding an adjective rating scale. *Journal of Usability Studies 4*, 3 (2009), 114–123.

[7] Barker, L., McDowell, C., and Kalahar, K. Exploring Factors That Influence Computer Science Introductory Course Students to Persist in the Major. *SIGCSE Bull. 41*, 1 (Mar. 2009), 153–157.

[8] Barnes, D., Fincher, S., and Thompson, S. Introductory problem solving in computer science. In *5th Annual Conference on the Teaching of Computing* (1997), University of Kent, 36–39.

[9] Barr, D., Harrison, J., and Conery, L. Computational Thinking: A Digital Age Skill for Everyone. *Learning & Leading with Technology* (March/April 2011), 20–22.

[10] Basawapatna, A., Koh, K., Repenning, A., Webb, D., and Marshall, K. Recognizing computational thinking patterns. In *Proceedings of the 42nd ACM technical symposium on Computer Science Education* (New York, NY, USA, 2011), SIGCSE '11, ACM, 245–250.

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

[12] Blackwell, A., Church, L., and Green, T. The Abstract is 'an Enemy': Alternative Perspectives to Computational Thinking. *PPIG, Lancaster* (2008).

[13] Bloom, B. Bloom's Taxonomy of Learning. Online: `http://www.uc.edu/content/dam/uc/cetl/docs/BloomsLevel.pdf` [Accessed 11 Dec. 2013].

[14] Bovee, C., Voogt, J., and Meelissen, M. Computer attitudes of primary and secondary students in South Africa. *Computers in Human Behavior 23*, 4 (2007), 1762 – 1776.

[15] Brennan, K., and Resnick, M. New frameworks for studying and assessing the development of computational thinking. In *Annual American Educational Research Association meeting* (Vancouver, BC, Canada, 2012), MIT Media Lab.

[16] Bryant, R., Sutner, K., and Stehlik, M., 2010. Introductory Computer Science Education at Carnegie Mellon University: A Deans' Perspective. Online: `http://reports-archive.adm.cs.cmu.edu/anon/anon/home/ftp/2010/CMU-CS-10-140.pdf` [Accessed 5 Nov. 2013].

[17] Bundy, A. Computational thinking is pervasive. *Journal of Scientific and Practical Computing 1*, 2 (December 2007), 67–69.

[18] Byrne, P., and Lyons, G. The Effect of Student Attributes on Success in Programming. *SIGCSE Bull. 33*, 3 (June 2001), 49–52.

[19] Carter, L. Why Students with an Apparent Aptitude for Computer Science Don'T Choose to Major in Computer Science. *SIGCSE Bull. 38*, 1 (Mar. 2006), 27–31.

[20] Cohoon, J., and Tychonievich, L. Analysis of a CS1 Approach for Attracting Diverse and Inexperienced Students to Computing Majors. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2011), SIGCSE '11, ACM, 165–170.

[21] Computer Science Teachers Association. A Model Curriculum for K12 Computer Science: Final Report of the ACM K12 Task Force Curriculum Committee. Tech. rep., Association for Computing Machinery, 2006.

[22] Computing at School Working Group. Computing: A curriculum for schools. Tech. rep., Computing at Schools (CAS) Association; British Computing Society, October 2011.

[23] Curzon, P., Black, J., Meagher, L., and McOwan, P. CS4FN.org: Enthusing students about computer science. *Proceedings of Informatics Education Europe IV, Freiburg, Germany, November* (2009), 5–6.

[24] Curzon, P., Peckham, J., Taylor, H., Settle, A., and Roberts, E. Computational thinking (CT): on weaving it in. *SIGCSE Bull. 41*, 3 (July 2009), 201–202.

[25] Denning, P. Great principles of computing. *Commun. ACM 46* (November 2003), 15–20.

[26] Denning, P. Is computer science science? *Commun. ACM 48*, 4 (Apr. 2005), 27–31.

[27] Denning, P. Computing is a natural science. *Commun. ACM 50*, 7 (July 2007), 13–18.

[28] Denning, P. The profession of IT: Beyond computational thinking. *Commun. ACM 52* (June 2009), 28–30.

[29] Felder, R., and Silverman, L. Learning and teaching styles in engineering education. *Engineering education 78*, 7 (1988), 674–681.

[30] Fisher, A., March 2005. 'Thinking Skills' and Admission to Higher Education. Online: `http://www.cambridgeassessment.org.uk/ca/digitalAssets/113982_Thinking_Skills___Admissions_to_HE.pdf` [Accessed 30 Apr. 2012].

[31] Forehand, M., 2010. Blooms Taxonomy. Online: `http://www4.edumoodle.at/gwk/pluginfile.php/109/mod_resource/content/5/forehand_bloomschetaxonomie02.pdf` [Accessed 25 Nov. 2013].

[32] Fuller, U., Johnson, C., Ahoniemi, T., Cukierman, D., Hernán-Losada, I., Jackova, J., Lahtinen, E., Lewis, T., Thompson, D., Riedesel, C., and Thompson, E. Developing a computer science-specific learning taxonomy. *SIGCSE Bull. 39*, 4 (Dec. 2007), 152–170.

[33] Futschek, G. Algorithmic Thinking: The Key for Understanding Computer Science. In *Informatics Education - The Bridge between Using and Understanding Computers* (2006), R. Mittermeir, Ed., vol. 4226 of *Lecture Notes in Computer Science*, Springer, 159–168.

[34] Galpin, V., and Sanders, I. Perceptions of Computer Science at a South African university . *Computers & Education 49*, 4 (2007), 1330 – 1356.

[35] Goode, J., and Margolis, J. Exploring Computer Science: A Case Study of School Reform. *Trans. Comput. Educ. 11*, 2 (July 2011), 12:1–12:16.

[36] Gouws, L., Bradshaw, K., and Wentworth, P. First Year Student Performance in a Test for Computational Thinking. In *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference* (New York, NY, USA, 2013), SAICSIT '13, ACM, 271–277.

[37] Gouws, L. A., Bradshaw, K., and Wentworth, P. Computational Thinking in Educational Activities: An Evaluation of the Educational Game Light-Bot. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education* (New York, NY, USA, 2013), ITiCSE '13, ACM, 10–15.

[38] Grover, S., and Pea, R. Computational Thinking in K-12: A Review of the State of the Field. *Educational Researcher 42*, 1 (February 2013), 38–43.

[39] Guzdial, M. Education: Paving the way for computational thinking. *Commun. ACM 51* (Aug. 2008), 25–27.

[40] Hambrusch, S., Hoffmann, C., Korb, J., Haugan, M., and Hosking, A. A Multidisciplinary Approach Towards Computational Thinking for Science Majors. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2009), SIGCSE '09, ACM, 183–187.

[41] Henderson, P., Cortina, T., and Wing, J. Computational Thinking. *SIGCSE Bull. 39*, 1 (Mar. 2007), 195–196.

[42] Hu, C. Computational thinking: what it might mean and what we might do about it. In *Proceedings of the 16th annual joint conference on Innovation and Technology in Computer Science Education* (New York, NY, USA, 2011), ITiCSE '11, ACM, 223–227.

[43] Jacobs, C., and Sewry, D. Learner Inclinations to Study Computer Science or Information Systems at Tertiary Level. *South African Computer Journal 45*, 0 (July 2010).

[44] Johnson, C., and Fuller, U. Is Bloom's taxonomy appropriate for computer science? In *Proceedings of the 6th Baltic Sea conference on Computing Education Research: Koli Calling 2006* (New York, NY, USA, 2006), Baltic Sea '06, ACM, 120–123.

[45] The Joint Task Force on Computing Curricula: Association for Computing Machinery and IEEE-Computer Society. *Computer Science Curricula 2013: Final Report 0.9 (Pre-release version).*

[46] The Joint Task Force on Computing Curricula: Association for Computing Machinery and IEEE-Computer Society. *Computer Science Curricula 2013: Strawman Draft (February 2012)*, February 2012.

[47] Kafai, Yasmin and Burke, Quinn. The Social Turn in K-12 Programming: Moving from Computational Thinking to Computational Participation. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2013), SIGCSE '13, ACM, 603–608.

[48] Kinnari, H. A study of the mathematics proficiency. In *Proceedings of the 1st International Workshop on Mathematics and ICT: Education, Research and Applications* (November 2010), I. Mierlus-Mazilu, Ed., Technical University of Civil Engineering Bucharest, 35–39.

[49] Leal, J. Testing the perception of time, state and causality to predict programming aptitude. In *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on* (2013), 721–726.

[50] Lu, J., and Fletcher, G. Thinking about computational thinking. *SIGCSE Bull. 41* (March 2009), 260–264.

[51] Mathematics Learning Study Committe; National Research Council. *Adding It Up: Helping Children Learn Mathematics.* National Academies Press, Nov. 2001.

[52] McGill, M., Settle, A., and Decker, A. Demographics of undergraduates studying games in the United States: a comparison of computer science students and the general population. *Computer Science Education 23*, 2 (2013), 158–185.

[53] McMaster, K., Rague, B., and Anderson, N. Integrating Mathematical Thinking, Abstract Thinking, and Computational Thinking. In *Proceedings of ASEE/IEEE Frontiers in Education Conference* (2010), vol. Session S3G.

[54] Michalewicz, Z., and Michalewicz, M. *Puzzle-based learning*. Hybrid Publishers, 2008.

[55] Mishra, A., and Garg, D. Selection of Best Sorting Algorithm. *International Journal of Intelligent Information Processing 2* (2008), 233–238.

[56] Moursund, D. *Computational Thinking and Math Maturity: Improving Math Education in K-8 Schools (Second Edition)*. University of Oregon, Eugene, Oregon 97403, September 2007.

[57] PACT. Principled Assessment of Computational Thinking. Online: `http://pact.sri.com` [Accessed 27 Nov. 2013].

[58] Papert, S. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc., New York, NY, USA, 1980.

[59] Perković, L., Settle, A., Hwang, S., and Jones, J. A framework for computational thinking across the curriculum. In *Proceedings of the fifteenth annual conference on Innovation and Technology in Computer Science Education* (New York, NY, USA, 2010), ITiCSE '10, ACM, 123–127.

[60] Pólya, G. *How to solve it: A new aspect of mathematical method*. Princeton University Press, 2008.

[61] Qualls, J., and Sherrell, L. Why computational thinking should be integrated into the curriculum. *J. Comput. Small Coll. 25* (May 2010), 66–71.

[62] Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., and Kafai, Y. Scratch: Programming for All. *Commun. ACM 52*, 11 (Nov. 2009), 60–67.

[63] Schaefer, M. Computational Thinking in a Liberal Arts Cryptology Course. Online: `compthink.cs.depaul.edu/documents/finalDrafts/finalDraft_CSC_233.pdf` [Accessed 5 Mar. 2012], April 2009.

[64] Scott, T. Bloom's taxonomy applied to testing in computer science classes. *J. Comput. Sci. Coll. 19*, 1 (Oct. 2003), 267–274.

[65] Selby, C., November 2012. Promoting computational thinking with programming. Online: `http://eprints.soton.ac.uk/346936/1/WiPSCE_ao_soton_eprints.pdf` [Accessed 2 May 2013].

[66] Sengupta, P., Kinnebrew, J., Biswas, G., and Clark, D. Integrating Computational Thinking with K-12 Science Education: A Theoretical Framework. *Proceedings of the 4th International Conference on Computer Supported Education* (2012), 10.

[67] Settle, A., Goldberg, D., and Barr, V. Beyond computer science: computational thinking across disciplines. In *Proceedings of the 18th ACM conference on Innovation and Technology in Computer Science Education* (New York, NY, USA, 2013), ITiCSE '13, ACM, 311–312.

[68] Snyder, L. CS principles pilot at University of Washington. *ACM Inroads 3*, 2 (June 2012), 66–68.

[69] Starr, C., Manaris, B., and Stalvey, R. Bloom's Taxonomy Revisited: Specifying Assessable Learning Objectives in Computer Science. *SIGCSE Bull. 40*, 1 (Mar. 2008), 261–265.

[70] Taylor, V., and Ladner, R. Data Trends on Minorities and People with Disabilities in Computing. *Commun. ACM 54*, 12 (Dec. 2011), 34–37.

[71] Thiruvathukal, G. Computational Thinking... and Doing. *Loyola University Chicago Faculty Publications* (2009).

[72] Thomas, L., Ratcliffe, M., Woodbury, J., and Jarman, E. Learning styles and performance in the introductory programming sequence. *SIGCSE Bull. 34*, 1 (Feb. 2002), 33–37.

[73] Thompson, E., Luxton-Reilly, A., Whalley, J., Hu, M., and Robbins, P. Bloom's taxonomy for CS assessment. In *Proceedings of the tenth conference on Australasian Computing Education - Volume 78* (Darlinghurst, Australia, 2008), ACE '08, Australian Computer Society, Inc., 155–161.

[74] Toth, D. Our experiences incorporating robotics into our service course: poster session. *J. Comput. Sci. Coll. 25*, 6 (June 2010), 256–258.

[75] Weinberg, A. *Computational thinking: an investigation of the existing scholarship and research.* PhD thesis, Colorado State University, 2013.

[76] Wentworth, P. Can computational thinking reduce marginalisation in the future Internet? In *Kaleidoscope: Beyond the Internet? - Innovations for Future Networks and Services, 2010 ITU-T* (2010), 1–5.

[77] Wing, J. Computational Thinking. *Communications of the ACM 49*, 3 (March 2006), 33–35.

[78] Wing, J. Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society* (2008), 3717–3725.

[79] Wing, J., Spring 2011. Research Notebook: Computational Thinking - What and Why? Online: `link.cs.cmu.edu/article.php?a=600` [Accessed 29 Feb. 2012].

Appendix A

CT Test Questions

Question A
What is the next number in the sequence?
6, 9, 12 ...

Question B
In a race, in which position would you be after passing the person in second position?

Question C
Mpho and Lerato are given a bag of Lego bricks. They share them out equally and are left with one remaining brick. How many bricks were in the bag?
a) 58 b) 79 c) 100 d) 276

Question D
The sum of Jack and Jill's ages is 13. What will the sum of their ages be in five years' time?

Question E
Pupil is to classroom as ..... is to library.

Question 1
An enormous sheet of paper, only 0.1 mm thick, can be folded double ten times. How thick will the folded sheet be?

Question 2
Jason is 5 years old. In three years' time Ramone will be twice as old as Jason. How old is Ramone now?

Question 3

Arrange four sixes using the standard mathematical operators (+ - × ÷) so that the answer is 1.

$$6 \quad 6 \quad 6 \quad 6 = 1$$

Question 4

LEAD is to DEAL as 9514 is to .....

a) 9514 b) 9451 c) 4519 d) 4159

Question 5

A frog is at the bottom of a well which is 19 m deep. The frog jumps 4 m up the side of the well, but then needs an hour to recuperate. During this hour the frog slides back 2 m. How many hours will it take the frog to get out of the well?

Question 6

Which number comes next in the sequence?

0, 3, 8, 15, 24, 35, 48, 63 ...

Question 7

One of the following statements is true. Which one?

a) Only one of these statements is false.

b) Only two of these statements are false.

c) Only three of these statements are false.

d) Four of these statements are false.

e) All five of these statements are false.

Question 8

Which of the following *cannot* be paid using only 5c and 7c coins?

a) 23c b) 24c c) 26c d) 77c

Question 9

What is the smallest decimal number consisting of only 1's and 0's that is divisible by 15?

Question 10

A message can be sent as a string of digits using the standard alphanumeric code where A=1, B=2, C=3, etc. Thus 121 could be ABA or LA or AU.
What common English word has the code 2 1 1 2 1 2?

Question 11

If you write down all the numbers from 1 to 100, how many digits have you written down?

Question 12

A man ate 100 bananas in five days, each day eating 6 more than the previous day. How many bananas did he eat on the first day?

Question 13

Long ago, South Africa used Pounds (lb) and Ounces (oz) to measure weight. This system is still in use in some parts of the world. You need to know that 1 lb is equal to 16 oz. Given only mass pieces of 1 lb, lb, lb, 2 oz and 1 oz:

a) What is the largest mass you can measure (in oz)?
b) Which mass pieces would you use to make up 11 oz?

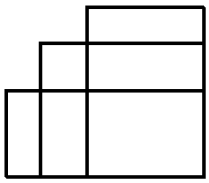
Question 14

Use the mathematical symbols +, -, ×, ÷ once each to resolve the equation. Write the symbols in the correct order:

$$(8 \quad 3 \quad 4) \quad 5 \quad 1 = 5$$

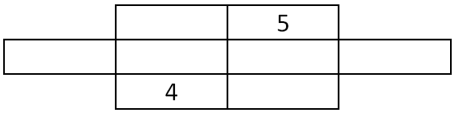
Question 15

How many rectangles can you see in the figure?



Question 16

Write the numbers from 1 to 8 in the squares, so that the squares containing consecutive numbers do not touch each other on a side or on a corner.



Question 17

A girl has an equal number of brothers and sisters, but each brother has twice as many sisters as brothers.

a) How many daughters in the family?
b) How many sons in the family?

Question 18

The example of modular arithmetic with which we are most familiar is time. We get up at 7, but 12 hours later it is 7 again, and 24 hours later it is 7 again. We can write this:
$19 = 7 \pmod{12}$ W [19 has a remainder of 7 when divided by 12] or $31 = 7 \pmod{12}$

We can also write:
$4 = 1 \pmod{3}$ [4 has a remainder of 1 when divided by 3]

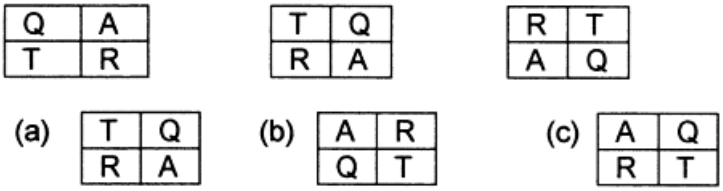
a) Write down the value of $5 \pmod{4}$
b) Write down the value of $13 \pmod{5}$.

Question 19

Z, Z, Y, Z, Y, X, Z, Y, X, W, Z, Y, X, W, ... Which letter comes next?

Question 20

Which square completes the series?

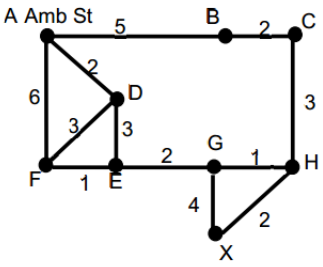


Question 21

Hali-Alu, the new volcano, doubled in height every day. After 30 days, it was as high as Table Mountain. How many days did it take to grow to half the height of Table Mountain?

Question 22

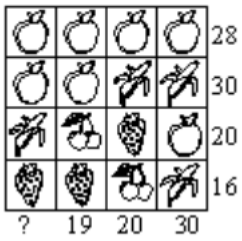
You need to drive your ambulance to X, the scene of an accident, as quickly as possible. Alongside is a map of the town, where A is the ambulance station, and each intersection is marked with a letter B, C, D, etc. The numbers along the road show how long it will take to drive that road.



Find the quickest route to get from the ambulance station to
the accident and indicate it with the letters for the intersections.
A, B, C, D, etc. Also include the total time for the route.

Question 23

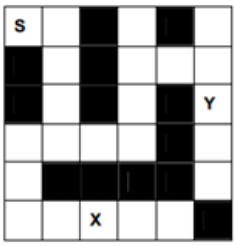
Look at the drawing. The numbers alongside each column and below each row are the total of the values of the symbols within each column and row. What value should replace the question mark?



Question 24

The following maze has two treasures marked as X and Y. Solid blocks show where walls are located, and the clear blocks show where a robot could travel. Your job is to program the robot to walk through the maze, collect the treasure, and bring it back. The commands you can give the robot are:

Fx - move forward x blocks
R - turn right 90
L - turn left 90
T - turn around (same as LL or RR).
P - pick up treasure
D - drop treasure



Initially the robot is at position S and is facing towards the right of the map. The robot can only pick up the treasure if it is on the same square of the map as the treasure. The robot must drop the treasure back at square S.

As an example, here is how the robot would collect treasure X and bring it back:
F1, R, F3, R, F1, L, F2, L, F2, P
T, F2, R, F2, R, F1, L, F3, L, F1, D

What commands would you need to program the robot with for it to fetch treasure Y?

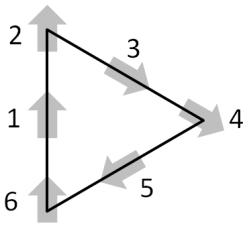
Question 25

We have a robot known as Big Bert (BB). BB always leaves a trail of his movements. He can only turn right for a maximum of 360 degrees, and cannot turn left at all. He can move forward any number of robot steps n, but cannot go backwards. Summary of commands:

T(n) Where n is the number of degrees to turn to the robot's right. Where $n \leq 360$.
M(n) Where n is the number of robot steps to move forward.

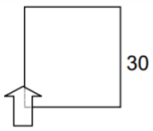
Example: To draw a shape such as an equilateral triangle with a side of 20 steps Big Bert must use the following list of commands.

M20
T120
M20
T120
M20
T120



You can see that BB was instructed to do M20, T120 three times. This instruction can be written more efficiently as 3 (M20 T120).

a) Write down the most efficient list of instructions to make BB draw a square, with the side of the square 30 steps. Remember, the robot must end its journey back in the starting position as indicated by the arrow.



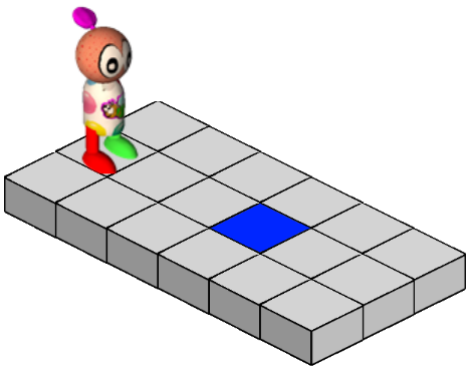
b) Write down the most efficient list of instructions to make BB draw a regular pentagon, with each side having 30 steps. The robot must end its journey in the position in which it started (indicated with an arrow).

Appendix B

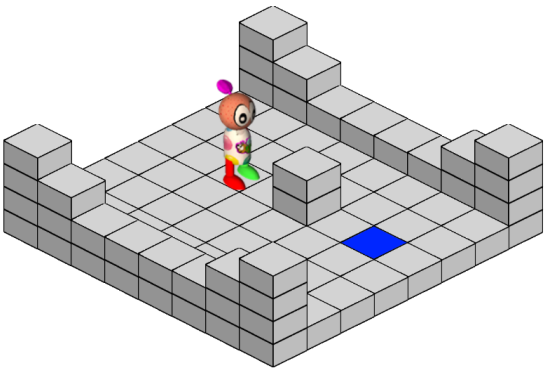
Rubot Levels

The figures below show the Rubot levels that were assigned to students for the user study on a weekly basis.

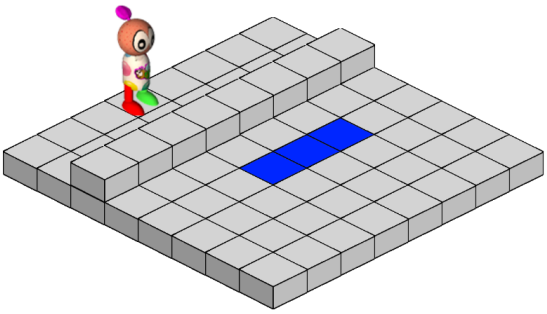
B.1 Week 1



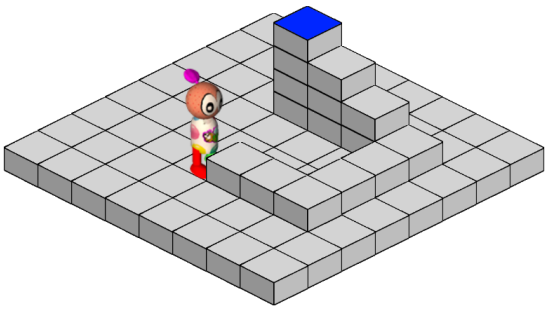
(a) Level 1.



(b) Level 2.

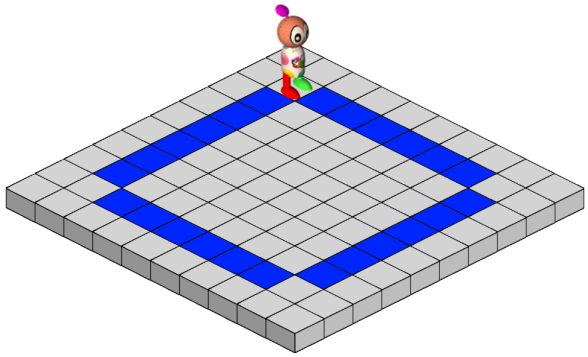


(c) Level 3.

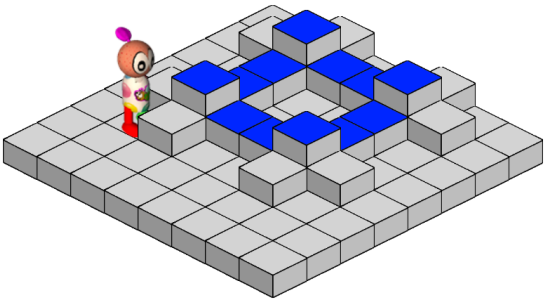


(d) Level 4.

B.2 Week 2

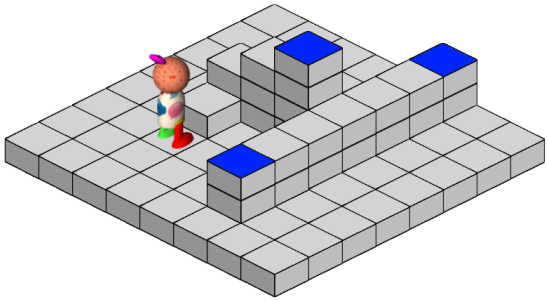


(a) Level 5.

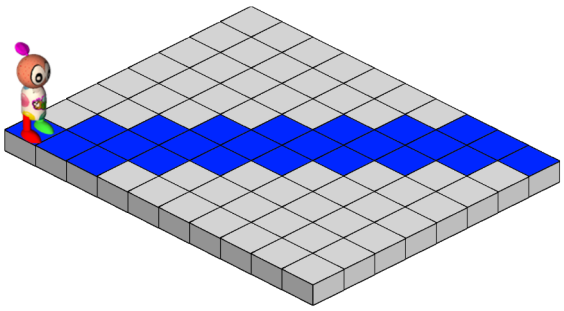


(b) Level 6.

B.3 Week 3

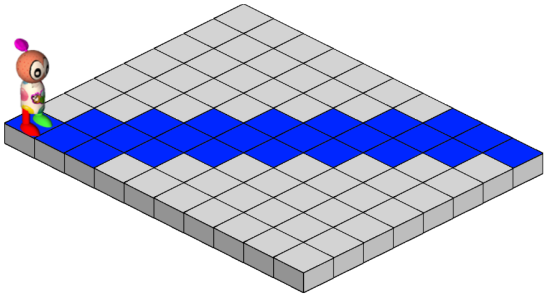


(a) Level 7.

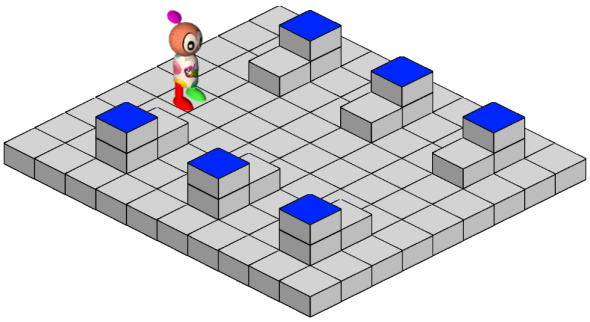


(b) Level 8.

B.4 Week 4

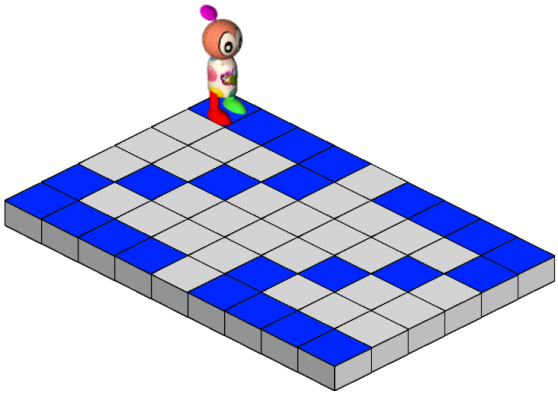


(a) Level 9.

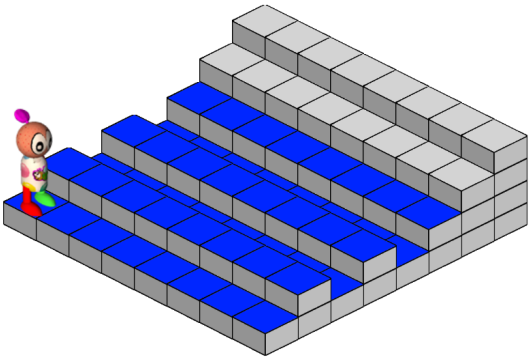


(b) Level 10.

B.5 Week 5



(a) Level 11.



(b) Level 12.

Appendix C

Rubot User Feedback Questionnaire

Section 1: Usability

For each of the questions below, indicate your response in the appropriate column.

	Strongly Disagree				Strongly Agree
	1	2	3	4	5
1. I think that I would like to use this game frequently					
2. I found the game unnecessarily complex					
3. I thought the game was easy to use					
4. I think I would need the support of a (technical) person to be able to use this game					
5. I found the various functions in this game were well integrated					
6. I thought there was too much inconsistency in this game					
7. I would imagine that most people would learn to use this game very quickly					
8. I found the game very cumbersome[1] to use					
9. I felt very confident using the game					
10. I needed to learn a lot of things before I could get going with this game					

List three things that you **enjoyed** about this game.

List three things that you **did not enjoy** about this game.

[1] **Cumbersome:** slow or complicated and therefore inefficient (Oxford Dictionary)

Section 2: Usefulness

For each of the questions below, indicate your response in the appropriate column.

	Strongly Disagree				Strongly Agree
	1	2	3	4	5
Relevance to the Computer Skills course					
1. This game helped me to understand computer skills concepts					
2. The difficulty of this game was appropriate for students in this class					
3. I understood the purpose of this game					
4. I think that future computer skills classes would benefit from playing this game					
Computational Thinking Characteristics					
5. I put a lot of thought into how to approach each level					
6. I often broke problems down into smaller sections that needed solving					
7. I understood how to look at a level and represent the solution using commands					
8. I was able to work effectively with different levels of detail (particularly with functions)					
9. Once I learned how to do something I used this knowledge over again					
10. It was useful to look for patterns in the problem					
11. I put thought into how to solve the levels efficiently					
12. I understood how to use and arrange groups of commands to perform a desired action					
13. I had to be creative to play this game					
14. I developed my own heuristics[2] to make the levels easier to solve					
15. It was easy to find and fix mistakes in my logic					
16. It was important to me to produce a good solution (rather than one that simply worked)					

Any other feedback?

[2] **Heuristic:** a technique designed for solving a problem more quickly, often identified through experience.

Appendix D

Puzzles

These puzzles are taken from Puzzles.com. Each question and the final solution is reproduced below. The URLs for accessing each puzzle are provided at the end of this appendix.

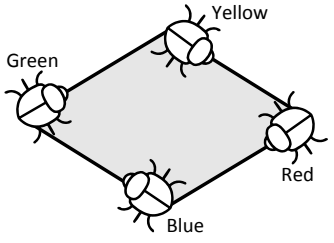
D.1 Bottle and Cork

A bottle costs a dollar more than a cork. Together they cost $1.10. How much does the bottle cost and how much does the cork cost?

Solution: The bottle costs $1.05 and the cork costs 5c; this makes exactly a dollar difference between the bottle and the cork.

D.2 Bugs Traffic

Four bugs, the Green, the Yellow, the Red, and the Blue occupy the corners of a square as shown in the illustration. Each side of the square is 10 units long. Simultaneously, the Green bug starts to crawl directly toward the Yellow one, the Yellow toward the Red, the Red toward the Blue and the Blue toward the Green.



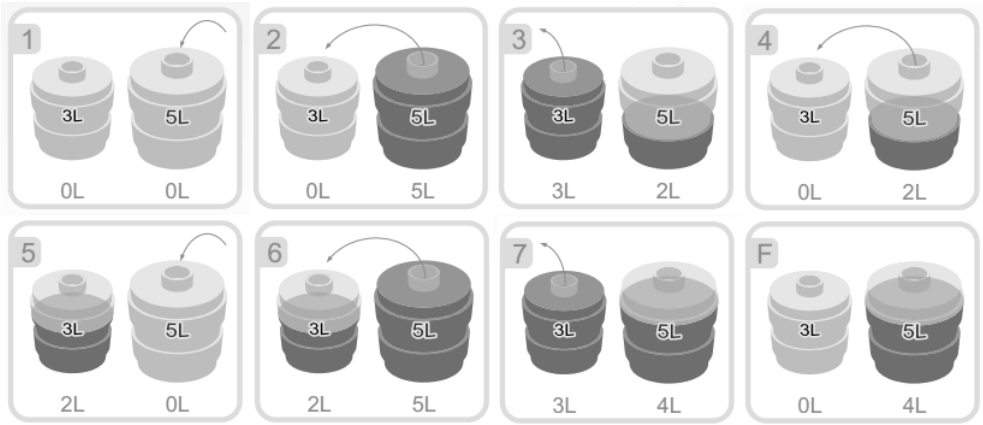
Since all four bugs crawl at the same constant rate, they will describe four congruent logarithmic spirals which meet at the centre of the square. Thus the question is: how far does each bug travel before they meet?

Solution: At any given instant the four bugs form the corners of a square which shrinks and rotates as the bugs move closer together. The path of each pursuer will therefore at all times be perpendicular to the path of the pursued. The length of each spiral path will be the same as the side of the square: 10 units.

D.3 Get 4 Litres

There are a 3L container and a 5L container available. The object is to measure exactly 4L of water with the help of these two containers and some immense supply of water (say, a river or lake). How this can be done?

Solution: The solution for measuring 4L of water in seven steps is shown below.



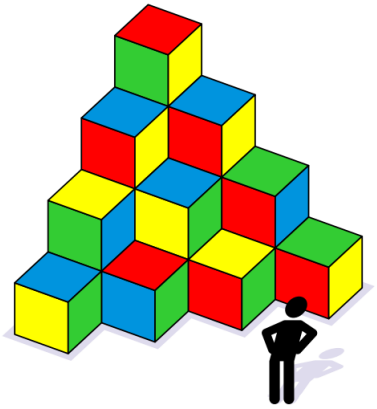
D.4 Kisses and Handshakes

Kent and Hannah invited some of their friends to a dinner. Some friends arrived with their spouses, while some arrived alone. Each guest greeted each of the two hosts as well as each other guest. When two men greeted each other there was handshaking. When two women greeted each other there was kissing. The same was true when a man and a woman greeted each other.

It is known six handshakes and 12 kisses have happened in total. Can you say how many guests arrived at the dinner, how many of them were in couples and how many of them were alone? Obviously, when two guests arrived as a couple they didn't greet each other.

Solution: Three men arrived at the dinner. Two of them arrived with their spouses, and one arrived alone. As a result, six handshakes and 12 kisses have happened.

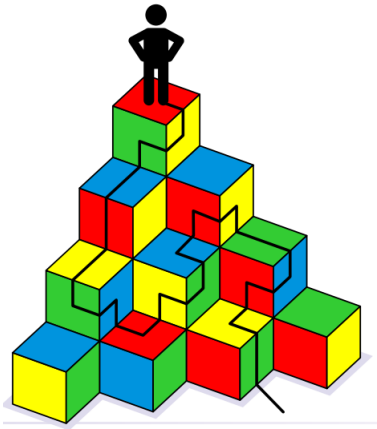
D.5 Rock Climber



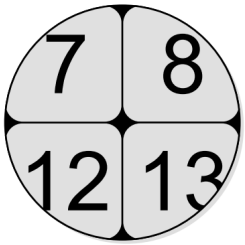
Suppose you are a rock climber, standing at a mountain's foot. The mountain consists of rocks coloured into four different colours, blue, green, red and yellow, as shown in the illustration.

Your objective is to get to the mountain's top, always climbing from one rock to another adjacent rock, but never jumping over the rocks. As you climb to the top your route must contain a consecutive series of four colours constantly repeating. For example, yellow-green-blue-red, and then again yellow-green-blue-red, and so on. How quickly can you get to the top?

Solution:



D.6 Square Size

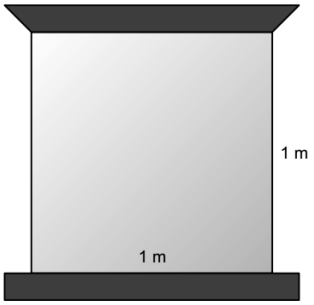


Numbered tiles (1 through N) form a square and are placed in it line-by-line in ascending order starting from its top left corner. A fragment of the square is shown. What is the size of the square?

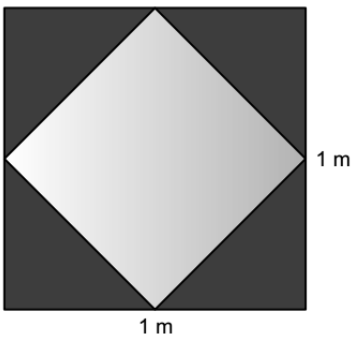
Solution: The numbered tiles form a 5 x 5 square.

D.7 Square Window

A nobleman had a sitting-room with only one window in it: a square window, 1 meter high and 1 meter wide, as shown in the diagram. He had weak eyes, and the window gave too much light. He sent for the builder, and told him to alter the window to give half the light. Only he was to keep it square — the same 1 meter high and 1 meter wide. How did he do it? The builder wasn't allowed to use curtains, or shutters, or coloured glass, or anything of that sort.



Solution:



D.8 Truth Telling

It is known only one character is telling the truth. Mr. April says that Mr. May tells lies. Mr. May says that Mr. June tells lies. Mr. June says that both Mr. April and Mr. May tell lies. Who is telling the truth? Explain your answer.

Hint: Consider whether each character in turn is telling the truth; you will end up with only one possible solution.

Solution: Mr. May is telling the truth. Mr. April lies when he says that Mr. May is lying. Mr. May is telling the truth when he says that Mr. June is lying. Mr. June is lying when he says both Mr. April and Mr. May are lying since one is telling the truth.

D.9 URLs

Bottle and Cork:
`http://www.puzzles.com/PuzzlePlayground/BottleAndCork/BottleAndCork.htm`

Bug Traffic:
`http://www.puzzles.com/PuzzlePlayground/BugsTraffic/BugsTraffic.htm`

Get 4L:
`http://www.puzzles.com/PuzzlePlayground/Get4L/Get4L.htm`

Kisses & Handshakes:
`http://www.puzzles.com/PuzzlePlayground/KissesAndHandshakes/KissesAndHandshakes.htm`

Rock Climber Maze:
`http://www.puzzles.com/PuzzlePlayground/RockClimberMaze/RockClimberMaze.htm`

Square Size:
`http://www.puzzles.com/PuzzlePlayground/SquareSize/index.htm`

Square Window:
`http://www.puzzles.com/PuzzlePlayground/SquareWindow/SquareWindow.htm`

Truth Telling:
`http://www.puzzles.com/PuzzlePlayground/WhosTellingTheTruth/WhosTellingTheTruth.htm`