The Relationship Between Programming Quality and Different Measures of Computational Thinking

Joint PhD

A dissertation submitted in fulfilment of the requirements
for the degree of Doctor of Philosophy

urn:nbn:de:gbv:18-ediss-101131

at the

Universität Hamburg

Fakultät für Erziehungswissenschaft

and

Macquarie University

Faculty of Human Sciences

Submitted

11/07/2019

by

Kay-Dennis Boom

SUPERVISORS

Principal Supervisor:	Jens Siemon (Universität Hamburg)
First Associate Supervisor:	Matt Bower (Macquarie University)
Second Associate Supervisor:	Amaël Arguel (Macquarie University)

Date of defence

28/02/2020

ABTRACT

Computational thinking (CT) is often promoted as the literacy of the 21st century and as the foundation of many concepts in computer science and related fields. Although most research has shown that including CT in computer science education has positive effects on programming, there are conflicting results. These inconsistencies may occur because of different frameworks, which lead to different ways of measuring that kind of thinking. This raises questions about the role of CT in programming, in particular, how is CT applied when solving a programming task (RQ1) and whether different measures of CT can be relevant predictors for programming quality (RQ2).

Based on a literature review from two fields, computer science and psychology, a conceptual framework of CT is developed in this thesis. This conceptual framework builds the foundation for an instrument to observe CT behaviour. In order to answer the research questions, participants worked in pairs (*n* = 27) to solve a programming task in Scratch. The solving process were videotaped and analysed based on CT activities. In addition, participants' Scratch projects were analyses based on programming quality criteria. A set of adjusted Bebras tasks were used as unplugged measure of CT. To control for confounding effects, a measure for nonverbal intelligence was completed by the participants as well.

Results showed that not all CT associated behaviour was equally often apparent while participants were working on the programming task. Participants engaged only infrequently in decomposition or abstract thought about a problem. Instead, of thinking about the problem, they tried to create solutions from the beginning. Correlations and regression analysis also revealed that CT measures differ in their suitability for revealing the relationship with programming quality. Only the behaviour based measure of CT revealed that relationship.

On this basis, it is recommended that educators should focus on different parts of CT in order to enhance that kind of thinking. In order to analyse the unique impact that CT might have on programming, instruments must be chosen with care.

Acknowledgements

Undertaking this PhD study was a life-changing experience and I met many people who influenced me and my work.

I would like to thank my supervisor, Matt Bower, for his patience and guidance through the whole journey. The discussions with him were always a source of great inspiration and encouragement. I am very grateful to had him as my supervisor.

I also would like to thank my associate supervisor, Amaël Arguel, for his critical comments. Because of him, I revaluated my view on specific topics such as intelligence and certain statistical approaches. He reminded me to be always critical about what you think you know.

Of course, many thanks to my principal supervisor, Jens Siemon, who encouraged me to undertake this joint PhD program in the first place. He taught me many things about the research process in general and what it means to have the “big picture” in mind. Without him, this thesis would not have been possible.

I greatly appreciate the support I received from my editor, Robert Trevethan. Thank you, Robert, for all your professional advice.

Many thanks also to my German colleague Sören Schütt and my Australian colleague Mark Gronow. You kept me encouraged through some difficult days and you always had some time spare for a quick office chat. I would not have enjoyed my journey that much without you.

CONTENTS

1 INTRODUCTION 1

1.1 Background and problem statement 1

1.2 Purpose of the study and research questions 4

1.3 Significance of the study 6

1.4 Structure of the thesis 7

2 CONCEPTUAL FRAMEWORK 8

2.1 How to define computational thinking? 8

2.1.1 The missing definition 8

2.1.2 Definition of computational thinking 9

2.2 Core characteristics of computational skills 14

2.2.1 Problem solving in general 14

2.2.1.1 Typology of problems 14

2.2.1.2 Problem-solving models 17

2.2.1.3 Problem-solving in mathematics 19

2.2.1.4 What problem solving means for computational thinking 23

2.2.2 Decomposition 24

2.2.2.1 The role of decomposition in computer science 25

2.2.2.2 The role of decomposition in psychology 27

2.2.2.3 What decomposition means for computational thinking 30

2.2.3 Abstraction 30

2.2.3.1 The role of abstraction in computer sciences 30

2.2.3.2 The role of abstraction in psychology 35

2.2.3.3 What abstraction means for computational thinking 39

2.2.4 Algorithmic design 40

2.2.4.1 The role of algorithmic design in computer sciences 40

2.2.4.2 The role of algorithmic design in psychology 42

2.2.4.3 What algorithmic design means for computational thinking 47

2.3 Relationship of components 47

2.4 Assessment of computational thinking 48

2.4.1 Using unplugged methods 49

2.4.1.1 The Bebras tasks 50

2.4.2 Using visual programming environments 53
2.4.2.1 Scratch 54
2.4.2.2 Comparison between unplugged and plugged methods 57
2.5 The relationship between computational thinking and other concepts 59
2.5.1 Intelligence as general problem-solving skill 59
2.5.1.1 Meaning of abstract thinking in theories about intelligence 64
2.5.1.2 Differences between computational thinking and intelligence 66
2.5.2 Programming quality 68
2.5.2.1 Assessment of programming quality 70
2.5.2.2 Computational thinking and programming 72
2.6 Summary 75
3 METHODS 77
3.1 Research questions 77
3.2 Procedure 77
3.2.1 Phase 1: Online study 77
3.2.2 Phase 2: In classroom programming task 78
3.2.2.1 Scratch programming environment 78
3.3 Justification for video study 81
3.4 Participants 84
3.5 Instruments and measures 85
3.5.1 The Bebras tasks 85
3.5.2 Test of nonverbal intelligence 88
3.5.2.1 Psychometrics and usage in this study 90
3.5.3 Programming quality rubric scheme 91
3.5.3.1 Richness of project 93
3.5.3.2 Variety of code usage 93
3.5.3.3 Organisation and tidiness 93
3.5.3.4 Functionality of code 94
3.5.3.5 Coding efficiency 95
3.5.3.6 Weighted score of sum and reliability assessment 96
3.5.4 Dr Scratch 97
3.5.5 Computational thinking behaviour scheme 98
3.5.5.1 Computational thinking components 99
3.5.5.2 Decomposing 99
3.5.5.3 Abstraction I – neglecting details 100

3.5.5.4 Abstraction II - recognising patterns 100
3.5.5.5 Designing and applying algorithms 100
3.5.5.6 Reliability assessment 103
3.6 Pilot study 104
3.7 Data analysis approach 106
3.7.1 Units of analysis 106
3.7.2 Addressing research question 1 106
3.7.3 Addressing research question 2 107
3.7.4 Statistical analyses 108
3.8 Research ethics approval 108
4 RESULTS 109
4.1 Overview of measures 109
4.1.1 Bebras tasks 109
4.1.1.1 Individual scores as the unit of analysis 109
4.1.1.2 Paired scores as the unit of analysis 110
4.1.2 Test of Nonverbal Intelligence 111
4.1.2.1 Individual scores as unit of analysis 111
4.1.2.2 Paired scores as unit of analysis 111
4.1.3 Programming quality 112
4.1.4 Dr Scratch 113
4.2 Answering the first research question 114
4.2.1.1 Lag sequential analysis of computational thinking behaviour 120
4.3 Answering the second research question 123
4.4 Additional results 125
5 DISCUSSION 126
5.1 Summary of the study 126
5.2 Discussion of the first research question 127
5.2.1 No or only barely abstract thinking 127
5.2.2 Rushing to the solution 129
5.2.3 Some prior mathematical knowledge required 131
5.3 Discussion of the second research question 132
5.4 Practical implications 136
5.4.1 Problem solving 137
5.4.2 Decomposition 138

5.4.3 Abstraction 140
5.4.4 Algorithmic design 142
5.5 Critical evaluation of the study 143
5.5.1 Methodological 143
5.5.1.1 Research design 143
5.5.1.2 Instruments and measures 145
5.5.2 Conceptual consideration 148
5.5.2.1 Limitation of the operationalisation 148
5.5.2.2 Computational thinking itself 151
5.6 Future work 153
5.7 Conclusion 155
6 REFERENCES 157
7 APPENDIX 181

1 INTRODUCTION

1.1 Background and problem statement

With steadily decreasing costs of data collection, storage, and processing on the one hand, and constantly increasing computer power on the other, digitalisation continues to shape our everyday lives (Organisation for Economic Cooperation and Development [OECD], 2017, p. 6). This is not without an impact to our society. The job market is developing so quickly that, in 2017, researchers from the Institute for the Future (IFTF) estimated that by 2030 up to 85% of today's school children will work in jobs that have not yet been created (IFTF, 2017, p. 14). Because no other areas are growing more quickly than are science, technology, engineering, and mathematics (STEM), it is likely that most of these jobs will be in these and related fields (OECD, 2016b).

This development also shifts the demand for required skills in two ways (OECD, 2016b). First, ICT-related skills will be in greater in demand. This includes the need for ICT specialists such as programmers. Technology is quickly blending in more jobs than previously, so generic ICT-related skills have become more relevant for many different areas (Burning Glass, 2014). Programming-related skills have relevance for jobs that have not previously been related to programming. Medicine, academia, and product management all rely on technology to some extent. Second, with the development of digital technologies, automation of labour is increasing (Autor, Levy, & Murnane, 2003). This does not influence all kinds of jobs to the same extent, however. Michaels, Natraj, and van Reenen (2014) stated that workers such as bank tellers and paralegals who perform routine tasks have decreased in demand in the past decades whereas nonroutine jobs are increasing. Employees should be able to generate and process complex information, think critically, and be flexible with new or ambiguous situations and open-ended problems (OECD, 2016a, 2016b).

Several authors have proposed that computational thinking (CT) can address this shift in demand for the required skills (see, e.g., Denning, 2009; Falkner, 2016; Swaid, 2015; Wing, 2006). A definition of CT is yet to materialise, and one of the tasks of this thesis is to clarify its scope and boundaries (see Chapter 2). For this introduction, CT can be regarded as the ability to reformulate problems in ways that computers can then be used to help in solving those problems (ISTE and CSTA, 2011). It is also seen as an

umbrella term for different kinds of (sub)skills such as the ability to decompose a problem, engage in abstract thinking, and design algorithmic solutions. These abilities are closely related to programming and are crucial for STEM-related fields. Swaid (2015) saw CT as the core aspect of STEM and recommended that educators include CT in their lectures.

Lu and Fletcher (2009) went even one step further and described CT as the underlying understanding of programming and as something that should be taught before programming. They compared CT with basic skills in different areas. In English, for instance, learners first encounter basic language proficiency before writing an essay or discussing Shakespeare. In mathematics, basic arithmetic builds the foundation for more advanced mathematical approaches such as stating a proof. According to Lu and Fletcher, the same is true for programming and CT in which CT sets the foundation for programming and related skills in computer science (CS). Teaching CT means preparing for the new generation of programmers to fill gaps in the job market.

Although CT has its origins in computer science, it is not bounded to only that field. Wing (2006) proposed CT as a fundamental skill not just for computer scientists but for everyone. Some scholars associate CT with an attitude of handling uncertainty and see it as a powerful tool especially for handling complex and open-ended problems (Barr & Stephenson, 2011). The taskforce on CT of the Computer Science Teachers Association (CSTA) provided several examples of how different areas—from biology to history—can benefit from including CT in their curricula (CSTA, 2011). Williamson (2016) even linked the ability to think computationally with the effectiveness of political participation in the future.

This is why many scholars perceive CT to be the literacy of the 21st century and something that should be taught from an early age (Bocconi et al., 2016; Gretter & Yadav, 2016; Tabesh, 2017). Indeed, CT has been considered in national curricula and has become more relevant in a number of countries, including some countries in the EU (Bocconi et al., 2016), Switzerland (Repenning, 2015), and Australia (Australian Curriculum, Assessment and Reporting Authority [ACARA], 2012).

Because CT is promoted by so many authors as a versatile tool, and because it is included in a variety of national curricula, it is important to investigate its role when people are solving complex problems. It is seen as the foundation of programming and

related skills. This is why the investigation of the relationship of both domains is so important.

Lye and Koh (2014) reviewed 27 articles about how programming in K-12 and higher education is implemented. Because there is no generally accepted definition of CT, these articles relied on differing perspectives and measures of CT, and this has led to inconsistent results. In general, however, Lye and Koh concluded that CT has positive effects on programming and could be used in regular classrooms. Moreover, Grover et al. developed the Foundation for Advancing Computational Thinking (FACT) for K-12 pupils, in which CT was used to promote programming (Grover, 2017; Grover, Pea, & Cooper, 2015). Although it must be noted that Grover used the term CT broadly and did not sharply differentiate CT from other elements in her approach, she concluded that CT was used effectively to enhance programming skills.

In contrast, Araujo, Santos, Andrade, Guerrero, and Dagiene (2017) saw the relationship between CT and programming more critically. They used the Bebras task as a measure for CT. Correlations between the CS students' performance in a set of Bebras tasks and their grades were only low to moderate. Moreover, performance in the Bebras task did not improve after students had been exposed to a programming course. It is possible that these conflicting results are caused by varying frameworks and ways of measuring CT.

Lye and Koh (2014) also suggested that future studies should be using thinking-aloud protocols and capturing on-screen programming activity to have a more in-depth perspective of the actual role of CT. This was partially done by Falloon (2016), who conducted a video study to investigate the impact of CT on an open-ended computational task. He recorded primary school students who worked in pairs on that kind of task. Results indicated different CT patterns, but the role of CT for programming was not further investigated. It is possible that different patterns of CT behaviour have different impacts on programming. This raises the question whether these thinking patterns are dependent on age and whether thinking patterns can be associated with programming quality.

To provide an answer to this, Wu, Hu, Ruis, and Wang (2019) conducted a similar video study, but with students who were enrolled in an educational technology major. The students worked in a collaborative setting on a programming task. Based on their performance, participants were divided into two groups (low versus high performing)

and their CT behaviour was analysed. Results revealed that the low performance group appeared to be tinkering around whereas the high-performance group worked more systematically. However, it is unclear how Wu et al. took into account group performance having been based on pairs.

Computational thinking is proposed as an important problem-solving approach for contemporary society and development of the world's workforce. However, it is still unclear what kind of role CT plays when people solve programming problems. Prior research about CT has not clearly distinguished between CT as a problem-solving approach and programming skills, and there has not been control of any confounding effects. This may be due to the various conceptual frameworks about CT, which also leads to different ways in which it is measured.

Nevertheless, despite the lack of a unified definition of CT, its significance is rather evident. Considering that CT has a border and general frame, it is a valid fundamental skill not only for computer users, but also for everybody, believed to take place in the basic skills (reading, writing and arithmetic) used by everyone in the near future (Wing, 2006). Consequently, increasing numbers of researchers have been paying attention to CT, including, experts in the field of educational technology who have emphasized the importance of CT as 21st century skillset (Korkmaz & Bai, 2019).

1.2 Purpose of the study and research questions

In recent years, there have been efforts to measure CT skills, abilities, knowledge, competencies (Korkmaz & Bai, 2019). Measuring computational thinking is particularly important for the K-12 practice field that serves as the foundation of CT training activities and the evidence of the training results. However, there is no widely accepted standard for measuring CT, except for a CT Scale by Korkmaz, Çakir and Özden (2017), a 29-item CT scale that measures five factors, namely, creativity, cooperativity, algoritmic- critical thinking and problem solving. This scale adopts multidimensional and hierarchical setting methods, as well as, certain content elements of science of computer and problem solving process. Notably, CT is divided into nine dimensions, namely, data collection, data analysis, data presentation, problem, decomposition, abstract, automation, simulation and parallel algorithm and process (Korkmaz & Bai,

2019). The problem at hand is that there seems to be no variables for measuring programming quality relative to CT, such as in a form of scale or questionnaire.

According to Wing (2006), CT requires extraction and decomposition in comparison to great complex systems or processes. These processes aid in selecting convenient representations for solving a problem or modeling in the parts related to the problem. Moreover, digital age individuals are expected to possess CT skills but currently, there is a dearth in evidence and knowledge about the extent to which these skills should be had, and the specific levels that allow adequate CT skills. These matters can be known through appropriate measurement and assessments.

Wing (2008) posits CT complements thinking in mathematics and engineering and focuses on designing systems that aid in solving complex problems humans face (Wing, 2008, Lu & Fletcher, 2009). The core CT concepts encompass (a) abstractions that serve as mental tools for computing and essential for solving problems, (b) layers or problems that have to be solved on different levels, and (c) relationships between layers and abstractions (Wing, 2008). The concept of abstraction and the ability to deal with different levels of abstractions, as well as to think algorithmically and understand the consequences of scale of big data, are fundamental to CT (Denning, 2009, Lu & Fletcher, 2009). Aho (2012) further explains that CT entails "thought processes involved in formulating problems so their solutions can be represented as computational steps and algorithms" (p. 832). On the other hand, according to Denning (2009), CT traces its history in computer science in 1950s when it was called algorithmic thinking, referring to "a mental orientation to formulating problems as conversions of some input to an output and looking for algorithms to perform the conversions" (p. 28). However, there are certain computer science educators who contend that programming is not essential in the teaching of CT (Lu & Fletcher, 2009). Lu and Fletcher (2009) even suggests that emphasizing programming could discourage students from getting interested in computer science. Overall, CT conceptually refers to "systematically, correctly, and efficiently process information and tasks" to solve complex problems (Lu & Fletcher, p. 261). It is important to note that despite the many albeit fragmented descriptions and definitions of CT, there is certainly a dearth in knowledge about how it should be measured especially in the context of programming and programming quality.

Thus, for the present study, a programming task was designed in which participants worked together collaboratively. Different measures with different perspectives on CT

were used to observe what kind of instruments can predict programming quality most effectively. A test of nonverbal intelligence was used to control for potential confounding effects. In order to define the problem, the following two research questions were asked:

RQ1: *How is computational thinking applied when solving a programming task?*

RQ2: *Can multimodal measures of computational thinking be relevant predictors for programming quality?*

1.3 Significance of the study

There are three anticipated contributions that this study could make. First, researchers could benefit from the framework developed in this research because it helps to have a more precise understanding of CT. Even after more than a decade of intensive research about CT, there is no final agreement concerning what CT looks like. Researchers in different studies tend to refer to the same concepts but with different terms. The framework developed in this thesis is an attempt to reduce the confusion by defining what the terms mean in the context of CS. Because CT is considered to be a specific problem-solving approach and is therefore associated with specific cognitive concepts, psychological perspectives are considered as well, for instance, what it actually means to think abstractly. Second, use of different CT measures and an instrument for a theoretically close concept help to further shape CT as a construct. Because CT is not clearly defined, different measures can lead to different conclusions about CT. Using several different measures based on different frameworks helps identification of the facets that each measure focuses on. This will help researchers to choose the most appropriate instrument(s) for their research. Third, this study could be beneficial for CS educators who teach CT in order to promote programming. The results show what kinds of CT-associated behaviour might be most relevant for programming. The results also help to identify deficits in CT behaviour that can help educators focus on appropriate aspects of CT behaviour.

1.4 Structure of the thesis

This thesis comprises five chapters. Following this introductory chapter, in Chapter 2, the conceptual framework is developed. In that chapter, reasons are considered about why there is no final definition but rather a general but vague agreement about CT and its most highly related components and skills. This general agreement will be identified based on different major works concerning the definition of CT. Furthermore, CT and its associated skills will be analysed from computer science as well as psychological perspectives to develop an action-based framework related to the assessment of CT. This should allow assumptions to be considered about how the different skills are related to each other. The chapter closes with presentation of a potential relationship with theoretically closely related concepts concerning nonverbal intelligence and programming (quality) and a short overview of how the research questions will be answered.

Chapter 3 provides insight into the methods of the study. This includes an overview of the research design and procedure, but also provides justification for a video-based study having been seen as the best choice to address the research questions. Demographic information about the participants is presented. The instruments used to assess CT, nonverbal intelligence, and programming quality are presented as well as the results of the previously conducted pilot study. Finally, a detailed outline is provided about how the data were analysed and what implications were to be considered.

The results are presented in Chapter 4. First, a general and descriptive overview of all measures is given to provide a holistic view of the results. Then, the two research questions are answered and further findings are presented.

In Chapter 5, the discussion chapter, the results from Chapter 4 are discussed in relation to the conceptual framework developed in Chapter 2. This chapter also includes a critical evaluation and consideration of the limitations of the study. Based on the theoretical interpretation and limitations, suggestions are made for future research. The chapter closes with conclusions relevant the whole study.

2 CONCEPTUAL FRAMEWORK

2.1 How to define computational thinking?

Computational thinking has been widely and intensively discussed. So far there is no universal agreement concerning a definition. Therefore, the development of a definition of the construct as used in this thesis is presented in the following.

2.1.1 The missing definition

Since its first major appearance in 2006 by Wing, there has been much discussion about what is actually meant by CT. Wing (2006) described CT broadly as a general attitude rather than providing concrete examples concerning what CT is and what it is not. In the subsequent years, different authors proposed a variety of definitions and perspectives about what CT is and what components characterise it. Some authors have proposed a broad description of CT (Guzdial & Wing, 2011; Hu, 2011). However, most authors (e.g., Aho, 2012; Barr & Stephenson, 2011; NRC, 2010) have emphasised the need for a clear and distinctive description on the basis that precise use and understanding of the terminology of a concept is crucial to communicate ideas clearly with other people. Different kinds of definitions also lead to different ideas about how to measure such concept. Only a definition of a concept that most scholars can agree on makes standardised assessment possible. An agreed-upon definition is important for two reasons. First, results from different studies can be compared with each other, which makes further research possible. Second, standardised assessment facilitates monitoring of a concept. This is especially important for education where CT is often praised as a new literacy of the millennium as mentioned in the introduction. As participants of the US National Research Council (NRC) on CT concluded, if CT is part of a curriculum it requires assessment, and without agreement on a common definition it is difficult to develop appropriate tools for assessment (NRC, 2010, p. 57). This is also why it is important to try to make clear what CT means in this study.

So far, tremendous work has been done in order to define CT. Nonetheless, new and alternative definitions of CT continue to appear. Regardless of the scope of a study (e.g., assessment of CT or its application in education), many researchers propose their own definitions and often use different terms for the same concepts. Kalelioğlu,

Gülbahar, and Kukul (2016, p. 591) stated that even 10 years after Wing's seminal paper on CT there is still no commonly accepted definition of CT that has been "scientifically proven". A reason for that might be the lack of a conceptual framework that could not only explain *what* kinds of skills can be seen as core concepts for CT, but also *why* and *how* those skills are used. Such a theoretical framework could be used as a solid foundation for further work on the concept and might have implications for standardised assessment and, furthermore, for development of curricula about CT.

2.1.2 Definition of computational thinking

In order to provide a clear and precise definition of CT, a conceptual framework about CT should comprise three steps. First, CT should be defined and its core components should be identified based on (1) systematic literature reviews and (2) major publications that summarise the opinions of experts. Although systematic reviews can already provide a valuable overview of a generally accepted consensus within a community, conclusions based on experts' perspectives can provide an additional level of content validity (see, e.g., Newman, Lim, & Pineda, 2013; Zamanzadeh et al., 2015) about a construct such as CT, which is important for its assessment. This means that results based on workshops and task forces by distinguished experts in the field of computer science (education) as well as surveys of experts should be considered. In a second step, CT and its associated skills should be analysed from a CS-related point of view. There is no doubt that the origin of CT lies in the field of CS but little work has been done to provide explicit examples of how CT components are rooted in CS. As a result there is no resolution about what decomposition mean in CS, the shape of abstraction, and the role that algorithms play. Making these concepts clear would provide reasons why specific skills are mentioned most often in the literature and by experts. Third, CT should be analysed from a psychological point of view. CT is often proclaimed as the ability to "think like a computer scientist" (Wing, 2006, p. 35), which emphasises CT being foremost a cognitive ability along with its associated skills. However, since its first appearance, it seems unclear what this actually means. There are uncertainties about how a problem is decomposed, how abstraction works and what it means to think algorithmically. Scrutinising these skills would help us understand how they are applied and what they look like in concrete situations.

To shed light on these concepts from different perspectives (e.g., what abstraction means in CS and what it means in psychology) also helps in identifying when

researchers use different terms but actually refer to the same concept and how different concepts might be related to each other. For example, Grover and Pea (2013) use decomposition and modularisation interchangeably when referring to structuring problems with CT. Angeli et al. (2016) used the term generalisation which would be described as pattern recognition in other works. Modelling and models are often associated with abstraction in the sense of models are abstract representations of the world (Hu, 2011). Abstracting and modelling are sometimes even used interchangeably (Denning et al., 1989).

There are some frameworks that have been used to, at least partially, analyse CT from proposed perspectives. For instance, in a short overview, Barr and Stephenson (2011) linked some typical CT skills (e.g., decomposition and abstraction) and their meaning in different fields such as CS. However, this overview consisted of only a few keywords, so a deeper investigation of the relationship between CT-associated skills and their origin in CS is not possible. Others have referred to CT-associated skills as “mental tools and concepts from computer science” (NRC, 2010, p. 3) or have declared that “CT is the basic principle of computing science” (Shi, Liu, & Hendler, 2014, p. 2512) and explained in a general sense how CT and its components are important for grasping CS concepts. However, again there was no detailed explanation concerning which components were being referred to and how they are specifically linked to CS. Another promising attempt was made by Kramer (2007), who explained why abstraction is a key concept in CS. His work is often referred to in order to provide reasons why the ability to abstract is one of the core skills in CT. Kramer linked abstraction as used in CS with its meaning in other fields such as art and cognitive development. Nonetheless, even this often-cited publication only scratches the surface of abstraction from a cognitive psychological perspective. In conclusion, there are some works that link CT skills with CS and psychological concepts, but in most work the analysis is insufficiently deep to provide a sound foundation for further research, for example, concerning standardised measurements.

For the purpose of defining CT in this thesis, four systematic literature reviews about CT were selected by the investigator. These publications were the most recent ones at the time this study was conducted. These reviews were (in chronological order) Selby and Woollard (2014), Kalelioğlu et al. (2016), Bocconi et al. (2016), and Shute, Sun, and Asbell-Clarke (2017). All reviews contained the search terms “computational thinking” in different databases, as shown in Table 2.1. Bocconi et al. (2016, p. 9) did

not specifically state which databases were searched, but they mentioned “a wide range of data sources, including both academic and grey literature (e.g., journal papers, reports, blogs, etc.)”. They also analysed MOOCs and grassroots initiatives, and they surveyed ministries of education to obtain official documents (e.g., policy strategies and national reports), all with regard to CT. In addition to their literature review, they interviewed 14 policy makers, researchers, and practitioners from nine different countries. The total number and kind of documents in the reviews is shown in

Table 2.2. Selby and Woollard (2014) did not specify the total number, but according to their reference list they accessed more than 35 documents. Conclusions in all reviews were based on consistency of usage and interpretation across the retrieved literature, and those conclusions largely indicated consensus in the community at the time of the study with regard to CT.

Table 2.1

Overview of Databases in Review Articles

Databases	Reviews
ACM Digital Library	Selby et al. (2014); Kalelioğlu et al. (2016)
Compendex	Selby et al. (2014)
EBSCOHOST	Kalelioğlu et al. (2016)
Engineering Village	Selby et al. (2014)
ERIC	Selby et al. (2014); Shute et al. (2017)
Google Scholar	Selby et al. (2014); Shute et al. (2017)
IEEE Explore	Selby et al. (2014); Kalelioğlu et al. (2016)
JSTOR	Shute et al. (2017)
PsycINFO	Selby et al. (2014); Shute et al. (2017)
Science Direct	Kalelioğlu et al. (2016)
Springer	Kalelioğlu et al. (2016)
Web of Sciences	Selby et al. (2014) ; Kalelioğlu et al. (2016)

Table 2.2

Numbers and Kind of Documents Used in Reviews About CT

Review	Number and kind of documents
Kalelioğlu et al. (2016)	125 articles
Bocconi et al. (2016)	> 350 articles published in conf. proceedings or journals > 210 documents identified as grey literature 3 curricula documents from England, France, and Finland 4 policy documents 12 grassroots initiatives and MOOCs > 30 policy papers
Shute et al. (2017)	45 articles

As major works that reflect the opinion of computer science (education) experts, four publications were selected. These were (in chronological order) by the NRC (2010), Barr and Stephenson (2011), ISTE and CSTA (2011), and Corradini, Lodi, and Nardelli (2017). In February 2009, the NRC conducted a 2-day workshop with 37 experts including Peter Denning, Roy Pea, Mitchel Resnick, and Jeannette Wing in order to define the scope of CT. Their final report (NRC, 2010) is widely seen as one of the benchmarks in the field because it is repeatedly mentioned throughout the literature. A similar often-mentioned publication is the comprehensive article by Barr and Stephenson (2011) that summarises the opinions of 26 "thought leaders" (not specifically identified) of the Computer Science Teacher Association (CSTA) and the International Society for Technology Education (ISTE). In the same year, CSTA and ISTE also conducted a (joint) survey in order to find an operational definition of CT and to gather feedback from nearly 700 computer science teachers, researchers, and practitioners. A similar approach was used by Corradini et al. (2017) who analysed responses from nearly 1,000 teachers in an online survey concerning what they believed CT to be.

As a result of the analysis of the proposed literature reviews and expert surveys, CT is defined in this thesis as a problem-solving approach that includes three skills identified as core concepts of CT. These skills are the ability to decompose a problem, the ability to engage in abstraction, and the ability to understand and design algorithms.

For a summary of the analysis, see Appendix A. All works made clear that CT is not necessarily limited to these skills but they are consistently mentioned throughout the literature and by experts and therefore identified as particularly relevant for CT.

2.2 Core characteristics of computational skills

In the following sections, the core characteristics of CT are analysed from both CS and psychological perspectives. In addition, didactical approaches in fields with a long tradition of problem-solving teaching will be presented. First, a general overview about problem solving will be presented, before decomposition, abstraction, and designing algorithms will be discussed.

2.2.1 Problem solving in general

In this section, a general definition of problem and different kinds of problems are presented. Based on that, the different problem-solving approaches and how to teach them are discussed. After that, a conclusion will be drawn about what this means for CT.

2.2.1.1 Typology of problems

The general ability to solve problems depends to some extent on the kind of problem at hand. Therefore, definitions and a categorisation of problems are presented first. Problems come in different forms and it is difficult to find a general definition for all kinds of problems. Yet, in the middle of the last century, Gestalt psychologist Karl Duncker (as cited in Gilhooly, 2012, p. 2) offered a definition that has remained suitable for most kinds of problems: “a problem exists when a living organism has a goal but does not know how this goal is to be reached.” Jonassen (2000) offered a similar definition in stating that a problem has two critical attributes. First, there is a noticeable difference between two situations: the current moment and a goal. Second, there is a social, cultural, or intellectual value in eliminating this difference. This means that if there is no one who perceives the difference between those two states or there is no one who is willing to eliminate it, there is any problem. Although different terms may have been used, this view about the definition of a problem is shared by several other authors of the field (e.g., Anderson, 2015, p. 183; Bransford & Stein, 1993, p. 7). In summary, a

problem can be generally thought of as a situation that is interpreted by someone as a challenge to be overcome.

A further analysis of problems is made by Reitman (as cited in Gilhooly, 2012, p. 3), who pointed out that most, if not all, problems can be portioned into three states: initial state, goal condition, and a set of various actions to transform the problem from the starting state to the goal condition. If all states and actions of a problem are specified, the problem can be classified as being well defined; if not all of the states and actions of a problem are specified, that problem is classified as being ill defined. It is important to state that the word specified does not mean the problem is familiar to the person who faces it. Specified means that there is a clear and unambiguous state of start, goal, and transition that transforms one state to another. Completely well-defined problems are relatively rare; rather, they are the scope of formal sciences or can be seen in forms of games. In addition, the term well defined should not be confused with easy to solve. For example, chess is a well-defined problem with a clear starting and goal conditions and specified rules that determine which actions are permitted in order to achieve the goal. That does not mean, however, that winning a game of chess is easy. On the other hand, ill-defined problems are most likely difficult to solve because of their ambiguous nature. The goal is always to specify as many steps as possible.

A slightly different classification of problems is made by Jonassen (1997) who distinguished between well- and ill-structured problems. For Jonassen, well-structured problems present all elements of a problem to the problem solver, require the application of a finite number of well-structured rules, and have comprehensible solutions where the relationship between decision choices and all problem states is known. Ill-structured problems, on the other hand, possess elements that are (at least partially) unknown to the problem solver; furthermore, multiple solutions are possible and there are multiple criteria for evaluating the solution. Also, some ill-structured problems may require judgments or expression of personal opinions or beliefs about the problem from the problem solver. So the distinction between well- and ill-structured problems can be seen in the degree of knowledge the problem solver has for every problem state. To put it simply, a well-structured problem has a well-known initial state, a well-known goal state, and a well-known limited number of logical operations to close the gap between initial state and goal state—and the opposite pertains for ill-structured problems. Jonassen also emphasised that the structure of a problem should be seen as continuum, with well and ill as end poles, rather than as a dichotomous categorisation.

Examples of different kind of problems, according to Jonassen, and their level of structure are seen in Table 2.3.

Table 2.3

Jonassen's (1997) Typology of Problems

Level of structure	Kind of problem	Description
Rather ill ← → rather well	Logical	Abstract tests of reasoning; examples are Rubric's Cube or Tower of Hanoi
	Mathematical	Algorithmic procedures in mathematics such as equation factoring or long division
	Math story	Mathematical problems embedded in stories
	Rule-using	Problems with several correct solutions in which the solver needs to choose the "best" one; examples are tax returns or some card games such as Bridge
	Decision making	Similar to rule-using but better solutions are less obvious. Different options results in different consequences. Jonassen refers to "life decisions" as decision making problems.
	Trouble shooting / diagnosis	Eliminating problems from a running system such as debugging in programming or fixing a car in mechanics
	Strategic performance	Problems that demand high situational awareness and flexibility in the process of handling such as combat missions or tactics in some sports games
	Case analysis	Analysing highly extraordinary cases for that domain; common in law or medicine
	Design	Design problems often have ambiguous specifications of goals, no determined path to solution, and require knowledge from different domains
	Dilemma	Similar to decision-making problems, but, because all solutions seem unsatisfactory, their outcome is highly unpredictable

The difference between Reitman's well/ill-defined problems and Jonassen's well/ill-structured problems lies in the specification of the problem states and the level of knowledge. A problem can be well defined but also ill structured to some extent. For instance, initial state, goal, and transition steps are all specified (problem is well

defined), but it is also possible to have multiple solutions to a problem (ill structured). To have a consistent and concise terminology throughout this thesis, problems are categorised based on Jonassen's system of the level of problem-structure.

2.2.1.2 Problem-solving models

According to Anderson (2015) "problem solving is a goal-directed behaviour that often involves setting subgoals to enable the application of operators" (p. 182). For Anderson, operators are all actions that transform the initial problem state to another state. Different theories about problem solving use different terms and also suggest different steps, but, according to Pretz, Naples, and Sternberg (2003), these steps can be generally summarised to (1) recognising that there is a problem, (2) analysing and defining the problem, (3) forming a strategy and solution, (4) organising knowledge about the problem, (5) allocating resources and applying the solution, and finally (6) monitoring the progress and evaluating the outcome.

According to Jonassen (1997) the problem-solving process also depends on the kind of problem, and some steps are more relevant than others depending on the extent of structure of the problem. For well-structured problems, Jonassen combined ideas from the theory of human problem solving (Simon & Newell, 1971), the model of the ideal problem-solver (Bransford & Stein, 1993), and Gick's general problem-solving strategies (Gick, 1986). As a result, Jonassen identified three major steps: (1) representation of the problem space (understanding the problem, its constraints and goals), (2) search for a solution, and (3) implementation of solutions. These steps involve different kinds of strategies such as mapping the problem onto prior knowledge, recalling analogical problems, identifying relevant subgoals and steps, and simplifying the problem. These strategies play different roles at different stages but may also occur simultaneously. The third step also includes testing and evaluation of potential solutions.

These three major steps of well-structured problem solving also play a role in Jonassen's model for solving ill-structured problems, but they are more divided due to the higher level of uncertainty in all problem states. Overall, this results in a slightly more complex model:

Step 1: Representation of the problem space

As with well-structured problems, understanding a problem is the first step. However, domain and context knowledge now play a bigger role for ill-structured problems.

Step 2: Identifying and clearing alternatives

Problem solvers may need to consider more than a single problem representation because of the variation of possible solutions. Each problem space must then be evaluated in order to decide which is the most relevant for the current situation.

Step 3: Generating possible solutions

The process of generating multiple solutions is often creative and, according to Jonassen, it relies not only on prior experience but also on unrelated thoughts and emotions at this early stage.

Step 4: Viability of alternative solutions

In order to choose the most valuable solution, solvers create an evaluation system based on their own beliefs and knowledge. This system may also include the opinions of others.

Step 5: Monitoring the problem space

This involves metacognitive strategies such as planning or allocating resources.

Step 6: Implementing and monitoring solution(s)

Possible solutions generated in Step 3 are implemented by the system generated in Step 4. In this respect, Jonassen emphasised the role of continuous performance assessment because of the ambiguous nature of ill-structured problems.

Step 7: Adapting solution(s)

Only few ill-structured problems might be solved with a satisfactory outcome at first try. It is more likely that solutions must be adapted and the solver needs to go back to some prior steps. That gives the whole process a more iterative character than is the case for well-structured problems.

Although slightly different terms might be used, similar steps are identified by others (see, e.g., Ge & Land, 2003). Jonassen (2000) further commented that the greatest difference between well- and ill-structured problems lies in their level of uncertainty and restricted knowledge (at least at the start of a problem-solving process). Although well-structured problems are more likely to be solved by a systematic search for solutions, the process for ill-structured problems is more "dialectical". The degree of appropriateness of a possible solution might change over time, and external variables can play a greater role than for well-structured problems. This is why Jonassen referred to a "design process" when he described the overall solving procedure for ill-structured problems.

2.2.1.3 Problem-solving in mathematics

Solving (mathematical) problems is a defining aspect of mathematics and mathematical didactics. The discussion of teaching mathematics *with* or *through* problem-solving is as old as the field itself. As CT emerged originally from CS and CS is closely linked to mathematics, some typical mathematical problem-solving strategies are presented. A good summary of this field is provided by Liljedahl et al. (2016). The focus of their work lies in heuristic methods and the phenomena of creativity and discovery in mathematics. These methods and phenomenon are discussed in more depth.

In the 20th century, mathematical thinking and problem solving were highly associated with heurism and heuristic strategies (see, e.g., Hadamard, 1945). The term *heuristic* emerged from a story about the legendary Archimedes how he was struggling over a problem by the King of Syracuse. The King wanted to know whether his crown was indeed pure gold as the goldsmith claimed. Archimedes had problems to figure out how to answer this question until he went on day into his bathtub and observed that the volume of water, he displaced was equal to the volume of his body. He suddenly understood how he could apply this observation to his problem. Because he was so excited about it, he jumped straight out of the bathtub and ran naked home while screaming his wife's name "Eureka".

This kind of sudden insight and the ability to apply a previous successful method to another situation are the core of heurism and heuristic strategies and methods. Such strategies highlight general terms and provide a general rule-of-thumb that may help to find a solution. For mathematical education this means to teach general problem-solving approaches to solve specific mathematical problems.

Teaching heuristic methods is largely accepted in mathematical education. It is more discussed which kind of methods are more important and in which way they should be taught. Kilpatrick (1985) suggested a taxonomy to summarise such methods:

> *Osmosis*: by solving many similar problems (with minor changes) learners develop implicitly a general problem-solving strategy (inferential learning).
>
> *Memorisation*: by memorising correct steps needed to be done to solve a specific kind of problem (deductive leaning)

Imitation: by showing how someone solves the problem in an ideally way (social learning with an expert)

Cooperation: by working on a problem together with others (social learning with peers).

Reflection: by promoting to use metacognitive strategies. Rather than "learning by doing" the main idea in this method lies in "learning by *thinking* about doing". What steps have been done in the past, what restrictions are there, what options are giving, and so on.

Kilpatrick (1985) further stated that these methods are not independent from each other but can be combined.

Applying and combining specific heuristic strategies is one way of teaching problem-solving in mathematics. However, these methods come with specific conditions which are not always been met. Sometimes there are no similar problems, any steps to remember, or it is not possible to reflect on the problem with others. A more general and holistic way of looking at mathematical problems comes with the mental agility model (Liljedahl et al., 2016).

Successful problem solvers tend to switch fast between perspectives. They can connect different components and see the relativity of circumstances. They show some level of mental agility. In general, typical signs for this kind of flexible thinking are (Bruder, 2000):

Reduction: reducing a problem to its essential core aspects. For example, using visualisations and structuring aid like graphs or tables to abstract important information.

Reversibility: reversing thought process by working through a problem from the end to the beginning and working backwards (e.g., using rough estimations for the possible result).

Minding of aspects: see different aspects or sides of a problems at the same time, (e.g., taking a complex figure and breaking it down to more simple structures).

Change of aspects: being able to switch perspectives of the problem that can prevent of getting stuck (e.g., proving geometric propositions by using vectors).

Using heuristic methods may compensate for less flexible and less successful problem-solver. The long-term goal of using heuristic methods in mathematical didactics is to teach how to break through some mental blocks. It can help to develop a different mindset before trying to solve a problem.

To achieve a more flexible thinking through heuristic methods, Bruder (2000) suggested three phases of training: first, students need to get used to such methods by using giving specific hints in the task description. These hints refer to typical signs of flexible thinking, e.g., "look for similarities" or "detach and attach elements". Second, students think out load while trying to solve the problem. They try to combine the hints with the referring sign like "when I solve a geometrical problem, I detach smaller and simpler parts like rectangles, triangles and circles." In the third phase, students try to solve the task by applying the methods.

Ironically, Archimedes, name giver of heurism, did not apply any specific (heuristic) method to solve his problem. Neither did Sir Isaac Newton in a similar famous story when the observation of falling apples lead to the sudden understanding of universal attraction. In both stories, no one was comparing they problems to others, memorising any steps, working with others, metathinking on the problems, and so on. They were not actively thinking about their problem or topic but busy with seemingly uncorrelated actions like submerging in a bathtub or just watching apples.

According to Hadamard (1945), "the sudden and immediate appearance of a solution at the very moment of sudden awaking" (p. 8) is the third of four stages of the invention in the mathematical field. During the first stage, a person would constantly think about the problem and make countless unsuccessful attempts to solve the problem. This stage is marked by high mental effort. The many unsuccessful attempts and feeling of disappointment lead to the second stage. The person stops engaging with the problem and gains some distance (e.g., taking a bath or going for a walk in the park). The pressure of the solving eases, results are being "digested" (p. 63), false leads and assumptions do not occupy someone's thought capacity anymore. This stage of incubation is defined by low (if any) mental effort while false leads and assumptions do not block the person's whole thought capacity anymore. Incubation is then suddenly interrupted by the third stage: illumination. Illumination is usually accompanied with a mixture of positive emotions like relief and proud to have found a solution finally. Eventually comes the stage of verification in which the ideas of solutions are being

evaluated. Details are being worked out and the solutions become formally correct proofs and such.

Hadamard (1945) emphasised how all these stages are interconnected to each other. Of course, discoveries can be made by just working on it without a break (only first stage). The same way some discoveries were produced just by chance (only second stage). However, more than often both are needed to lead the mathematician to an appropriate solution.

In addition, CT is related to computation skill development in school mathematics (Li et al., 2020). Computation is a familiar idea to many people, particularly to parents and students in elementary school. Indeed, students are required to learn to compute with numbers (CCSSI, 2010; NRC, 2002). Computational skill is usually considered as important not only in a person's day to day like, but also in preparing for, and in conducting, numerous professions, such as science, engineering, insurance, and finance, or other professions where numbers are used. Computation is also usually considered as a basic skill, and parents and the public would be seriously disappointed if children do not learn such basic skills through school education (Li et al., 2020).

Computation has historically been loosely connected to thinking until such time that mathematics educators began realising the significance of students making sense of what they do when they are engaging in computation (Li & Schoenfeld, 2019). Combining the construct of computation with thinking in this restricted sense makes CT not new to mathematicians, mathematics educators, and teachers at all. In this regard, CT thus emphasizes the significance of thinking and understanding in, and for, performing computations. The CT construct is likely to have been readily accepted because of its importance to every student in learning mathematics. Nevertheless, mathematics educators now use alternative terms conveying the same meaning, such as "number sense" (Sowder 1992) and "symbol sense" (Arcavi 1994). Based on these, what is the relevance of CT to other individuals aside from students? Why must the significance of CT be advocated by computer scientists as important to everyone, when computation as used in mathematics is usually regarded as merely as basic skill?

2.2.1.4 What problem solving means for computational thinking

Problem solving takes place in different steps. The earlier steps focus on the problem itself. This includes steps about understanding the problem as well as constraints and rules. Grasping the problem space or problem representation marks the beginning of the solving process. The last step is concerned with actual solutions. That may involve developing a solving strategy and implementing prospective solutions as well as monitoring and evaluating those prospective solutions. This general approach can be mapped onto the more specific problem process of CT. It is conceivable that the CT core skills focus on different aspects of the overall problem-solving process as well, and they also play different roles at different stages. This point will be elaborated on in the assessment model developed later within this thesis where the CT core skills will be discussed in more detail and will also be analysed using this perspective as a foundation.

Problems also can be categorised based on their level of structure. Well-structured problems present all elements of the problem, have a limited number of well-known rules and constraints, and possess correct and convergent answers. In contrast, ill-structured problems are less clear to the solver, have more uncertainty, and may have several possible solutions that need to be evaluated and eventually adapted during the whole process. Although the boundaries between those types of problem are sometimes unclear, the kind of problem has an impact on the kind of solution, as Jonassen's model implies. The solving process for a well-structured problem appears to be more straightforward and streamlined, whereas ill-structured problems require a more complex solving process with iterative steps.

At first glance, CT shares some similarities with features of problem-solving processes of well-structured problems. For instance, described strategies such as identifying subgoals, simplifying the problem, and recalling analogical problems, as well as implementing, testing, and evaluating solutions, can be seen as parallel descriptions for decomposition, the ability to abstract, and designing algorithmic solutions, respectively. This makes CT apparently better suited for well-structured problems. However, the literature about CT is rich with references about its usefulness for ill-structured problems. The jointly proposed definition from ISTE and CSTA lauded CT as the "the ability to deal with open ended problems" and declared that CT goes along with attitudes such as "confidence in dealing with complexity" (ISTE and

CSTA, 2011, p. 1). This view is widely adopted by many others (e.g., Barr & Stephenson, 2011; Bocconi et al., 2016, p. 16; Corradini et al., 2017; Kalelioğlu et al., 2016; Selby & Woollard, 2014; Weintrop et al., 2016). Shute et al. (2017) even concluded that CT "relates not only to well-structured problems, but also to ill-structured problems (i.e., complicated real-life problems in which solutions are neither definite nor measurable" (p. 2).

To put everything together, CT is based mainly on typical solving strategies for well-structured problems but is applied for ill-structured problems. This initially appears to be a contradiction, but it makes sense when looking at the field of CS and related areas like mathematics and mathematical didactics. Here, machines (e.g., computers) are used as tools to solve problems. Machines, however, are bounded to problem-solving methods for well-structured problems. Humans can handle multiple solutions and undefined constraints and can deal with ambiguous elements, but machines cannot. This makes CT a more holistic and flexible problem-solving approach, like the mental agility model. The mental agility model describes how successful problem-solvers can change quickly their view on the circumstances of problems before applying a solving method. CT describes the agile mental activity in formulating a problem so that a machine can help to solve it (Wing, 2008, 2011)—that is, to transform to some extent ill-structured problems into more structured ones. This way, it can be understood as a more elaborated heuristic strategy in comparison to some other typical problem-solving methods in neighbouring fields.

In summary, CT should be seen as a problem-solving approach especially useful for open-ended problems with multiple possible solutions, but it involves strategies usually employed for well-structured problems. This should be taken into account in order to have an optimally cohesive measurement model. Ill-structured problems as well as well-structured problems should be used to capture all facets of CT and to assist with understanding the full process of CT.

2.2.2 Decomposition

As defined earlier, the problem-solving process in CT is associated mainly with three skills: decomposing of problems, the ability to abstract, and creating algorithmic solutions, each of which will be discussed in the next sections. To do this, they are viewed from a CS perspective to provide an indication about why these abilities are

mentioned throughout the literature and are constantly emphasised by experts. Within the next subsections, the three skills are also analysed in terms of their meaning in psychology in order to acquire better insights about how they are applied in concrete situations.

2.2.2.1 The role of decomposition in computer science

In a very general sense, decomposition means to deconstruct or to factorise a complex system into its simpler parts (Booch, 1994, p. 14; MDESE, 2016, p. 50). Complex systems in CS can refer to different things and concepts on different levels and so the core idea of decomposition takes place in different forms (Najafi, Niu, & Najafi, 2011). Regardless of whether it refers to organising working project, the basis of whole programming paradigms, or as a vital concept in specific programming languages, decomposition plays a vital role in CS.

On a macro-level, decomposition takes place as a crucial element in agile project management, which has its roots in software engineering and is still popular in this field. Agile management is an umbrella term for many different approaches such as Scrum and Extreme Programming, with the same methodological foundation. The core idea of these approaches is to be able to create first drafts of solutions or products quickly and to quickly adapt to changes during the working process (D. Cohen, Lindvall, & Costa, 2004, p. 8). That involves many circles in the production process and rounds in communication with different team members and stakeholders on until the final product is delivered.

To cope with having these different steps, the original task must be broken into subtasks. In Scrum, for instance, the production process is divided into many relatively short working periods, called *sprints*, with successive meetings involving customers to obtain constant feedback. For sprints, it is important be able to define specific goals before the next meeting takes place. This is similar to the *planning game* in Extreme Programming in which the overall goal of a project is translated into user stories with different parts. Each part of the story focuses on different problems and requirements. This approach helps participants to organise responsibilities among the team members according to their capabilities (D. Cohen et al., 2004, pp. 13–15). These two examples show how the deconstruction of tasks or problems can be vital in the working process in software engineering.

At a lower level, decomposition can be interpreted as the core component of modular programming. Modularity can generally be seen in CS as the "development of autonomous processes that encapsulate a set of often-used commands performing a specific function and might be used in the same or different problems." (Atmatzidou & Demetriadis, 2016, p. 664). The complex system here is the whole programming paradigm, which is decomposed in several smaller (partially independent) modules or packages. Boudreau, Tulach, and Wielenga (2007, p. 9) pointed out, that in a time of open-access software and programming languages, many kinds of software are no longer developed by a single developer or single team. Instead, many people all over the world contribute to it in forms of modular applications. This diversity leads to many solutions to different kinds of problems. Modules are isolated programs that contain a limited number of subroutines that relate to a very specific kind of problem. They usually work independently of each other and are organised in libraries. To use modules, programmers need to explicitly call them up. This modular design helps to prevent chaotic "spaghetti code"[1] because only the subroutines needed for a particular problem are activated.

For instance, R is a statistical programming language that is organised in modules. In general, R can be seen as a simple but very potent calculator in which the most fundamental mathematical operations are provided. Many statistical processes demand more sophisticated mathematical models. It would be tedious to near impossible for individual users to write every statistical routine. Instead, different users create different modules, called packages, with some of those packages being more advanced statistical procedures. So, instead of writing a function that would perform a hypothesis test such as an independent-samples *t*-test, the user needs only to load a package that includes that particular *t*-test. This applies to other statistical procedures.

At an even deeper level, decomposition is implemented as one of the core concepts of another current popular programming paradigm. In object-oriented programming, clean isolation and reuse of code is a vital concern (Najafi et al., 2011) and is summarised in the concept called *encapsulation*. Encapsulation means that some features are excluded or encapsulated from the rest of the program (Dale, Weems, &

[1] Spaghetti code refers to a set of code in which many GO-TO statements are used to transfer code actions to another place in the program as often occurs in programming languages such as FORTRAN or BASIC. This kind of code often appears to be as unsorted as a bunch of spaghetti and thus is difficult to read and should be avoided (Boudreau et al. (2007, p. 14).

Headington, 2004, p. 177). As a result, objects such as variables or operations do not communicate with each other, and neither do they directly or automatically influence each other. Instead, programmers need to explicitly indicate what objects in their program have access to each other. This way, the purpose of a program is deconstructed into different chunks of codes. This helps to prevent unwanted interactions from different parts of the program that could cause errors. For example, there might be a function, F1, that uses “x” as its name, and, coincidentally, “x” could also represent another function, F2. Without encapsulation, the argument in F2 would be interpreted as the function F1 but that might not be the original referent. Encapsulation reduces the impact of changes and makes it easier to keep control of functions. Altering parameters or deleting functions or methods does not have an impact across the whole program. That way, execution of codes is safer and more stable. Encapsulation shows how a program is decomposed into chunks of codes that not only work independently but also work together without unintentionally influencing each other.

These examples demonstrate how the concept of decomposition appears in different aspects or stages of CS and associated areas from life-circle design and process modelling of projects to actual development and implementation of programs. Regardless of the level of action, people are constantly confronted with decomposing complex systems into smaller components. This underlines the impact of decomposition that also indicates why it is considered so often in the literature as such a crucial skill for CT.

2.2.2.2 The role of decomposition in psychology

Breaking down problems into smaller problems had been recommended as a general problem-solving strategy in psychology long before the appearance of CT (Anderson, 2015, p. 182; Jonassen, 1997). The general idea behind problem decomposing is to divide the initial problem into smaller problems as long as they are sufficiently small that a potential solution seems obvious (Polson & Jeffries, 1985). A more specific example of a decomposition model was developed by F. J. Lee and Anderson (2001). It was loosely based on the goal, operators, methods, and selection rule (GOMS) model by Card, Moran, and Newell (1983). The GOMS is a cognitive model with the goal of predicting humans’ behaviour when they interact with computers to improve usability experiences. Lee and Anderson’s model is more general and focuses only on task analysis. In their model, they distinguish between three different layers of task

decomposition. The most general level is the *unit task level*, which is still closely linked to the overall task. At this level, the main task is divided into subgoals that can be achieved independent of each other. Next is the more specific *functional level*. At this level, the operations that are needed to achieve the subgoals are defined. The last level consists of primary cognitive goals up to motor actions such as a single keystroke—which is why F. J. Lee and Anderson (2001) called this level the *keystroke level*. However, operations on this level are not confined to action made on keyboards. It is the most specific level and cannot be broken down any more. On this level, fundamental actions needed to achieve the sub goals on the functional level are described.

The distinction between the different levels is important for illustrating the dependencies between them. At the two higher levels, the subgoals and tasks are only dependent on the level above. Regardless of the particular system or platform that any solution will be run on, goals on the unit task levels are dependent only on the overall main task, and the subgoals of the functional level are dependent only on the unit task level. No other knowledge is needed. Nevertheless, the system might have an impact on these subgoals. The keystroke level, as the lowest level, is highly dependent on the system in which the solution might be applied. This means that there must be knowledge about the operational platform before the steps needed at this level are identified. Operators need to know what kinds of actions are possible in general and whether any assumptions or conditions need to be met.

As an example of a decomposition process, the overall task might be to write an essay about someone who overcomes his or her major fear. On the unit task level, three subgoals might be identified: (1) clarifying who the person is, (2) clarifying the specific fear, and (3) developing a plan for overcoming the fear. Each of these goals can be deconstructed at the functional level. Developing a plan, for instance, can be further deconstructed into developing (1.1) a beginning, (1.2) an end, and (1.3) a turning point. The words and phrases used to write the essay represent the keystroke level for this example because they can be seen as comprising the atomic unit. The specific words that can be used depend on the language the account will be written in. The language represents the system or platform here. The unit tasks and functional tasks are not dependent on the system. Regardless of the language, it is likely that the subgoals remain the same. Nonetheless, the language proficiency of the writer may have an influence on the subtasks.

F. J. Lee and Anderson (2001) suggest a top-down analysis for task decomposition, where given tasks are analysed as entities beginning at the top with the most general terms and going down to more specific goals where the most basic and simple operations are located. Breaking down a task in this way may have three benefits.

First, it may help to gain a better overview. A large task can appear to be overwhelming and unclear. Too much information needs to be processed at once. On the other hand, several problem chunks can be ordered by different features including their priority or approximate time to achieve a solution. This provides an overview for the process. A better overview can also help to identify potential challenges in the later solution processes, thus making the overall process more robust.

Second, it is possible to identify subgoals and tasks that can be achieved and solved independent of each other. This means that resources such as time, materials, and manpower can be allocated efficiently. For example, there might be two people with different strengths who work on the same task. If two subgoals on the unit task were identified, it will be probably more efficient if the people worked independently on the different tasks according to their skill level rather than simultaneously working on the same main goal. F. J. Lee and Anderson (2001) demonstrated how participants solved a complex air traffic controlling task significantly faster when they decomposed it into simpler subtasks.

Third, different subproblems may result into different solving approaches, which can involve different levels of cognitive load. In mathematics, for example, some complex problems are broken down into simpler ones so that the required solving strategy shifts from a calculation strategy to a memory strategy (Bull & Espy, 2007, pp. 114–115). The Trachtenberg system (Trachtenberg, 1960) can be seen as such an example. With the Trachtenberg system, the complex and cognitively demanding problem of multiplication of two numbers larger than two digits can be broken down into a set of comparatively simple steps that include only addition and multiplication of numbers with only one digit. Both operations require significantly less cognitive effort.

Although task decomposition has many benefits, there are some constraints. According to Polson and Jeffries (1985), the effectiveness of this strategy depends on the level of the problem-solvers' knowledge about the problem. The more knowledge they have, the easier it is for them to formulate subtasks. However, Polson and Jeffries also emphasised it as being a generally useful strategy.

Decomposition as a strategy is more part of the problem-planning phase than the actual solving process. As Lee and Anderson’s model shows, decomposition is related to reformulating the problem in order to plan steps for the solution. Although decomposition depends on the level of knowledge the problem solver has about the problem, it can generally be regarded to be a powerful mental tool.

2.2.2.3 What decomposition means for computational thinking

Decomposition has been shown to be a concept that appears in different forms and at different stages in CS and related fields. From a psychological perspective, decomposition comprises methods in which problems or tasks are deconstructed into smaller chunks, as seen in F. Lee and Anderson's (2001) model of task decomposition. Decomposition is concerned with reformulating the problem itself rather than formulating a solution. Thus, it can be assumed that decomposition should take place at a very early stage of the process if problems are to be solved with CT.

Concrete signs of decomposition derive from steps of problem deconstruction. For example, breaking originally complex problems into smaller and less complex ones and what the next steps could be in order to deal with these subproblems can be seen as part of decomposition. Also, how these steps are related to each other and the main problem can be seen as decomposition according to Lee and Anderson’s model.

2.2.3 Abstraction

In the literature, one of the most frequently mentioned skill relating to CT is the ability of abstract thinking. Abstraction appears in different forms in CS. After analysing these different forms below, the psychological meaning of abstraction is discussed. Then, a final conclusion is drawn about what abstraction means with relation to CT.

2.2.3.1 The role of abstraction in computer sciences

In computer science and related fields, abstraction is regarded to be a fundamental concept. CS is rich in references about abstraction, such as data abstraction or procedural abstraction, which are used to describe the separation of logical properties of data and a procedure, respectively (Dale & Walker, 1996, pp. 4–5). Denning et al. (1989) refer to abstraction as one of the main paradigms in their idea of computing as a discipline. They also consider abstraction to be the main focus in CS as well as in

software engineering. Aho and Ullman (2000) even declared CS to be a "science of abstraction". This alone underpins the relevance of abstraction in CS.

According to the *Encyclopaedia of Database Systems*, abstraction is defined as a concept that "allows developers to concentrate on the essential, relevant, or important parts of an applications" (Thalheim, 2009, p. 6). According to Ward (1995, p. 443), there are several different ideas of subforms and usages of abstraction, but the main ideas can be reduced to three core principles:

1. Abstracting specification say what a program does without necessarily saying how it does it.
2. Abstraction is a process of generalisation, removing restriction, eliminating detail, removing inessential information (such as algorithmic details).
3. Abstract specifications have "more potential implications", moving to a lower level means restricting the number of potential implementations.

To summarise these points loosely, abstraction in CS implies reasoning about common structures in data or mathematical entities while certain properties that differ from instance to instance are ignored (Pease, Smaill, & Guhe, 2009), or, to say it differently, an abstract algorithm presents a solution without fully revealing how the result was achieved (Haberman, 2004).

In programming, abstraction can also mean "giving things names" (Stein, 2002, p. 1). Things, in this case, may be algorithms, data, objects, and so on. These can be seen as computing entities. Behind these of entities lies considerable information that is usually not needed. Giving them names can help to abstract out the unnecessary information. An often-used example for abstraction in programming is the task of drawing a square (Wentworth, Elkner, Downey, & Meyers, 2012). Several steps must be followed and assumptions made to create a square, specifically, drawing four lines of the same length, all connected to each other; two lines are orthogonally connected to each other whereas the two opposite lines are parallel. Most humans know intuitively what a square is. They do not always need those detailed instructions. However, a computer does not have intuitions and therefore does not know what a square is. It always needs precise instructions. It would be very tedious and confusing for a programmer to always specify all these steps and assumptions just to get a computer to draw a square. Luckily there is no need to do this. Instead, programmers apply abstraction and write a function that includes those steps and they give it a name such as "square". In this case, the word

"square" is an abstraction of a more detailed procedure hidden behind the word. This word or name is the only information the programmer is interested in. The function hides all the unnecessary details the programmers do not need to know to draw a square. Programmers only need to know which function they need to call on to complete the task by the name of the function. Of course, this could be done with different kinds of figures. For example, a triangle would require drawing three lines with all connected to each other at their corners.

This opens the opportunity to complete more complex tasks with higher levels of abstraction. If the task is to draw a house, the intuitive idea of a house might be a square with a triangle on top. Instead of writing a new function with explicit steps, it is possible to use abstraction and to combine functions with each other to create a new one. The function "house" could consist of the functions "square" and "triangle". This can lead to even more complex tasks such as creating villages (a collection of houses), and so on. Here the programmer operates on different levels or layers of abstraction and switches between them.

Another example of the principle of different layers of abstraction is seen in the open system interconnection (OSI) model (Colburn & Shute, 2007). The OSI model is a framework for computer network architecture. It describes how communication is performed in seven layers within which data are exchanged between systems in different ways. Each level represents a different layer of abstraction. The layer with the lowest possible abstraction is the physical level where data are transmitted using electric currents that turn data into on and off (i.e., 1 and 0) binary states. Data are then processed into higher order layers of abstraction up to the level of end-user applications (e.g., a web page).

Without abstraction, programmers would still have to program in machine code on a physical level. They would need to translate their data and instructions in binary form. Of course, it is not feasible for humans to do this. With abstraction, however, it is possible to convert information from binary form to a more complex level in a bottom-up process. Programmers are not interested in exactly how the computer is carrying out the procedure and they do not need to know. Programmers only want to draw a square and sometimes a triangle on the top.

Abstraction not only enables communication between humans and machines. It also provides opportunities for efficient and clean coding. A code that appears to be elegant

and easy to read has a high level of abstraction. A well-written code does not produce convoluted solutions and does not provide more information or results than needed (Kramer, 2007). A high level of abstraction means a high level of generalisation. As Ward (1995, p. 450) described it, "a program S_1 is an abstraction of another program S_2 if each of the possible execution sequences for S_1 consists of a subsequence of possible execution sequences for S_2". Essentially, this means that the more concrete a specification becomes, the more degrees of freedom are lost. It also means that a more abstract code has greater possibility to be used and has a higher level of generalisation. A program has a high level of generalisation when it is easy to apply for different situations and therefore few specifications need to be altered. That helps to reduce unnecessary duplications of codes. Instead of repeating the same statement with different arguments, loops or recursion could be used. In that way, abstraction helps to reduce the amount of information that needs to be understood and it also reduces the level of complexity. That is why code written on a relatively high level of abstraction appears to be easier to read. As shown in Section 2.5.2.1, readability is seen as a hallmark of a good code.

Stein (2002, pp. 5–6) provided an example how abstraction can make code more efficient and easier to read. Imagine there is a bank account with a method that shows:

```
1  Int getBalance( Signatory who ) throws InvalidAccessException
2  {
3      if ( !who == this.owner )
4      {
5          throw new Invalid AccessException( who, this)
6      }
7      // else
8      return this.balance;
9  }
```

The first lines define the bank balance as an integer. The verification takes place with the throw command, which is linked to the crucial if command in the second line. If the owner cannot verify with a valid identification, the balance is not shown, but if it is correct (line 8), the balance will be shown.

Of course an account holder might also want to withdraw money, in which case the balance in the account would change as well. This is what a solution could look like:

```
1  Public Instrument withdraw( int amount, Signatory who ) throws
2   InvalidAccessExcept
3  {
4      if ( !who == this.owner )
5      {
6          throw new Invalid AccessException( who, this)
7      }
8      // else
9      this.balance = this balance - amount;
10     return new Cash ( amount );
11  }
```

Compared with the routine before, the only changes appear in the first and ninth lines. The first line now defines a procedure where the balance is not shown but an amount of money as an integer can be withdrawn. As for the first routine, this is linked to an if command, which is exactly the same as the first one. In the ninth line, the balance is overwritten as a result of a subtraction of the original balance and the recently withdrawn amount of money, and is finally shown in line 10.

This solution would work, but the solution appears to be convoluted with some redundancy and duplicates. In addition, if any changes needed to be made, both routines would have to be altered. That would slow the whole work process down. Instead, it is possible to abstract the common pattern here, which is the verification procedure:

```
1  private void verifyAccess ( Signatory who ) throws
2   InvalidAccessExcept
3  {
4      if ( !who == this.owner )
5      {
6          throw new Invalid AccessException( who, this)
7      }
8  }
```

This separated verification routine can be now implanted into a new routine where the routines for showing the balance and withdrawing the money are combined:

```
1  Int getBalance( Signatory who ) throws InvalidAccessExcept
2  {
3      this.verifyAccess ( who );
4      return this balance;
5  }
6  public Instrument withdraw( int amount, Signatory who ) throws
7   InvalidAccessExcept
8  {
9      this.verifyAccess ( who );
10     this.balance = this.balance - amount;
11     return new Cash( amount );
```

This solution appears to be more concise and easier to understand because of the reduced redundancy of code. In addition, modification for the verification procedure can be done at one place instead of two, which makes this solution more efficient than the first one. This shows how abstraction makes code easier to read and optimises the work flow.

These examples demonstrate how abstraction is applied in CS and why it has become such a vital concept. Abstraction can be seen as the core of some programming paradigms as well as in handling data and procedures. Code that is written “more abstractly” is also easier to read and understand, and it appears to be “more elegant”. In general, abstraction enables communication between machines and humans. CS would probably not exist without any kind of abstraction. This might also explain why abstraction is considered to be important for CT.

2.2.3.2 The role of abstraction in psychology

Although it has its roots in philosophy back to the time of Aristotle (Burgoon, Henderson, & Markman, 2013), empirical research about abstraction is traditionally located in psychology. It gained more attention during the cognitive revolution in the second half of the 20th century (see, e.g., Posner & Keele, 1968; Rosch, 1978). Unfortunately, even in this field the concept of abstraction lacks a final and clear definition (Barsalou, 2003). That leads to a situation where there are nearly as many theories about what abstraction is as there are methods and approaches for studying it.

Posner et al. (1968) described the ability to abstract as the ability to infer rules based on observations and to apply these rules to instances that the person has never encountered, for example, “when a man correctly recognizes an animal he has never seen before as a dog, he has manifested an ability to generalize from previous experience” (p. 353). Posner could show that participants generate a kind of scheme or concept of patterns (prototype) based on presented stimuli in a training session. These prototypes were easier and faster to recognise than were any other kind of pattern even though participants had not seen them before. That means that abstraction is strongly associated with learning. It is an efficient way of interpreting and storing information.

In addition, the stimuli in the training session in Posner’s research (Posner, 1968) were all different. However, these differences were ignored, and instead participants implicitly focused on shared attributes. Posner concluded that participants had *abstracted* a concept, a mental representation, of something that they had not experienced before. They recognised a set of rules that determined what belonged together and what not. It is important to point out that abstraction does not involve learning about actual physical stimuli and attributes of things, but rather the relationship(s) between them (Posner, 1969). Although the stimuli in the training session were different, participants developed an idea of what all stimuli had in common (Posner & Keele, 1968). This makes abstraction crucial for learning. It is not *memorising* but *inferring*, which goes further in higher-order thinking.

For Piaget, abstract thinking played a crucial role in cognitive development. He distinguished between two kinds of abstraction. *Empirical abstractions* are inferential projections based on former experiences. They are described as belonging to reality (Moessinger & Poulin-Dubois, 1981). This kind of abstraction refers to the processes of inferring and developing rules based on actual observation and comes close to Posner’s idea of abstraction. It is predominantly part of what Piaget called the concrete stage in his theory of development. The term concrete refers to the content of thinking, which is still bounded to an experience in the real world. *Reflective abstraction*, in contrast, mainly refers to metacognition such as thinking about one’s own thinking (Campbell & Bickhard, 1986, p. 88). It also describes the ability to think about “things” that are not physically bounded to the real world, such as laws, ideas of others, and symbols in general. Possible outcomes are derived from imagination and thoughts without regard to whether they had been actually experienced. This kind of abstract thinking becomes more dominant in the formal operational stage—the last stage of cognitive development

in Piaget's theory (Siegler, DeLoache, & Eisenberg, 2014, pp. 134–145). That underlines how, according to Piaget, abstraction is part of the later stages of cognitive development and therefore part of higher-order thinking. Abstract and concrete thinking are the end poles of the same dimension. An abstract concept is only a thought process, a fuzzy image in the mind or a loose idea of something, and there is not necessarily a connection to the real world. In contrast, *concrete* means there is a manifestation in the real world that can be straightforwardly projected into the real world. This is why Piaget saw the ability of abstraction as part of higher-order thinking and even as the peak in human cognitive development.

Abstraction is in particular associated with Rosch's principle of categorisation (Rosch, 1978; Rosch, Mervis, Gray, Johnson, & Boyes-Braem, 1976). In her theory, a category contains a number of objects that are considered as somehow equivalent (i.e., representatives of a category share the same features). Taxonomy refers to a system of how categories are related to each other by means of inclusiveness, and this is where abstraction comes into play: "The greater the inclusiveness of a category within a taxonomy, the higher the level of abstraction" (Rosch, Mervis, Gray, Johnson, & Boyes-Braem, 1976, p. 383). The level of abstraction determines a specific level of inclusiveness. A category within a taxonomy of a higher level of abstraction contains more objects than does a category at a lower level because there are more objects considered to be equivalent because, in turn, there are fewer features that need to represent a category (i.e., the level of inclusiveness is higher). Thus, the most abstract level in a taxonomy is also the most inclusive level, and the least abstract or most concrete level is the least inclusive.

An example of a categorical system with different layers of abstraction is shown in Table 2.4. The somewhat abstract level contains fewer objects than the highly abstract level, but more than the concrete level. The level of inclusiveness or abstraction is higher in the next more abstract level. That also means that objects in more abstract categories often appear to be more different among each other than they do in less abstract levels. However, these differences are (implicitly or explicitly) ignored and considered to be unnecessary details. That becomes more difficult the higher the level of abstraction becomes. Or, to put it differently, it becomes more difficult to recognise shared features the higher the level of abstraction becomes. In contrast, the more concrete a category becomes, the more similar the objects appear to be. This has an impact on how easy it is to imagine the objects and leads to the same conclusion Posner

made, albeit for different reasons: Concrete objects are easier to imagine than are abstract objects because fewer features need to be omitted.

Table 2.4

An Example of Different Layers of Abstraction

Concrete	Somewhat abstract	Highly abstract
Bob	Human	Living being
Football	Sports	Free-time activity
Preparing a report for next quarter	Writing a text	Working
Holding hands	Love	Emotion
White oak	Tree	Plant

Abstraction does not only play a role in the categorisation of things. Other research investigated its role for memory (see, e.g., Hintzman, 1986, e.g.,), or how abstraction is connected to language (Barsalou, 1994, e.g.,). However, these works are based at least partially on the ideas of Posner, Piaget, and Rosch. They also are not crucial for the process of abstraction in the context of CT, and, as a result, they are not considered more deeply in this thesis.

In summary, abstraction is a thought process that is used to achieve organised thinking (Shivhare & Kumar, 2016). In Posner's theory, abstraction is associated with learning about rules or recognising patterns across observations. Piaget went one step further by arguing that reasoning is based on prior experiences at the beginning but develops to a more unbounded way of thinking. Patterns and rules are able to be learnt even without self-made observations but with thinking about thinking. For Rosch, abstraction is accompanied by the level of inclusiveness of features: the higher the level of abstraction, the higher the level of inclusiveness. That also means that some features are interpreted as important (for that level of abstraction) and others need to be ignored in order to fit within a specific level of a taxonomy.

In conclusion, the ability for abstraction can be broken down into two processes. The first is the ability to distinguish between important information and unimportant details. The second is the ability to identify invariant features over different instances and to recognise patterns and rules. Although viewed differently within psychology, there is a

general acceptance that abstraction plays a role in learning (Burgoon, Henderson, & Markman, 2013). This includes the ability to infer patterns and rules, even without prior first-hand experience, as the result of a thinking process. In addition, all theories of abstraction emphasise that abstraction is part of higher-order thinking. Of course these theories are not without limitations and flaws, and Piaget's scientific methods have been the target of considerable criticism (Lourenço & Machado, 1996). However, there is general support for Piaget's findings, which underlines the importance and impact of his theories about abstraction as a cognitive process.

2.2.3.3 What abstraction means for computational thinking

As shown in the previous section, decomposition is clearly associated with the problem itself (i.e., reformulating the problem). This may not be as obvious for abstraction. The psychological analysis in this thesis revealed that abstraction comes in two components: neglecting unimportant details and recognising patterns. Both components can refer to the problem itself as well as to possible solutions. For instance, the initial problem as well as possible solutions can both be discussed in order to be simplified, and simplification can be interpreted as a sign of abstraction in the sense of neglecting unimportant details. On the other hand, mentally comparing different problems as well as different possible solutions can be seen as a process of identifying patterns throughout instances. Thus, it can be assumed that abstraction can take place at a very early stage of the problem solving process as well as in the middle stages.

Some authors have referred to CT as the ability to look at the same problem from different layers of abstraction (Priami, 2007). This way, different and more insights about the original problems might be possible. In that sense, CT means neglecting unimportant details and recognising patterns in such way that emerged models of the reality can be interpreted by other humans or machines (Wing, 2008). This duality of abstraction should be taken into account when measuring CT. Abstracting in the sense of neglecting details can be seen in forms such as simplifying features crucial to the problems (i.e., the problem itself, possible solutions, constraints, rules, etc.). These are operations that show that the problem solver focuses on important elements of the problem through abstraction. Abstraction also means identifying similar structures in (sub)problems and possible solutions. If the problem solver detects patterns, this should be seen in some kind of a reaction of sudden realisation or enlightenment such as "aha moments" (Piaget, 1952, p. 7; Posner & Keele, 1968), which are similar to

Archimedes sudden insight while sitting in the bathtub. Such aha moments could be seen as aspects of abstraction in the sense of CT as well.

2.2.4 Algorithmic design

The last major core skill associated with CT that will be discussed in this thesis is the concept of algorithms and the ability to think in algorithms. As for the other skills, it will be first discussed what algorithms are and what their role is in CS and related fields. This includes the design and evaluation of algorithms. After that it will be revealed that our mind at least partially can be seen as organised in algorithms and what it means to think algorithmically. Conclusions will then be drawn about what the design of algorithms means for CT.

2.2.4.1 The role of algorithmic design in computer sciences

Although undoubtedly one of the most important concepts in computer science, there is a several decades-long dispute about what an algorithm is, and a formal definition is yet to be agreed on. There are, however, some characteristics that are mentioned more often than others and that will be the foundation for an operational definition used in this study. Cormen, Leiserson, Rivest, and Stein (2014, p. 4) described an algorithm as a "well-defined computational procedure that takes some values, or set of values, as input and produces some value, or set of values, as output." This indicates that an algorithm is a tool to accomplish computational problems, although the word computational must be used very broadly in this context. Algorithms can be used to accomplish advanced tasks such as identifying human genes on the basis of several billion possible chemical pairs, or finding the shortest way to drive from one place to another, or accomplishing less complex task such as letting an avatar speak or walk when a specific key is pressed. All these tasks can be translated into computational problems regardless of whether the produced data are genes, distance coordinates, or electronic keyboard signals (Cormen et al., 2014, pp. 6–9). There are various versions of algorithms that all accomplish different kinds of computational tasks (see, for an overview, Sipser, 2013).

In its most general way, an algorithm can be seen as a cooking recipe (Sipser, 2013) with the ingredients and quantities as inputs, a set of rules and sorted steps that describe what to do with the ingredients, and the finished meal as output. The really delicious part of an algorithm lies in the procedure that describes the set of rules that make the algorithm appear to be a black box. For the outcome, no knowledge is needed about

exactly how the data are used. The data are always processed automatically and autonomously once the algorithm has been activated. It does not matter who executes the algorithm. This is where the real power of an algorithm lies. Once an algorithm has been created, anyone can solve the same problem without thinking about it. The executer of an algorithm does not need to understand what the purpose of the algorithm is or exactly how it works. The executer needs only to follow the steps described in the algorithm mechanically. That makes computers perfectly suited for being algorithms. They blindly follow steps provided in the algorithm.

Algorithms can be presented in different forms such as pseudocode (a set of rules written in plain language) or as a flowchart. Algorithms for computers are usually referred to as programs (Cormen et al., 2014, p. 6). A program is nothing other than an algorithm translated in a specific computer language. It is important to note that errors can occur during this translation, in which case the program will not behave as originally intended. Maybe the set of rules or the rules in the flowchart make sense, but the program does not work in the same way. These programming errors are called "bugs"[2] and the process of fixing them is known as "debugging".

Bugs can occur in all phases of a program's lifetime. Especially at the beginning, programs rarely work as originally intended. Programmers spend roughly more than one third of their time finding errors and validating code (O'Dell, 2017), so debugging plays a major role in software development. Debugging involves several steps, and although exact names might vary slightly throughout the literature they can be roughly described as reproducing, diagnosing, fixing, and, sometimes as fourth step, reflecting (Butcher, 2009, pp. 17–18). Through the first three steps, test runs are conducted regularly. Therefore, constant evaluation can be seen as the foundation of the whole debugging process. Reproducing a bug on a reliable basis is the first goal. Being able to reproduce an error on demand provides insights about its cause and helps to rule out alternative explanations. After that, the second, "experimental phase", begins. Butcher (2009, p. 49) pointed out how diagnosis takes place within the debugger's mind, not within a computer, and it refers to conducting little experiments. Experiments in this case might again be conceived of as test runs, but they have slightly changed parameters. Butcher stressed the importance of making only one change at a time in order to increase the accuracy of conclusions about the bug's cause. After being sure about the origin of the

[2] The term "bug" was most likely coined by Grace Hopper who observed how a moth flew into a computer causing some malfunctions (McFadden, 2018, September 13).

bug, fixing it is attempted and new tests are run. In the final step, a thoughtful debugger should reflect on how the bug was created and on how to install mechanisms to prevent similar bugs in the future. In addition, if bugs of the same kind appear frequently, it might be useful to reflect on one's own behaviour to determine at what point in the programming process the errors were created.

In summary, although not formally defined, algorithms can be seen as a set of (computational) steps. Algorithms appear in different forms, for example as written plain language style (e.g., spoken English), as flowcharts, or as coded programs. The design of algorithms plays a significant role in CS. As Cormen et al. (2014, p. 14) commented: "having a solid base of algorithmic knowledge and technique is one of the characteristics that separates the truly skilled programmer from the novices". This also includes the maintenance and evaluation of algorithms and therefore debugging is naturally part of any algorithmic design process. This conclusion provides good reasons why understanding the concept of algorithms and thinking algorithmically appear to be so important for CT.

2.2.4.2 The role of algorithmic design in psychology

During the mid-1950s, many psychologists found themselves unsatisfied with behaviourism as the main approach in psychology at that time. Behaviourism followed the idea that psychology should be based solely on behaviour because behaviour is the only objective source of data. In the eyes of a behaviorist, the mind was something subjective and unobservable. Opposing pure behaviourism, some psychologists assumed that the human mind is a real "thing", more than just a black box, but observable, and slowly the cognitive revolution in psychology began (Miller, 2003). To conceptualise their new ideas about how the human mind works, many cognitive psychologists were inspired by an equally new emergent field: computer science. Cognitive psychologists tended to compare the human mind with computational processes and they used a nomenclature nuanced by CS. For instance, Miller studied how human memory works (see, as a classic example, Miller, 1956). Together with his colleagues, he coined the term "working memory" for the short duration when information is temporarily stored and first processed (Miller, Galanter, & Pribram, 1960, p. 65). This term was purposely chosen to be analogous to random access memory (RAM) in computers, which is seen to work in a similar fashion and is colloquially referred as "working memory" as well. The liaison between psychology

and computer science, together with linguistics (with special contributions from Chomsky) as well as neuroscience, anthropology, and philosophy, eventually resulted in a new, interdisciplinary research area: cognitive science (Miller, 2003).

Not only were similar terms used to describe human thinking, but later psychologists proposed that our knowledge is stored in a way that corresponds to algorithms. In the late 1970s, Schank and Abelson presented their theory of *scripts*. Scripts are ideas of idealised events that follow stereotypic sequences of actions (Schank & Abelson, 1977). An often-used example is going to a restaurant (Figure 2.1). From the initial state "enter a restaurant" to the outcome "leave the restaurant" a set of rules guide the whole process in a way that is similar to an algorithm. An algorithm works as a black box where the operator does not need total knowledge about the situation but only follows the rules. Although this is not entirely true for scripts (e.g., most of us do not blindly follow a set of rules when we enter a restaurant), further studies suggested that scripts reduce people's cognitive load in their working memory (Bower, Black, & Turner, 1979). In conclusion, even though scripts should not be seen as humans' autopilot for behaviour, they are something like algorithmic systems that help us to deal with common events.

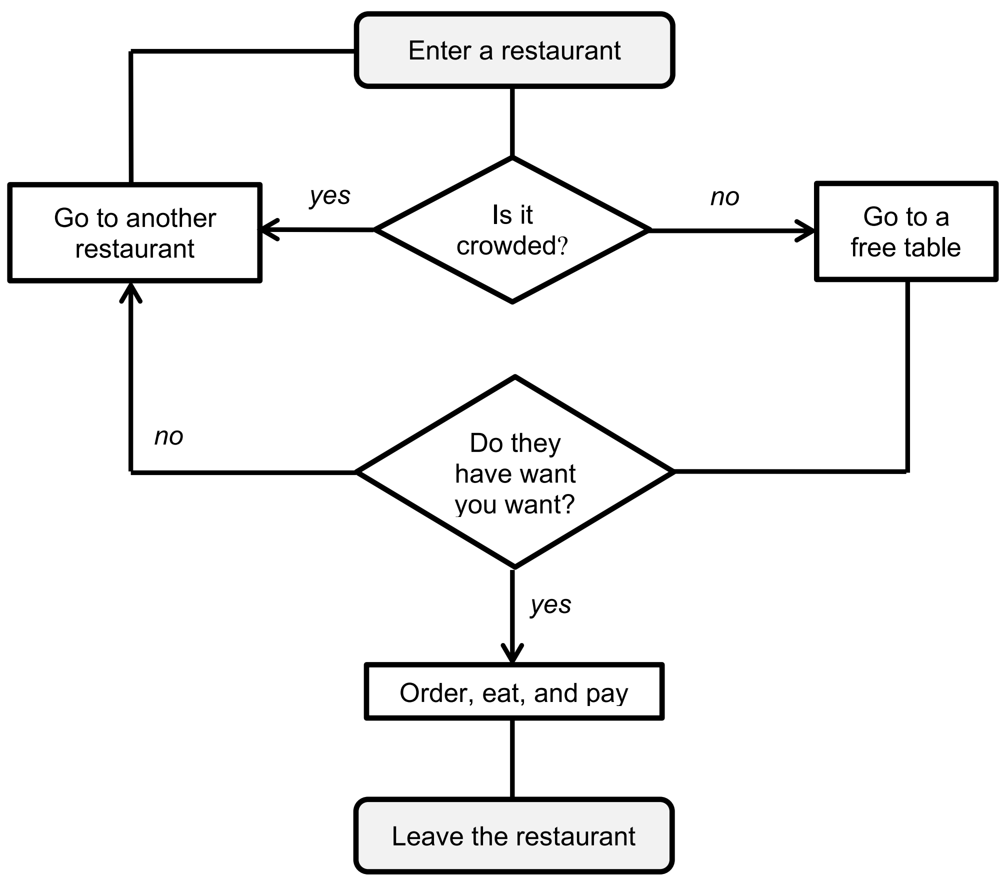


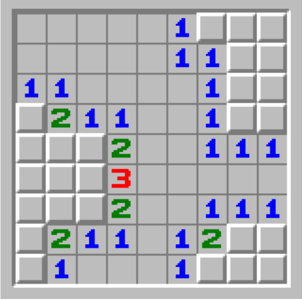
Figure 2.1. Script of going to a restaurant presented as flowchart (example inspired by Schank & Abelson, 1977).

Scripts are implicitly learnt over several similar instances and time. They are often deeply internalised so that people are not always aware that they follow specific procedures (Schank & Abelson, 1977). A set of rules for a sequence of behaviour that was not internalised but purposely designed for future actions is called a *plan*. Plans are more obvious and require more consciousness than do scripts. They are also actively created whereas scripts are usually not. Although scripts can be seen as implicit algorithms, plans are more explicit algorithms for behaviour and situations.

Some cognitive scientists even conceptualised humans' information processing mechanisms as algorithms. This view is seen especially in information processing theory (IPT). There are different kinds of IPTs, but they are all similar in that information is seen as input that must be processed through several steps in order to create an output. The content and context of the problem are not of interest, and neither are subjective elements such as people's motivation or their perception of the problem. An effective example is the work of Newell, Shaw, and Simon (1958), which is also widely seen as the foundation for what is known today as artificial intelligence. Newell

et al. postulated a theory based on three conditions: (1) a number of memories, which contain symbolised information and are linked by various relations; (2) a definite number of primitive information processes, which operate on the information in the memories; and (3) a definite set of rules for combining these processes. Newell et al. transformed these conditions into a program called *The Logic Theorist* (LT) and compared its outcome with results from humans. Although the tasks they used to test LT were limited to mathematical proofs, they concluded that LT "is qualitatively like that of humans faced with the same task" (Newell et al., 1958, p. 155). Despite justified criticism, and acknowledging the limitations, of their work (see, e.g., Fetzer, 1998), Newell et al. succeeded in at least partially imitating the human mind based on the same logic as typified in algorithms.

So far in this section, knowledge representation and information processing were discussed based on the idea of algorithms. To some extent, humans also think algorithmically. This can be seen in in forms of specific games. A classic example of such algorithmic game is Minesweeper. At the beginning of the game, the player is presented with a board of grey squares. The squares are either empty or have a mine underneath. By left clicking, the user reveals what is underneath. The first left click never reveals a mine but opens several neighbouring squares. Some of these squares have numbers that indicate how many neighbouring squares contain mines. The goal of the game is to reveal all empty squares without left-clicking on a square with a mine, which would instantly result into losing the game. As a little help for orientation, players can flag squares by right clicking when they think a mine could be lurking there. Refer to Figure 2.2. Each square has up to eight neighbouring squares, so the player needs to consider information from more than only one square to determine where mines are and where it is safe to left click.



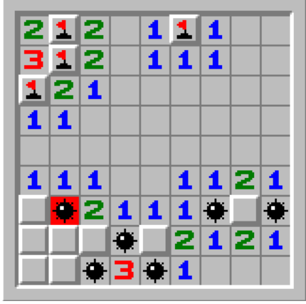


Figure 2.2. An example of Minesweeper. At the left a typical initial state of a game is shown; in the middle there is a lost game where a mine was triggered; and on the right is a game in which all mines had been correctly identified.

The gameplay mechanics can be written in plain English or pseudo-code:

```
1) Squares have two states {closed; revealed}

2) IF left-click (for the first time) on square THEN
      square is empty → change state to revealed
            AND change state of neighbouring squares to revealed
            IF they do not face at least 1 mine

3) IF left-click on closed square THEN
      EITHER square is empty → change state to revealed
            AND change state of neighbouring squares to revealed
            IF they do not face at least 1 mine
            ELSE indicate the total number of mines they face
      OR square has mine underneath → change state to revealed
            AND game ends with FAIL

4) IF right-click place flag

5) IF all empty closed squares are revealed THEN
            game ends with SUCCESS
```

Understanding the algorithm of the game mechanics is essential for winning the game. Probably the player needs some attempts to figure out how the gameplay works. For instance, a first-time player could think that every square with a number faces a mine. Of course this rule is not correct and the player needs some incorrect trials to realise the mistake. To put it differently, the mental algorithms must be debugged in some test sessions. As Simmons (1988) stated, debugging also means using reasoning techniques to handle problems. For instance, a player first states, and then tests, hypotheses about the origin and cause of the problems and how to logically eliminate them. Finally with experience, however, the player will probably understand the rules of this game and mentally designed an algorithm that can be followed for success.

Earlier algorithms were defined as tools to accomplish transformation of information from an input state to an output state with a definite number of well-ordered steps. Regardless of who operates the algorithm, the output will be the same (as long as the input is the same). It was earlier stated that debugging is an evaluation process that is inseparable from the overall design of algorithms. Consequently, algorithmic thinking in the sense of being able to mentally design algorithms means being able to recognise what input is available or had been used (e.g., left click or right click, and different states of variables), what steps in what order are needed (e.g., taking the information of

several different squares into consideration), and what the outcome is (e.g., empty square means winning; mine means losing). Of course, this also incorporates testing and adapting drafts of algorithms (i.e., testing ideas for rules and algorithms).

2.2.4.3 What algorithmic design means for computational thinking

The process of designing an algorithm involves several steps. Every action indicating the creation of algorithms (e.g., writing pseudocode, creating a flowchart, or coding a program) can be seen as the initial step in algorithmic design. This also includes clues by which users actively try to follow certain sequences in order to solve a problem. In addition, all actions aimed for adjusting the algorithm based on testing can be seen as following steps and are identified as debugging and evaluating, respectively. In contrast to the other two CT core skills (decomposition and abstraction), the design of algorithms is considered to be only part of a solution. So, clues of algorithmic design are likely to be evident at a later stage of the whole CT process.

2.3 Relationship of components

In the earlier sections, CT was analysed from two different perspectives to justify why specific skills are more likely to be associated with CT than others and how these skills are applied in CT. Although much research has been done to define and assess CT, little work has been done in order to identify the relationship between the associated skills and at which state of the problem-solving process particular skills might be more likely to occur.

All elements are interrelated and they all play a role at different times. To successfully apply an efficient step-by-step solution, it is necessary to recognise patterns in the problem. To do so, the problem has to be deconstructed into its elements. Similarities can be identified only if there are elements to compare with each other. In addition, subproblems and subsolutions can be handled independently. Not all elements of the initial problems have to be handled at the same time, but only the elements of the subproblem. This makes it easier to find patterns and to apply an efficient solution. Equally important is to focus on important information and to neglect details. It is possible to identify patterns only if distracting but unnecessary information is ignored. In addition, abstraction requires focusing on the main aspects of a problem and therefore only main aspects will be considered in the solution. Therefore, decomposition and

abstraction allow identifying patterns in the problem which leads to an efficient solution. This process is seen, for instance, in programming. Coding a function means to decompose a larger concept into a set of steps at the next level of abstraction (Martin, 2009, p. 36).

Although these skills are linked to each other, it is also plausible that the different skills are more dominant or occur more often at different times. Decomposition is considered to be part of reformulating the problem and so is seen as part of the preparation process, whereas algorithmic design is seen as part of the process of implementing the solution. That means at the beginning of the overall problem-solving process, actions of decomposition should take place more often than at later stages, and the opposite is true for algorithmic design. Ideally, problem solvers first analyse and decompose the problem before formulating any algorithmic solutions. Unimportant details could be neglected when analysing the problem or when considering several possible solutions, and the same is true for recognising patterns, which can occur over different instances of (sub)problems or possible solutions. Therefore, different forms of abstractions could be equally distributed over the whole problem-solving process.

2.4 Assessment of computational thinking

The assessment of CT has been the goal for many studies and workshops over the last few years (Moreno-León & Robles, 2014) and many different approaches have been developed. Because CT is based on major CS concepts, it is no surprise that there are some studies in which programming languages such as Python (Brancaccio et al., 2015) or VPython (Aiken et al., 2012) have been used to assess CT. However, using a programming language is not without problems. Participants need to be literate in that specific programming language. If there is one misplaced comma or semicolon, the program will not work as intended. Of course, some languages are more sensitive and complex than others, but they all have in common that minor syntactical errors can have a huge impact on the outcome. In addition, the relationship between CT and programming ability is still debatable (this will be discussed in more detail in section 2.5.2). That means using programming language, which requires considerable prior specific knowledge, might be not appropriate for assessing something more general such as CT.

An approach that requires less specific knowledge is use of pseudocode. Pseudocode fills the gap between informally describing, in plain language, what is happening in an algorithm and a coded algorithm written in a programming language (Roy, 2006). Pseudocode is not a single or precise system but rather a meta-language usually written in a spoken language such as English (Cormen et al., 2014, p. 17). The focus here lies more on semantics than on syntactics. That means it is more about knowing whether a “loop” or “if” command is needed in order to create a specific algorithm instead of knowing whether specific commands are separated by a comma or semicolon. This makes pseudocode less sensitive and more “forgiving” to the user and gives more freedom to observe more general thinking approaches such as CT. Pseudocode has been widely used for teaching programming and even for general problem solving long before CT emerged in the CS community (see, e.g., Olsen, 2005). So, it is plausible to use it as a method for assessing CT. Indeed, Davies (2008, p. 3) used pseudocode “emphazising computational thinking” and Grover, Pea, and Cooper (2015) used pseudocode along with other kind of assessment in their educational framework on CT to enhance programming skills. Although pseudocode is easier to understand than any programming language, some sort of specific knowledge is required. People still need to learn a set of vocabularies and a communication style (Roy, 2006). Especially for novices such as students with no or only little prior CS background, this appears to be a challenge and requires some preparation time. This is why two other approaches appear to be very promising when observing CT: using unplugged methods such as logical quizzes, and using programming environments.

2.4.1 Using unplugged methods

The term unplugged method, coined in the early 1990s, can be summarised as a collection of learning activities that teach computer science concepts without a computer (Bell & Vahrenhold, 2018; Rodriguez, Kennicutt, Rader, & Camp, 2017). These methods include haptic or kinaesthetic activities as well as logical quizzes. For instance, Curzon, McOwan, Plant, and Meagher (2014) created several different workshops in which students first learnt about the conceptual idea behind CT (i.e., what skills involve CT). In one of those workshops, the students applied these skills to analyse specific magic card tricks or to “program” a human robot. This included deconstructing the tasks in their core elements and finding algorithmic solutions. A similar task was used by (Rodriguez, Kennicutt, Rader, & Camp, 2017) in which

students needed to connect houses in a village by using a minimum numbers of stones. They also let students decipher a message by using binary code or explicitly naming the hidden "interaction rules" between a fruit vendor and a customer, showing that algorithms are part of everyday life (see "scripts" in Section 2.2.4.2). Brackmann et al. (2017) developed quizzes and tasks in which students needed to draw typical Tetris figures by only hearing the commands such as "start", "up", and "left" from another student to show algorithmic thinking.

Despite the popularity of unplugged methods, the effects on learning are still unclear. Thies and Vahrenhold (2013) tested whether unplugged methods can be useful to teach CS concepts such as binary numbers, binary search, and sorting networks to 25 students aged 11 to 12. For that, they assigned the students to a treatment and control groups. Members of the treatment group were taught the CS concepts by unplugged methods involving activities and actions. For instance, binary search was introduced to the unplugged method group by playing the classic game Battleship. For the control group, the same concept was introduced in a classic textbook fashion. The results indicated that for none of the three introduced CS concepts the kind of method makes a difference in learning. The authors concluded that unplugged methods have at least no negative effects on learning. To put it differently, no positive effects were found either.

2.4.1.1 The Bebras tasks

Arguably, among the most influential unplugged methods for assessing CT are the Bebras[3] tasks. These tasks are part of an annual international contest on informatics and CT, with over 2.6 million contestants[4] from over 45 countries in 2018. The tasks originated in 2004 as a competition for children and young adults (school year levels 3 to 12) in Lithuania (Dagienė, 2006). The tasks are mainly divided into five different age groups which vary slightly between the countries (Dagienė & Stupuriene, 2015). Studies show that these age categories parallel Piaget's theory of cognitive development (Lutz, Berges, Hafemann, & Sticha, 2019). In general, there is a strong tendency to use concrete material (e.g., realistic pictures) for younger groups and more abstract material for older groups. Each age group is categorised into three levels of difficulty (easy, medium, and hard). Analyses also indicate that these assumed categories substantially match with the perception of difficulty by the participants (Bellettini et al., 2015).

[3] Bebras is the Lithuanian word for Beaver and was chosen to encourage younger participants.
[4] https://www.bebras.org/?q=statistics

Before a Bebras task is accepted for the contest, it has to go through an intensive validation process on different national and international levels (Dagienė & Stupuriene, 2015, 2016). In the first step, a group of computer scientists and teachers of computer science create a draft of tasks. Nearly each country that participates in the Bebras contest has such a group of experts. The tasks are created based on the official guidelines for Bebras tasks. For instance, the problem should be clearly presented, easy to understand, and not be tricky; not be too easily solved (solutions should be attained between 1 and 4 min.); be at an appropriate difficulty level for the proposed age group; and be independent from any coding language but related to the CT concepts as described above (Dagienė & Futschek, 2008). In the second step, an annual workshop is held and experts selects the set of tasks being proposed and rejects, refines, or simply accepts those tasks for use in that year's contest (Dagienė & Stupuriene, 2015). Participants of this workshop are also computer scientists and teachers of computer science. This two-step process ensures a satisfying amount of content validity and provides a sound basis for assessing contestants' CT skills.

The main idea behind Bebras tasks is to create problems that require specific cognitive abilities rather than technical knowledge or coding experience (Dagienė & Stupuriene, 2015). This makes them promising for assessing CT and might be also the reason why they are not only being used in international challenges but are also intensively used in research (Dagienė & Stupuriene, 2016). According to the official guideline for creating Bebras tasks (Dagienė & Futschek, 2008), categorisation of the cognitive abilities is mostly congruent with the proposed CT major skills in this thesis. This is seen especially in the Australian versions of the Bebras challenge from 2014 and 2015 (see, for an overview, Schulz & Hobson, 2015; Schulz, Hobson, & Zagami, 2016, respectively), which were also the most recent tasks at the time of data collection of this study. The cognitive abilities are classified in four categories, the first three of which are breaking down problems into parts, interpreting patterns and models, and designing and implementing algorithms, which all correspond to the major CT skills proposed in this thesis as decomposing a problem, the ability of abstraction, and algorithmic design, respectively. The fourth category, described as organising data logically, also shows similarities to algorithmic design as used in this thesis.

An example of a typical Bebras task shows how CT skills must be applied to achieve a solution (Figure 2.3). The task is presented as a general problem in which someone wants to reach a target but cannot directly see it. To solve this problem, test takers have to think about several trials they have to make and think about where the arrow will land (decomposing the problem). To find the correct solution, contestants have to apply a binary search algorithm (applying algorithms). Abstraction in the sense of neglecting unimportant information may be involved here as well to be able to focus on the binary trials. Understanding binary search also includes some abstraction in the sense of pattern recognition.

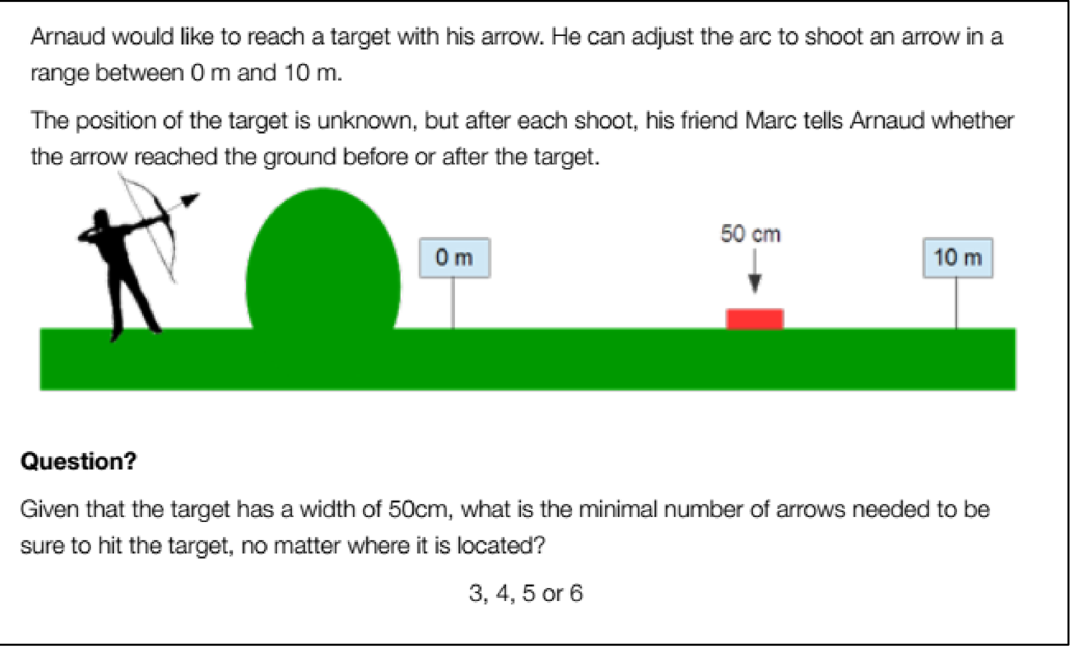


Figure 2.3. An example of as hard categorised Bebras task for school level year 11 and 12 students

To review the psychometric structure and features of the test, Araujo, Andrade, Guerrero, and Melo (2019) conducted a confirmatory factor analysis. They analysed answers from over 1,500 Lithuanian students to a set of the Bebras tasks used in 2015. They identified a two-factor solution. Factor one included skills such as decomposition, abstraction, and generalisation. Factor two was identified as comprising algorithmic thinking or reasoning. This result mostly fits the conceptual idea of CT as stated in this thesis with two perspectives, one focussing on reformulating the problem (i.e., decomposition and some abstraction) and the other focusing on the solution (i.e., some abstraction and algorithmic design). In summary, the Bebras tasks can be seen as a

promising way of assessing CT. First, the skills required to solve the logical quizzes fit to the identified major CT skills as described in this thesis. Second, the structure of these skills is compatible with structure as stated here.

Using unplugged methods such as Bebras tasks is not without problems however. As discussed earlier, CT is highly associated with the use of technology and is especially suitable for ill-structured problems. The Bebras tasks, on the other hand, are well-structured quizzes (i.e., there is always a single correct answer), intentionally having no association with technology. Thus, it is possible that not all facets of CT are covered when only using the Bebras tasks. In addition, it is not clear whether the Bebras tasks have a unidimensional structure and measure only CT. The Bebras tasks are deliberately developed to trigger CT-related skills such as decomposing, abstracting, and designing algorithmic solutions. However, it is not yet clear whether other cognitive constructs are measured as well. This is why another approach to assess CT should be considered as well.

2.4.2 Using visual programming environments

Visual programming environments (VPEs) are also a promising way to evoke and asses CT. VPEs are not based on any kind of written code or text. Instead, users “code” by assembling several different graphical elements representing specific functions. As with Lego® bricks, connectors on the blocks suggest how they should be put together. Blocks are shaped to fit together only in ways that make syntactic sense. This makes VPEs appear more like a puzzle-game than a programming language. This reduces the required knowledge about the environment to a minimum. Users only need to think about what commands generally make sense and not which blocks fit together. During the last decade several of those programming environments have emerged (see, for comprehensive overviews, Ching, Hsu, & Baldwin, 2018; Eguíluz, Garaizar, & Guenaga, 2018; Lye & Koh, 2014) and are favoured for assessing CT. For instance, Werner, Denner, and Campe (2012) designed three independent tasks in *Alice* in which participants needed to accomplished specific tasks to demonstrate CT-associated skills such as algorithmic thinking and abstraction. Werner et al. (2012) assessed the level of CT with a rubric-based scoring system that was designed for that purpose. Participants were highly motivated by this kind of task. Other popular approaches involve use of visual programming environments together with hardware applications such as *BBC*

micro:bit (Sentance, Waite, Hodges, MacLeod, & Yeomans, 2017), *Arduino* (García-Peñalvo, Reimann, & Maday, 2018), or *Calliope mini* (Lübbers & Jansen, 2018).

One reason for VPEs being popular might be because they are perfect to use for design problems when users can freely work on a problem. As mentioned earlier, design problems, in which multiple solutions might be equally favourable, are archetypal examples of ill-structured problems (Jonassen, 1997). CT was particularly discussed as the ability to reformulate and solve ill-structured problems, so it is no surprise that many studies about CT are based on such design tasks (e.g., creating a short story or game) using VPEs (see, for an overview, Lye & Koh, 2014).

2.4.2.1 Scratch

Among Alice, probably the most popular VPE for studying CT might be *Scratch* (Weintrop & Wilensky, 2018). Scratch emerged from a project by MIT's Lifelong Kindergarten Lab in 2002. The first full version was then created under the lead of Mitchel Resnick in 2007 (Resnick et al., 2009). Since then, it has been continuously enhanced and it has inspired many other programming environments because of its sophisticated design and user-friendly interface (Eguíluz, Garaizar, & Guenaga, 2018). Seiter and Foreman (2013) designed a CT measurement based on Scratch. Their idea of CT is similar to the conceptualisation in this thesis, with decomposition, abstraction, and algorithms as major concepts. Brennan and Resnick (2012) even claimed they could measure long-term effects of both conceptual understanding and the application of CT skills using Scratch over time. Their assessment design also included different design tasks with different levels of difficulty. In addition, Grover et al. used Scratch among other things for the formative and summative assessment of CT in their FACT (Grover, 2017; Grover, Pea, & Cooper, 2015), and Cernochova, Dorling, and Williams (2015) concluded that they successfully improved students' CT skills using Scratch. Even for younger participants such as elementary school students, a modified and simplified version of Scratch (namely *Scratch Jr*) has been shown to be useful in order to observe CT development (Falloon, 2016; Portelance & Bers, 2015). Overall, Scratch appears to be a promising tool to measure CT because of its low level of required prior knowledge about the system and its extensive use in computer science education research (Lye & Koh, 2014).

Another influential reason to use Scratch is Dr Scratch. Dr Scratch is an open web application in which projects created in Scratch can be autonomously analysed

according on their level of CT (Moreno-León & Robles, 2014). The analysis of a Scratch project using Dr Scratch is as simple as providing the URL of the project or uploading its saved file when working offline. The tool provides information regarding the development of (bad) programming habits (Moreno-León & Robles, 2015; Moreno-León, Román-González, Harteveld, & Robles, 2017). Its view on CT is more attached on programming than other models of CT, e.g. the idea of CT used in this thesis. Dr Scratch assesses the code of the programs for the purpose of assigning a score on different aspects of this competence, including, abstraction, problem-decomposition, logical thinking, synchronization, parallelism, flow control, user interactivity and data representation (Moreno-León & Robles, 2015). Although phrasing is different, the core dimensions named in Dr Scratch are still linked to the aspects of CT as used in this thesis. Each of these dimensions are measured between 0 and 3 points, and an overall CT Score is assigned by summing up the partial scores. Meanwhile, with regard to errors and bad programming habits, for every evaluated project, Dr. Scratch searches for codes that are never executed, checks the correctness of message synchronization among characters, searches for object properties that are wrongly initialized, discerns codes that are repeated in the programs of the characters and points to objects that are not named in a personalized manner. Based on the score, the feedback provided by Dr Scratch is different. There are three levels of CT development created, namely, basic, developing, master, as shown in Table 2.5. The purpose of these three levels is to prevent overwhelming novice learners and offering all available information to experienced users. In the light of these, the feedback information that is basically, the number of tips and errors showed in the report, is incorporated at each level.

Table 2.5

Level of Development for Each CT Dimension (Moreno-León & Robles, 2015)

CT aspect as used in this thesis	Dimension in Dr Scratch	Basic	Developing	Proficiency
Abstraction & Decomposition	Abstraction and problem decomposition	More than one script and more than one sprite	Definition of blocks	Use of clones
	Parallelism	Two scripts on green flag	Two scripts on key pressed, two scripts on sprite clicked on the same sprite	Two scripts on when I receive message, create close, two scripts when %s is > %s, two scripts

				on when backdrop change to
Algorithmic design	Logical thinking	If	If else	Logic operations
	Synchronisation	Wait	Broadcast, when I receive message, stop all, stop program, stop programs sprite	Wait until, when backdrop change to, broadcast and wait
Abstraction	Flow control	Sequence of blocks	Repeat, forever	Repeat until
Decomposition	User interactivity	Green flag	Key pressed, spite clicked, ask and wait, mouse blocks	When % is >%s, video, audio
Algorithmic design	Data representation	Modifiers of sprites properties	Operations on variables	Operations on lists

There had been attempts of standardised assessment of CT before. For instance, Brennan and Resnick (2012) visualised how often particular code chunks were used by different user profiles. In 2013, Boe et al. designed Hairball, a tool that tries to detect errors in projects. However, both approaches had been used to provide valuable overall evaluations of projects but did not make any inferences the level of CT. Inspired by Hairball, Dr Scratch was created to fill this gap. It is one of the first of approaches that provides a score for CT based on quantitative analysis of Scratch projects.

Nonetheless, it is critically to note that the developers did not clearly state how they chose their dimension for Dr Scratch and how dimensions are operationalised. That leads to the question whether all of the seven stated CT dimension in Dr Scratch are indeed crucial for indicating CT. For instance, flow control and user interactivity might rather crucial concepts for programming (Watt & Findlay, 2004) rather than for CT. In addition, it is not obvious how the decision was made how the usage of specific kinds of code chunks indicates a basic, developing or proficiency level.

To validate their instrument, the developers compared the Dr Scratch metrics with classic software engineering metrics such as cyclomatic complexity and Halstead's metrics. For more details, see McCabe (1976) and Halstead (1977), but, in short, both measures take into account variables such as the number of distinct operators in the

software and the overall complexity of a program (Moreno-León, Robles, & Román-González, 2016). It must be noted, however, that Dr Scratch CT assessment solely based on coding elements as used in Scratch. Someone's level of CT is based on how well they handle Scratch. To further validate, the developers compared Dr Scratch results with the judgment of computer science (education) experts (Moreno-León, Román-González, Harteveld, & Robles, 2017). They asked the experts to provide "an assessment for the technical mastery of the project" (p. 2791). High correlations between Dr Scratch scores and experts' judgement indicate high convergent validity. However, it is possible that the experts rather evaluated the programming proficiency of the Scratch project rather the level of CT.

For Armoni, Meerbaum-Salant and Ben-Ari (2015), Scratch is not "real" programming. They studied the use of the Scratch environment for teaching CS concepts to middle school students after investigating how these concepts were successfully learnt. In their 2015 study, they explored CS within the visual Scratch environment in middle school in comparison to CS within a professional textual programming language (C# or Java) in secondary school. Armoni et al. (2015) found that programming knowledge and experience of students who had learnt Scratch significantly facilitated learning the more advanced material in secondary school. Students who learnt CS through the visual Scratch environment in middle school learnt new topics more quickly, encountered less learning difficulties, and attained higher cognitive levels of understanding of most concepts. Moreover, these same students who learnt CS using Scratch had higher levels of motivation and self-efficacy in enrolling in advanced CS classes. Overall, Armoni et al. (2015) assert that because of these findings, teaching CS in general and visual programming are well-justified. In conclusion, Dr Scratch is a valuable tool for the assessment of Scratch projects, but the Dr Scratch CT-mastery score might be biased toward programming ability of Scratch rather providing a holistic measurement of CT.

2.4.2.2 Comparison between unplugged and plugged methods

Interest in programming education has exponentially increased over the past decade (Hermans & Aivaloglou, 2017). Indeed, the number of countries where schools are including programming and CT in the curricula of elementary schools, has rapidly increased. Introducing programming to education raises questions about how best to teach programming and CT, including the role of the computer, and whether teachers

should used plugged tools and instruments or unplugged methods where there is no need for computers.

Hermans and Aivaloglou (2017) emphasise the need for children to be able to apply programming concepts using a computer, which means that it is necessary to know how plugged approaches and systems compare to unplugged ones. Therefore, they conducted a study to determine whether it is better to start with plugged lessons immediately, or first use unplugged materials. Specifically, they were interested in determining which method is more effective in (a) facilitating understanding of programming concepts, (b) motivating and supporting the students' self-efficacy in programming tasks, and (c) motivating students to explore and use programming constructs in their assignments. To answer these research question, they conducted a two-phase experiment through which they compared starting with unplugged lessons with starting on the computer.

The researchers taught 35 elementary school children aged eight to 12 years old who were designated to two different random groups for eight weeks. For the first four-week phase, 17 children were taught Scratch, while the remaining half (18 students) used unplugged materials only. Both the plugged and the unplugged lessons covered the same concepts of loops, conditionals, procedures, broadcasts, parallelisation, and variables. After the four weeks, both groups were provided with two weeks of Scratch lessons, so that that can practice Scratch programming at greater depth. In these lessons, the same concepts were covered as in the first phase. Meanwhile, for the unplugged group, a special lesson was taught to the students wherein they learnt to use unplugged to concepts in Scratch. After the two weeks, two more weeks followed wherein the children created their own games in Scratch. The experiments were concluded by administering a test to students in order to evaluate how correctly they used programming concepts in Scratch. The results of this study show that after a total of eight weeks, (a) there was no difference between the two groups in their mastery of programming concepts, (b) the unplugged first group demonstrated stronger self-efficacy, and (c) they also used a wider vocabulary of Scratch blocks, including more blocks that were not explained in the course materials.

2.5 The relationship between computational thinking and other concepts

There is a vigorous debate about what CT is, but there is little discussion about what CT is *not*. On the one hand, the concept of intelligence shares some remarkable overlap with CT. Programming, on the other hand, is often mentioned as a skill that eventually can emerge from CT because of some similarities. The conceptual overlaps between CT on one hand and intelligence and programming skills on the other, are investigated more extensively in the following sections.

2.5.1 Intelligence as general problem-solving skill

Because CT is a cognitive skill, it is important to investigate its relationship to potentially similar skills. What is broadly known as intelligence shares two properties that lead to the assumption of some conceptual overlaps. First, both concepts are considered as (general) problem-solving approaches, and second, in both concepts the ability of abstract thinking plays a predominant role. Both aspects are discussed in more detail immediately below.

Although intelligence has historically been a controversial construct, many definitions propose intelligence as the ability to solve problems and to reason abstractly. Gardner, for example, described intelligence as a summary of such skills (Gardner, 1983, pp. 60–61):

> A human intellectual competence must entail a set of skills of problem solving — enabling the individual to resolve genuine problems or difficulties that he or she encounters, and, when appropriate, to create an effective product — and must also entail the potential for finding or creating problems — thereby laying the groundwork for the acquisition of new knowledge.

In the following decade, a group of experts in research of intelligence and similar fields (Gottfredson, 1997, p. 13) signed a statement about intelligence:

> Intelligence is a very general mental capability that, among other things, involves the ability to reason, plan, solve problems, think abstractly, comprehend complex ideas, learn quickly and learn from experience. It is not merely book learning, a narrow academic skill, or test-taking smarts. Rather it reflects a broader and deeper capability for comprehending our surroundings — ‘catching on’, ‘making sense’ of things, or ‘figuring out’ what to do.

Later in the statement it is emphasised how intelligence is different from other psychological concepts such as creativity or personality and how intelligence is strongly connected to problem comprehension and problem solving. Although there is no official and final definition of intelligence, this statement has been often used as a benchmark for other researchers who also included or even emphasised problem solving and (abstract) reasoning. Jensen (2002, pp. 39–40), for instance, described intelligence as:

> an open-ended category for all those mental processes we view as cognitive, such as stimulus apprehension, perception, attention, discrimination, generalization, learning and learning-set acquisition, short-term and long-term memory, inference, thinking, relation education, inductive and deductive reasoning, insight, problem solving, and language.

Nickerson (2011, p. 108) defined intelligence as “the ability to learn, to reason well, to solve problems, and to deal effectively with challenges — often unpredictable — that confront one in daily life.” In summary, many experts share the same view about how problem-solving abilities and abstract reasoning (among some other skills) together define intelligence.

Furthermore, these views of experts overlap mostly with those of non-experts. Sternberg, Conway, Ketron, and Bernstein (1981) asked nearly 200 subjects in college libraries, at train stations, or at supermarket entrances about their idea of intelligence. They generally tended to believe that a concept of intelligence should include problem-solving abilities. Sternberg asked 200 professors in art, business, philosophy, and physics to rate the characteristics of intelligence and their idea of an ideally intelligent person (Sternberg, 1985). Results differed little from previous studies and again naïve theories about intelligence consist of logical thinking (including good memory and good vocabulary). An intelligent person was mainly described as a good problem-solver and as someone who thinks reasonably.

Sternberg (2004) also stated that implicit theories are culturally dependent to some extent. For instance, in Asian as well as African cultures, interpersonal social skills are more closely associated with intelligence than in western cultures. However, Sternberg also made clear that neither Asian nor African cultures base their ideas of intelligence only on social aspects. General cognitive abilities such as problem solving or reasoning are highly associated with intelligence regardless of the cultural background but social notions play a greater role in eastern and African cultures. Interestingly, this cultural influence has apparently a bigger impact when participants are supposed to describe

their own intellectual abilities instead of others. When Korean participants were asked to estimate other people's intellectual abilities, they emphasised problem-solving ability higher than social skills (Lim, Plucker, & Im, 2002).

It is important to note that a person who is considered to be intellectual is perceived as a *general* problem solver. Wechsler, who developed one of the most popular tests on intelligence, regarded intelligence as "aggregated" or "global" and that intelligence is composed of different abilities that are qualitatively different to some extent (Wechsler, 1958, p. 7). This idea of intelligence as a general cognitive ability is mainly based on the findings of Spearman (1904). Investigating the relationship between different cognitive tests, Spearman noticed that all tests were positively correlated to a nearly equally large extent with each other. Spearman concluded that each test measures its own specific factor (s-factor) but also a general factor that is in common to all tests. He coined this commonality g-factor for general ability or general intelligence.

Stadler, Becker, Gödker, Leutner and Greiff (2015) conducted a study to determine the empirical relationship between complex problem solving (CPS) and intelligence by meta-analytically summarizing the various research findings on the correlation of CPS and intelligence. The researchers also sought to determine the moderating factors that might help explain the contradicting findings. Showing that there is a considerable albeit far from perfect correlation between various measures of CPS and intelligence, Stadler et al. (2015) provide essential information on the construct validity and nomological classification of CPS. Following a definition by Buchner (according to Frensch & Funke, 1995, p. 14), where CPS is understood as:

> (…) the successful interaction with task environments that are dynamic (i.e., change as a function of the user's interventions and/or as a function of time) and in which some, if not all, of the environment's regularities can only be revealed by successful exploration and integration of the information gained in that process.

According to this definition, it becomes obvious why CPS is usually compared to intelligence on a conceptual basis to (a) establish discriminant validity, or to (b) define individual attributes that help explain performance in CPS tasks (Stadler et al., 2015). On the one hand, some defining characteristics of CPS such as integrating information is part of nearly all definitions of intelligence (Sternberg & Berg, 1992). Meanwhile, the dynamic and transparent characteristics of complex problems are not well-established aspects of the present conceptualization of intelligence such as the Cattell–Horn–Caroll (CHC) theory of human intelligence (McGrew, 2009). This view of CPS can potentially

add to the understanding of human abilities (Stadler et al., 2015). These divergent theoretical views are evident in empirical findings on the relation between CPS and intelligence. Many studies have previously shown that although performance in CPS tasks greatly vary between individuals, psychological assessments of general intelligence cannot explain this variability (Rigas & Brehmer, 1999).

Kluwe, Misiak, and Haider (1991) synthesized 11 previous studies on performance in CPS tasks and relationship between CPS and intelligence, and found that the majority of these studies fail to show a close relationship between intelligence scores and CPS performance measures. Consequently, some researchers began to suggest that CPS is a cognitive construct that is largely independent from intelligence (Stadler et al., 2015). Rigas and Brehmer (1999) explain this perspective according to the different-demands hypothesis. According to this hypothesis, the weak correlations researchers observed between measures of general intelligence and CPS performance, is explained by how CPS tasks demand performance of more complex mental processes than intelligence tests typically do, such as, the active interaction with the problem to acquire knowledge on the problem environment, which, in turn, leads to weak empirical correlations between the constructs.

Wirth and Klieme (2003) conducted a study on the same constructs and found a correlation of $r = .84$ between a latent factor of different measures of CPS and general intelligence. The latent factor scores on MultiFlux, a newer measure of CPS showed a latent correlation of $r = .75$ with various aspects of the Berlin Model of Intelligence Structure (BIS) test, which is an established intelligence test (Kröner, Plass, & Leutner, 2005). More recent investigations on relationships between CPS and intelligence also show moderate to strong latent correlations of the two constructs (Greiff, Wustenberg, Molnar, Fischer, Funke & Csapo, 2013; Sonnleitner et al., 2012; Wüstenberg, Stadler, Hautamäki, & Greiff, 2014; Wüstenberh, Greiff & Funke, 2012). Notably, these empirical investigation also show incremental value over and above measures of intelligence in predicting school grades (Wüstenberg et al., 2012) and job success (Danner, Hagemann, Schankin, Hager & Funke, 2011) despite strong correlations in other studies, and in support of the different demands hypothesis.

Stadler et al. (2015) explain that inconsistent findings about the relationships between CPS and intelligence could be due to the conceptualization of intelligence. Nearly all theories of psychometric intelligence at this point in time include one or two

very broad, latent factors of general intelligence that capture a significant proportion of all cognitive abilities, including, abstract reasoning, memory, or factual knowledge (McGrew, 2009). From this viewpoint, researchers have undertaken studies on the relationships between CPS and intelligence and usually covered broad measures of general intelligence as well as, used different tasks to evaluate and measure cognitive abilities, such as, factual knowledge or general crystalized intelligence (McGrew, 2009). In contrast, subsequent studies focused more on specific sub-facets of intelligence, and in particular, reasoning, which was theoretically and empirically determined to be conceptually closest to CPS (Stadler et al., 2015). Reflecting on the different-demands hypothesis, broad measures of intelligence do address different aspects that may not be relevant for the successful solution to a complex problem, including, factual knowledge, thereby constraining the empirical relation between CPS and intelligence. However, evaluations that focus on reasoning reflect “the use of deliberate and controlled mental operations to solve novel problems that cannot be performed automatically” (McGrew, 2009, p. 8) are conceptually more similar to CPS than very broad measures of general intelligence. This could result in considerably stronger correlations between CPS and intelligence (Greiff et al., 2013, Wittmann & Hattrup, 2004; Wittmann & Süß, 1999). Based on these, researchers’ conceptualisation of intelligence in a study may impact the relationship between CPS and intelligence found with higher correlations of CPS, and reasoning than of CPS and broad measures of general intelligence.

Overall, theoretically, researchers have hypothesized the two constructs of CPS and intelligence to be everything from completely separate to identical (Stadler et al., 2015). Over the course of roughly four decades of research, empirical studies have been showing that either CPS and intelligence are totally different from each other, or CPS and intelligence are nearly identical in characteristics. The meta-analysis of 47 studies containing 60 independent samples and a total sample size of 13,740 participants, indicate a considerably strong correlation between CPS and intelligence with an average effect size of $M(g) = .433$ (Stadler et al., 2015). Moreover, the same researchers studied whether the operationalization of CPS and intelligence moderates this correlation. Although there have been no major correlation differences considering the operationalization of intelligence, the approach used for measuring CPS moderates the correlation of CPS and intelligence (Stadler et al., 2015). In particular, the most recent approach toward assessing CPS yields the strongest correlations between CPS and intelligence (Stadler et al., 2015).

2.5.1.1 Meaning of abstract thinking in theories about intelligence

Some even consider the ability to abstract as being the distinction between human and nonhuman intelligence (Deacon, as cited in Gabora & Russon, 2011, p. 329). As Terman (1921, p. 128) stated: "An individual is intelligent in proportion as he is able to carry on abstract thinking." This is why abstract reasoning plays a major role in developmental psychology and various theories of intelligence.

Binet, for example, at the beginning of the 20th century developed one of the first intelligence tests (Mackintosh, 2011, p. 5). The purpose of this test was to identify children who were in need for special education and those who are not. To do this, Binet's concept of intelligence consisted of different "higher order" thinking processes such as abstract thinking. Especially Piaget (1952, 1960) focused in his theory of intellectual development of children's ability to think abstractly. According to Piaget, the ability to reason abstractly is a crucial element in the formal operational stage (the last stage of development in his theory). In addition, empirical studies could show how younger children have a lower ability for abstract thinking and they need more concrete examples for learning than do older children or teenagers. Younger children also tend to use concrete examples rather than abstract ones when they are instructed to explain concepts (Fischer & Kenny, 1986; Kitchener, Lynch, Fischer, & Wood, 1993). Brooks (1981) illustrated how younger children and children showing lower intelligent behaviour have difficulties recognising and learning from prototypical images. Because abstract reasoning is the crucial aspect of interpreting prototypes, Brooks further concluded that this ability was less developed in both groups. Using abstract thinking as an indicator for the development of human cognition underpins the strong relationship between abstract thinking and intelligence.

This strong relationship also explains why the ability to abstract or abstract reasoning is often prominently incorporated in different theories about intelligence. Thurstone (1938) proposed seven human cognitive skills that he identified as "primary mental abilities". Thurstone emphasised the independence of these abilities, but many of them are based on being capable of recognising patterns. For instance, the ability "inductive reasoning" includes recognising patterns in a sequence of numbers. This idea is similar to Gardner's theory of multiple intelligences (Gardner, 1983). As Thurstone had done, Gardner claimed to have identified qualitatively independent abilities, and, as in Thurstone's theory, many of those multiple intelligences have to do with abstract

thinking in the sense of recognising patterns. For instance, someone who has high "spatial intelligence" can quickly and easily interpret visual patterns.

Probably the most prominent emphasis on the ability to abstract as a facet of intelligence may be seen in the work of Cattell and Horn (Cattell, 1963; Horn & Cattell, 1967) with a major revision by Carroll (Carroll, 1993), resulting in one of the most popular theoretical frameworks about intelligence: the Cattell-Horn-Carroll theory (CHC). According to the CHC, human intelligence is organised in a three-stratum hierarchy with a list of 70 to nearly 100 narrow cognitive abilities at its lowest stratum (the number varies across the literature, see Flanagan & Dixon, 2014; McGrew, 2005). These specific abilities are factorised into eight broader abilities at the second stratum and the g-factor at the third-topmost stratum. One these eight abilities is the general or fluid reasoning factor g(f) (Carroll, 1993, pp. 196–200) that subordinates any kind of abstract thinking across a variety of domains including novel problems. It includes deductive reasoning (top-down inference; drawing a conclusion about a specific case based on a general statement) and inductive reasoning (bottom-up inference; drawing a conclusion about a general statement based on a specific case). Because little or no language is involved, g(f) is sometimes referred to as nonverbal intelligence.

The close relationship between abstract thinking and general intelligence can be also seen empirically. Carroll (1993, p. 233) emphasised the high loading of g(f) on g throughout different studies, and Kvist and Gustafsson (2008) even demonstrated that there is basically no difference between g(f) and g for homogenous samples (i.e., when subjects have had equally good or bad opportunities to develop abilities). It would be an exaggeration to assume that abstract thinking and what is usually known as general intelligence should be regarded as the same concept. However, theoretically and empirically, abstract thinking can be seen as the core of human cognitive competence (Lohman & Lakin, 2011, p. 430). That also leads to the conclusion that abstract reasoning might be the best estimation of general intelligence if more sophisticated measurement of intelligence might not be possible or available (e.g., time issues).

It is worth mentioning that not all definitions of intelligence refer explicitly to problem solving or abstract thinking. Humphrey, once chairman of the APA Task Force on ability and achievement testing (Lubinski, 2004), saw intelligence rather as "the resultant of the processes of acquiring, storing in memory, retrieving, combining, comparing, and using in new contexts information and conceptual skills" (Humphreys,

1979). Mayer and Salovey suggested with their idea of *emotional intelligence* (see, for an overview, Mayer, Salovey, Caruso, & Cherkasskiy, 2011) a new perspective on the whole concept that does not focus on abstract thinking and problem solving. Nevertheless, as shown here, an overwhelming number of theories about intelligence as well as naïve ideas indicate that problem solving and abstract thinking are regarded as crucial parts of human intellectual competence.

2.5.1.2 Differences between computational thinking and intelligence

Despite the similarities of CT and intelligence are still different concepts. First, intelligence has an assumed strong biological component in contrast to CT. Second, CT is more associated with technology than is intelligence. Third, the scope of problems in both concepts is different.

Since its first appearance, it is assumed that intelligence has strong links to biological processes. Galton who was one of the first who worked on intelligence in the late 19th hundred was convinced that a more intelligent person has finer sensory discrimination and therefore is capable of storing and acting upon more sensory information (Mackintosh, 2011, pp. 3–4). Although we now know that not every kind of sensory perception is linked with intelligence, there is no doubt today that such linkages exist (Haier, 2011, p. 351). A number of imaging studies have shown how different intellectual activities are mapped onto our brain and how g has a high hereditary component (Toga & Thompson, 2005). However, a biological link has not yet been proposed for CT. Frameworks about CT emphasise that it is a skill that can be acquired over time and with practice by everyone (see, e.g., Brennan & Resnick, 2012; Lockwood & Mooney, 2018a; Wing, 2006).

As seen in the definitions of intelligence provided above, no specific technology is involved when talking about intelligence. Intelligence is simply described as a characteristic of the human mind. This is different for CT which is often mentioned in conjunction with some sort of technology such as using computers to solve problems. Nevertheless, it is important to acknowledge that CT does not occur only in conjunction with technology, it is more strongly associated with the use of computers than is intelligence.

In addition, Wenke and Frensch (2003) questioned the generality of intelligence. They had difficulties finding convincing empirical evidence that people who score high

on intelligence tests are also generally good problem solvers. Although it is not unusual to find high correlations when problems are simple and well-defined, Wenke and Frensch reported that correlations decline with an increase in the complexity of problems. They further stated the high correlations could be caused by the high similarities between the used instruments. Intelligence tests are traditionally based on well-structured problems seldom use complex and ill-structured problems. That leads to the conclusion that intelligence can be seen as the general ability to solve problems but only when the problems are well-structured. Correlations are lower when problems are more complex and ill-structured such as those that govern a fictional city (Dörner, Kreuzig, Reither, & Stäudel, 1983). On the other hand, as explained in section 2.2.1, CT is regarded to be an approach for solving ill-structured problems with methods usually used for well-structured problems. So the scope of CT is different. Although people with high intelligence scores might do well in solving well-structured problems, people with high CT levels might excel in solving complex problems.

In summary, CT and intelligence show some conceptual overlaps. First, both are concerned with solving problems. Although both concepts are not associated with the same kind of problems, mutual dependencies are still conceivable. Second, the distinct role of abstraction is present in both concepts. Some theories about intelligence even do not distinguish between high intellectual performance and high abstract thinking, and in CT the ability to abstract is often considered as a keystone. Despite the difference, a study with the goal of observing the effects of CT should take these similarities into account and consider a measure of intelligence with a focus on abstract thinking. This way the unique attributes of CT can be analysed more precisely.

In addition, algorithmic thinking could be a distinctive facet of CT whereas the ability of abstraction is not. Thus, a more distinct definition increases the divergent validity of CT and could lead to instruments with higher discriminate validity. On the other hand, the strong correlation between CT and intelligence could also be an indication that they two are naturally related. For example, CT may even be regarded as a component of general intelligence. Indeed, there are theories on intelligence based on the idea of several mental abilities or multiple intelligences, such as that of Gardner (1983). Spearman (1904) already explained that positive relationships between these different abilities can be summarized as a g(eneral) factor of intelligence. Here, CT could be a cognitive ability that may be classified as a g factor. The only possible

explanation as to why CT is not considered as a g intelligence is because it is new and its relation to other cognitive abilities is unclear.

2.5.2 Programming quality

CT is a cognitive skill and this is why some theoretical overlaps with other cognitive concepts such as intelligence can be assumed. CT also emerged from CS, so technology and computers play a role. Although there is a common understanding that CT and programming are not the same (Lockwood & Mooney, 2018a; Wing, 2006), it is conceivable that CT overlaps at least to some extent with CS-associated skills.

Wing famously stated that CT is primarily about humans and not about machines. CT is "a way that humans, not computer, think" (Wing, 2006, p. 35) because obviously computers do not think. Additionally, participants of the NRC workshop concluded that technology is part of CT (NRC, 2010). From that perspective, CT means to find and apply the appropriate technology to reformulate a problem so that a computational solution is possible (Bocconi et al., 2016). CT involves asking "How would I get a computer to solve this problem?" (Wing, 2008). The computer can be understood as an agent for "computational thoughts". The computer executes the human's abstract cognitive procedures in concrete actions and arrives at the solution. It may be an exaggeration to say that CT depends on using computers, but it can be assumed that computers can play a crucial role for solving problems in the context of.

One way to solve problems through computers is programming. The European Digital Competence Framework for Citizens defines programming as the ability to "plan and develop a sequence of understandable instructions for a computing system to solve a given problem or perform a specific task" (Vuorikari, Punie, Carretero, & van den Brande, 2016). This is similar to the definition published by the Massachusetts Department of Elementary and Secondary Education (MDESE), which referred to programming as "the craft of analysing problems and designing, writing, testing, and maintaining programs to solve them." (MDESE, 2016, p. 56) Most people do not distinguish between programming and coding and use both interchangeably in daily conversation. Nonetheless, to be precise, coding has a narrower meaning than programming. Coding is the act of writing a computer program in a specific programming language (MDESE, 2016, p. 48). In this sense, coding means to implement a specific solution to a programming problem. For Bornat (as cited in Bruce

& McMahon, 2002, p. 23), knowing how to code means having knowledge about a specific programming language and how to write a line of code that is syntactically correct. Knowing how to program means having a broader knowledge about the principles of different programming languages.

This is why CT is often strongly associated with programming. As for CT, programming is concerned with solving problems. The difference, however, lies in the kind of problems at hand. Although programming is concerned only with problems that can be solved using coding, CT is not limited to that. Its scope is wider and not strictly limited to CS-related problems (Lamprou & Repenning, 2018; Shute, Sun, & Asbell-Clarke, 2017). As stated earlier, CT is a specific way of transforming and approaching a problem so that a computer can help to solve it. These problems can be programming tasks but also extend beyond that. Despite that, programming problems are always computational thinking problems (Figure 2.4).

This is the assumed relationship in theory and so programming activities are sometimes used to teach CT (Ching, Hsu, & Baldwin, 2018; Lye & Koh, 2014) or CT is used to teach programming (Davies, 2008; Grover, 2017; Grover, Pea, & Cooper, 2015). However, it is still unclear how CT is actually applied when solving programming problems and how much CT is actually involved in programming activities. Researchers have so far looked at both concepts separately but have not analysed the whole programming process and how much CT is actually involved.

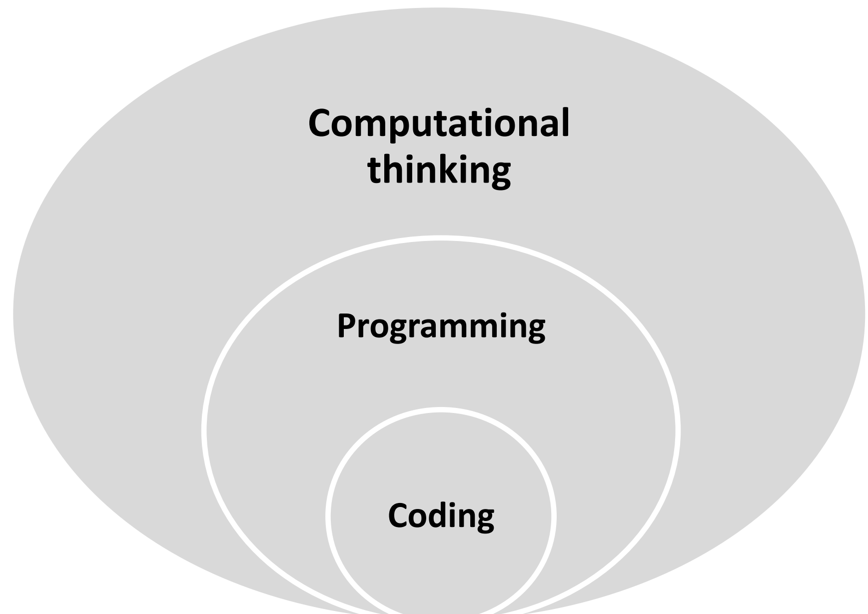


Figure 2.4. Assumed conceptual relationship between CT, programming, and coding.

2.5.2.1 Assessment of programming quality

Assessment of programming skills has been shown to be difficult. Unlike intelligence, for which different well-developed tests exist, there are no standardised measurements for programming. There are some frameworks about marking programming skills, but they mainly focus on mistakes. For example, Tabanao and colleagues developed a system to analyse the kind of errors that occur when novice CS students compile their code in Java (Rodrigo, Tabanao, Lahoz, & Jadud, 2009; Tabanao, Rodrigo, & Jadud, 2011). Luxton-Reilly et al. (2017) went one step further and drew conclusions about students' knowledge about crucial CS concepts, but they also made their inferences based on wrong syntax or wrong commands in code. These approaches come in very handy when the goal is to simply focus on correct coding in a specific programming language, but, as described above, programming consists of more than that. Correct coding is only one facet of the ability to program.

In his book *Clean Code*, Martin (2009) proposed the idea of programming as craftsmanship. Instead of focussing on a specific programming language, Martin gave an overview of general principles concerning what he called good programming practice. Based on that, several major aspects that distinguish good code from bad code

are derived. These principles allow a more holistic view about programming quality than is possible by looking only for someone's erroneous code.

Most programmers may think that "getting the code working" might be the most important task for a professional programmer. However, this is often not true. Of course, a code needs to work but the functionality might change over time and version of releases. The readability, on the other hand, has a profound effect on the ability of others to maintain the code. If the code is difficult to read, it is difficult to adjust its purpose and functions over time. As Martin (2009, p. 76) stated, "the coding style and readability set precedents that continues to affect maintainability and extensibility long after the original code has been changed beyond recognition". The readability is an indicator for the future usage of the code. A code is readable if it is simple and clear, and if expressions are in a tight order. This also includes the formatting style. For example, functions related to each other (i.e., call themselves or have similar purposes) should be closely located.

The readability of a code is also determined by its clearness of intention. If the intention is clear, it is simple to maintain and extend it not only for the original programmers but for other programmers as well. A good code "should clearly express the intent of its author. The clearer the author can make the code, the less time others will have to spend understanding it" (Martin, 2009, p. 175). One strategy to achieve this is to adhere the *keep it simple and stupid* (KISS) principle. The meaning of KISS is twofold. First, if there are more solutions for a problem, the simplest one is the best. This principle is comparable to similar principles in different disciplines such as Occkham's razor in science in which the simplest model or theory is preferable. Second, it means one function should do only one thing. Several nested functions would do more than one thing. This causes the level of complexity to rise and so the level of readability decreases (Martin, 2009, pp. 35–36).

A good code is an efficient code and a code is efficient when just the right number of expressions and commands are used and duplication of expressions and commands is avoided. This is meant by the *once and only once* or *don't repeat yourself* (DRY) principles. Duplications inflate the code unnecessarily. It becomes more difficult to read and understand. Martin (2009, p. 289) called duplications "missed opportunity for abstractions" to a higher level. As described in more detail in Section 2.2.3.1, abstraction in codes creates the opportunity for freedom and independence. With a high

abstract code, fewer manual changes are needed in later maintenance or extensions, and, with less manual interference, errors are less likely.

2.5.2.2 Computational thinking and programming

Computational thinking and programming are related to each other. "CT is that type of thinking used when programming on a computer or developing an application for another type of digital device" (Fraillon, Ainley, Schulz, Duckworth, and Friedman, 2019, p. 3). CT is considered as a specific function for computer scientists for learning how to use a computer (Li et al., 2020). Prevailing views that associate CT with computers and/or programming may easily lead people to believe that CT is specifically for computer science professionals. In view of this, it would be more challenging for many to understand why CT is important to everyone and most professions. With a certainty programming, at least in the way it is perceived from the work of professional programmers, is perceived to be difficult (Li et al., 2020). For example, to develop a software for a computer's internal operations would be highly relevant for professionals in computer science, but out of reach to many others. Similarly, abstraction and modelling with the use of CT in many professional fields beyond computer science would be seen as unimportant and of marginal concern for most people.

With emphasis on computing and programming in CT, it may be said that CT has not been highlighted in traditional school education wherein course requirements in computer science or programming are minimal or altogether absent. Wing (2006) must be credited with the understanding of CT as future-oriented and important to everyone. By directly associating CT and concepts that are essential to computer science, Wing (2006) substantially contributed to the current movement of computer science education for all in the United States (PITAC, 2005; White House, 2017). However, there are continuing challenges for teachers and education researchers who have been accosted with the difficulties in seeking to understand the meaning of CT, its assessment as well as usefulness for everyone (Denning, 2017). To note, accessing and using CT could be undermined in different ways by equally different expectations of training in computer science as a pre-condition. It cannot be overstated that computer science itself is no longer considered as the study of phenomena pertaining to computers but rather is understood as the study of computational information processing both natural and artificial (Denning, 2007). On the other hand, human thinking can also be defined according to specific models of information processing when undertaking different

tasks (Anderson, Bothell, Byrne, Douglass, Lebiere & Qin, 2004). The associations between computing and human thinking in information processing implies the possibility of taking the CT construct to a more generalizable level.

In particular, CT needs to be regarded as a model of thinking, which makes CT more about thinking rather than computing (Denning, 2007). Because computing is the study of natural and artificial information processing, this means to say that CT pertains to the search for various ways to process information "that are always incrementally improvable in their efficiency, correctness, and elegance" (Denning, 2007). In turn, this necessitates improvement in using appropriate strategies, such as, abstraction and modelling, practice, skill acquisition and improvement. In this regard, there are various forms of information at varying levels of abstraction, and as such, seems like various representations that are customizable and used in different disciplines for problem solving, modelling, and system building (Li et al., 2020). In the same way that people design and can design, every person including students also process information, and the task of educators is to help them (Li et al., 2020).

Even though programming and coding can be part of CT, the latter should not be constrained to computer science because as mentioned earlier, CT is widely applicable to diverse professional fields and in day-to-day living. For instance, computational modelling is currently harnessed for summarizing and analysing data (as code in CT) in various ways to help forecast ongoing trends in the coronavirus crisis, in multiple countries (Li et al., 2020). In the event that there is no accurate data for coronavirus or CT, government agencies and health organisations cannot effectively monitor and manage the crisis and in turn, this can lead to massive loss of life (Li et al., 2020). Moreover, if there is no adequate attention to improving information processing efficiency and elegance, there could be a resulting loss in opportunities to nurture students' CT and develop skills that prepare them to deal with global crises. Thus, it is utterly important that school curricula and instruction integrate CT in students' subject content learning, rather than just limiting CT to computer science and mathematics. CT should be applied in other STEM disciplines and beyond (Li et al., 2020).

It is also important to point out that although commonly used definitions of CT are emerging in literature, these new definitions tend to be ambiguous about how people acquire CT so that they can transform into computational thinkers. To clearly determine whether is truly a super-set of programming, as well as, its importance, it is helpful to

recall the definition of CT and what this form of thinking is and what it is not. To note, CT is the process through which a person conceptualises, which means it is not the process through which a person programs (Wing, 2006). In addition to this, CT defines a way of thinking at various levels of abstraction, rather than just the skill for programming. It cannot be emphasized enough that the CT process begins before the first line of code is even written (Wing, 2006). Second, CT is a basic functional skill rather than a mechanical one. Third, the term CT may contain the word “computer”, but this primarily pertains to the way humans think rather than the computer equipment. Computers do not think it is the people who does the thinking for computers, which means to say that it cannot be CT. Fourth, CT is not for a person to think like a computer but instead, thinking with a computer. Moreover, CT complements and blends mathematical and engineering thinking. Just as importantly, the products of CT are ideas and concepts used for approaching and solving problems, which also means that these are not artifacts (Wing, 2006). Overall, CT can be considered as a set of specific cognitive skills and problem-solving processes.

Although CT is relatively new, the process described by Wing (2006) may be considered as a computationally-enhanced version of the well-established scientific method (Lamprou & Repenning, 2018). For instance, in accordance with Wing’s conceptualization, CT nay be regarded as combining mathematical-analytical thinking with natural sciences, engineering, and other disciplines. In other words, CT is conceptualised as thinking instead of a physical object that is the computer. CT is considered and utilised as a way of thinking that harnesses the computer as an instrument for supporting human thought processes, to envision the results of this thought process, as well as, to formulate a problem so that a computer supported solution may be introduced. Based on Wing’s (2006) definition, the CT process can be pictured into three stages.

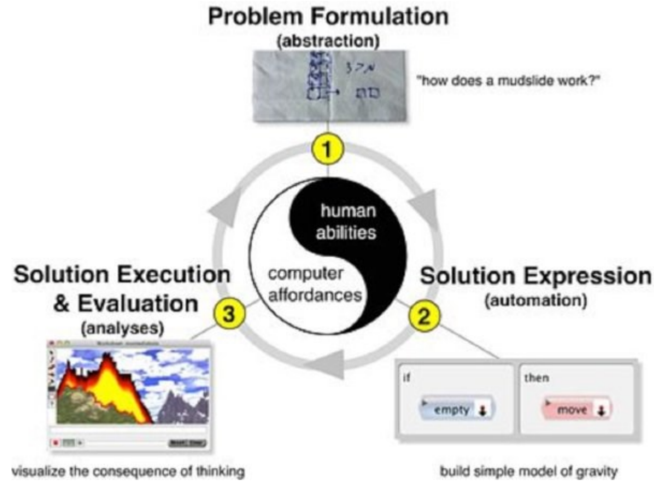


Figure 2.5: Illustration of the CT Process Stages (Lamprou & Repenning, 2018).

As seen in Figure 2.5 above, the first step of CT is formulating a problem which takes place through abstraction (Lamprou & Repenning, 2018). The individual formulates a question usually about how something works while visualising the problem using a diagram on a piece of paper. It is through abstraction that clearly demonstrates how CT does not have to be initiated through the use of a computer. The second stage is solution expression (automation), which is a "non-ambiguous expression of the solution so that the computer can carry it out through computer programming" (Lamprou & Repenning, 2018). The third stage is executing the solution and assessing it through Analysis, whereby the computer shows direct consequences of one's own thinking.

2.6 Summary

The idea of thinking as a computer scientist has drawn much attention of many researchers. The attempt to consolidate this idea resulted in many different definitions, which leads to the perception of a CT being a fuzzy concept. So, the first step of the thesis thesis was to find a way to narrow this concept in a working definition to enable empirical assessment. To do this, systematic literature reviews and major publications with summaries of important experts were analysed to identify the repeatedly mentioned core characteristics and aspects of CT.

Its three core aspects consist of decompose the problem into smaller sub-problems, abstract the problem as a way to simplify it, and designing an algorithmic solution. While the first two facets comprised the analysis of the problem, algorithmic design describes how the problem is actually being solved. Because CT neighbours many

different research fields and because of the vast number of different definitions, all aspects were discussed from different perspectives, i.e., psychology, education, and computer science, to have a better insight of what CT is and where it comes form.

Computational thinking is widely seen as a way to solve different kinds of problems. Complex problems and especially the understanding of algorithmic solutions became more important in the last two decades. That is why it is important to investigate how CT is applied when solving such problems and to find ways to systematically observe it. This research lies the foundation of future training plans on CT.

3 METHODS

3.1 Research questions

This study had the goal to answer the two research questions. The first research question ("How is computational thinking applied when solving a programming task?") as well as the second research question ("Can multimodal measures of computational thinking be relevant predictors for programming quality?") can be both described as exploratory because no specific hypotheses were defined. The focus of RQ1 was what CT looked like and what CT skills are most dominant, whereas RQ2 was about to reveal the relationship between CT and programming quality. To address both questions different methods and measures were used. To answer RQ1, participants were filmed, their voices were recorded, and their screen activities were captured while solving a programming task. To answer RQ2, different measures for CT were used to see how much variance in programming quality could be explained. In addition, control variables such as nonverbal intelligence, sociodemographic information, and prior knowledge were assessed as well.

3.2 Procedure

3.2.1 Phase 1: Online study

The study was divided into two phases. For the first phase, participants logged in on the university provided learning management system. One measure of CT was based on unplugged method. This (unplugged) CT measure, a measurement of nonverbal intelligence, some demographic data, and prior programming experience assessment were obtained from this learning management system. As unplugged CT measure, a set of adapted tasks from the Bebras contest was used. Nonverbal intelligence was assessed with the 3rd edition of the Test of Nonverbal Intelligence (TONI-3). The Bebras tasks and the TONI-3 are described in more detail in section 3.5. In addition, participants were asked to provide demographic information (e.g. age, gender) and whether they had any prior programming experience and whether were familiar with Scratch.

The participants received an email with general information about the study and an invitation link to the Bebras tasks and the TONI-3. Both tests were administered independently to give participants a break between the testing sessions. Participants had up to one week to complete all questions and tasks before the second phase began.

3.2.2 Phase 2: In classroom programming task

The second phase of data collection took place within university classrooms. Students were organised in pairs and were given the following task to work on in Scratch:

> "Program a story or a game where a hero has to overcome a challenge in order to defeat the villain(s)."

This task can be categorised as a typical designing task. Participants are constraint by some limitations and multiple solutions are equally favourable, which makes this kind of task an "archetypical example of ill-structured problems" (Jonassen, 1997). As described before, ill-structured tasks should be used when investigating CT. To analyse when and how much time participants actually spent on CT-relevant behaviour while working on this task, a coding manual was design. The results participants created for this task is also further analysed with regard to their programming quality based on a rubric scheme and Dr Scratch. The CT behaviour coding manual, the programming quality rubric scheme, and Dr Scratch are explained in more detail in section 3.5.

The reasons for working in pairs are described in more detail in section 3.3. The CT coding manual is dependent on observable clues compromising both verbal and nonverbal communication. To provoke such communication clues, a social setting needed to be created. It was anticipated that encouraging students to work in pairs would facilitate observation of CT. The programming pairs were formed based on their Bebras scores to minimise any effects due to large differences in competences. In total, 37 programming pairs were formed.

3.2.2.1 Scratch programming environment

Scratch is a prominent VPE. It was officially recommended to teachers to enhance CT abilities (CSTA, 2011) and has been used in multiple studies (see, e.g., Brennan & Resnick, 2012; Moreno-León, Román-González, Harteveld, & Robles, 2017; Wang & Zhou). As such, Scratch provided an opportunity to both build upon previous studies and contribute to the growing body of research relating to its use. In addition, it was

expected that the participants had no or only little prior programming knowledge and Scratch is easy and rapid for novices to learn. Scratch also provides a considerable degree of flexibility and power to users, enabling them to respond to more ill-structured tasks such as the one used in this study. Accordingly, Scratch was chosen for this study.

In Scratch, the user can chose between different sprites. Sprites can be dinosaurs, animals, or things. For every sprite, there is a different programming window so the users prepare their separately for each sprite. Codes are not written in Scratch; instead code chunks are being connected to each other by drag and drop. The code chunks are similar to Lego® blocks and can be connected only if the connections make sense. This is the biggest difference to normal programming languages. There is no possibility of syntax errors or illogical programming caused by simple lack of knowledge about the programming environment. A bunch of connected code chunks is usually called a script. An example of a Scratch project as created in this study is shown in Figure 3.1.

Figure 3.1. A screenshot of a Scratch project. On the bottom left are shown the three different used sprites. A script of connected code chunks for the dinosaur sprite is shown on the right. The window in the top left shows the effects and actions of the codes.

Code chunks are divided into 10 categories. These chunks have their own colour based on their category, which makes it easier to read the later programmed project (Figure 3.2). Motion code chunks are deep blue and make sprites move through the

space. Looks code chunks (dark purple) change appearance (or costumes) of sprites and activate chat or think bubbles. Sound code chunks (bright purple) enable sound effects. Brown event code chunks initiate the codes to run, and the category control (ochre) contains mostly if-then code chunks and loop commands. Bright blue sensing code chunks react in combination with the environment, for example touching a sprite of specific colour or position. Mathematical and logical operations can be performed with code chunks from operators (bright green), and data code chunks (orange) create a scoring system. There are some pen code chunks (dark green) that draw when used and there is the possibility to use customised code chunks (dark violet) if more blocks are used. However, the last two are infrequently used in general and were not used in this study as well.

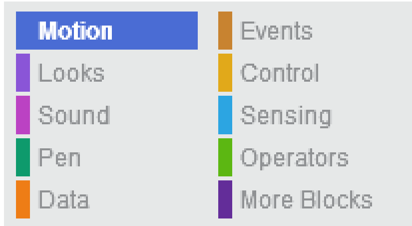


Figure 3.2. The categories of code chunks in Scratch.

There is a qualitative difference in usage of motion, looks, sounds, and event code chunks on one hand, and control, sensing, operator and data code chunks on the other. Code chunks from the first category (which will be referred to as Level 1 chunks) can be easily connected to other chunks and cause outcomes by themselves. Code chunks from the second set (which will be referred to as Level 2 chunks) cannot be connected as simply as Level 1 chunks. Most of Level 2 chunks have to be connected to other chunks first or they cannot cause an outcome directly. Level 2 chunks are most comparable to functions in other programming environments or programming languages that need an extra argument to work, whereas Level 1 chunks do not need any arguments. These restrictions of usage make Level 2 chunks more complex and they are, therefore, cognitively more demanding than are Level 1 chunks. For instance, the sensing code chunk *touching colour* can only be integrated in four different control chunks that additionally need to be connected to a Level 1 chunk to work. The differentiation between Level 1 and Level 2 will be considered in the rubric scheme to assess the programming quality.

Before the actual task was assigned, students were given an opportunity, in a 40-minute warm-up phase, to become familiarise to Scratch. Participants were not recorded during this warm-up and it was not part of later data analyses. The investigator recommended completing four tutorials prepared in Scratch: (1) Getting Started with Scratch, (2) Make it Fly, (3) Create a Story, and (4) Catch Game. The full complements of Level 1 and Level 2 chunks were introduced in these tutorials. Therefore, it was expected that all participants would have similar level of knowledge about Scratch by the time the programming task was commenced, and that they would also understand the principles of Scratch such as how to connect code chunks by means of drag and drop and how to program their avatars or sprites in general.

Participants' activities were captured with Open Broadcasting Software (OBS, version 18.0) while working on the programming task. Participants' verbal interactions were recorded with a headset around their necks. Along with voice recording, built-in webcams filmed most of the participants' upper buddy. This way, facial expressions and other nonverbal clues were observable so interactions were easier to interpret than with voice recordings alone. In addition, participants' operations on screen were captured as well.

Out of the 37 pairs, 1 pair did not give consent to be recorded, 5 pairs turned off (unintentionally) the microphone or shut down the recording software, and in 4 instances the software froze while recording. There were complete and unproblematic recordings from 27 pairs, which were used for later analyses.

3.3 Justification for video study

Direct observation of behaviour is the best choice of methods when actions are the centre of the research question and self-report is not valid enough or not practical. The first is true when measuring performance is the aim the latter is true when an intervention is going (Chorney, McMurtry, Chambers, & Bakeman, 2015). Both are partially true for CT as seen in this study. CT is seen as a thinking product but also a product of actions. In addition, the goal of this study is to observe what kind of CT associated behaviour and skills people use while working on a programming task. Therefore, an observational video study seems to be the right choice.

Others used different approaches. Brennan and Resnick (2012) interviewed children about their Scratch projects. Brennan and Resnick emphasised the positive side of interviews that they provide a deeper insight in the thoughts and (intended) meaning behind these projects. They also, however, pointed out that the interviews were limited by what the children remembered, which was sometimes not correct. For example, some answered the question what they did when they stuck that they never got stuck, which is highly implausible.

CT is a cognitive process and therefore difficult to observe directly. However, in verbal and nonverbal communication people express their thoughts and make them accessible to other people. They talk about what they want to do and what they intend to do. In other words, people *talk* about what they *think*. To certain degree, this is true for nonverbal communication as well. Facial expressions, body language, and gestures can provide to some extent clues about people's intentions and thoughts. Accordingly, in order to study CT it seems appropriate to create a social situation in which people are encouraged to communicate verbally and nonverbally with each other. For this study, that kind of social situation was created by building programming-pairs so students were working collaboratively on a programming task. Moreover, the investigator of the study instructed the participants "to express their thoughts" and "to talk to each other".

To capture all CT-relevant moments during this social situation, participants were filmed while working on the task. As Knoblauch, Tuma, and Schnettler (2013) put it, videography is an especially useful tool to investigate communication and social interactions because no other means of recording is able to collect data in such detail. In video recordings, it is possible to pause and repeat single frames to unveil hidden micro interactions that show CT interaction patterns. For instance, before an algorithm is applied, the problem often needs to be put into chunks. Especially with video data, it is possible to identify these patterns. In this regard, Knoblauch et al. (2013) have called videography "the microscope of the social scientists"—something that provides a broad opportunity to analyse social interactions.

Silverman (2013) has pointed out that video data can be analysed in two ways. On the one hand, a researcher can engage in "mapping the woods" to explore the data on the surface and identify empirical clues for theoretical concepts. On the other hand, a researcher can engage in "chopping the trees" and analyse a video by a fine-grained

sequential analysis based on a theory. To answer the research questions, both ways were needed.

In order to address RQ1, the video data needed initial "mapping" to identify events that could be related to CT and that are not, for example, irrelevant chatting. In a second step, the CT-relevant data needed to be "chopped" with a coding scheme in which they could be classified in more detail so that the role they play for CT became clearer. The more detailed classification of CT events allowed the investigation of the relationship between CT and programming quality that provided an answer to RQ2.

There are different ways to code the video material, which are defined by the level of inference a rater needs to make and the sampling method. If there is only little inference required then the coding scheme is referred as low inferential. Coded behaviour in low inferential schemes is mostly easy to observe, such as hands coming up during a lesson Low inferential coding schemes deal with behavioural clues or interactions between people in a short time period (i.e., a couple of minutes, or even seconds). In contrast to low inferential coding schemes, high inferential coding schemes provide a judgment involving a longer period of time, for example, a whole lesson. Codes are less closely related to an actual behaviour. The theoretical concept is an attribute, or a feature, or a set of different behaviours. The intensity of this attribute is judged on a rating scale. An example would be classroom climate during a school lesson, rated on a Likert-type scale. Codes of high inferential schemes are more complex and therefore they are more open to interpretation. Low- and high-inferential coding are both often used in video studies in education (Pauli & Reusser, 2006). For this study, CT is assumed to be a mixture of both with some elements being rather low and some being rather high inferential.

There are two kinds of sampling methods for coding schemes: time sampling and event sampling. When using time-sampling, the whole session is divided into time intervals, usually 5–20 seconds (Lotz, Gabriel, & Lipowsky, 2013). Each interval is assigned to one distinct code of behaviour. If more than one critical behaviour clues are shown during the interval, the dominant behaviour is coded for the whole interval. Time sampling is used when there is already some prior knowledge about when the behaviour will occur. In contrast to time sampling, event sampling is based on the start and the end of a behaviour sequence. Event sampling makes it possible how often and for how long participants showed a specific behaviour (Bakeman & Quera, 2011, p. 27). Event

sampling is usually used when little is known about the occurrence of the construct. Because of the novelty of the coding scheme, CT will be coded based on event-sampling.

Meanwhile, it must be noted that an added data collection method for this study would have been the interview method of experts, instrument development, or validation study. However, such data collection is time consuming, as Brennan and Resnick (2012) emphasized. Apart from the time-intensive nature of such data collection, interviews would have to be repeated at several points over time to obtain a developmental portrait (Bresnan & Resnick, 2012). The goal of this study was to find out how CT might be applied during a programming task. An interview would have been disrupting the process and hindsight comments might be biased. In light of these points, a video-based study of this design is well justified.

3.4 Participants

Because CT is a highly discussed concept especially in computer science education, the sample for the main study consisted of 108 pre-service students completing a digital creativity and learning course at an Australian University, in March 2017. It must be noted that students' convenor was also the author's supervisor. However, participation in the study was voluntary regardless of the relationship to the investigator and convenor. Students' participation and performance in the study were unrelated to their university assessments and the convenor was not present during any time of data collection. It is important to point out that the sample size and the selectiveness used for sampling may lead to an issue pertaining to non-generalization to the broader population from which the sample was derived. The goal of this study was to observe CT while working on a problem. This requires at least some level of CT and so a selective sample was drawn. A complete random sample would have likely resulted in a sample with very low level of CT. In comparison, a sample of skilled programmers or engineering students may have also confounded the results. Thus, a sample from students enrolled a digital creativity learning course was drawn. Because of these considerations and because of the lack of specific studies like the current one, made it difficult to plan the required sample size in beforehand and a post-hoc power-analysis is provided later. Moreover, the convenor was not present during data collection, thereby eliminating any risks for bias.

There were more female (*n* = 73; 68%) than male students (*n* = 33; 30%) among the participants (*n* = 2; 2% preferred not to say). On average, students were 23.9 (*SD* = 5.2) years old. Because the university in which the study was conducted has a significant number of international students, participants were asked about their English proficiency on a scale from 1 (*poor*) to 4 (*native speaker*). The vast majority of participants (*n* = 105; 97%) indicated that they spoke English fluently or were native speakers. To be sure all participants have the same level of programming knowledge, participants were asked on a scale from 1 (*no prior experience*) to 5 (*professional level*) how familiar they were with programming. Again, nearly all participants (97 %) had either no or only little prior programming experience.

3.5 Instruments and measures

3.5.1 The Bebras tasks

To measure CT based on unplugged methods, participants solved an online version of adapted Bebras tasks. The Bebras contest itself is described by (Dagienė, 2006) in more detail. In total, 20 tasks were used and which were all chosen from the Australian versions of the Bebras contests from 2014 (Schulz & Hobson, 2015) and 2015 (Schulz, Hobson, & Zagami, 2016). Nine tasks were derived from the 2014 version and 11 from the 2015 version (

Table 3.1). The versions from 2014 and 2015 were the most recent ones at the time of creating the CT test for the present study (end of 2016) and both were freely accessible. Only Australian versions were considered because the study was conducted in Australia and it was hoped this would avoid any problems associated with linguistic phrases and idioms.

One requirement was that the overall testing time for the Bebras tasks would not exceed 60 minutes. Based on the results of former studies (Dagienė, 2006; Dagienė & Futschek, 2008) and the pilot study, it was expected a person would need 3 minutes on average to solve one question. Therefore, 60 / 3 = 20 tasks seemed to be an adequate number. Moreover, in previous studies (Dolgopolovas, Jevsikova, Savulionienė & Dagienė, 2015; Lee, Lin, & Lin, 2014), the same number of tasks or even fewer had been used to assess participants' level of CT.

The tasks were chosen from tasks relevant to oldest age group available for the original Bebras contests (i.e., adolescents 16 to 18 years of age; school levels 11 and 12). As mentioned in section 3.4, the average age of participants in this study is 24 years. Although there is a big gap in age, it was not expected that this difference would cause any problems (e.g., ceiling effects) for three reasons. First, no published evidence could be found that 16 and 24 year-olds differ significantly in cognitive skills such as the abilities to decompose a problem, use algorithmic thinking, and use abstract reasoning which all are needed to solve the Bebras tasks. Second, participants of the pilot study who were similar in age to the participants of the main study were asked how difficult they perceived the tasks. Their answers corresponded with the categorisation of the difficulty of the tasks. They tended to mark as easy the tasks that had been labelled easy, to mark as medium the tasks that had been labelled medium, and to mark as hard the tasks that had been labelled hard. Third, using the Bebras tasks to assess the level of CT for participants older than the suggested age group have been used in previous studies, for instance, for vocational students (Lee, Lin, & Lin, 2014) and novice engineering students (Araujo et al., 2017). In conclusion, it was not expected that the participants of the study would solve the Bebras tasks differently from the way in which the originally intended age group had done.

There were three levels of difficulty: easy, medium, and hard. Participants received two, three, or four points when they solved tasks at each of these levels, respectively, and they did not lose any points when they gave incorrect answers. This scoring scheme relies on the recommendations for scoring in the Australian Bebras contest since 2014 (Schulz & Hobson, 2015). The maximum achievable score was 57.

Table 3.1 contains the CT tests for the present study based on a balanced mix of Bebras tasks from both Australian Bebras contest versions and levels of difficulty.

Table 3.1

The Distribution of CT Tasks by Origin and Level of Difficulty

Level of difficulty	Tasks from 2014	Tasks from 2015	Total
Easy (2 points)	4 (8 p.)	4 (8 p.)	8 (16 p.)
Medium hard (3 points)	3 (9 p.)	4 (12 p.)	7 (21 p.)
Hard (4 points)	2 (8 p.)	3 (12 p.)	5 (20 p.)
Total	9 (25 p.)	11 (32 p.)	20 (57 p.)

For the purpose of the present study, some of the Bebras tasks were revised slightly. Some Bebras tasks are presented with iconic beavers or other comic pictures, or instructions refer to beavers as in "beaver did … beaver went …" (see overview for original Bebras tasks in (Schulz, Hobson, & Zagami, 2016). This is not surprising because the Bebras contest was developed for school students from 8 to 18 years of age (school levels 3 to 12). This beaver theme was intended to keep younger contestants motivated during the test. Older contestants expressed problems with these beaver stories and preferred a neutral presentation of tasks (Vaníček, 2014). Therefore, the tasks were presented without any reference to beavers in order to be more appropriate for the sample in this study. An example of an original task and its revised version is shown in Figure 3.3 (see Appendix B for the complete set of tasks used). The structure of the task of the revised version is unchanged in that participants have to apply the same cognitive strategies to find the solution. Only the comic elements and the beaver references were changed or deleted.

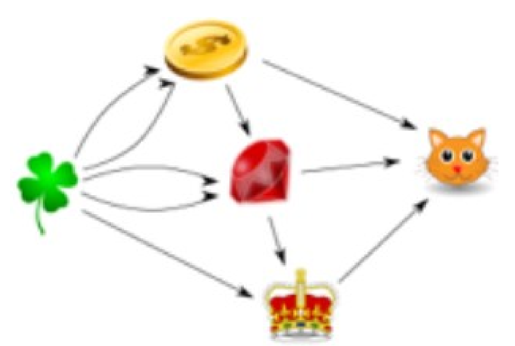
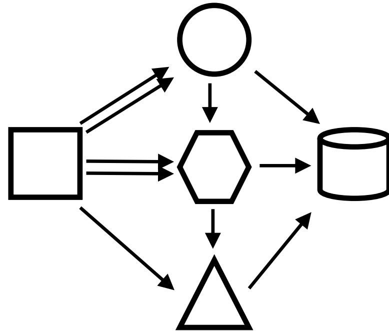
Task: ***Original ("Beaver the Alchemist")***	**Task:** ***Revised***
Beaver the Alchemist can convert objects into new objects. He can convert: - two clovers into a coin - A coin and two clovers into a ruby - A ruby and a clover into a crown - A coin, a ruby, and a crown into a kitten After an object has been converted into another object, it disappears immediately.	In the following you can see how objects convert into other objects. The rule is: - two squares convert into one circle - One circle and two squares convert into one hexagon - One hexagon and one square convert into one triangle - One circle, one hexagon, and one triangle convert into one cylinder After an object has been converted into another object, it disappears immediately.
	
Question: How many clovers do Beaver the Alchemist need to create one kitten? **Answer**: 11	**Question**: How many squares do you need to create one cylinder? **Answer**: 11

Figure 3.3. Example an original Bebras task as used in the Australian Bebras contest 2014 on left and its revised version on the right.

Although CT is considered as multifactorial with emphasis on the cognitive abilities as described in Román-González et al. (2017), this study seeks to determine (a) in RQ1, what CT looks like and what CT skills dominate among the participants; and (b) in RQ2, the relationship between CT and programming quality. As such, to answer RQ1, participants were filmed, their voices were recorded, and their screen activities were captured while they solved a programming task while to answer RQ2, different CT measures were used to determine the extent of variance in programming quality. Consequently, attention has been on the analyse/apply (through Bebras) and create/evaluate (through Dr Scratch) cognitive components of CT (Román-González et al., 2017).

3.5.2 Test of nonverbal intelligence

To measure participants' nonverbal intelligence or general problem-solving skills, the Test of Nonverbal Intelligence (3rd edition; TONI-3) was selected, developed by Brown, Sherbeernou, and Johnson (1997). The TONI-3 is a culture fair test (i.e., minimally linguistically demanding). With a 15-minute average testing time, it is relatively fast to administer. The TONI-3 was developed to assess the cognitive aptitude of children and adults from 6 to 90 years of age, which lies in the range of the participants' age. Moreover, the authors claim that the test is measurement of intelligence with a theoretical and psychometrical foundation. Nevertheless, the TONI-3 may be limited in terms of reliability and correlations with achievement measures. Because of this, it must be pointed out that future studies using this instrument may be undertaken to collect additional validity and reliability evidence and diverse samples.

The focus of the test lies "on abstract reasoning and problem-solving" as cognitive concepts and although the developers pointed out the TONI-3 was not developed based on a specific theory, these concepts play a crucial role in several prominent theories of intelligence. Therefore, the TONI-3 score can be interpreted as one of Thurstone's mental abilities, as one of Gardner's multiple intelligences, or as facet of fluid intelligences or as facet of fluid intelligence described by Cattell, Horn, and Carroll.

In general, the test material consists of two test forms, A and B. Both forms have 45 equivalent abstract pictures as test items and five identical exercise items to ensure the participants understand the procedure and materials. For this study, test form A was used. Every item was divided into two parts, as shown in Figure 3.4. The first half of an item showed an uncompleted set of geometrical figures. In the second half, six similar

figures were provided. The participants had to choose one of the six figures from the second half that completed the set of figures of the first half. In some items (Figure 3.5), the task was slightly changed so that only one figure was presented and the participants had to choose one set out of four sets of figures that completed the row. Nevertheless, the task in all test items was always to identify patterns and to complete a set or a row of abstract figures, which is a typical test procedure for figural/ abstract problem-solving.

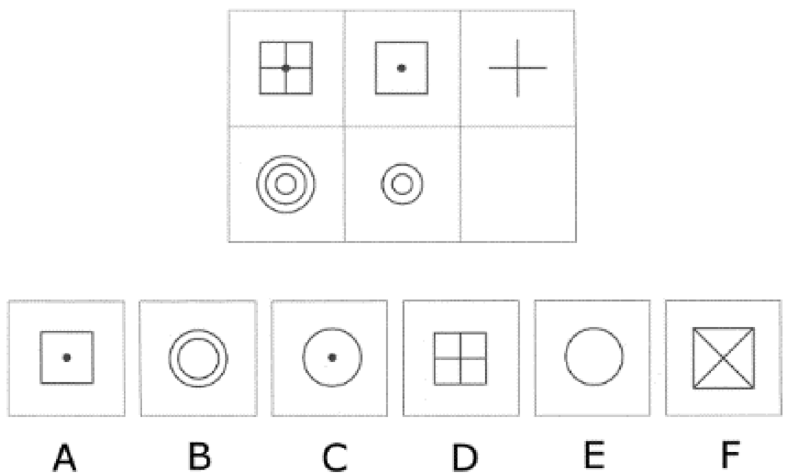


Figure 3.4. Item number 27 from test form A; one figure completes a set of figures.

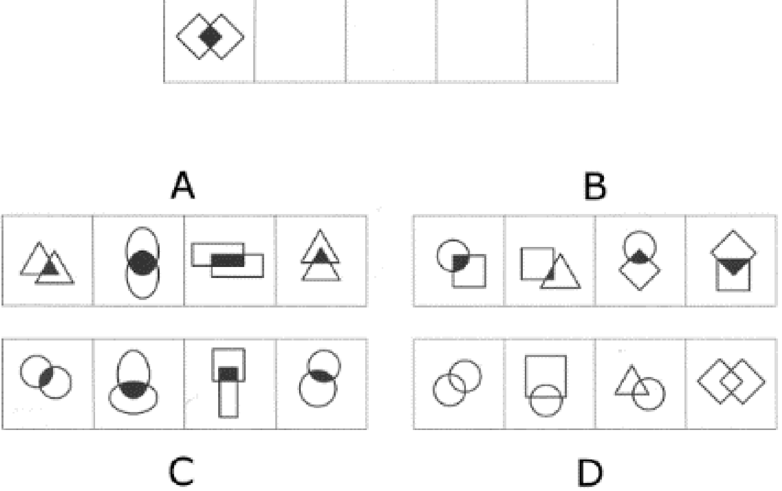


Figure 3.5. Item number 36 from test form A; a set of figures completes a row.

The test was not timed. Participants could take time as much as they needed. The items were ordered by difficulty. The test began with the easiest tasks and progresses in level of difficulty. In the original test, this permitted having a ceiling item in the testing procedure to reduce the testing time for the participants. The ceiling item is defined as the last item of the last five attempted items in which the participant has made three mistakes. The testing continued until the ceiling item had been reached or until last item (item number 45) had been solved. Every correct identified figure scored one point and all points of all correct solved items until the ceiling item or the last item were added to yield a total score. Therefore, the raw scores ranged from 0 to 45.

3.5.2.1 Psychometrics and usage in this study

The raw scores can be transformed via norm samples into IQ scores with a mean of 100 and a standard deviation 15. The reliability assessment shows that the TONI-3 is relatively stable over time with a retest-test reliability of $r_{tt} > .90$ and the internal consistency of Cronbach's $\alpha = .93$ indicates it assesses a latent construct with a little measurement test error (Brown, Sherbeernou, & Johnson, 1997).

The validity of the TONI-3 is also satisfactory. For the content validity, the original test material for the first version was reviewed by psychologists, psychometricians, and educators with expertise in experimental and developmental psychology (Brown, Sherbeernou, & Johnson, 1997). In addition, item response analyses methods were used to identify any biased items. High criterion validity is indicated by middle and high correlations between the TONI-3 and the figural part of other intelligence tests. High correlation between the TONI-3 and school achievements indicated high construct validity Banks & Franzen, 2010; Brown, Sherbeernou, & Johnson, 1997). The validity coefficients are sufficiently high to indicate that the TONI-3 satisfactorily measures abstract problem-solving ability.

To optimise the reliability and validity, the TONI-3 was used according to the manual as much as possible. Some changes still had to be made because the original test comes as a printed hardcopy and the IQ estimations in this study were completed online. First, the material from the book was scanned so it was possible to present it online. Second, during the original test setting, an examiner guides the participant through the

complete process. For instance, the examiner checks whether the participant has understood the test process and materials. There was no examiner in during the test session and participants were on their own while doing the test online. An exercise section was implemented to ensure the participants became familiar with the test situation despite an examiner not being present. In the exercise section, participants received immediately feedback about their initial solving attempts, as shown in Figure 3.6. Nevertheless, this exercise section was separated from the actual testing section, so it was possible for the participants to simply skip the exercise section and it is not possible to determine whether or not this did actually happen. Apart from these changes, the TONI-3 was conducted and analysed based on the instructions in the manual.

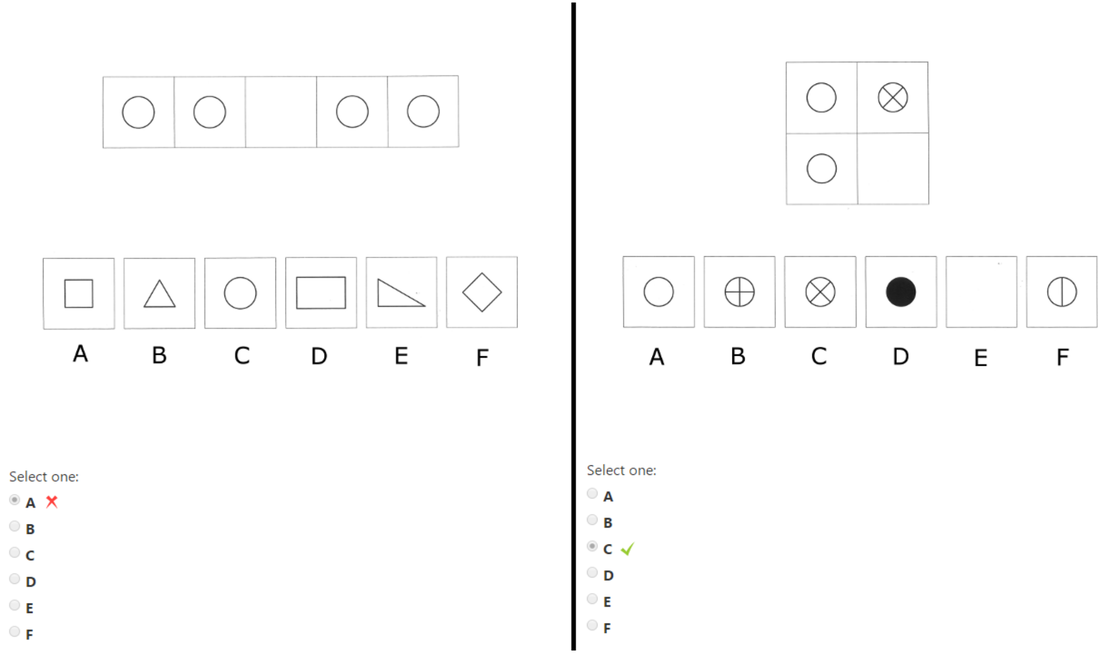


Figure 3.6. On the left, an example of an as correct solved marked item; on the right, an example of an as false solved marked item.

3.5.3 Programming quality rubric scheme

For this study, to measure programming quality, a rubric scheme for participants' solutions of the programming task in Scratch was developed. In general, a rubric scheme is a scoring tool for complex student work (Dawson, 2017). Such scheme usually contains qualitative and quantitative dimensions. The vertical dimension often represents the qualitative criteria of the attribute that is intended to be assessed. This is the content dimension. The quantitative levels are represented on the horizontal

dimension. These indicate the extent to which each category has been achieved. This is the rating dimension. The two dimensions create a matrix where the advantage of rubric schemes lie. Each element of the matrix provides a concise definition for every criterion on each level. These descriptors make the judgment more transparent, clear, and fair. Scoring with a rubric is usually more consistent and more reliable than without one (Jonsson & Svingby, 2007). In addition, the gradations of quality over all criteria allow the strengths and weaknesses to be investigated in more detail as well calculating an overall score for the attribute.

Measuring programming quality with a rubric is not a common approach but had been more popular over the years. Lister and Leaney (2003) suggested a more criterion-references grading for programming to ensure a higher level of clarity. They did not develop a rubric scheme for specific test situations but made a general recommendation how to measure programming quality for novice students.

More recent, Fagerlund, Häkkinen, Vesisenaho, Viiri (2020), developed a scheme with the specific purpose to analyse Scratch projects. Their scheme included two content related areas (vertical dimension). One area specifically analysed programming patterns such as “animation”, “speech and sound”, “collision”, “data manipulation”, and “user interaction”. The other area focused on computational thinking related concepts based on Brennan and Resnick (2012). They analysed over 300 Scratch projects by 57 fourth graders with the purpose to have basis for educational feedback. However, Fagerlund et al. (2020) did not specifically mention the rating itself (horizontal dimension).

In a similar recent study, Basu (2020) also developed a rubric scheme and a description with such horizontal rating dimensions. In the vertical dimension are concepts such as readability and correctness as well as specific kinds coding patterns such as use of loops and conditions. The rating dimensions states from 0 (“lack of use”) to 3 (“exceeding grade level proficiency”). Inter-rater reliability with CS teachers of 90% or higher can be seen as sufficient and comparison between 160 Scratch projects from middle school students and their grades indicated a high criterion validity. In summary, Basu concluded that well developed rubric schemes provide valuable insights on programming skills.

Therefore, a rubric scheme was also developed in this study. The first two vertical criteria, (1) *richness of project* and (2) *variety of code usage,* were developed to take

into account the kind of programming task. The task was ill-structured and open ended and so a variety of solutions are possible to develop. These two dimensions assess how much participants use the possibilities provided by Scratch. The other three vertical criteria, (3) *organisation and tidiness*, (4) *functionality of code* and (5) *coding efficiency* are mainly derived from good coding practice described in more detail in section 2.5.2.1. To ensure content validity, all five criteria were discussed with two computer science education professionals. One was the supervisor of the author of this thesis who specialised in research of usage of technology for educational purposes and a retired computer science teacher with over 30 years of experience. These criteria were horizontally rated in five steps from 0 (*not evident*), 1 (*poor*), 2 (*satisfactory*), 3 (*good*), and 4 (*excellent*). In total, the rubric scheme has 25 descriptors. For more details, see Appendix C.

3.5.3.1 Richness of project

The criterion extent and richness of code was based on what and how much was happening in the final Scratch project. A Scratch project received lower scores when there was only one programmed element that did only one thing than did a Scratch project that included several different elements that did several things and that were related to each other. For instance, the level *poor* meant there was only one sprite that moved in one direction whereas level *excellent* meant there were more than two sprites that could move and change appearance after being triggered and that interact with each other. In general, a higher level Scratch project simply contained more sprites and code chunks.

3.5.3.2 Variety of code usage

The variety of code usage described how many different code chunks from different code chunks were used. Many different Level 2 chunks resulted in higher scores than only a few Level 1 chunks. Participants with a higher Level in this category used nearly all the opportunities that Scratch had to offer. For example, a Level *poor* Scratch project contains only sprites that move and make sound. To achieve Level *excellent*, a majority of chunks from Levels 1 and 2 needed to be used.

3.5.3.3 Organisation and tidiness

The organisation of the whole workspace played a role in the assessment of programming quality as well, in terms of how messy or clean the workspace appeared to

be. An often-mentioned feature of a *good* code is its readability (Martin, 2009). A code is easy to read when the coding environment is free of "dead scripts". Dead scripts are pieces of code chunks that play no role for the whole program. Examples of dead scripts are comment-out, deactivated, or incomplete codes. Dead scripts unnecessarily fill the program console. This decreases the readability of the whole program because the user has to actively ignore these codes in order to understand the whole program. It is neither possible to comment out codes in Scratch, nor to deactivate codes. However, it is possible to have incomplete code chunks. A code chunk has an effect only when it can be initiated by an event chunk as shown in Figure 3.7. So, if a code chunk or if a sequence of code chunks were not connected to an event chunk they did not play a role for the whole program and were defined as dead scripts. In addition, the correct order enhances readability. Readability referred to whether the codes appeared where the reader would expect them and whether they were eventually align with the screen. The fewer code chunks a Scratch project had and the more it looked organised, the higher the readability and the higher the score. A Scratch project rated as *poor* had many dead scripts and looked messy. In contrast, a Scratch project rated as *excellent* had no dead scripts; they are organised and eventually aligned with the screen. In summary, the appearance of the whole workspace looked tidy and was easy to read for a higher level.

Figure 3.7. An example of a valid list of connected codes with an event code at the beginning is shown on the left. On the right is shown an example of a dead script.

3.5.3.4 Functionality of code

Functionality is crucial for a code. In this study, functionality was assessed on the basis of two questions. First, was the intention of the Scratch program clear? Second, did the program work as intended? Some coding attempts were so basic that it was not even clear what the participants wanted to do. To achieve at least a *poor* level, the intention of the code needed be clear even if it did not necessary work. For example, several connected code chunks from the moving category very likely indicate that the

participants tried to program a sprite to move. It might not have worked because of other problems (e.g., no event chunk is connected to initiate the movement) but at least the intention was clear. If the intention was clear and if it worked, the next question was how well it worked and how smoothly the code ran. Most of the Scratch project might have worked without any error but a text box appeared and disappeared too quickly to read. Another example could be a reaction game where the user had to control a sprite and reacted to its environment, but it moved too slowly or the scoring system did not work properly (e.g. points were not counted correctly). These were examples of working Scratch projects with clear intentions but that did not run smoothly. For an *excellent* Scratch project, the reading time needed be reasonable, speed of moving elements needed be adequate, and the score systems needed to make sense and worked correctly.

3.5.3.5 Coding efficiency

The category efficiency described the usage of controlling code chunks and the number unnecessary duplications of codes. Duplication was unnecessary if an opportunity was missed for abstraction to a higher level. Duplications show a violation of the “Don’t repeat yourself” or “Once and only once” principles. So, for example, if a function needed to be repeated, instead of copy paste the same code several times, it would have been better to loop this code with the correct control function. The correct usage of these kinds of control functions required a higher level of abstraction in thinking, but the code would become more efficient and elegant. Figure 3.8 contains an example of two codes from two different Scratch projects. Both codes did the same in that they controlled movements in four directions of a sprite while using arrow keys. However, in the code script on the right, a forever code chunk was used to control all directions of movement whereas the code script on the left there were just copies of controlling codes for one direction four times. The script on the left shows duplications that could have been avoided. A poor Scratch project showed many duplicates, and few control chunks were used. In contrast, an *excellent* Scratch project did not have any duplicated chunk, and participants demonstrated a comprehensive and complete use of control chunks.

Figure 3.8. Two examples of the same function but coded differently. An example with unnecessary duplicates is shown on the left and a more efficient version is seen on the right.

3.5.3.6 Weighted score of sum and reliability assessment

Whereas organisation was fully independent from the other categories, the extent and richness, variety, functionality, and efficiency were partially dependent. For instance, a low score in extent and richness means participants did not use many code chunks. If there were few code chunks, it was impossible to have many different code chunks contributing to a higher score in variety. It was also very likely that the functionality of the Scratch project was limited and that it was coded efficiently. However, that was not true for higher scores. High scores in richness as well as in variety do not necessarily mean the Scratch project is well coded and runs smoothly.

To have an overall judgment about the Scratch project, a weighted mean over all categories was calculated. The weighting represented the importance of each category for programming quality and is based on Martin (2009). Assessments in extent and richness, variety, and functionality equally contributed 20% to the final score. In efficiency, participants showed a higher level of thinking. In particular, it might be challenging for participants to see opportunities for using control code chunks that allowed increasing layers of abstraction. Efficiency represented programming quality at slightly higher level than the other categories. On the other hand, to keep the workplace tidy and aligned was not unimportant but seemed to be less essential to be a good programmer. It represented a programming quality slightly less than the others. Therefore, assessments in efficiency were weighted 30%, and organisation 10%, when

determining the final score. In total, the theoretical range of weighted means ranges from 0 to 5.

To assess consistency of the program rubric scheme, the interrater reliability assessment between the investigator of the study and a former teacher of computer science with over 30 years of work experience was computed. The investigator of this study trained the former teacher for one hour with the programming rubric. All categories, levels, and descriptors were discussed in detail. In addition, typical examples of overall *poor*, *satisfactory*, *good*, and *excellent* Scratch project were reviewed.

In total, 42 Scratch projects were assessed with the rubric scheme. To assess the interrater reliability, an intraclass correlation coefficient (ICC) for the weighted overall mean was calculated (Shrout & Fleiss, 1979). The ICC is used when the variable is metric, which is assumed for the weighted mean. The raters were set as fixed and only participants were set as a source of randomness that is referred as *model 3* (Shrout & Fleiss, 1979). Because there was only one occasion of measurement, the form for the ICC was 1. As type for the ICC, *absolute agreement* was chosen because the assessment of the agreement between both raters was important and not how consistent the raters were (see, for more details regarding ICC for reliability assessment, Kim, 2013; Trevethan, 2017). Confidence interval for ICC(3,1) with type *absolute agreement* indicates that the rubric scheme is a sufficient reliable instrument, 95% CI [.87, .96]. The investigator rated the Scratch projects slightly more liberal (M = 2.13; SD = 0.97) than the former computer science teacher (M = 2.06; SD = 0.93) did. However, this difference was not significant, $t(41) = |1.16|$, $p = .253$.

3.5.4 Dr Scratch

To have an additional score for Scratch projects, Dr Scratch was used. Different to the developers' intentions, Dr Scratch is not considered as an assessment of CT but rather an alternative measure for programming quality of Scratch in this study. The seven dimensions of CS relevant concepts (i.e., abstraction and problem decomposition, parallelism, logical thinking, synchronization, algorithmic notions of flow control, user interactivity and data representation) are judged as *Not evident* (worth 0 points), *Basic* (1 point), *Developing* (2 points), and *Proficient* (3 points) (Moreno-León, Robles, & Román-González, 2015). In order to obtain a general evaluation, all points will be summed to obtain an overall score, referred to as a mastery score. Mastery scores

between 8 and 14 are regarded as *general developing*; lower than 8 is regarded as *generally basic*, and more than 14 as *general proficient*. Scores are based on the number of dead scripts, correctness of messages synchronisation, object properties that are (not) correctly initialised, and (unnecessary) repeated code chunks (Moreno-León & Robles, 2015).

3.5.5 Computational thinking behaviour scheme

In order to assess when and how much time participants spent on CT-relevant behaviour during the Scratch session, the computational thinking behaviour scheme (CTBS) was built. Besides of the literature about CT, the results of the pilot study were considered as well. The development of the coding manual is based on the general recommendations for developing and modifying a behavioural coding schemes by Chorney, McMurtry, Chambers, and Bakeman (2015). The CTBS is a low inferential coding scheme with high inferential elements, and the coding method is event-sampling.

In this study the goal was to explore the way CT occurs and what attributes of CT are shown more often and what kind of patterns can be detected. These attributes are shown in specific behavioural clues. Some of these clues can be identified with rather low inference (e.g., algorithmic design) while other clues are more complex and require a higher inference (e.g., problem decomposition and thinking abstractly). Thus, the scheme can be regarded as a mixture of low and high inferential clues. Not all behaviour exhibited by the participants is coded, only actions and utterances that indicate CT. The CTBS is based on event sampling, which means it analysed how often and for how long specific behavioural clues occur and how long they last.

The coded events in the CTBS can be understood as latent constructs, so it is not possible to observe them directly. However, it is possible to see the effect of someone's thinking in their behaviour. The behaviour can be understood as the manifestation of these latent constructs. Therefore, statements and conclusions about latent variables can be made based on their manifest counterparts. Based on the behaviour we see we make conclusion about the intentions and thoughts. This is an example of inferential reasoning and does not come without any problems. Inferential reasoning always implies a level of uncertainty and false predictions. As we are not always right about our conclusions of someone's thoughts based on just observing them, it is not possible to explain perfectly relationship between a latent variable and a manifest variable. This

is why identifying the correct manifest variables is crucial to make inferential statements about a latent variable with the highest validity possible. The manifestations of the latent components of CT as used in this study are described in the following.

3.5.5.1 Computational thinking components

Based on the literature review, four components were identified as main features of CT and which are the latent constructs in the CTBS: decomposition, abstraction as in ignoring unimportant details, abstraction as in recognising patterns, and designing and applying algorithms. Decomposition and both types of abstraction deal with the problem itself, whereas designing and applying algorithms are concerned with the solution to the problem. The CTBS includes behavioural clues as manifestations for these four components. Every kind of behaviour that indicates one of these four components is coded as an event. An event can be (1) an action of a single person (2) what a single person says or (3) be part of a discussion between people. Events are as mutually exclusive (i.e., every observed CT related behaviour can only be assigned to one code). In the following paragraphs, the behavioural clues of each latent construct are described in detail and the whole scheme is shown in Table 3.2.

3.5.5.2 Decomposing

In CT, a crucial part of handling a problem is dividing the problem into smaller chunks as a technique for reducing complexity. This is called the decomposition of a problem. The idea is that a couple of less complex problems are easier to solve than a single, more complex problem. Thus, any actions or spoken words that refer to putting the task into subtasks are coded as a clue for decomposition. This also includes being aware of the fact that there are several steps to make until a solution has been achieved, for instance, the steps suggested by Jonassen’s model for problem solving. One possible manifestation of decomposition is the discussion of the immediate next step to perform in order to accomplish the task. When participants ask what the next step(s) would be or when they explicitly state what to do next, they make clear that they divided the main task into subtasks and in which order they would like to complete them. For instance, an event is coded if a participant makes a clear order of tasks such as (1) which sprite should be the hero, (2) which sprite should be the villain, (3) what is the story about? This way, participants divide the main problem into sub-problems and sub tasks. In addition, a discussion of how story elements can be implemented as a code can be a clue for decomposition as well, for instance, when participants start talking about how they

want their sprites having specific features such as moving in one direction or shooting something.

3.5.5.3 Abstraction I – neglecting details

Another component of CT is abstraction with its two subprocesses. Clues indicating ignoring unimportant details but focus on relevant information can be seen when someone literally says to ignore something and to focus on something instead. Furthermore, rephrasing or simplifying the main task, sub-problems, functions, or the meaning of code chunks, can indicate deliberately neglecting of unimportant details as well. However, just reading the main tasks again or simply repeating what anyone said does not constitute ignoring unimportant details. Participants must show an understanding of what they simplified. An event is coded only when their rephrasing is clearly an attempt of simplification.

3.5.5.4 Abstraction II - recognising patterns

Recognising patterns, or the ability to identify similar characteristics across several items, can be seen as the other component of abstraction. An event of pattern recognition is coded when participants explicitly say something or show in another way that they understand or see a pattern. Saying or doing something that directly refers to patterns can be understood as clues for pattern recognition. In addition, clues when participants showed they were able to transfer what they learnt during the tutorial can be considered as pattern recognition as well. For instance, participants learnt how to use codes in a specific way in the tutorial. When they realised they can use the codes in a different way as well during the actual test session, it is coded as a pattern recognition event as well. This kind of realisation of patterns can be very sudden are called aha moments in this study. These transferred learning situations can be also seen when participants copy and paste code sequences and alter them subsequently. When participants use the same piece of code in different situations that must mean they identified similarities in these situations; therefore, this behaviour is coded as an event for pattern recognition as well.

3.5.5.5 Designing and applying algorithms

The components described immediately above focus on manipulation of the problem. The category designing and applying an algorithm focuses on the solution. Algorithm describes a sequence of operations. In a usual programming environment, an algorithm

comes in the form of written code. In Scratch, algorithms are not written in codes; rather, predefined code chunks are combined. Therefore, an event is coded as designing an algorithm when code chunks are connected. Applying of the algorithm is shown by executing it. Codes in Scratch can either be executed by double clicking the sequence of codes or by clicking on the green flag. Both actions will be coded as events for applying an algorithm. Only in rare cases does the freshly coded algorithm work as intended. More likely, participants need to adjust the code. This adjustment is defined as debugging in the CTBS.

Table 3.2

Computational Thinking Behavioural Scheme

CT components (latent variables)	Behavioural clue (manifest variables)
Decomposing	Talking about the immediate next step
	Put problem into pieces / building sub tasks or problems
	Discussing if then relations of the story or game (is related to programming elements)
Abstraction I – neglecting information	Focusing on important information; neglecting unimportant details
	Simplifying anything (problem, sub problem, functions, code bocks, etc.)
Abstraction II – pattern recognition	Identifying similar characteristics (sub problems, functions, code blocks, etc.)
	Use of copy-paste
	Aha moments (must be related to an event when student understood relationship between things)
Designing and applying an Algorithm	Putting code chunks together
	Testing and judging algorithm (i.e., clicking on run or double click on sequence or actively observing a running sequence)
	Debugging - try to find error and adjust algorithm

3.5.5.6 Reliability assessment

The consistency of the CTBS was estimated by the interrater reliability between the investigator of the study and a second person. This person was a PhD student at the same department but was not involved in the study. This person was trained by the investigator during a practice session. The CTBS was discussed together and a video from the pilot study was used for practising. After that, five of the 27 videos were selected and the interrater reliability was assessed. The videos were chosen based on a mixture of high and low Bebras and Scratch scores in order to have a representative subsample of videos. This procedure is also based on the guide on developing and modifying behavioural coding scheme from Chorney et al. (2015).

To estimate interrater reliability in this instance, κ coefficients for each of selected videos were computed (Cohen, 1960). Cohen's κ is widely used to determine the degree of stability and agreement of two or more judges for nominal variables. It is similar to the frequency of agreement but adjusted for agreements, some of which can be expected to occur by chance alone. Cohen's κ distinguishes only between agreement and disagreement and was not designed for an event-sampling design in video studies. This creates a problem. To have an agreement for two coded events, not only the content of the events must be the same (e.g. both identified an event as "debugging" or "discussing if then relation") but also the onset and offset. Because it is virtually impossible two people start and stop an event at exactly the same time, an interval of tolerance can be defined between "still an agreement" and "already disagreement". For this study, the tolerance for an agreement was set when both events overlapped at least 50% of the time. If two events did not overlap, it still counts as an agreement as long as their onsets differed by 5 seconds or less. Because no conventions or recommendations could be found in the literature known to the author, these settings are arbitrary. However, the author attempted to find a compromise between being overly rigorous and overly lenient in order to obtain valid results that could be regarded as valid.

As a result, the range of the frequency of agreement lies between 66.67% and 72.50%. At least two third of the events were identified from both raters. In addition, the range of κ coefficients, from .58 to .67, indicates moderate reliability (Landis & Koch, 1977). For the most part, disagreement occurred without any systematic bias or recognisable patterns. However, sometimes the codes within a category had been mistaken. For example, events that indicated debugging were coded as an event when

participants had simply created the code, and vice versa. Because both codes are manifestations of the same latent construct (designing and applying an algorithm) this disagreement was not considered to be serious.

3.6 Pilot study

The pilot study was held two months before the main study, at the end of January 2017. Ten University students were randomly asked to participate. Most of the participants did not know each other but some were acquaintances and have met in the past. The average age of the participants was 26.10 years (*SD* = 4.93). Eight participants were male and eight had no prior programming experience. All participants spoke English fluently. Therefore it can be assumed that the sample for the pilot study was comparable to the sample of the main study. No one of the pilot sample was part of the main sample.

The purpose of the pilot study was to test the procedure, some of the instruments, and the programming task. As for the main study, the pilot study was divided into two phases. In the first phase, participants solved the Bebras tasks online at home beforehand. In the first version, the Bebras tasks for the pilot study contained 13 items from the Australian version of the Bebras contest from 2014 and 13 tasks from 2015, (i.e., 26 tasks in total). Seven tasks were categorised as easy, 10 as medium, and nine as hard. The TONI-3 was not part of the pilot study because it is already a well-established instrument and there was therefore no need to test its usability. In the second phase, participants solved the programming task in classrooms at Macquarie University few days after they finished the online CT test. The programming tasks had to be completed in Scratch. So that all participants had the same level of knowledge about Scratch, the participants completed a 20 minutes tutorial before the actual task began. The instruction for the programming task was “Program a game or a story where a hero has to overcome a challenge in order to defeat the villain(s)”.

For the Bebras tasks, participants had the opportunity to flag unclear tasks during the testing process. In addition, participants were asked to rate the level of difficulty of each task as *easy*, *medium-hard*, or *hard*. The average time a participant needed to complete the test was 81.25 minutes (*SD* = 16.87). With a total number of 26 tasks this means that on average participants needed slightly more than 3 minutes per item. On average, participants achieved a total score of 46.10% (*SD* = 11.32) from the maximum possible

score of 80 points. Some of the questions categorised as *hard* were challenging for most participants. Results for the Scratch programming tasks were less complex than expected. Many participants recreated the code they had seen in the tutorial before. Other codes and opportunities, which were not shown in the tutorial, were ignored and not used. Thus, the results of the programming tasks were less rich in terms of complexity than had been expected. Many actions indicating algorithmic design were observed and just little less often actions indicating decomposition was seen. Participants discussed quite intensively the task and how to approach it. However, any kind of utterance or actions with regards to abstraction was barely observed. The investigator concluded that the time for tutorials was not sufficient to give the participants an adequate overview of the possibilities in Scratch.

To reduce the overall time of the test to 60 minutes and to avoid ceiling effects in the main study, some tasks needed to be removed. Selection of items to remove was based on two criteria. The first was usability of the tasks. Some tasks were unclear in their presentation or instruction and were flagged by some participants. The second criterion was solvability of the tasks. Some tasks were solved by only a few or none of the participants. Six of these items were deleted. The remaining 20 tasks were considered to be unambiguous and potentially solvable, and were therefore used as the final version of the CT test for the main study. In addition, the Scratch-tutorial time was doubled from 20 to 40 minutes. The tutorial was created from introduction videos provided by Scratch. The programming tasks was slightly adjusted to “Program a story or a game where a hero has to overcome a challenge in order to defeat the villain(s)” because the new tutorials explained not only how to program a game but also a story.

In addition, the CTBS was slightly changed as well. Some codes of some categories were renamed, for example, “discussing if then relations” was named “recognition of several steps” before. Moreover, the behavioural clue “talking about the immediate next step” was added to the category decomposition. Although any behaviour indicating abstraction was barely observed, the categories were kept for the main study because of theoretical importance of both facets.

3.7 Data analysis approach

3.7.1 Units of analysis

In phase 1, participant solved the Bebras tasks and completed TONI-3 online. That means that for each participant an individual value were obtained. In phase 2, participants were paired based their Bebras scores to solve the programming task in Scratch. That means that the remaining measures (i.e., the programming quality, the time they spent on CT-relevant behaviour, and the Dr Scratch score) were paired as well. That means that two participants always had identical values on these variables. These variables can be referred to as between-pair variables (Gonzalez & Griffin, 2012; Kenny, Kashy, & Cook, 2006).

To have individual and paired data has implications for how the data are handled and analysed together. The Bebras score and the TONI-3-IQ can be analysed in two ways. First, variables can be analysed individually (i.e., the unit of analysis is each participant). Second, variables can be analysed pairwise (i.e., the unit of analysis is combined). In the latter case, the mean for the Bebras score and for the TONI-3-IQ is calculated for each pair. This variable is referred to as a within-pair variable and enables analysis of relationships with the other already paired variables. To assess how similar the values of the later combined variables indeed are, the level of “nonindependence” (Kenny, Kashy, & Cook, 2006, p. 26) is assessed by the Intraclass-Correlation-Coefficient (ICC; not to be confused with ICC to estimate reliability). According to Kenny, Kashy, and Cook (2006), a high ICC indicates statistical nonindependence, meaning the scores within each pair are more similar than between the pairs. That would justify using the mean of the pairs for further analysis. A low ICC would indicate that the scores within each pair are not more similar than between the pairs. Under those circumstances, further analyses based on the mean over the pairs must be interpreted with caution. An explanation of ICC for nonindependence is provided in Appendix D.

3.7.2 Addressing research question 1

The first research question (RQ1) was: “How is CT applied when solving a programming task?” To address this question, participants were recorded while solving a task in Scratch. Recordings were then coded using the CTBS. Recordings were analysed based on the CTBS in INTERACT software (Mangold, 2018). This way, it

was analysed how often a specific CT-relevant behaviour was exhibited and for how long.

In addition, coded events were analysed for any kind of specific patterns and whether it is possible to predict with a certain probability the occurrence of one CT-relevant behaviour given another CT-relevant behaviour. This kind of analysis is done by lag sequential analysis (LSA; Bakeman & Quera, 2012). LSAs are based on Markov chains and provide transition probabilities form one event to another in order to identify any typical sequences which may be more likely than others. To test whether transition probabilities are significant different from zero, *Z*-scores are calculated based on the difference between the empirical frequency and the expected probability. An explanation concerning how Z-scores are calculated is shown in Appendix E. *Z*-scores greater than 1.96 are regarded a statistically significant because the corresponding probability is less than .05 and consequently, indicate a pattern of actions of interest (Ivanouw, 2007). An example question for LSA would be, "Given a programming pair shows 'debugging behaviour', does this increase the probability of the pair showing 'testing behaviour' next?".

3.7.3 Addressing research question 2

The second research question (RQ2) was: "Can multimodal measurements of CT be relevant predictors for programming quality?" To address this question, the relationship between the two different measures for CT (Bebras score, and the time spent on CT-relevant behaviour based on the computational thinking behaviour scheme) on one side and the programming quality (based on the programming rubric scheme) on the other side was analysed. Two kinds of analyses for the relationship were conducted. First, correlation coefficients for each CT measure and programming quality score were obtained. This revealed any linear relationship between each of the CT measures and programming quality alone. The correlation between programming quality and the measure of nonverbal intelligence (TONI-3-IQ) was obtained also to see whether programming quality shared variance with a measure for IQ as well. Second, it was analysed whether programming quality can be predicted by the different CT measures and TONI-3-IQ. This was done by conducting a multiple linear regression analysis. The regression analysis showed the (linear) relationship between programming quality and each predictor when the effect of all other predictors was held constant. Because Dr Scratch was considered as an evaluation for Scratch projects, the same analyses were

conducted with Dr Scratch mastery score as well. If any, positive correlations were expected and so all tests of significance for all analyses were conducted one-sided.

3.7.4 Statistical analyses

For most quantitative analyses, the free statistical programming language R (R Core Team, 2017) was used. Power analysis for regression were performed with G*Power (Faul, Erdfelder, Lang, & Buchner, 2007). Because of the small sample size, some analyses are based on both, parametric as well as non-parametric tests. Some of the measures were used for further analyses in regression models. Regression models rely on the assumption of normality; therefore, these measures were tested with the Shapiro-Francia test (Royston, 1993; Shapiro & Francia, 1972) to determine whether their distributions differ significantly from normal. According to Yap and Sim (2011), the Shapiro-Francia test is the most powerful test among the most common tests for normality. Effect sizes are interpreted based on Cohen's (1988) criteria. The threshold on significance for all tests used in this study was set at .05 based on the common convention.

3.8 Research ethics approval

Before the study was conducted and any data were collected, the research ethics committee from Macquarie University had to approve the procedure. To receive the approval, a number of principles had to be met: (1) describing of all potential participants and giving reasons for choosing them; (2) fair recruitment of participants without any pressure to participate; (3) minimising the risk of any harm to participants; (4) protecting participants' privacy and confidential information; (5) obtaining participants' consent; (6) fully debriefing participants and giving them appropriate information.

The ethics committee were satisfied that all principles would be met and granted approval on 16 December 2016 (reference number 5201600918). Recruitment of participants for the pilot study commenced the same month. A research report concerning progress needed to be sent to the committee each year until completion of the research. A final progress report is due 2021.

4 RESULTS

An overview of the descriptive results of Bebras tasks, TONI-3-IQ, programming quality, and Dr Scratch is first given before the research questions are being answered. The first research questions is then answered by an overview of the time participants spent on CT-relevant behaviour during programming task in Scratch, based on the CTBS and the LSA to reveal any patterns in the participants behaviour. The second research question is answered by correlations between programming quality and the different measures of CT. Also the results of the regression analysis with programming quality as outcome and the different CT measures and TONI-3-IQ as predictors are presented. The result chapter closes with correlation patterns between the different CT measures and TONI-IQ-3 as further analysis.

4.1 Overview of measures

The Bebras tasks as well as the TONI-3-IQ are obtained individually whereas the remaining measures are based on pairs. For the Bebras tasks and TONI-3-IQ that means that descriptive results of both measures are first presented as individual and then as paired scores. Results of remaining measures are based on pairs from the beginning.

4.1.1 Bebras tasks

4.1.1.1 Individual scores as the unit of analysis

The maximum achievable score for the Bebras tasks (57) was set as 100 %. In total, 110 students completed all tasks. One participant achieved 100 %; the lowest observed score was 21 %. The close range between mean, trimmed mean (10 %), and median indicated a normal distribution (Table 4.1). This was supported as well by visual inspection (see Appendix F) and the result of the Shapiro-Francia test with $W' = 0.99$, $p = .559$.

The average Bebras score did not raise any concerns that the test was too easy or too difficult. No ceiling or bottom effects could be found for any tasks (i.e., there was no items solved by everyone and there was no items solved by no one). There were three levels of difficulty for the Bebras tasks: easy, medium, and hard. As easy labelled tasks were expected to be solved more often than medium labelled tasks and medium labelled tasks were expected to be solved more often than hard tasks. Overall, this pattern could

be found in the results. However, as hard labelled tasks were slightly more often solved than expected. For a more detailed item analysis, see Appendix G.

On average, participants were recorded as taking 198.74 min (*SD* = 635.01) to finish the Bebras tasks. The trimmed mean (10%) was 59.52 min and the median was 55.00 min. The range to complete the whole test lay between 5 and 5607 min. The Shapiro-Francia test revealed a significance deviation from the normal distribution, $W' = 0.26$, $p < .001$, and skewness of $v = 6.5$ indicated that there were more extreme values on the right side of distribution than on the left (i.e., some participants were recorded as taking longer to finish the test). Because of skewed data, Spearman's ρ were used to analyse the relation between achieved Bebras score and needed time. Analysis revealed a positive medium large and significant correlation, $\rho = .40$, $p < .001$, meaning the more time participants took to work on the Bebras test the higher their scores.

Although the trimmed mean and the median were close to the expected maximum time of 60 min (see section 3.5.1), the other statistics indicate problems with extreme values. The fact that completion of the tasks was not supervised may be an explanation for these results. The very slow completion could be explained by participants taking breaks between. There are no assumptions breaks could have influenced the outcome of the test, so no participants were excluded from further analysis because they needed too long.

4.1.1.2 Paired scores as the unit of analysis

Based on their individual Bebras score, participants were organised in pairs to solve the programming task in Scratch together. To assess how close participants for each pairs generally were, the ICC over all 37 pairs was calculated. The significant ICC of .75, $F(36,37) = 7.05$, $p < .001$, indicated that the scores of both participants in each pair were indeed quite close to each other and can be interpreted as statistically "nonindependent" (Kenny, Kashy, & Cook, 2006, p. 26). This supported the approach to use the mean per pair for further analysis. The mean, the trimmed mean (10%), and the median of all pairs were only slightly higher than the values calculated individually (Table 4.1). Although the Shapiro-Francia test indicated a normal distribution, $W' = 0.97$, $p = .352$, visual inspection leaded to the conclusion that this might not be the case (see Appendix F) and further analyses must be interpreted with caution.

Table 4.1

Overview of Bebras Scores

Individual Bebras scores				Paired Bebras scores			
M (*SD*)	*Trimmed M* (10 %)	*Mdn*	*N*	*M* (*SD*)	*Trimmed M* (10 %)	*Mdn*	*N*
57.03 (18.60)	56.98	57.89	110	58.93 (17.17)	58.71	61.84	37

4.1.2 Test of Nonverbal Intelligence

4.1.2.1 Individual scores as unit of analysis

The TONI-3 was completed by 71 participants with a range for IQ from 76 to 140. As for the Bebras scores, TONI-3-IQ were normally distributed with nearly no difference between mean trimmed mean (10%), and the median (Table 4.2). In addition, the Shapiro-Francia test indicated a normal distribution, $W' = 0.97$, $p = .093$, as well as the visual inspection did (see Appendix F).

The general findings for the TONI-3-IQ were similar to the results of the Bebras tasks. Participants needed generally longer than expected. The expected maximum time to complete the TONI-3 is 15 minutes, but participants needed more than 10 min longer, $M = 26.06$ min ($SD = 16.75$). The non-normal right shifted distribution ($W' = 0.71$, $p < .001$, $v = 2.59$) and the range from 5 to 103 min indicated that some participants may have taken breaks between working on different items. This may explain the surprisingly high average completion time. A significant positive and medium large correlation between TONI-3-IQ and needed time was found, $\rho = .42$, $p < .001$. Similar to the Bebras test that means the more time participants spent on the test the higher their TONI-3-IQ. As for the Bebras tasks, no participants were excluded from further because they needed longer than expected. Even though 5 min seemed to be quite fast, it is still plausible to finish the TONI-3 with a reasonable result in that time (Brown, Sherbeernou, & Johnson, 1997). Thus, no participants were excluded based on the TONI-3 test time.

4.1.2.2 Paired scores as unit of analysis

There are complete data of 33 pairs for the TONI-3. Although smaller than for the Bebras score, the significant ICC of .49, $F(32,33) = 3.00$, $p = .001$, indicated that the IQ

scores within pairs were quite close to each other and justified to use the mean over pairs for further analysis. The mean, trimmed mean (10%), and for the IQ over all pairs were all close to the results when the unit of analysis were individuals (Table 4.2). The visual inspection (see Appendix F) and Shapiro-Francia test, $W' = 0.97$, $p = .425$, revealed no significant difference from normal.

Table 4.2

Overview of TONI-3-IQ

Individual TONI-3-IQ				Paired TONI-3-IQ			
M (*SD*)	*Trimmed* *M* (10 %)	*Mdn*	*N*	*M* (*SD*)	*Trimmed* *M* (10 %)	*Mdn*	*N*
112.49 (14.17)	113.12	113	71	114.74 (12.98)	115.41	115	33

4.1.3 Programming quality

Because the programming assessment and Dr Scratch measurement were both based on the usage of different code chunks and sprites, a short overview of those Scratch metrics of al 37 pairs is given first. On average, pairs had $M = 4.14$ ($SD = 2.32$) sprites, used $M = 44.73$ ($SD = 31.42$) code chunks, and created $M = 6.65$ ($SD = 5.18$) scripts in the allocated time of roughly 40 min. Not all coding pairs used all kind of code chunks provided in Scratch. There were three pairs that did not use any kind of Level 2 chunks. In general, participants used significantly more Level 1 chunks ($M = 28.81$, $SD = 19.25$) than the Level 2 chunks ($M = 17.32$, $SD = 14.07$), $t(33) = |5.79|$, $p < .001$.

As shown in Table 4.3, the full range of rating scale (0 to 4) was used by all programming pairs. The distributions of all five programming dimension had their centre at around 2 (*satisfactory* level). To be more specific, satisfactory means for the category extent and richness that overall there is “one thing” happening (i.e., sprites are mainly moving, changing, counting, switching, or making sounds). For the variety of code usage it means that on average the projects showed many different chunks but mainly Level 1 chunks and only few Level 2 chunks. A satisfactory Level for organisation means that the workspace of the projects looked tidy and scripts were organised, but there were some dead scripts as well. Satisfactory level for the dimension functionality means that Scratch projects worked as intended with only some minor

problems. For programming efficiency satisfactory means that a few code chunks and scripts were copied and the programming pairs used a few control code chunks overall. In total, the programming quality of all pairs over all dimensions was at a satisfactory level and visual inspection (see Appendix F) as well as the Shapiro-Francia test indicated that the weighted mean is normally distributed, $W' = 0.97$, $p = .369$.

Table 4.3

Overview of Programming Quality Dimensions

Programming dimensions	*M* (*SD*)	*Trimmed M (10 %)*	*Mdn*
Extension	1.86 (0.89)	1.81	2.0
Variety	2.19 (1.02)	2.19	2.0
Organisation	1.84 (0.87)	1.84	2.0
Functionality	1.92 (0.95)	1.94	2.0
Efficiency	2.08 (1.21)	2.10	2.0
Weighted mean	2.00 (0.91)	2.03	2.2

Note. *N*(pairs) = 37

4.1.4 Dr Scratch

Descriptive statistics for all dimensions are shown in Table 4.4. The full range for assessment was used only for the dimensions synchronization and parallelism. In more detail, results for flow control indicated most pairs managed to use at least one kind of loop chunk to keep their workflow running more smoothly. Results for data representation indicated that most pairs coded actions for their sprites on a basic level (e.g., editing X- and Y-axes manually). Only a few pairs used a more developed approach such as sensing code chunks in combination with editing X- and Y-axes. Abstraction and problem decomposition for the most pairs were on a basic level as well. That means most pairs managed to have several scripts and more than one sprite but did not define their blocks and did not use clones. Many programming pairs used basic event code chunks such as “clicking green flag” to interact with the user, but most pairs actually used more developed approaches by ask and wait chunks or interacting chunks. Results for synchronization show a widespread use of different approaches. As many pairs used basic approaches (like using wait chunks) as higher advanced chunks (like

wait until and broadcast interactions). However, most pairs did not use any code chunks that implied any kind of synchronization in their coding. A similar broad result can be seen for parallelism where most pairs managed to achieve a basic level (i.e., two scripts were built with simple event code chunks). Only some pairs coded more than two scripts that ran for one sprite or coded even more advanced scripts with more codes running in parallel. Some pairs did not use any kind of parallelism in their coding. Finally, most pairs only used basic if-statements instead of more developed if-else statements, and no pair used logical operations. Many pairs did not use any of these code chunks, which caused a very low score for logical thinking.

Overall, the average mastery score was 9.03 (*SD* = 2.70). Considering the range of 0 to 24, the mean indicated a low but still developing level for the sample in general. Visual inspection (see Appendix F) and the Shapiro-Francia test revealed no significant difference from normal, $W' = 0.96$, $p = .210$.

Table 4.4

Overview of Dr Scratch Dimension

Dr Scratch dimension	Absolute frequency of level				*M* (*SD*)	*Mdn*
	0	1	2	3		
Abstraction and problem decomposition	2	35	-	-	0.95 (0.23)	1
Parallelism	5	21	4	7	1.35 (0.95)	1
Logical thinking	15	20	2	-	0.65 (0.59)	1
Synchronisation	14	11	1	11	1.24 (1.26)	1
Flow control	-	9	28	-	1.76 (0.43)	2
User interactivity	-	11	25	1	1.73 (0.51)	2
Data representation	1	22	14	-	1.35 (0.54)	1

Note. *N*(pairs) = 37

4.2 Answering the first research question

To synchronise all videos, the starting time for all videos was set when the investigator of the study said “Happy coding” in the recordings. The end time was set when the pairs saved their work at the end of the Scratch programming session and no further relevant activity was observed. Out of the 27 unproblematic and complete recordings, the longest video was 42 minutes and 54 seconds and the shortest video was 38 minutes and

19 seconds. The average video was 40 minutes and 5 seconds long. To control for the effect that video length varied slightly over the different pairs, the duration of each CT event were divided by the overall duration of the video to obtain the percentage over time. Overall, 18 hours were recorded, in which 1,438 CT-relevant activities were identified.

It is notable that not all kinds of CT-relevant behaviour were observed (Table 4.5). No pair showed any behaviour that would indicate putting problems into pieces (part of decomposition) or identifying similar structures (part of pattern recognition). This is also true for any kind of behaviour that would indicate neglecting unimportant information. Pattern recognition could be identified only a few times during the whole recorded time and not for every pair. All expected behavioural clues for algorithmic design were seen in all recordings. Together, algorithmic design made ups to 75% of all coded events. Typical examples of coded events are seen in Table 4.6.

Table 4.5

Overview of Coded Events and Time Spent on CT-relevant Behaviour

CT component		*N (pairs)*	*E* (*E/total number of event)*	Percentage of CT-relevant behaviour	
				M (*SD*)	*Max – Min*
Decomposition	Next step	26	110 (.08)	1.30 (0.78)	2.88 – .29
	Problem pieces	-	-	-	-
	Discussing if then	27	200 (.14)	7.41 (3.46)	16.00 – 2.00
	Overall	27	310 (.22)	7.77 (5.35)	22.61 – 1.03
Abstraction – neglecting information	Ignoring details	-	-	-	-
	Simplifying problems	-	-	-	-
	Overall	-	-	-	-
Abstraction –pattern recognition	Identifying similar structures	-	-	-	-
	Copy paste	12	30 (.02)	0.92 (0.98)	3.75 – .18
	Aha moments	10	26 (.02)	1.34 (0.90)	3.41 – .36
	Overall	17	53 (.04)	1.43 (1.05)	3.75 – .18
Algorithmic design	Putting code chunks	27	340 (.24)	21.01 (8.05)	35.21 – 3.42
	Testing	27	479 (.33)	7.88 (3.34)	16.23 – 2.63
	Debugging	27	253 (.18)	8.57 (5.82)	23.91 – .35
	Overall	27	1,072 (.75)	37.46 (12.26)	61.06 – 10.39
CT overall		27		46.14 (14.96)	70.42 – 15. 74
Coded events in total			1,438		

Note: *E* = number of events

In general, behaviour that indicated any kind of pattern recognition took less time than decomposing the problem or algorithm-designing behaviour. Just over the half of all 27 pairs showed any kind of recognising pattern behaviour. On average, pairs spent about one third of the Scratch session with putting code chunks together and nearly half of their time with any kind of CT-relevant behaviour. Visual inspection (see Appendix F) and the Shapiro-Francia test revealed no significant deviation from normality, $W' = 0.97$, $p = .598$, for the percentage of overall CT-relevant behaviour.

It was expected that behaviour indicating problem decomposition would appear rather at the beginning of the session, when participants probably discussing the problem, while algorithmic design would be more dominant at the end of the session, when participants discussing possible solutions. For pattern recognition no specific accumulation of behaviour at any time was expected. To analyse what kind of CT behaviour occurs at which time during the Scratch session, the behaviour was visually mapped as seen in Figure 4.1. As expected, the beginning of the session any kind of problem decomposition was the most dominant behaviour along with non-CT relevant, such as private utterance (i.e., white areas in the Figure indicate no coded events). Interestingly, problem decomposition was not only showed at the beginning but throughout the whole session even at later stages. Contrary to expectations, algorithmic design was not only dominant at the end but throughout the whole session with some pairs started as early as minute one. As expected, pattern recognition was quite equally distributed over the whole session. As mentioned earlier, algorithmic designing was the most dominant behaviour from all observed behaviour for all pairs. There were long (i.e., putting code chunks together) as well as rather short periods (i.e., testing). This is different to behaviour indicating decomposition, which was mostly short events lasting only a couple of seconds. Exceptions of this are pair number 6 and 25. In both cases, participants intensively discussed the plot and mechanics of the game they intended to create. If pattern recognition was identified, it was only for a short period of time and it equally likely at the beginning, the middle, or end of the session as expected.

Table 4.6

Typical Examples of Coded Events

CT component		Utterance or actions
Decomposition	Next step	Pair 3: “Okay. Should we pick our hero and villain first?” Pair 4: “Okay! Next one. Who’s our villain gonna be?” Pair 13: “Oh! We need a sound effect now!”
	Discussing if then	Pair 16: “When he says that [points on dialog bubble] we then wait [points on code chunk called wait] and that’s how we could do the delay, I think?” Pair 19: “Then it’s like when it gets to 10 points or something we add a second ghost and that’s like it gets to the second level and how the game could progress, I reckon.”
Abstraction – pattern recognition	Copy paste	Participants copy a chunk of code to reuse it somewhere else in their project
	Aha moments	Pair 8: “Ah! When that one [points on a sprite] goes on that one [points on another sprite] the score goes like infinitely higher.” “Really? Oh!” Pair 9: “Oh, no! It happened because […] it’s set to when touching sprite1 then go to x. So when the purple [points on code chunk] is gone it has nothing to touch and it just keeps going! So we need to make to green one go!” Pair 10: “Oh! You know what? Because it’s not connected itself! It’s like the one.”
Algorithmic design	Code chunks	Pairs put code chunks together
	Testing	Pairs run their code
	Debugging	Pairs alter their code after they realised their code does not work as intended

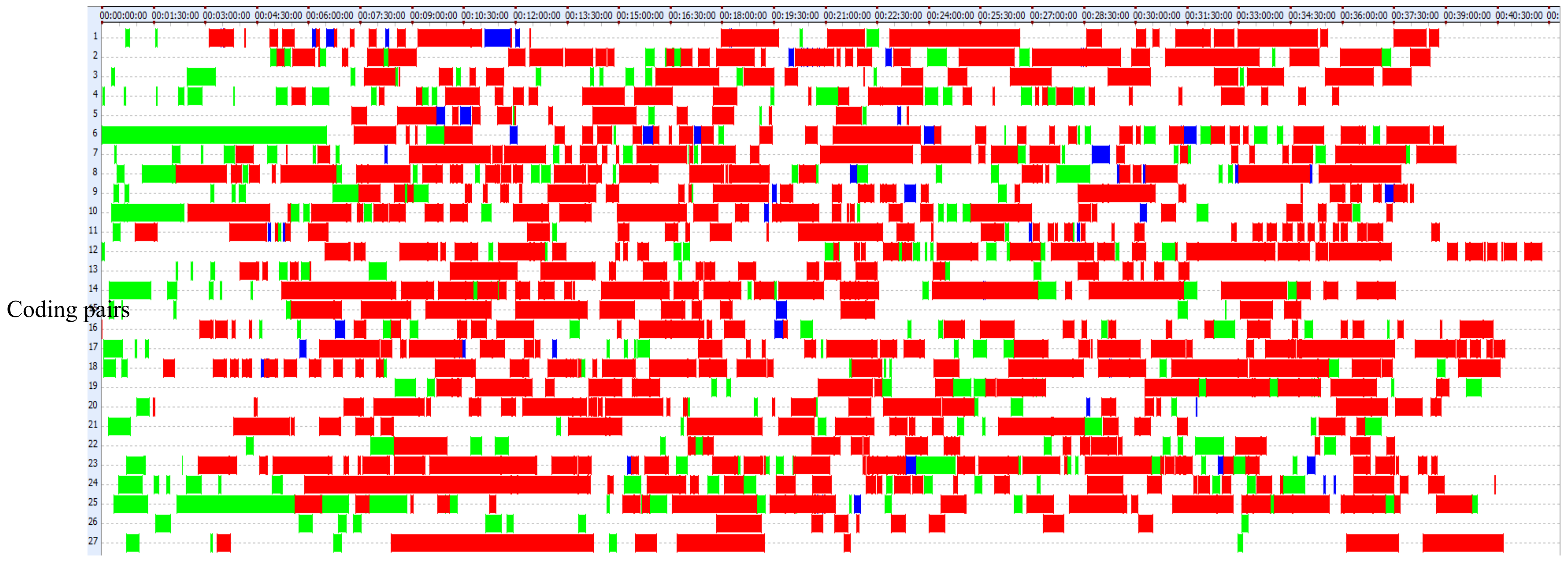


Figure 4.1. Distribution of any CT associated behaviour with green indicating decomposition, blue indicating abstraction, and red indicating algorithmic design.

It is worth mentioning that some pairs had more problems with Scratch than others. This resulted for some to delete everything in the middle of the session and created a new project. Also, some had difficulties with mathematical expressions. For instance, some pairs intensively discussed how to code their sprites so these are able to move in all direction. To do so there is a motion code chunk which refers to the Cartesian coordinate system using the parameter Y for moving up and down and X for moving right and left (Figure 4.2). Some pairs failed to make their sprites moving as they wanted because they lack the knowledge that X and Y stand for different directions.

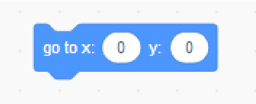


Figure 4.2. Motion code chunk using Y and X parameters.

4.2.1.1 Lag sequential analysis of computational thinking behaviour

To identify any patterns in behaviour, a lag sequential analysis (LSA) was conducted. LSA for overall CT components revealed that with any kind of CT-relevant behaviour it was very likely that it is followed by any kind of algorithmic design (XXX). This is no surprise due to the generally high occurrence of algorithmic design behaviour in the data. Although it is worth mentioning that only the self-occurrence (algorithmic design → algorithmic design) was significant here. The only other significant sequence was the self-occurrence for decomposition. In nearly one third of the time, decomposition was followed by any other kind of decomposition (decomposition → decomposition).

To have a deeper look, the relationships between the single behavioural clues were analysed as well (Table 4.8). Results showed that pairs nearly equally likely started to put code chunks together after they were talking about the immediate next step (next step → putting code chunks, .50) or discussed if then relation regarding the mechanics of their Scratch project (if then → putting code chunks, .48). In up to 50 % of cases pairs showed any kind of decomposition, they started to work on their code afterwards. Pairs generally showed only occasionally copy paste behaviour and so its probability to observe was generally low. However, when it occurred, it was significantly likely to be after pairs were talking the immediate next step (next step → copy paste, .06). The clue “aha moments” was similar rarely as “copy paste”. It appeared with a low but significant probability after another “aha moment” in beforehand (aha moments → aha

moments, .08). In over one third of cases the pairs had “ah moments”, they were trying to debug their code (aha moments → debugging, .35). Pairs tended to test their code chunks every time they worked on it, either after they started to put them together (putting code chunk → testing, .69) or after they tried to debug it (debugging → testing, .82). Although debugging usually occurred after testing (testing → debugging, .44), pairs also significantly likely talked about the next immediate next step after they tested their code (testing → next step, .12).

Table 4.7

Transition Probability Over all CT-relevant Behaviour

CT components	Decomposition	Abstraction – pattern recognition	Algorithmic design
Decomposition	.32*	.06	.62
Abstraction – pattern recognition	.13	.05	.82
Algorithmic design	.19	.04	.77*

Note: *two-sided $p < .05$.

Table 4.8

Transition Probability Over all CT-relevant Behavioural Clues

CT component	Behavioural clue	Decomposition		Abstraction – pattern recognition		Algorithmic design		
		Next step	If then	Aha moments	Copy paste	Putting code chunks	Testing	Debugging
Decomposition	Next step	.14*	.22*	.03	.06*	.50*	.01	.04
	If then	.11*	.20*	-	.04	.48*	.08	.09
Abstraction – pattern recognition	Aha moments	.04	.12	.08*	-	.23	.19	.35*
	Copy paste	-	.10	-	.03	.37	.47	.03
Algorithmic design	Putting code chunks	.02	.09	.02	.02	.15	.69*	.01
	Testing	.12*	.15	.03	.01	.24	.01	.44*
	Debugging	.02	.04	-	.01	.04	.82*	.06

Note: *two-sided $p < .05$.

4.3 Answering the second research question

To analyse the relationship of programming quality and CT Pearson's *r* were computed as a first step (Table 4.9). Significant positive correlations with programming quality were found for the unplugged CT measure (Bebras score) with a medium large effect size. Correlations between programming quality and time spent on CT-relevant behaviour overall and in particular of algorithmic design were found significant and positive with quite large effects. Remaining correlations with TONI-3 IQ and time spent on any other kind of CT-relevant behaviour were not statistically significant.

Correlations between Scratch project evaluation (Dr Scratch mastery score) and remaining measures showed a similar pattern overall. Correlation between programming quality and Dr Scratch mastery score indicated a large positive and significant relationship, $r = .64$, $p < .001$. For visual inspection and correlations based on Spearman's ρ see Appendix and I, respectively.

Table 4.9

Pearson's r Correlations Between Programming Quality, Dr Scratch and Different Measures

	Programming quality		Dr Scratch mastery score		*N* (pairs)
	r	*p*	*r*	*p*	
Bebras score	.30	.038	.28	.048	37
Time of CT-relevant behaviour (overall)	.62	< .001	.61	< .001	27
Time of decomposing	.24	.113	.28	.079	27
Time of pattern recognition	.12	.326	-.17	.252	17
Time of algorithmic design	.63	< .001	.60	< .001	27
IQ based on TONI-3	.23	.099	.13	.234	32

Note: one-sided *p*-values.

Regardless of what kind of programming measure was used (programming quality or Dr Scratch), there is a small till medium large positive relationship with unplugged CT, which means the higher the score for unplugged CT assessment the higher the programming quality of the Scratch project. An even stronger relationship was found

for time. The longer and more often participants spent on CT-relevant behaviour (in particular working on their solutions) the better were their programming results. No such statement was possible for nonverbal IQ and programming.

In a second step, two regression models were estimated with programming quality and Dr Scratch as outcome, respectively, and the CT measures and TONI-3 IQ as predictors. Standardised parameter estimations and tests of significance of the regression model are shown in Table 4.10. The regression models only partly supported the findings from the correlations with the relation between the Bebras score vanished for both programing outcomes, programming quality and Dr Scratch mastery score, even when taking into account the effect of TONI-3 IQ. Similar to the interpretation of the correlation, that means the more time participants spent on CT relevant behaviour the higher the programming quality of the Scratch project (when controlling for nonverbal IQ).

Table 4.10

Regression Models

	Programming quality			Dr Scratch mastery score		
Predictors	β	*t*-value (*SE*)	*p*	β	*t*-value (*SE*)	*p*
Bebras score	-0.41	-1.95 (1.24)	.066	-0.14	-0.62 (4.27)	.542
Time of CT-relevant behaviour (overall)	0.74	4.31 (0.01)	< .001	0.70	3.86 (0.03)	< .001
TONI-3 IQ	0.36	1.82 (0.01)	.084	0.11	0.53 (0.05)	.599
R^2 (R^2_{adj})		.50 (.42)			.44 (.36)	
F(3,20)		6.60	.003		5.29	.008

Note: N = 24. The intercept is omitted for better overview.

Post hoc analyses for both regression models were performed for power estimation. Based on the given parameters ($N = 24$, number of predictors = 3, effect size = $R^2_{pro.qual} = .50$, $R^2_{DrScratch} = .44$, and $\alpha = .05$), a power of > .99 for both models was achieved. Because of the small sample size, assumptions about linear multiple regressions such as homoscedasticity, multicollinearity, and residuals were rigorously checked (see Appendix J). No serious violations of any assumption could be found but the residuals when the outcome is programming quality are not normally distributed, *W'(Y=programming quality)* = 0.88, p = .011. In conclusion, the power of both

regression models were sufficiently high enough and the regression coefficients can be interpreted as "best linear regression estimations" (BLUE).

4.4 Additional results

Further analysis revealed a medium large positive and significant correlation for Bebras score and time, which means the higher participants scored on the test for unplugged CT the longer they spent on CT-relevant behaviour during the programming task, $r(27) = \rho = .39$, $p = .022$.

Because of some (partially) conceptual overlaps between nonverbal intelligence and CT, the correlations between the TONI-3 IQ and CT measures were obtained as well (Table 4.11). As expected, the correlation between TONI-3 IQ and Bebras score was significant and positive with a medium and large effect sizes. The higher the participants' nonverbal IQ the higher they scored on in unplugged CT. No correlation between TONI-3-IQ and any CT-relevant behaviour was significant. Because data for the Bebras score and TONI-3-IQ were originally obtained individually, correlations based on individual scores were computed as well. However, with $r(71) = .53$, $p < .001$, and $\rho(71) = .57$, $p < .001$, results were similar to paired ones and did not alter the overall interpretation that nonverbal IQ and unplugged CT are highly positively correlated.

Table 4.11

Pearson's r for TONI-3-IQ and Different Measures

	TONI-3-IQ				*N* (pairs)
	r	*p*	ρ	*p*	
Bebras score	.52	.002	.49	.002	33
Time of CT-relevant behaviour (overall)	.06	.767	.09	.346	24
Time of decomposing	.01	.963	-.06	.382	24
Time of pattern recognition	.38	.157	.35	.103	15
Time of algorithmic design	.05	.811	-.01	.981	24

Note: *p*-values are one-sided.

5 DISCUSSION

5.1 Summary of the study

In this study, the goal was to analyse the role of CT when working on a programming problem. Because there is still discussion about what CT actually is, the first task was to develop an operational definition and to identify the core elements and major skills associated with CT. To do this, major publications with the goal of defining CT by experts from CS (education) as well as systematic literature were considered. This led to the conclusion that CT is a problem-solving approach, including decomposing a problem, the ability to engage in abstraction, and the ability to understand and design algorithms in order to create a solution to a problem. Because CT emerged originally from CS and is also considered to be a thought process, the CT components were analysed based on their meaning in CS and psychology. This way an explanation was given concerning why these skills are considered to be crucial, what they could look like, and what kind of potential behavioural clues could indicate CT.

In a next step, the relationship between these skills was discussed to determine at which time during the overall problem-solving process the CT-associated skills might be predominant. Moreover, the two most dominant methods for assessment with different perspective and implications on CT were discussed. The one were the Bebras tasks. The Bebras tasks are short quizzes, which claim to measure CT without using any kind of technology and thus are referred as “unplugged methods”. The other method was Scratch. Scratch is a visual programming environment, which claims to measure CT by providing an opportunity, in which users can work free and creatively on their projects.

To answer the RQ1 (how CT is applied when solving a programming task) and RQ2 (what kind of CT measurement might be relevant for predicting programming quality), participants solved a set of slightly altered Bebras tasks and worked together in pairs on a programming task in Scratch. The solving processes of the Scratch task were analysed based on the time participants spent on CT-relevant behaviour. Participants’ solutions were then evaluated and which CT measures were the best predictors for programming quality were identified. In addition, a measure for nonverbal intelligence was assessed to control for potential confounding effects.

5.2 Discussion of the first research question

5.2.1 No or only barely abstract thinking

Analysis of the recordings revealed that abstraction and to some extent problem decomposing were difficult to observe. Based on the literature review, two subcomponents of abstract thinking were identified: neglecting details and recognising patterns. Neglecting details was operationalised as any kind of behaviour that would indicate some kind of simplification of, or actively focussing on, (sub)problems, functions, codes, or solutions, or any other kind of entity. However, nothing like this was observed in the recordings and so there was no behaviour observed that would have indicated people were actively neglecting information while working on a problem. Recognising pattern was operationalised by actions or utterances referring to identifying similar characteristics of entities, copy and paste actions, and aha moments. No behaviour was found that would have directly indicated participants identified similar structures, and aha moments and copy-paste procedures were observed only rarely.

There are various possible reasons for this. First, participants did not show any kind of this behaviour. Second, participants are able to problem decomposition and thinking abstractly but were not able to utilise it. On one hand, it seems unlikely that participants though abstractly to only a small extent. Abstract thinking is an inherent part of human cognition (Rosch, 1978). New experiences are constantly compared with prior knowledge to coordinate and consolidate new information. For example, humans are able to classify an unknown animal by simply comparing it with known animals (Piaget, 1952). By doing so, given information is being evaluated whether it is important or unimportant for a particular categorisation. The same is true for the situation in the Scratch sessions. There are many instances in which participants could have abstracted information and recognised patterns, particularly because, in this study, participants completed a tutorial beforehand. It is very likely that they recognised at least some (sub)problems or (partial) solutions from that.

On the other hand, it must be pointed out that participants were not directly instructed to use CT. Participants had no prior knowledge in programming or CS related concepts and were not familiar with Scratch. It is also very likely that the concept and associated components such as abstract thinking were unknown for the most participants, and, as a result, they did not actively engage in any CT-relevant behaviour.

Maybe participants were able to identify patterns over instances, but they were not able to utilise them for the current task.

Vyn Dyne and Braun (2014) developed a CT workshop to prepare students for more advanced CS-relevant topics. The workshop was concerned with problem solving and covered topics such as decomposition, abstraction, analysis of trends and patterns, and algorithm development. These topics are similar to the CT components as used in this study. Results of later evaluation showed a significant improvement in analytical thinking and reasoning skills and logical thinking. No such improvement was found for students in a control group. Results implicate that CT does not occur naturally but must be trained.

Touretzky, Marghitu, Ludi, Bernstein, and Ni (2013) also designed a framework to foster understanding of fundamental programming concepts. They did this by introducing different VPEs (i.e., Alice, NXT-G, and Kodu) to 31 students. All of the used VPEs have similar commands and provide similar opportunities to users but named differently. By using the same concepts in various VPEs and various names it was hoped students would abstract the essences of the mechanism from syntactic details like names. For instance, WHEN/DO in Kodu are SWITCH blocks in NXT-G and are IF/THEN commands in Kodu and they all can be seen as conditional commands (i.e., the essence of mechanism) even though they look different (syntactic details). Results suggested that students not only enjoyed learning in different VPEs but also managed to switch smoothly between them. Authors concluded that participants indeed recognised patterns over the different VPEs. So, different VPEs were used in order to enhance participants abstract thinking abilities, which was found to be a successful approach. Again, this result also indicated that abstraction does not occur naturally in such context.

A similar approach was used by Basogain, Olabe, Olabe, and Rico (2018) in which several CT skills was taught in preparation courses for novice CS students. In one session, students needed to create a project in Scratch with four main scenes and two sub scenes. This approach is similar to the task used in the current study, in which participants needed to create a story or a game with different facets. The core idea of the task in the study of Basogain et al. (2018) was to introduce the students to a top-down design process in which students needed to decompose the main task in several sub tasks. Students were supposed to recognise that they needed to think from the goal backwards. In another session, abstract thinking was promoted by giving the students

the task to design a project in Scratch in which students needed to program their own customised code chunks with the goal to draw geometric patterns with various parameters. It was hoped that coding their own code chunks requires students to abstract the functionality of already implemented code chunks. Students would need to compare what they have and what they need and are so forced to neglect unimportant details but focus on the important information. Results showed that students indeed improved their grades and gained confidence in their CT skills during the time of the courses.

It is important to improve students' problem decomposition ability and context relevant abstract thinking because it has been shown that both have positive impacts on programming ability. Alaoutinen (2012) analysed different coding style by 145 CS students. One dimension described how information are processed with "being active" and "being reflective" as end poles. Active coders were described as someone who tends to need to actively doing something in order to processed information while reflective coders tend to think tasks through before they start to work (use more decomposition and think more abstractly). One result of the study was that reflective coders had better grades in programming. This underpins the role of decomposition and abstract thinking. In the current study, most pairs would be labelled as active coders based on Alaoutinen's coding style taxonomy. It is possible that more decomposition and abstract thinking could have led to better programming quality.

This leads to the conclusion that at least some crucial CT related skills (i.e., proper problem deconstruction and thinking abstractly in a programming session) do not appear naturally and need to be introduced and trained properly. One session might not be enough to observe them.

5.2.2 Rushing to the solution

A remarkable finding is that all kinds of CT associated behaviour were seen nearly equally distributed over the whole Scratch sessions. Based on the literature, it was concluded that decomposition is more associated with the problem itself whereas algorithmic design is more associated with finding and creating a solution. Abstract thinking, in contrast, was associated with both, the problem in a sense of recognising pattern in sub problems and the solving process in sense of recognising patterns in possible different solutions. Therefore, it was expected that any kind of behaviour associated with problem decomposition was more likely to appear at the beginning of the Scratch session while any kind of behaviour regarding possible solutions would

have been observed subsequently. Behaviour indicating abstract thinking was expected to see equally often over the whole programming task. However, as the visual inspection revealed, a slightly different pattern of CT associated behaviour was revealed. Although problem deconstruction was indeed seen more often and longer at the beginning, such behaviour was also frequently shown throughout the whole session. Also, actions indicating working on the solution (i.e., algorithmic design) started from a very early stage and were shown constantly until the end. Pairs also did not decompose the problem only at the beginning but over the whole session. In general, pairs did not spend much time discussing the problem in comparison to working on the solution. They spent over one third of the overall Scratch session with algorithmic-associated behaviour and spent much less time on decomposing and abstracting. Results indicated that pairs were working on the solution without much thinking about the task or the solution itself.

This is a typical behaviour for novice programmers (McDonald, 2018). Novices tend to dive straight into the task without thinking what they want to accomplish. They think trial and error is an appropriate way to produce results. That leads to the idea that this is a more efficient way to produce a result rather spending time designing a computer program on paper in form of flow charts to organise their ideas and thoughts. However, McDonald (2018) further concluded that is always better to structure the program on paper first. He compared programming without a plan with constructing a building without blue prints. Of course it is possible to just pile a bunch of bricks and creating a house just by doing it. It will fulfil the purpose of giving shelter but it also may have some unfortunate features like skewed walls or a bathroom connected to the dining room. It is the same with programming. Of course it is possible to create a program by just writing some code or putting some code chunks together. It also may fulfil its purpose and work as intended. Nonetheless, as Martin (2009, pp. 200–201) emphasised the primary goal is not to get the program working. It is about planning ahead and knowing the goal of the program. This involves the CT crucial such as problem decomposition and abstract thinking. Novice programmers and people, who do not have any experience in programming like the participants in this study, lack these kinds of CT associated skills.

Results of the LSA revealed that, when pairs showed any kind of decomposing behaviour, they usually then started to put code chunks together. The probability of discussing the next step was highest when they had tested their code immediately

beforehand. Also, when they worked on their solution it was very likely that their next behaviour would have something such as testing or debugging their code to do with the solution. In general, the circle of putting code chunks together, testing, debugging, and testing again showed participants were primarily focus on their coded solution. None of the pairs stated clearly what kind of steps they needed to make through the whole process. For instance, many steps suggested by Jonassen's (2000) problem-solving model for ill-structured problems (representation of the problem-space; identifying and clearing alternatives, monitoring the problem space) were not observed.

These results are comparable to results in prior studies. Falloon (2016) conducted a study to investigate what kind of CT processes of young children (5 to 6 years old) are mostly evident and how these processes are applied when working on a task that was similar to the task used in this study. The second most frequently exhibited behaviour was associated with debugging and testing. In addition, children were mostly occupied with the same kind of behavioural circle: creating, testing, debugging, and testing again. These results imply that this kind of behaviour might be typical people who have no prior knowledge about programming or CT.

In conclusion, pairs rushed to the solution with little forward planning. Issues were most often discussed only when they arose. Participants worked on their code after aha moments and showed copy–paste behaviour mainly after discussing a subsequent step. Pairs often did not properly discuss what their goal was but started to work on a solution from the beginning. It is possible that this kind of behaviour might be typical for this kind of tasks if not instructed otherwise.

5.2.3 Some prior mathematical knowledge required

The participants of this study had no significant prior programming knowledge. Thus, to analyse CT during a programming session, the programming environment had be easy for novices to learn. Scratch seemed to be a good choice because of its low threshold and easy access. Although Scratch can be used without programming skills, it still relies on some knowledge of mathematics. For instance, some pairs failed to code their sprites as they might have wanted to because they did not know about the Cartesian coordinate system with X representing horizontal movements and Y representing vertical movements. This result bolsters the opinion of some who argue that CT overlaps, to at least some extent, with mathematics (Shute, Sun, & Asbell-Clarke, 2017). Others even

link CT directly to the knowledge of the Cartesian system (Mensing, Mak, Bird, & Billings, 2013).

Specific knowledge is an important factor for successful problem solving, though (Bransford & Stein, 1993, p. 4). People's ability of solving problems is strongly connected to the amount of knowledge about the area of the problem. Bransford and Stein (1993) further stated that the effect of general problem solving skills are often overestimated while the role of knowledge is underestimated. People tend to make inference of someone's level of intelligence when observing failed or successful solving a problem although the reason might be simply lie in the level of knowledge. The same might be true about CT. As intelligence, CT is seen as a problem-solving approach not necessarily limited to a specific area. Failing or being successful in solving a programming problem does not only depend on the level of CT but also on the level knowledge a person has about the problem area.

5.3 Discussion of the second research question

Two different measures for CT were used to investigate the relationship between programming quality and CT. One measure, the Bebras tasks, is considered as unplugged method and is based on abstract problems with no obvious link to CS concepts. The other measure, the CTBS, focused on behaviour participants showed while solving a programming task. Both CT measures were positively correlated with each other with a medium large effect size. This indicates a certain level of convergent construct validity. Convergent construct validity refers to whether a test is measuring the construct it claims to be measuring (Cronbach & Meehl, 1955) and is established by comparing different measures of the same construct with each other as done in this study. Correlations between both CT measures indicate that they may tap the same construct, but it is possible that they do this from different perspectives. While the Bebras tasks capture the more abstract parts of CT the CTBS covers more the later stage of the solving process when people design and implement solutions. The "only" medium large effect size might reflect these different perspectives on the same construct.

Both CT measures were positively correlated with programming quality. As a consequence, a general interpretation could be that the higher the level of CT the better the programming quality. However, this interpretation would be premature because the

regression analysis revealed that only one—the time participant spent on CT-relevant behaviour—was a significant predictor of programming quality when controlling for other variables such as the level of nonverbal intelligence and other CT measure. A second regression analysis with Dr Scratch mastery score as a measure for evaluation of Scratch projects supported this finding. Again, the reason why the two different CT measures predict programming differently well might lie in different perspectives the measures have on CT and the perspective might mediate the relationship with programming.

The Bebras tasks might focus on the abstract parts of CT. Correlations between the Bebras score and the TONI-3-IQ were high regardless of whether the units of analysis were individual or paired scores or the correlations were based on Pearson's *r* or Spearman's ρ. As for the most instruments for nonverbal intelligence, TONI-3 is based on pictures in which participants need to identify similar instances and recognise patterns. Many of the Bebras tasks are designed in a similar fashion. The original idea behind the Bebras tasks was to create a test about CS concepts "independent from specific systems" to avoid contestants being dependent on prior knowledge of any specific IT system (Dagienė & Futschek, 2008, p. 22). This led to some items being similar to those of nonverbal intelligence tests.

As found in some prior studies, this also caused confusion for some Bebras contestants. Vaníček (2014) asked participants for their opinions about the Bebras tasks. Some questioned the purpose and validity of the test, stating, "I wonder what the contest questions have to do with informatics. Maybe nothing at all?" If (at least some) Bebras tasks are similar to those of nonverbal intelligence tests and there is a high and significant positive correlation between both measures, it is possible that both tests measure similar constructs. This would explain why the relationship between the Bebras scores and programming quality vanished when controlled for TONI-3-IQ. The Bebras tasks are validated by several studies (Dagienė & Stupuriene, 2016; Dolgopolovas, Jevsikova, Savulionienė, & Dagienė, 2015; Lockwood & Mooney, 2018) but none of these studies controlled for any potential confounding effects on similar psychological constructs such as nonverbal intelligence. So far there is only one study in which the potential relationship between the Bebras tasks and nonverbal intelligence has been discussed with similar findings to this study (Román-González, Pérez-González, & Jiménez-Fernández, 2017). Thus, it is possible that the Bebras tasks indeed measure CT but mainly the facet of abstract thinking related to pattern recognition.

It is possible that this rather abstract part of CT alone is not a good predictor for programming ability because more cognitive effort is required to transfer the needed skills to apply in a different situation and setting. According to the authors of the Bebras tasks, participants need to apply the same cognitive abilities as needed for programming tasks such as problem deconstruction, thinking abstractly, and being able to understand, design, and evaluate algorithms (Dagienė & Sentance, 2016). However, the content of the Bebras tasks (as for the most unplugged methods) is very different from real programming tasks. Even though the same skills are required to solve both kinds of tasks, the Bebras tasks as well as the programming task in this study, it would require a high level of transferability from these abstract logical quizzes to real applied programming situations. This is similar to the conjecture playing Sudoku or other brain training games would generally improve cognitive abilities. However, that is likely not the case.

Stojanoski, Lyons, Pearce, and Owen (2018) tested this hypothesis by designing an experiment with a treatment and a control group. Participants of both groups played a game with the goal to identify correct items with logical clues given as support. The task was adaptive, which means it increased its difficulty automatically along with the increasing capabilities of the participants in order to being constantly a challenge. The treatment group were trained in this game for over two weeks while participants of the control group played the game only in a pre and post session. To test for the transferability of what participants might have learnt during the test session, a second game was implemented. The gameplay mechanics and goal of both games were the same but the given clues for the second game were changed so slightly different cognitive abilities were required. With no surprise, participants of the treatment group scored higher in the trained game than participants of the control group, which means the cognitive ability required for this game has indeed improved. However, no such improvement was found for the non-trained test game. Performance of the treatment group was similar to the control group. A second experiment with a different game and altered game mechanics supported the original finding. Authors concluded that despite the improvement in the trained games, there is no evidence for any transferable gains in performance for untrained tests was found. The authors also emphasised how the overall design of the trained and test games was quite similar and still no transfers in rather simple cognitive abilities such as recognising and identifying could not be found. However, that does not mean that no transfer from any cognitive training is possible. Kelly et al. (2014) analysed in a meta-study randomised controlled trials about the

effectiveness and transferability of cognitive trainings. In total, 21 studies reported some sort of transfer effects but were constraint on rather simple cognitive abilities (i.e., executive functions such as processing speed, memory and recall measures). Results were found most reliable within the same cognitive domain but effects were rather small. With no surprise, the more different the trained tasks were to the tested situation the smaller the effects. Thus, it is unlikely that tests such as the Bebras tasks have much of an impact on programming. The comical, logical quizzes of the Bebras contests are very different from programming tasks. Using them the Bebras task or other unplugged methods might increase people's motivation, but it is questionable whether the developed and improved skills can be useful generalised to other areas.

In contrast to the Bebras tasks, the focus of the CTBS lies on participants' actions. Correlations indicated that the more participant spent on CT associated behaviour the better the programming quality of their Scratch project. It must be pointed that this was mostly due to algorithmic design and algorithmic design is the more hands-on activity of CT. As stated before, participants were working on their code from the start of the session and so the interpretation would be the longer and the more participants coded the better. Even after controlling for other measures, this relationship was still significant and persisted in both regression models with programming quality and Scratch project evaluation as outcome, respectively.

These results indicate that hands-on tasks are more useful to enhance CT skills than more abstract ones. Such hands-on practices can be designed in VPEs such as Scratch. Indeed, Scratch has been found to be an effective tool for teaching CS concepts and programming in the past. In a more recent study, Chen, Haduong, Brennan, Sonnert, and Sadler (2019) asked over 10,000 second and fourth years CS students about their experience with VPEs and their first "real" classic programming language. Results can be summarised that VPEs have positive effects on programming. Xu, Ritzhaupt, Tian, and Umapathy (2019) conducted a meta-analysis on 13 studies about the effect VPEs have on cognitive and affective student learning outcomes. Cognitive outcome were measured by achievement in sorts of problem-solving often based on Bloom's taxonomy and improvement in programming skills. Even though effect sizes were generally small, results revealed a positive effect in favour of VPEs on cognitive measures. Also direct effects of using Scratch to teach programming has been found. Armoni, Meerbaum-Salant, and Ben-Ari (2015) investigated the effects of Scratch courses on the level of understanding of CS concepts such as conditional or repeated

executions and handling of variables. For this, high school students from four different schools were introduced with a Scratch preparation course for programming. Results can be described as small but in favour for Scratch. Students who had worked with Scratch had fewer difficulties and needed less time to learn new concepts and had a better understanding of the concepts. Scratch is a more hands-on tool than the more abstract tasks of the Bebras contest and therefore less cognitive effort is needed to transfer to solve programming tasks which might explain the strong relationship with the time participants spent on CT-relevant behaviour and programming ability.

In conclusion, the answer to the question how CT and programming quality are related to each other should be simply it depends. To solve the Bebras task, the same CT-related skills are required as the CTBS has discovered during the programming session. However, the focus of the Bebras tasks lies more on the abstract parts of CT while the CTBS focus more the hands-on part. It turned out that, if the view on CT is rather focusing on the abstract parts, only a little relationship can be found, which vanished after a more behaviour-focused measure is taken into account. The reason for this might lie in the level of cognitive effort needed to transfer CT related skills.

5.4 Practical implications

Results of previous studies implied that CT must be trained to be useful in a programming situation and results of this study support this implication. Therefore, in the following an overview of a developing educational framework about CT skills is presented.

CT involves different skills such as problem decomposition, thinking abstractly and algorithmic design. Results of the current study indicated that these skills should be taught in a programming setting in order to keep the level of transferability low. Unplugged methods alone might be not enough to teach CT effectively and should not be used as a stand-alone teaching unit (Bell & Vahrenhold, 2018). Instead, it should be embedded into the curriculum. If the goal is to use CT skills to solve programming tasks then CT should be included into programming curriculum. It should not be assumed that this alone would raise a general level of CT and people would be suddenly able to use CT in different settings. If, for example, CT is supposed to be a useful tool for other STEM areas such as biology or physics as Wing original stated (Wing, 2006) then it should be implemented in curriculum about biology and physics, respectively.

Not only the context plays a role but also the target group. Often the target group of CT are younger students. In this case, VPEs have been found to be an effective tool in order to enhance CT for programming purposes. However, Xu, Ritzhaupt, Tian, and Umapathy (2019) concluded that the effectiveness of VPEs depends how they are implemented in the educational setting emphasising the importance of the correct educational framework. Just letting students working in VPEs will barely increase their level of CT or enhance their level of understanding about programming concepts. Chen, Haduong, Brennan, Sonnert, and Sadler (2019) critically remarked that VPEs are not generally superior to classic programming languages with regards to learning programming, though. VPEs are more effective when introduced at an early age while no such positive effects were found when students were already teens or older. Based on these results, Chen et al. (2019) further questioned the recent trend of using VPEs to introduce programming concepts or teaching CT for university students.

5.4.1 Problem solving

Computational thinking is just one problem solving approach out of many. It is suited for a specific kind of problems. Thus, it is important to teach the scope but also its limits of CT and what kind of other problem solving strategies there are. CT is considered as problem-solving approach especially useful for ill-structured problems. So students first need to learn the taxonomy of problems and what it means to face an ill-structured problem in comparison to a well-structured one. A revised version of Jonassen's (2000) model for solving ill-structured problems might be a first step. Jonassen's model defined seven steps (1. representation of problem space, 2. identifying and clearing alternatives, 3. generating possible solutions, 4. viability of alternative solutions, 5. monitoring the problem space, 6. implementing and monitoring solutions, 7. adapting solutions), which can roughly be summarised in three stages: planning, implementing, and evaluation.

Pairs in this study barely talked about the problem itself. This might be a sign they lack attention for the planning stage. During this stage, participants should first be sure to fully understand the problem and being clear about the overall goal. This is meant by representation of the problem space. The task in this story was to program a story or a game where a hero has to overcome a challenge in order to defeat the villain(s). A first step would be to clarify what these elements mean to the problem. What do villain, overcome a challenge, and hero mean in this context? Bransford and Stein (1993) also

emphasised the role of prior knowledge and experience especially for solving ill-structured problems. So the planning phase also includes comparing similar problems in the past with the current situations. What are some similarities and what are the differences? This is when problem solver use abstract thinking and trying to identify similar structures while ignoring unimportant details like the context of the past problems.

Usually there might be more than one possible solution for ill-structured problems. This can be overwhelming and can cause some confusion. Therefore, it is crucial to focus on one goal and one approach at a time. To decide what possible solution should be tried first an evaluation system should be created based on knowledge and own beliefs (Jonassen, 2000). For instance, if working on one possible solutions takes too long because unforeseeable problems occur, it might be wise to switch to another possible solution. Some pairs in this study did exactly this because they became frustrated with their first attempt. An evaluation system helps to identify faster suitable solutions and can guide through the whole process.

In summary, in order to use CT effectively for programming, students need general knowledge about different types of problems and what solving steps are needed. This includes clarifying the problem representation and creating an evaluation system for possible solutions. Ill-structured problems usually have more than one possible solution, which means students need to be prepared that solutions might require adjustment and students should not fear to go back to prior steps of the process.

5.4.2 Decomposition

The core idea of problem decomposition is to identify the different levels a problem can have. Based on Lee and Anderson's (2001) model of task analysis, there are three main levels. At the unit-task level, the main goal is divided into several subgoals which can mainly be completed mostly independently. The level below is the functional level in which these subgoals are further deconstructed. The lowest level is the keystroke level which always represents the most atomic unit. Problem at this level cannot be further decomposed. The task used in this study could be divided into the following subtasks on the unit-level: (1) develop a plot, (2) create a hero, and (3) create a villain. These tasks can be further deconstructed into smaller chunks on the functional level. Developing a plot, for example, can be further deconstructed into developing (1.1) a beginning, (1.2) an end, (1.3) a challenge, and (1.4) a turning point. The usage of code chunks in Scratch

and creating coding sequences would be representing the keystroke level as the lowest one.

In order to teach such problem deconstruction approach, the benefits should be emphasised such as reduction of complexity. Problems of the lower levels appear to be less complex and so less cognitive effort is needed to achieve a subgoal. In addition, it is possible that identified subgoals can be achieved independently, which may increase the efficiency of the overall problem-solving process. As mentioned before, knowledge plays a crucial role in any kind of problem solving. This is especially true for the lowest level in problem decomposition. The lowest possible level is always dependent on the system in which it is carried out, which also means that the problem solver has enough knowledge about the system in order to plan the steps on the lowest level. Or to put it differently, if a problem solver encounters some serious difficulties to plan throughout the problem-levels, this might be a sign that some more knowledge about the system is needed before attempting to create any solutions.

Interestingly, problem decomposition has been perceived by students as one of the most difficult CT skill to master (Selby, 2015). Although the concept of problem decomposition seems often straight forward, students often struggle to use it effectively. Whether students are able to successfully deconstruct a problem appears to be dependent on the level of familiarity of the problem. Selby further stated that it is more likely students recognise the potential subgoals and different levels when students already know the solution or understand the problem well. Another reason why problem decomposition appears to be a challenge might be that the connections between the levels are not clear to the problem solver. Identifying subgoals on different levels is not enough. It is crucial to recognise how subgoals and levels are linked with each other. A problem solver needs to fully understand top-down process of decomposition. The results of this study show that was likely not always the case. Many students talked about the next immediate step to do but no serious top-down process was apparent.

Selby's results and results of this study imply that decomposition should be trained on familiar and maybe already solved problems. It might be then easier to understand for students what subgoals on which levels there are and how these are connected to each other. This way the full top-down problem decomposition process might come clear to the students.

5.4.3 Abstraction

The core of abstract thinking is being able to understand the relationship between different instances. It means to be able to see through the unimportant details and to recognise the deeper lying structure what these instances have in common. One way to enhance this ability is using analogies (Anderson, 2015, pp. 188–191). Analogy can be described as a process in which specific operators are taken from one problem and are mapped onto a solution to another problem. A classic example of an analogy is Rutherford's model about atoms in which electrons surround the nucleus of atoms the same way the planets surrounding the sun. Although the elements in both instances are different, the structure and relation of these elements remains the same.

The underlying structure of abstract reasoning is logical reasoning and so Nickerson (2011) also promoted teaching (formal) logic in order to enhance abstract thinking abilities. Logic reasoning provides a clear sequence of arguments with is also crucial for understanding algorithmic solutions. In addition, programming concepts are also strongly based on logic such as Boolean algebra. Nickerson critically stated, though, that logic is difficult to teach and transfer of this concept is particularly low. To keep the transfer level low, teachers should create simple and short tasks in case learners are not familiar with formal logic.

In general, abstract thinking has proven to be particularly tricky to teach because of its known high dependencies of context (Kelly et al., 2014). In case the goal is to improve abstract thinking in order to improve programming skills then using analogies and teaching formal logic alone will be barely effective. Although all are high correlated, Lohman and Lakin (2011) distinguished different forms of abstract reasoning like verbal reasoning, quantitative reasoning, and figural reasoning. Verbal reasoning is about the understanding of concepts and problems expressed in words, quantitative reasoning is about problems in numbers and figural reasoning is about the relation between geometrical forms. In that sense there might be also "computational thinking reasoning" which describes the ability to recognise patterns in algorithmic problems and solutions. To enhance the later one, exercises should be created in such programming setting (e.g., in VPEs or in actual programming languages).

If learners spontaneously try to solve novel problems by using actively abstract thinking, it is possible they are guided by superficial similarities. Ross (1984) taught different problem-solving methods by using examples of problems such as estimating

the probability of two dice sum up to seven. Only when the test examples illustrated the same principle as needed (e.g., same principle of probability), participants were able to solve new problems. When it did not and participants tried to abstract the structure by themselves, they tended to focus on superficial similarities (e.g., using dice or not) and so they were not able to solve any new problems. Unsupervised training can lead to wrong conclusions by learners especially if learners are new to the field.

The low level of abstract thinking of the participants in the current study also suggested that participants needed help to focus the crucial features so they would have been more capable of recognising patterns. Participants had no prior experience in programming or whatsoever and so it might have been difficult for them to focus on the crucial elements. Therefore, at least at the beginning of the learning process, a teacher should actively help learners to identify the crucial features of the learning material. For example, when introducing students to a new VPE and a warm-up phase is used in which learners learn the mechanics, teachers should encourage the students to actively point out similarities between the elements of the tutorial session and how the same code chunks can be used in different situations or how the same goal can be achieved by using different code chunks. This is similar to Touretzky, Marghitu, Ludi, Bernstein, and Ni (2013) who used different VPEs in order to teach the same concept. The name of the code chunks differed over the various VPE but the principle were all the same. This approach might enhance the abstract thinking abilities of learners.

To avoid learners focus on the wrong details, the training material must be chosen wisely. That means the training and test items should not be too close to each other because that would be just repetition of familiar material (Ross, 1984). On the other hand, if both are too far away from each other learners might fail to recognise similarities. Participants of this study probably did not use much what they learnt in the warm-up phase probably because the link between the tutorials and the actual programming tasks was not clear enough to them.

In summary, using analogies and teaching formal logic alone might be a good foundation for generally enhance abstract thinking but might not be enough to enhance abstract thinking in context of CT because of the low level of transferability of cognitive abilities. Especially at the beginning when material and concepts are new to learners, the learning process should not be unsupervised because novices tend to focus on superficial and not critically important similarities. There must be a right balance between the learning material at the beginning and the material at an advanced learning

stage. Materials must not be too similar and neither too different in order to provide learner the chance to recognise patterns.

5.4.4 Algorithmic design

Algorithmic design summarised the whole process of creating, testing and debugging a solution in CT. To be able to create an algorithmic solution learners need to know what algorithms are and how they work. Learners need to understand that an algorithm consists of different commands with different purposes and how these commands are semantically related to teach other. For instance, the general idea of a simple algorithm might be “when left click then move avatar 10 units right”. The general knowledge about algorithms is the first step. In a second step learners need to know how to implement their solutions. This implementation is highly dependent on the current used system. If an algorithm is supposed to be implemented in Scratch then learners not only need to understand the general sequences of commands but also how to create these commands in Scratch. How does “left click” look like in Scratch and what units are used to move the avatar to the right? The same is true for any other kind of used system. Domain specific knowledge is crucial for designing algorithmic solutions.

There are several educational frameworks about CT and the focus of many of them lies on the algorithmic part of CT. For example, Grover et al. (2019; 2015) developed several educational framework on algorithmic problem solving. The core idea of their frameworks is always to teach CT and CS principle such as abstraction and problem decomposition with different tasks using VPEs (in most cases Scratch). For example, to teach the idea of loops a set of tasks is design and dived into different units. In one unit, students are supposed to create a spiral by using the loop-code chunk. In later units the task becomes more complex and different forms of loops (e.g., nested loops) are included or the code must be further altered by adding variables and conditional expressions (Grover, Jackiw, & Lundh, 2019). This step-by-step procedure gives students the opportunity to understand how the different code chunks are related to each other.

When translating the general idea of an algorithm into the current system, it is very likely that mistakes are going to happen and programs do not work as intended. This is the reason why debugging and fixing sequences of codes often play a dominant role in most of educational frameworks (Grover, Jackiw, & Lundh, 2019; Grover, Pea, & Cooper, 2015; Voogt, Fisser, Good, Mishra, & Yadav, 2015). To practice debugging

skills there are often other than the own coding attempts are used. This has a specific reason. As every writer has his or her own style in writing so has every programmer his or her style in programming (Martin, 2009, p. xxii). When learners start to practice from scratch they quickly start to develop their own style. When practice only on the own codes they will learn to avoid certain mistakes they personally usually tend to do. When learners also see different styles of coding they encounter also different kinds of mistakes. Learners then need to understand the thoughts and intentions of the original designer. This is a way to improve reflecting skills (Grover, Pea, & Cooper, 2015), which might have a positive impact on debugging capabilities and might also enhance learners' general understanding of algorithmic solutions.

In summary, designing of algorithmic solution is dependent on the fundamental knowledge of how algorithms work but also about the current system in which an algorithm is supposed to be created. Teachers need to be sure that not only the concept and purpose of code chunks are understood by learners but also how these code chunks are related to each other. Only then an algorithmic solution can be designed. Debugging plays a crucial role and can be practiced by using malfunctioned codes.

5.5 Critical evaluation of the study

Within this section, this study is evaluated critically. Methodological and theoretical limitations are discussed, including the overall design of the study, the used instruments and the limited conceptual view on CT as seen in this study.

5.5.1 Methodological

5.5.1.1 Research design

Because the goal of this study was to analyse students' CT-associated performance while they solved a programming task, a video an observational video study seemed appropriate. Although video studies have many advantages, they also have some drawbacks. Participants could have acted differently because they knew they were being recorded. Some pairs actually made comments indicating that they were aware of the camera. However, none of this behaviour was prolonged for any of the participants. Soon after the recording had started they noticeably focused on the tasks. None of the participants mentioned the recording after minute 5, and none of the participants looked

straight into the camera or showed any other sign of awareness of the recording. Some pairs even discussed private issues with each other or were looking at their phone, which implied that they forgot about the camera. Thus, no serious concerns need be raised regarding the participants' natural behaviour.

The design of the study also led to the situation that different measures (Dr Scratch, programming ability, and CT-associated time) were based on Scratch. Although the focus of all instruments differed, this might have had an effect on the measures and should be considered when interpreting results.

Students worked in pairs because it was hoped that this would provoke social interaction and make otherwise unobservable thoughts accessible. Moreover, pair-programming settings have been used in prior studies in terms of measuring CT and programming knowledge for novices (Denner, Werner, Campe, & Ortiz, 2014; Wu, Hu, Ruis, & Wang, 2019). Nonetheless, this approach came with some challenges.

The performance of pairs might be dependent on the people who work together and how they get along with each other (Hanks, Fitzgerald, McCauley, Murphy, & Zander, 2011). Participants did not freely choose their partners but were paired according to their Bebras scores in order to prevent broad discrepancies in their levels of CT. It was assumed that huge differences could have negative effects on their performance during the Scratch session. Most participants did not know each other well. The 40-minute tutorial session before the actual tasks was the first time they worked together. It was hoped this was sufficient time for getting to know each other and bond. This appeared to be the case for most pairs, although some pairs talked or interacted with each other to only a small extent.

Another challenge is the usage of paired values as unit of analysis. It is questionable whether there is something like common levels of CT, intelligence, or programming quality. Some might argue that the results and overall conclusion might have been different if all measures were obtained and analysed solely on an individual basis. However, it must be noted that the Bebras scores as well as TONI-3-IQ were originally obtained individually and paired later. This made it possible to run some analyses based on individual as well as paired scores and compare the results with each other. These results were similar. Despite these challenges, the use of dyads is justified by added findings on collaboration such as the study conducted by Denner et al. (2014). Notably, according to the findings of Denner et al. (2014), students that work collaboratively in

pairs attain considerably higher CT scores than students working alone. Just as importantly, working in pairs is advantageous for students with little programming experience (Denner et al., 2014). In addition, working in pairs is quite common in the field of programming. Analyse a programming product as a team effort secures a certain level of external validity.

With a total sample of over 108 participants, the study reached a sufficient level of power for the calculated regression model, when expected effects lie in the "medium size" ranges (Cohen, 1988). However, some argue that post hoc power analyses are not particularly meaningful and misleading. Calculating post hoc power may seem to provide more statistical arguments but power is just an inverse function based on test probabilities and effect size (Aberson, 2019, p. 15). In case of rejecting the null hypotheses, it means power was sufficient to detect an effect by the given sample size and test probabilities. It is just a p-value in another shape. Therefore, post hoc power analysis adds no new information should not be overestimated.

In addition, the sample size reduced dramatically for some analyses based on paired scores. Pairs for the programming task in Scratch were created only after both participants completed the Bebras tasks and the TONI-3, which resulted in 37 pairs. Due to technical problems (e.g., when a computer froze, turning off microphones or webcams by accident) resulting in 27 complete and unproblematic recordings, which further limits the generalisation of quantitative analyses.

5.5.1.2 Instruments and measures

Because participants were all Australians and to avoid problems with use and recognition of idioms, only the Australian version of the Bebras tasks from 2014 and 2015 were used in this study. Although the conception and structure of the tasks do not differ much across the countries and years, it is possible that results might change with different Bebras tasks. It must be also noted that the Bebras tasks are originally designed elementary and high school students. There are studies in which CT of university students is measures by Bebras tasks but is this not a common approach. Based on the results of the pilot study, there were no reason of concerns and so the Bebras tasks available for the oldest age group were used in this study. However, results must be interpreted by caution.

Fortunately, the nonverbal intelligence assessment did not raise any concerns. It is generally assumed that intelligence is normally distributed in the population with μ =

100 and σ = 15 (Sternberg, 2017). The TONI-3-IQ does not generally differ from the assumed population in shape and dispersion, which indicates a good fit. The reason why the average mean is nearly 1 SD higher than the general population might be explained by the fact that only university students have been observed.

The TONI-3 was chosen because of its satisfactory psychometric properties, existing normative data relative to specified subgroups, and overall good conceptual fit. However, all items are geometric forms and figures in which participants need to identify patterns. Some argue that this is only one of several facets of nonverbal intelligence and figural reasoning alone might not be enough to sufficiently measure nonverbal intelligence (Wilhelm, 2005). Furthermore, the whole concept of intelligence was never without controversy. Scholars suggest use of diffuse concepts and terms such "cognitive abilities" instead of intelligence (Urbina, 2011, p. 35). This is because cognitive processes are easier to define and there is less heated discussion and there are fewer emotional associations with them.

The CTBS was created for the purpose of this study. That means this instrument was not used in other studies yet. Interrater reliability assessments indicated a satisfactory level of agreement on the different CT-relevant behavioural clues but the results of the CTBS still must be interpreted with caution because some indicators of some CT-relevant behaviour are dependent on the used environment. For instance, the CT component algorithmic design subordinates all utterance and actions with the purpose of designing an algorithmic solution to a problem. The programming task in this study was designs in Scratch in which the only way to create algorithmic solutions was to put code chunks together. If another programming environment would be used, other indicators could be identified. This limits the generalisation of the results of the study.

In addition, it is also possible the CTBS was not sufficiently sensitive to assess abstract thinking on a satisfactory level. Indeed, the results of the pilot study indicated that observing abstract thinking might be a challenge. As for the main study, no behavioural clues indicating neglecting information were found in the pilot study and pattern recognition was observed only rarely. Because abstraction plays such crucial role in CT, the investigator of the study still decided to keep abstraction in the CTBS to be sure to catch any signs of abstraction in case there might be any. Abstract thinking has been described as a complex information process of higher-order thinking. The process of recognising patterns and especially neglecting unimportant information might primarily take place automatically, with people often being unaware of it

(Barsalou, 2003; Carlson & Dulany, 1985). If people are not aware of it, it is difficult to observe in social interactions and other measures of abstract thinking might be more sensitive.

For the purpose of this study, a rubric scheme was developed to measure the programming quality of students' Scratch projects. Quality concepts such as variety of used code chunks and coding efficiency. While rubric schemes became more popular over the last few years, there are also some critical voices. Menéndez-Varela and Gregori-Giralt (2016) investigated the validity of rubric-based performance assessments of 84 first year students studying Conservation–Restoration and Design. They compared scored based on a rubric with ratings of two teachers and three student tutors. They concluded that rubrics contribute to students' learning performance. The strength of rubrics lies in promoting shared understanding of learning objectives and it is helpful when providing feedback. In that sense, rubrics are a good tool for formative assessment. However, rubrics reduce the complexity of a learning outcome. It depends on the topic of a course or projects whether such reduction may the usage of rubrics less favourable. Thus, Menéndez-Varela and Gregori-Giralt (2016) see rubrics as a scoring tool for summative assessment critically.

Panadero and Jonsson (2020) came to a similar critical conclusion about rubrics. In a meta-analysis of 27 publications about rubrics, they identified several "themes" mentioned in these papers. Among them were "standardisation and narrowing the curriculum" and "limitations of criteria". This shows how the biggest strength of the rubrics are also their biggest weakness. Reduction of complexity to achieve higher reliability may also result in less validity. Programming quality is a complex concept which has not been generally defined. Although the rubric score is based on the literature and checked by a former computer science teacher with several decades of experience, it is likely that there are criteria of programming, which were not included in the vertical dimensions of the current rubric. It is also possible the quantitative steps are too broad or too narrow. Theses biases may result in underestimating or overestimating true scores of programming quality. Therefore, conclusion about programming quality should be only made with similar definition and criteria used in this study and generalisation of the results must be made with caution.

In this study, Dr Scratch was used as an instrument to evaluate Scratch projects but not to measure someone's level of CT as it is usually used in the field. This decision was first made based on conceptual reasons. Dr Scratch predominantly relies on the use

of code chunks, which, in turn, has an effect on overall CT measurement. In prior studies, the developers of Dr Scratch emphasised its strong foundation in programming concepts (Moreno-León, Robles, & Román-González, 2016, 2017). In one of their studies, Román-González, Moreno-León, and Robles (2017) referred to Dr Scratch as "computational practices" based on Brennan and Resnick's framework on CT. The category "Computational practices" comprised mainly concrete CT actions including testing and debugging (Brennan & Resnick, 2012). These are facets subordinated under the concepts of algorithmic design in this study. Hoover et al. (2016) compared CT assessment based on Dr Scratch with qualitative analysis of the Scratch projects. Their results also showed that Dr Scratch usually produces lower CT scores. In addition, the quite high correlation between Dr Scratch mastery score and programming quality as used in this study also supports this approach empirically.

5.5.2 Conceptual consideration

5.5.2.1 Limitation of the operationalisation

There is still no sufficient theoretical framework for CT. The conceptual framework about CT as used in this thesis was developed based on systematic literature reviews and major publications of distinguished experts of computer science (education) experts. This resulted in a view on CT that has a strong view on skills and actions. Other CT frameworks may focus on other aspects.

One example is International Computer and Information Literacy Study (ICILS; Fraillon et al., 2019). The study was conducted 2013 for the first time with the goal to examine students' abilities to use computers and to investigate, create and communicate participate effectively in different environments like school, workplace or at home. The second and latest cycle in 2018 continued to examine students' computer and information literacy but additionally investigated students' CT. The authors defined CT as a two-dimensional construct: *conceptualising problems* and *operational solution* with three and two aspects, respectively.

Conceptualising problems describes the ability to understand the problem before any kind of solution may be developed. It contains the following aspects:

(1) *Knowing about and understanding digital systems* refers to the ability to understand a system by observing their interactions with other systems and how their components interact with each other. A person understands a sequence of

actions and how events are dependent and is able to use visuals tools like tree diagrams or flow charts to describe a system on a conceptual level. This also contains the ability to monitor a running system and make educated assumptions why a system may not work.

(2) *Formulating and analysing problems* describes the ability to break down a complex problem into smaller and more manageable parts. It also contains the ability to specifying the characteristics of the problem so that a computational solution might be applied.

(3) *Collecting and representing relevant data* may be relevant to make effective judgments. Analysing data can help to observe the behaviour of a system and to identify patterns or characteristics that are otherwise difficult to detect. This aspect also includes the use of simulations.

The aspect of *operational solutions* describes all process associated with the solution itself. It includes the process of creating, implementing, and evaluating a computer-based system to a problem. It contains two aspects:

(1) *Planning and evaluating solutions* refers to the ability of thinking ahead and establishing parameters of a system that are needed to achieve the desired outcome. It also refers to the ability to implement and evaluate the solutions. This includes developing a test strategy and being able to make critical judgment and detect faulty solutions.

(2) *Developing algorithms, programs and interfaces* focus on logical reasoning. It does not mean to be able to use specific programming languages but being able to think in steps and rules in order to solve a problem. It describes the underlying ability to design or debug simple algorithms and to create interfaces that allows interactions between users and (digital) systems.

This short summary of the CT framework in ICILS shows some similarities but also some differences with the framework developed in this thesis. Both frameworks divide the CT process in two stages or strands, respectively. In both frameworks decomposition the problem plays a major role in the first stage while algorithmic design (or designing operational solutions) is part of the second stage. The ability of abstract thinking with neglecting unimportant information and pattern recognition is also part of both frameworks. The biggest difference, however, comes with the role of data

handling. Collecting and interpreting data are not considered as a single core aspect of CT in this thesis. Instead, it is seen as part of algorithmic design, i.e., the second stage of CT. This is different to ICILS in which collecting and representing relevant data is already part of conceptualising of the problem. Whether data handling is an independent core aspect of CT or part of another CT ability may have an impact on conclusion on the level of CT someone has.

Another instance of different framework is shown in Brennan and Resnick (2012), who developed an educational framework based mainly by analysing Scratch projects and interviewing children using Scratch. In their framework CT is rather seen as a three-dimensional construct: computational concepts (major concepts in programming such as parallelism), computational practices (typical actions programmers do such as testing and debugging), and computational perspectives (perceived relation between programmers and the technological world). This different kind of perspective on CT may result in different ways of measuring.

Contemporary theories about CT are based on Wing's comment in 2006. His view about CT popularised and shaped the general idea of CT even though she was not the first who mentioned CT. In 1980, Seymour Papert—one of the pioneers of CS education—coined the term in his work *Mindstorm: Children, Computers, and Powerful Ideas* (Papert, 1980). Indeed, the perceptions about CT have changed over time.

Papert was a computer scientist and inspired by the automation processes of the 1960s. He was quickly convinced that CS should become part of school education and developed LOGO, a programming language designed mainly for children. At this time he worked with Piaget who developed different learning theories, which can be summarised in the basic principles of constructivism (Tabesh, 2017). Inspired by Piaget's work, Papert developed his own theory of learning, which is comes from the philosophical view of constructivism (Siegler, DeLoache, & Eisenberg, 2014, pp. 134–136). Piaget's constructivism suggested that learners construct their knowledge by comparing new information with prior experience, while Papert's constructionism adds the idea that learning happens best when learners construct a product that is meaningful for them (Ackermann, 2001). The focus in both theories lies on the individual. Knowledge cannot be transmitted by a teacher. Instead, learners need to be active, it is something playful; it means exploring and tinkering around (Papert & Harel, 1991). This is the core aspect of constructionism and this is also Papert's view about CT. For

him, CT is a way “to forge ideas” (Papert, 1996, p. 116). It also means tinkering around and conducting little “experiments”. In that sense, creativity would have probably played a bigger role for CT. This might mark the biggest difference between CT, as perceived by most scholars today, and as Papert saw it.

CT as used in this thesis did not explicitly include creativity as a component for two reasons. First, it is not presented as major facet but it is mentioned only incidentally along with other minor concepts. Second, CT as used in this thesis has a focus on skills and behaviour whereas creativity is an umbrella term for different psychological concepts in the same way that intelligence is. However, if creativity would be seen as a major component, it also might change how CT is seen by teachers and learners. Overall, CT is a fuzzy concept with multifactorial views and dimensions.

Notably, CT skills are learnt and/or developed (Palts & Pedaste, 2020). Indeed, CT skills can be developed according to three larger stages, namely, defining the problem, solving the problem, and analysing the solution. To define a problem, students learn how to formulate the problem, abstraction, problem reformulation, and decomposition. To solve the problem, students learn to collect and analyse data, algorithmic design, parallelization and iteration, and automation. To analyse the solution, students learn to generalize, test, and evaluate. Through these three stages, the students learn CT skills for problem-solving from start to finish.

5.5.2.2 Computational thinking itself

The whole concept of CT is not without any critics. Hemmendinger (2010) proposed four major critical points in his plea for modest concerning CT. His first point is that CT is nothing new. Philosophers have been thinking about thinking since philosophy exists. The same is true for CT relevant concepts like abstraction, which is traditionally associated with mathematics or psychology. So discussion is just old wine in new skins. Although Hemmendinger’s is correct that this kind of thinking not new, to put all these different skills and concepts under one umbrella term is a new approach and provides some new opportunities. The idea of computational thinking is to have a specific look at problems and it emphasizes the relationship between different skills like to decompose a problem to be able to find algorithm solutions. Results of this study revealed that this kind of thinking does not come naturally and particular training is needed.

Second, CT would be just a way of thinking for a specific domain and so it is not special. Mathematicians use mathematical thinking, historians use historical thinking,

chemists use chemical thinking and so on. Just because these ways of thinking are useful in their kind of domains does not necessarily mean they are useful in any other domain. Why should not be the same true for CT as a domain specific way of thinking for CS? This point, however, is only partially true. Digital devices are everywhere and become more important in basically every part of our everyday life and so the way of thinking computationally becomes more important. Different to other styles of thinking, CT becomes more important for more people. The participants of this study, for examples, were not CS students but preservice teacher students and CT was included in their work course. This underpins how CT is considered as a skill for everyone and not just people with a major in CS.

Third, advocating CT sometimes has an "imperialistic flavour" (Hemmendinger, 2010, p. 4). Some people tend to perceive the world through theoretical glasses. Hemmendinger (2010) quotes an interview when a computer scientist analyses a video about a science lessons. The computer scientist makes analogies referring to CT and compares the science lesson with CT relevant aspects. Hemmendinger further argued that perception of this person might have been biased just because the person was a computer scientist. An artists or a chemist might had been seen the video in a different way. So it is questionable whether there is indeed so much CT everywhere or whether some people just have this kind of perception. The contradiction of this point would be that CT is not meant to see all problems the same way but to offer problem-solver *one* way to approach problems. CT is presented as a problem-solving approach, which based on the idea to break down a problem into smaller pieces (decomposing the problem) and apply an algorithmic procedure. Not every problem can be break down into smaller pieces and not all problems can be solved with an algorithm. One participant who discussed the relevance of CT in law in the workshop on the scope and nature of CT also argued that the application of CT has its limits when the problems are highly context dependent (NRC, 2010, pp. 38–39). The same problem or case in law may have different outcomes because of the unique and different circumstances. Even though, law tries to follow a logical and objective procedure such as CT does, it has also subjective components. These cannot be considered in a computational thinking way of problem solving. The same is true for questions of ethics or philosophy. The (ultimate) question of life, the universe, and everything should not be answered by a computer. However, CT is a strong tool to solve problem, which can be deconstructed and algorithmic solutions are possible. Participants of this study were observed how much and how long CT-related behaviour they show when solving an ill-structured programming problem.

That does not mean that they needed to solve this kind of problem always this way but for this particular problem CT was suitable.

Fourth, CT might be not the right way to think about problems and it is rather “dumbing down” our thinking. Humans’ way of dealing with problems is simply different from computers. Whereas computers nearly always use step-by-step procedures, humans’ approach is more holistic and sometimes it feels artificial to specify control flow and explicit iterations or conditions. To teach this approach might not be beneficial, according to Hemmendinger (2010). It is, however, a misunderstanding to consider CT as replacement of other style of thinking or problem-solving. Creativity and “out of the box thinking” are (still) something unique of humans’ mind that cannot be emulated by machines (Kaufman & Plucker, 2011). Creativity plays a role in solving ill-structured problems. CT is considered as useful for those kinds of problems and so the overall perception of CT should be more like a useful addition of approaching problems instead of seeing as dumbing down cognitive processes. It is more like as Leo Cherne noted in the late 70’s: “The computer is incredibly fast, accurate, and stupid. Man is unbelievable slow, inaccurate, and brilliant. The marriage of the two is a force beyond calculation.” (Shoemate, 2008).

5.6 Future work

So far it is not clear whether pairs did not show much decomposition and abstraction during the programming task because they were not trained in CT, or whether the instruments used were not sufficiently sensitive. To analyse this, future studies could involve preparing students with a workshop or course about CT based on the practical implication suggested in this thesis. Results might differ if participants were instructed about different methods to improve CT-associated skills such as task decomposition (as suggested, for example, in the model of F. Lee and Anderson’s, 2001) and trained in recognising patterns in different (sub)problems as well as (sub)solutions. Doing this would also help to clarify what impact CT has on programming. It is also important to note that according to Lewis and Shah (2015), pair programming interactions in a sixth-grade CS enrichment program designed to promote equity reveals instances of inequity. They measured inequity through the documentation of students’ questions, commands, and total talk within four pairs. Data analysis indicates that less equitable pairs wanted to finish their tasks quickly, thereby leading to patterns of marginalization and

domination. However, in more equitable pairs, the emphasis on speed was not documented.

In this study, Scratch was used to assess CT but also programming quality, which means that the measures are dependent to some extent to each other. Results showed that the relationship between CT and programming was also dependent on how close the CT measure was to programming elements. The CT measure based on Scratch had stronger relationships to programming quality than did the CT measure independent from Scratch. Although studies with similar goals had a similar approach (see, e.g., Kazimoglu, Kiernan, Bacon, & Mackinnon, 2012), it might cause problems if Scratch is used to measure both concepts. Visual programming environments such as Scratch are usually used to introduce CT or programming concepts to people who have no knowledge about programming, as was the case in this study. In future, researchers should take this into account and analyse how CT is applied when experienced programmers solve a programming task in a programming language such as Java or C++. The way programmers approach problems develops over time as they gain more knowledge (Teague & Lister, 2014). It is possible that the level of CT for experienced programmers differs from the level of novices, which might mediate the relationship between both concepts.

Many analyses of this study were based on paired values which has different problematic theoretical and statistical implications. Even though it is not unusual to see CT as a concept which occurs in a collaborative setting (see, e.g., Falloon, 2016; Wu, Hu, Ruis, & Wang, 2019), this view is fairly uncommon for other concepts like programming quality and nonverbal intelligence. Some measures in this study were also available as individual scores which enabled less problematic analyses. However, future research should analyse all concepts also individually to have a more reliable view on the relationship of these concepts.

It is also worth mentioning that the CTBS was based on event-sampling method. Event sampling is usually used when not much is known of the construct (Bakeman & Quera, 2011, p. 27). Because of the novelty of the coding scheme, this was the case of this study. However, the results of this study provided some information about duration, occurrence and relations between the different CT skills. In future studies, time-sampling methods should be considered as well. When using time-sampling a specific time slot is divided into intervals (e.g., 5, 10, or 20 seconds) and the most dominant

behaviour is coded. This might provide some more details insights in the duration and occurrence of CT relevant behaviour.

The results also revealed that CT overlaps conceptually as well as empirically with nonverbal intelligence (see also Boom, Bower, Arguel, Siemon & Scholkmann, 2018). This could be because the CT associated skills (task decomposition, abstract thinking, and algorithmic design) especially play a role for well-structured problems. Jonassen (1997) presented decomposing as a typical method to solve well-structured problems. Algorithms are useful when all problem states are clear and there are no ambiguous possibilities or several solutions are possible. Therefore, algorithms are useful especially for well-structured problems. Moreover, the solution to well-structured problems are usually associated with intelligence, especially with abstract thinking (Wenke & Frensch, 2003). High correlations between problem-solving and intelligence are found only if the problems are well structured. Interestingly, CT is often not associated with well-structured problems. Instead, CT is promoted by many scholars as an ability to deal with complexity and open-ended or ill-structured problems (see, e.g., Barr & Stephenson, 2011; Shute, Sun, & Asbell-Clarke, 2017). However, solutions to ill-structured problems are not associated with intelligence (Wenke & Frensch, 2003). Creativity is needed to deal with ill-structured problems because those problems have unknown elements, vaguely defined goals, different evaluation criteria, and a level of uncertainty (Jonassen, 1997; Kaufman & Plucker, 2011). The reason why CT is associated with ill-structured problems might be reminiscent of Papert's original view of CT in which he associated CT more with tinkering around, exploring, and creativity as most scholars do today. Overall, this might lead to the conclusion that CT not only shares some properties with intelligence due to the associated skills but also with creativity due to its purpose. To further investigate these possibilities, more studies are needed.

5.7 Conclusion

CT is promoted as the literacy of the 21st century and is already implemented in various curricula all over the world. Some refer to CT even as the foundation programming and CS (Lu & Fletcher, 2009). Thus, the goal of this study was to analyse the role of CT for programming. Students with no prior significant knowledge about CT or programming were working in pairs on a programming task in Scratch. Results revealed that not all

facets of CT were equally apparent during the programming task. Any behaviour indicating decomposition and especially any behaviour indicating abstraction was barely found during the programming task. Instead, participants were spending most of the time with designing solutions without thinking much about the problems itself. There are many different frameworks how to implement CT in education even in curricula without any technical background (see, e.g., Perković, Settle, Hwang, & Jones, 2010). However, there is the question where the biggest challenges and difficulties may occur when implementing CT. Based on the results of this study, it is clear that the most important part for any educational framework should lie on abstract thinking and the ability to decompose a problem. Nonetheless, future studies need to evaluate how large the impact of abstract thinking and the ability to decompose a problem on programming actually is. It is possible that the used instrument might not have been sensitive enough to capture all facets of abstract thinking.

6 REFERENCES

ACARA (2012). The shape of the Australian curriculum: Technologies. Retrieved from http://www.acara.edu.au/verve/_resources/Shape_of_the_Australian_Curriculum_-_Technologies_-_August_2012.pdf

Aberson, C. L. (2019). *Applied Power Analysis for the Behavioral Sciences* (2nd ed.). Routledge.

Ackermann, E. (2001). Piaget's constructivism, Papert's constructionism: What's the difference? *Future of learning group publication*, *4*(3), 438–448.

Aho, A. V. (2012). Computation and computational thinking. *The Computer Journal*, *55*(7), 832–835. https://doi.org/10.1093/comjnl/bxs074

Aho, A. V., & Ullman, J. D. (2000). *Foundations of computer science* (6th ed.). *Principles of computer science series*. New York, NY: Computer Science Press.

Aiken, J. M., Caballero, M. D., Douglas, S. S., Burk, J. B., Scanlon, E. M., Thoms, B. D., & Schatz, M. F. (2012). Understanding student computational thinking with computational modeling. In *AIP Conference Proceedings, Physics Education Research Conference* (pp. 46–49). AIP. https://doi.org/10.1063/1.4789648

Alaoutinen, S. (2012). Evaluating the effect of learning style and student background on self-assessment accuracy. *Computer Science Education*, *22*(2), 175–198. https://doi.org/10.1080/08993408.2012.692924

Anderson, J. R. (2015). *Cognitive psychology and its implications* (8th). New York, NY: Worth Publishers.

Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, C., & Qin, Y. (2004). An integrated theory of the mind. *Psychological Review*, 111(4), 1036–1060.

Angeli, C., Voogt, J., Fluck, A., Webb, M., Cox, M., Malyn-Smith, J., & Zagami, J. (2016). A K-6 Computational thinking curriculum framework: Implications for teacher knowledge. *Educational Technology & Society*, *9*(3), 47–57.

Araujo, A. L. S. O., Santos, J. S., Andrade, W. L., Guerrero, D. D. S., & Dagiene, V. (2017). Exploring computational thinking assessment in introductory programming courses. In I. F. i. E. Conference (Ed.), *FIE 2017: Frontiers in Education, October 18-21, 2017, Indianapolis, Indiana, USA : 2017 conference proceedings* (pp. 1–9). Piscataway, NJ: IEEE. https://doi.org/10.1109/FIE.2017.8190652

Arcavi, A. (1994). Symbol sense: Informal sense-making in formal mathematics. *For the Learning of Mathematics*, 14(3), 24–35.

Armoni, M., Meerbaum-Salant, O., & Ben-Ari, M. (2015). From Scratch to "real" programming. *ACM Transactions on Computing Education*, *14*(4), 1–15. https://doi.org/10.1145/2677087

Atmatzidou, S., & Demetriadis, S. (2016). Advancing students' computational thinking skills through educational robotics: A study on age and gender relevant differences. *Robotics and Autonomous Systems*, *75*, 661–670. https://doi.org/10.1016/j.robot.2015.10.008

Autor, D. H., Levy, F., & Murnane, R. J. (2003). The skill content of recent technological change: An empirical exploration. *The Quarterly Journal of Economics*, *118*, 1279–1333. https://doi.org/10.1162/003355303322552801

Bakeman, R., & Quera, V. (2011). *Sequential analysis and observational methods for the behavioral sciences*. Cambridge: Cambridge University Press. https://doi.org/10.1017/CBO9781139017343

Bakeman, R., & Quera, V. (2012). Behavioral observation. In H. Cooper, P. M. Camic, D. L. Long, A. T. Panter, D. Rindskopf, & K. J. Sher (Eds.), *APA handbook of research methods in psychology, Vol 1: Foundations, planning, measures, and psychometrics* (pp. 207–225). Washington, DC: American Psychological Association. https://doi.org/10.1037/13619-013

Banks, S. H., & Franzen, M. D. (2010). Concurrent validity of the TONI-3. *Journal of Psychoeducational Assessment*, *28*(1), 70–79. https://doi.org/10.1177/0734282909336935

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

Barsalou, L. W. (1994). Flexibility, structure, and linguistic vagary in concepts: Manifestations of a compositional system of perceptual symbols. In A. F. Collins (Ed.), *Theories of memory.* Hove, UK: L. Erlbaum Associates.

Barsalou, L. W. (2003). Abstraction in perceptual symbol systems. *Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences*, *358*(1435), 1177–1187. https://doi.org/10.1098/rstb.2003.1319

Bartels, R. (1982). The Rank Version of von Neumann's Ratio Test for Randomness. *Journal of the American Statistical Association*, *77*(377), 40–46. https://doi.org/10.1080/01621459.1982.10477764

Basogain, X., Olabe, M. Á., Olabe, J. C., & Rico, M. J. (2018). Computational thinking in pre-university blended learning classrooms. *Computers in Human Behavior*, *80*, 412–419. https://doi.org/10.1016/j.chb.2017.04.058

Basu, S. (2019). Using Rubrics Integrating Design and Coding to Assess Middle School Students' Open-Ended Block-Based Programming Projects. *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 1211–1217. https://doi.org/10.1145/3287324.3287412

Bell, T., & Vahrenhold, J. (2018). Cs unplugged: How is it used, and does it work? In H.-J. Böckenhauer, D. Komm, & W. Unger (Eds.), *Lecture notes in computer science Theoretical computer science and general issues: Vol. 11011. Adventures between lower bounds and higher altitudes: Essays dedicated to Juraj Hromkovič on*

the occasion of his 60th birthday (pp. 497–521). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-98355-4_29

Bellettini, C., Lonati, V., Malchiodi, D., Monga, M., Morpurgo, A., & Torelli, M. (2015). How challenging are Bebras tasks? In V. Dagienė, C. Schulte, & T. Jevsikova (Chairs), *the 2015 ACM Conference,* Vilnius, Lithuania.

Bocconi, S., Chioccariello, A., Dettori, G., Ferrari, A., Engelhardt, K., Kampylis, P., & Punie, Y. (2016). *Developing computational thinking in compulsory education - Implications for policy and practice: JRC Science for Policy Report*: Publications Office of the European Union.

Boe, B., Hill, C., Len, M., Dreschler, G., Conrad, P., & Franklin, D. (2013). Hairball: Lint-inspired static analysis of scratch projects. In R. McCauley (Ed.), *Sigcse'13: Proceedings of the 44th ACM Technical Symposium on Computer Science Education; March 6 - 9, 2013, Denver, CO* (pp. 215–220). New York, NY: ACM. https://doi.org/10.1145/2445196.2445265

Booch, G. (1994). *Object oriented design with applications* (2nd ed.). *Benjamin/Cummings series in Ada and software engineering*. Redwood City, CA: Benjamin/Cummings.

Boom, K.-D., Bower, M., Arguel, A., Siemon, J., & Scholkmann, A. (2018). Relationship between computational thinking and a measure of intelligence as a general problem-solving ability. In I. Polycarpou, J. C. Read, P. Andreou, & M. Armoni (Eds.), *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education - ITiCSE 2018* (pp. 206-211 TS-CrossRef). ACM Press. https://doi.org/10.1145/3197091.3197104

Boudreau, T., Tulach, J., & Wielenga, G. (2007). *Rich client programming: Plugging into the NetBeans platform*. *Safari Books Online*. Upper Saddle River, N.J.: Prentice Hall. Retrieved from http://proquest.tech.safaribooksonline.de/9780132354806

Bower, G. H., Black, J. B., & Turner, T. J. (1979). Scripts in memory for text. *Cognitive Psychology*, *11*(2), 177–220. https://doi.org/10.1016/0010-0285(79)90009-4

Brackmann, C. P., Román-González, M., Robles, G., Moreno-León, J., Casali, A., & Barone, D. (2017). Development of computational thinking skills through unplugged activities in primary school. In E. Barendsen & P. Hubwieser (Eds.), *Proceedings of the 12th Workshop on Primary and Secondary Computing Education - WiPSCE '17* (pp. 65–72). New York, NY: ACM Press. https://doi.org/10.1145/3137065.3137069

Brancaccio, A., Marchisio, M., Palumbo, C., Pardini, C., Patrucco, A., & Zich, R. (2015). Problem posing and solving: Strategic Italian key action to enhance teaching and learning mathematics and informatics in the high school. In *2015 IEEE 39th Annual Computer Software and Applications Conference (COMPSAC)* (pp. 845–850). IEEE / Institute of Electrical and Electronics Engineers Incorporated. https://doi.org
/10.1109/COMPSAC.2015.126

Bransford, J., & Stein, B. S. (1993). *The ideal problem solver: A guide for improving thinking, learning, and creativity* (2nd. ed.). New York, NY: Freeman.

Brennan, K., & Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. In *Proceedings of the 2012 annual meeting of the American Educational Research Association, Vancouver, Canada*.

Brodnik, A., & Lewin, C. (Eds.). (2015). *Ifip TC3 Working Conference "A New Culture of Learning: Computing and next Generations"*. Vilnius, Lithuania. https://doi.org/10.13140/RG.2.1.2855.9206

Brooks, P. H. (1981). The Abstraction of prototypes as an aspect of intellectual development. *Intelligence*, *5*(3), 279–290. https://doi.org/10.1016/S0160-2896(81)90000-3

Brown, L., Sherbeernou, R. J., & Johnson, S. K. (1997). *Test of nonverbal intelligence-3*. Austin, TX: PRO-ED.

Bruce, C., & McMahon, C. (2002). *Contemporary developments in teaching and learning introductory programming: Towards a research proposal*. QLD, Brisbane: Queensland University of Technology.

Bruder, R. (2000). *Eine akzentuierte Aufgabenauswahl und Vermitteln heuristischer Erfahrung –Wege zu einem anspruchsvollen Mathematikunterricht für alle*.

Bull, R., & Espy, K. A. (2007). Working memory, executive functioning, and children's mathematics. In S. J. Pickering (Ed.), *Working memory and education* (pp. 93–123). Amsterdam [etc.]: Elsevier. https://doi.org/10.1016/B978-012554465-8/50006-5

Burgoon, E. M., Henderson, M. D., & Markman, A. B. (2013). There are many ways to see the forest for the trees: A tour guide for abstraction. *Perspectives on Psychological Science : a Journal of the Association for Psychological Science*, *8*, 501–520. https://doi.org/10.1177/1745691613497964

Burning Glass (2014). STEM | Real-time insight into the market for entry-level STEM jobs. *Burning Glass Technologies*. Retrieved from http://burning-glass.com/research/stem/

Butcher, P. (2009). *Debug It!: Find, repair, & prevent bugs in your code*. *The pragmatic programmers*. USA: Pragmatic Bookshelf.

Campbell, R. L., & Bickhard, M. H. (Eds.). (1986). *Human development: Contributions to human development: vol. 16*. *Knowing levels and developmental stages*. Basel, München u.a.: Karger. https://doi.org/10.1159/issn.0301-4193

Card, S. K., Moran, T. P., & Newell, A. (1983). *The psychology of human-computer interaction: Xa-GB*. Hillsdale, NJ: Erlbaum.

Carlson, R. A., & Dulany, D. E. (1985). Conscious attention and abstraction in concept learning. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *11*(1), 45–58. https://doi.org/10.1037/0278-7393.11.1.45

Carroll, J. B. (1993). *Human cognitive abilities: A survey of factor-analytic studies*. Cambridge: Cambridge University Press. Retrieved from https://doi.org/10.1017/CBO9780511571312

Cattell, R. B. (1963). Theory of fluid and crystallized intelligence: A critical experiment. *Journal of Educational Psychology*, *54*(1), 1–22. https://doi.org/10.1037/h0046743

Cernochova, M., Dorling, M., & Williams, L. (2015). Developing computational thinking skills through the literacy from Scratch project, an international collaboration. In A. Brodnik & C. Lewin (Eds.), *IFIP TC3 Working Conference "A New Culture of Learning: Computing and next Generations"* (pp. 40–50). Vilnius, Lithuania: Vilnius University.

Chen, C., Haduong, P., Brennan, K., Sonnert, G., & Sadler, P. (2019). The effects of first programming language on college students' computing attitude and achievement: a comparison of graphical and textual languages. *Computer Science Education*, *29*(1), 23–48. https://doi.org/10.1080/08993408.2018.1547564

Ching, Y.‐H., Hsu, Y.‐C., & Baldwin, S. (2018). Developing computational thinking with educational technologies for young learners. *TechTrends*, *62*(6), 563–573. https://doi.org/10.1007/s11528-018-0292-7

Chorney, J. M., McMurtry, C. M., Chambers, C. T., & Bakeman, R. (2015). Developing and modifying behavioral coding schemes in pediatric psychology: A practical guide. *Journal of Pediatric Psychology*, *40*(1), 154–164. https://doi.org/10.1093/jpepsy /jsu099

Cohen, D., Lindvall, M., & Costa, P. (2004). An Introduction to agile methods. In M. V. Zelkowitz (Ed.), *Advances in computers: Vol. 62. Advances in software engineering* (Vol. 62, pp. 1–66). San Diego, CA: Academic Press. https://doi.org/10.1016/S0065-2458(03)62001-2

Cohen, J. (1960). A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, *20*(1), 37–46. https://doi.org/10.1177 /001316446002000104

Cohen, J. (1988). *Statistical power analysis for the behavioral sciences* (2nd ed.). Hillsdale, NJ: L. Erlbaum Associates.

Colburn, T., & Shute, G. (2007). Abstraction in computer science. *Minds and Machines*, *17*(2), 169–184. https://doi.org/10.1007/s11023-007-9061-7

Common Core State Standards Initiative (CCSSI). (2010). Common core state standards for mathematics. Retrieved from http://www.corestandards.org/Math/Practice.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2014). *Introduction to algorithms* (3rd ed.). Cambridge, MA, London: MIT Press.

Corradini, I., Lodi, M., & Nardelli, E. (2017). Conceptions and misconceptions about computational thinking among Italian primary school teachers. In J. Tenenberg, D. Chinn, J. Sheard, & L. Malmi (Eds.), *Proceedings of the 2017 ACM Conference on International Computing Education Research - ICER '17* (pp. 136–144). New York, New York, USA: ACM Press. https://doi.org/10.1145/3105726.3106194

Cronbach, L. J., & Meehl, P. E. (1955). Construct validity in psychological tests. *Psychological Bulletin*, *52*, 281–302. https://doi.org/10.1037/h0040957

CSTA (2011). Computational thinking teacher resources. *Csta.acm.org*, 1–69. Retrieved from https://csta.acm.org/Curriculum/sub/CompThinking.html

Curzon, P., McOwan, P. W., Plant, N., & Meagher, L. R. (2014). Introducing teachers to computational thinking using unplugged storytelling. In C. Schulte, M. E. Caspersen, & J. Gal-Ezer (Eds.), *Proceedings of the 9th Workshop in Primary and Secondary Computing Education on - WiPSCE '14* (pp. 89–92). New York, NY: ACM Press. https://doi.org/10.1145/2670757.2670767

Dagienė, V. (2006). Information technology contests: Introduction to computer science in an attractive way. *Informatics in Education*, *5*(1), 37–46. Retrieved from http://dl.acm.org/citation.cfm?id=1149707.1149711

Dagienė, V., & Futschek, G. (2008). Bebras International Contest on Informatics and Computer Literacy: Criteria for good tasks. In R. T. Mittermeir & M. M. Sysło (Eds.), *Informatics Education - Supporting Computational Thinking: Third International Conference on Informatics in Secondary Schools - Evolution and Perspectives, ISSEP 2008 Torun Poland, July 1-4, 2008 Proceedings* (pp. 19–30). Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-69924-8_2

Dagienė, V., & Sentance, S. (2016). It's computational thinking! Bebras Tasks in the curriculum. In A. Brodnik & F. Tort (Eds.), *Lecture notes in computer science. informatics in schools: 9th International Conference on Informatics in Schools: Situation, Evolution, and Perspectives, Proceedings* (Vol. 9973, pp. 28–39). Cham: Springer Verlag. https://doi.org/10.1007/978-3-319-46747-4_3

Dagienė, V., & Stupuriene, G. (2015). Informatics education based on solving attractive tasks through a contest. *KEYCIT 2014 - Key Competencies in Informatics and ICT*. (7), 51–62.

Dagienė, V., & Stupuriene, G. (2016). Bebras - A sustainable community building model for the concept based learning of informatics and computational thinking. *Informatics in Education*, *15*(1), 25–44. https://doi.org/10.15388/infedu.2016.02

Dale, N., & Walker, H. M. (1996). *Abstract data types: Specifications, implementations, and applications*. Lexington, MA: Heath.

Dale, N., Weems, C., & Headington, M. R. (2004). *Programming and problem solving with Java*. Princeton, N.J.: Recording for the Blind & Dyslexic.

Danner, D., Hagemann, D., Schankin, A., Hager, M., & Funke, J. (2011). Beyond IQ: A latent state-trait analysis of general intelligence, dynamic decision making, and implicit learning. *Intelligence*, *39*(5), 323-334.

Davies, S. (2008). The effects of emphasizing computational thinking in an introductory programming course. In *2008 IEEE Frontiers in Education Conference (FIE),* Saratoga Springs, NY, USA.

Dawson, P. (2017). Assessment rubrics: Towards clearer and more replicable design, research and practice. *Assessment & Evaluation in Higher Education*, *42*, 347–360. https://doi.org/10.1080/02602938.2015.1111294

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

Denning, P. J. (2009). The profession of IT - Beyond computational thinking. *Communications of the ACM*, *52*(6), 28–30. https://doi.org/10.1145/1516046.1516054

Denning, P. J. (2017). Remaining trouble spots with computational thinking. *Communications of the ACM*, *60*(6), 33–39.

Denning, P. J., Comer, D. E., Gries, D., Mulder, M. C., Tucker, A., Turner, A. J., & Young, P. R. (1989). Computing as a discipline. *Communications of the ACM*, *32*(1), 9–23. https://doi.org/10.1145/63238.63239

Dolgopolovas, V., Jevsikova, T., Savulionienė, L., & Dagienė, V. (2015). On evaluation of computational thinking of software engineering novice students. In A. Brodnik & C. Lewin (Eds.), *IFIP TC3 Working Conference "A New Culture of Learning: Computing and next Generations".* Vilnius, Lithuania: Vilnius University.

Dörner, D., Kreuzig, H. W., Reither, F., & Stäudel, T. (Eds.). (1983). *Lohhausen: Vom Umgang mit Unbestimmtheit und Komplexität*. Bern: Huber.

Eguíluz, A., Garaizar, P., & Guenaga, M. (2018). An evaluation of open digital gaming platforms for developing computational thinking skills. In D. Cvetković (Ed.), *Simulation and Gaming.* InTech. https://doi.org/10.5772/intechopen.71339

Fagerlund, J., Häkkinen, P., Vesisenaho, M., & Viiri, J. (2020). Assessing 4th Grade Students' Computational Thinking through Scratch Programming Projects. *Informatics in Education*, *19*(4), 611–640. https://doi.org/10.15388/infedu.2020.27

Falkner, K. (2016). SCIS | Computational thinking as the new literacy. Retrieved from http://www2.curriculum.edu.au/scis/connections/issue_95/articles/computational_thinking_as_the_new_literacy.html

Falloon, G. (2016). An analysis of young students' thinking when completing basic coding tasks using Scratch Jnr. On the iPad. *Journal of Computer Assisted Learning.* Advance online publication. https://doi.org/10.1111/jcal.12155

Faul, F., Erdfelder, E., Lang, A.‑G., & Buchner, A. (2007). G*Power 3: A flexible statistical power analysis program for the social, behavioral, and biomedical sciences. *Behavior Research Methods*, *39*(2), 175–191. https://doi.org/10.3758/BF03193146

Fetzer, J. H. (1998). People are not computers: (Most) thought processes are not computational procedures. *Journal of Experimental & Theoretical Artificial Intelligence*, *10*, 371–391. https://doi.org/10.1080/095281398146653

Fischer, K. W., & Kenny, S. L. (1986). The environmental conditions for discontinuities in the development of abstractions. In R. A. Mines & K. S. Kitchener (Eds.), *Praeger*

special studies Praeger scientific. Adult cognitive development: Methods and models (pp. 57–75). New York, NY: Praeger.

Flanagan, D. P., & Dixon, S. G. (2014). The Cattell-Horn-Carroll theory of cognitive abilities. In C. R. Reynolds, K. J. Vannest, & E. Fletcher-Janzen (Eds.), *Encyclopedia of special education: A reference for the education of children, adolescents, and adults with disabilities and other exceptional individuals* (Vol. 121, p. 219). Hoboken, NJ: Wiley. https://doi.org/10.1002/9781118660584.ese0431

Fraillon, J., Ainley, J., Schulz, W., Duckworth, D., & Friedman, T. (2019). *EA International Computer and Information Literacy Study 2018: Assessment framework*. International Association for the Evaluation of Educational Achievement (IEA).

Frensch, P. A., & Funke, J. (Eds.). (1995). *Complex problem solving*. The European perspective. Hillsdale, NJ: Lawrence Erlbaum.

Gabora, L., & Russon, A. (2011). The evolution of intelligence. In R. J. Sternberg & S. B. Kaufman (Eds.), *The Cambridge handbook of intelligence* (pp. 328–350). Cambridge: Cambridge University Press. https://doi.org/10.1017 /CBO9780511977244.018

García-Peñalvo, F. J., Reimann, D., & Maday, C. (2018). Introducing coding and computational thinking in the schools: The TACCLE 3 – Coding Project experience. In M. S. Khine (Ed.), *Computational thinking in the STEM disciplines: Foundations and research highlights* (Vol. 55, pp. 213–226). Cham: Springer. https://doi.org/10.1007/978-3-319-93566-9_11

Gardner, H. (1983). *Frames of mind: The theory of multiple intelligences*. New York, NY: Basic Books.

Ge, X., & Land, S. M. (2003). Scaffolding students' problem-solving processes in an ill-structured task using question prompts and peer interactions. *Educational Technology Research and Development*, *51*(1), 21–38. https://doi.org/10.1007/BF02504515

Gick, M. L. (1986). Problem-solving strategies. *Educational Psychologist*, *21*(1-2), 99–120. https://doi.org/10.1080/00461520.1986.9653026

Gilhooly, K. J. (2012). *Human and machine problem solving*: Dordrecht, The Netherlands: Springer.

Gonzalez, R., & Griffin, D. (2012). Dyadic data analysis. In H. Cooper, P. M. Camic, D. L. Long, A. T. Panter, D. Rindskopf, & K. J. Sher (Eds.), *APA handbook of research methods in psychology, Vol 3: Data analysis and research publication* (pp. 439–450). Washington DC: American Psychological Association. https://doi.org/10.1037/13621-022

Gottfredson, L. S. (1997). Mainstream science on intelligence: An editorial with 52 signatories, history, and bibliography. *Intelligence*, *24*(1), 13–23. https://doi.org /10.1016/S0160-2896(97)90011-8

Gretter, S., & Yadav, A. (2016). Computational thinking and media & information literacy: An integrated approach to teaching twenty-first century skills. *TechTrends*, *60*, 510–516. https://doi.org/10.1007/s11528-016-0098-4

Greiff, S., Wustenberg, S., Molnar, G., Fischer, A., Funke, J., & Csapo, B. (2013). Complex problem solving in educational settings ! something beyond g: Concept, assessment, measurement invariance, and construct validity. J*ournal of Educational Psychology, 105*(2), 364-379.

Grover, S. (2017). Assessing algorithmic and computational thinking in K-12: Lessons from a middle school classroom. In P. J. Rich & C. B. Hodges (Eds.), *Emerging Research, Practice, and Policy on Computational Thinking* (Vol. 31, pp. 269–288). Cham: Springer. https://doi.org/10.1007/978-3-319-52691-1_17

Grover, S., Jackiw, N., & Lundh, P. (2019). Concepts before coding: non-programming interactives to advance learning of introductory programming concepts in middle school. *Computer Science Education*, *29*(2-3), 106–135. https://doi.org/10.1080/08993408.2019.1568955

Grover, S., & Pea, R. (2013). Computational thinking in K–12: A review of the state of the field.

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

Guzdial, M., & Wing, J. M. (2011). A definition of computational thinking from Jeannette Wing. *Computing Education Blog*. Retrieved from https://computinged.wordpress.com/2011/03/22/a-definition-of-computational-thinking-from-jeanette-wing/

Haberman, B. (2004). High-school students' attitudes regarding procedural abstraction. *Education and Information Technologies*, *9*, 131–145. https://doi.org/10.1023/B:EAIT.0000027926.99053.6f

Hadamard, J. (1945). *The psychology of invention in the mathematical field*. New York, NY: Dover Publications. Retrieve from: http://worrydream.com/refs/Hadamard%20-%20The%20psychology%20of%20invention%20in%20the%20mathematical%20field.pdf

Haier, R. J. (2011). Biological basis of intelligence. In R. J. Sternberg & S. B. Kaufman (Eds.), *The Cambridge handbook of intelligence* (pp. 351–368). Cambridge, UK: Cambridge University Press. https://doi.org/10.1017/CBO9780511977244.019

Halstead, M. H. (1977). *Elements of software science*. *Operating and programming systems series: Vol. 2*. New York, NY: Elsevier.

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

Hemmendinger, D. (2010). A plea for modesty. *ACM Inroads*, *1*(2), 4. https://doi.org/10.1145/1805724.1805725

Hermans, F. & Aivaloglou, E. (2017). To Scratch or not to Scratch? A controlled experiment comparing plugged first and unplugged first programming lessons. In Proceedings of WiPSCE ’17, Nijmegen, Netherlands, November 8–10, 2017. doi: 10.1145/3137065.3137072

Hintzman, D. L. (1986). "Schema abstraction" in a multiple-trace memory model. *Psychological Review*, *93*(4), 411–428. https://doi.org/10.1037/0033-295X.93.4.411

Hoover, A. K., Barnes, J., Fatehi, B., Moreno-León, J., Puttick, G., Tucker-Raymond, E., & Harteveld, C. (2016). Assessing computational thinking in students' game designs. In A. Cox, Z. O. Toups, R. L. Mandryk, P. Cairns, V. vanden Abeele, & D. Johnson (Eds.), *Proceedings of the 2016 Annual Symposium on Computer-Human Interaction in Play Companion Extended Abstracts - CHI PLAY Companion '16* (pp. 173–179). New York, New York, USA: ACM Press. https://doi.org/10.1145/2968120.2987750

Horn, J. L., & Cattell, R. B. (1967). Age differences in fluid and crystallized intelligence. *Acta Psychologica*, *26*, 107–129. https://doi.org/10.1016/0001-6918(67)90011-X

Hu, C. (2011). Computational thinking – What it might mean and what we might do about it. In G. Rößling, T. Naps, & C. Spannagel (Eds.), *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education - ITiCSE '11* (p. 223). New York, New York, NY: ACM Press. https://doi.org/10.1145
/1999747.1999811

Humphreys, L. G. (1979). The construct of general intelligence. *Intelligence*, *3*(2), 105–120. https://doi.org/10.1016/0160-2896(79)90009-6

IFTF (2017). *The next area of human machine partnership: Emerging technologies' impact on society & work in 2013*. Palo Alto, CA: Institute for the Future.

ISTE and CSTA (2011). Operational definition of computational thinking for k–12 education. Retrieved from https://csta.acm.org/Curriculum/sub/CurrFiles
/CompThinkingFlyer.pdf

Ivanouw, J. (2007). Sequence analysis as a method for psychological research. *Nordic Psychology*, *59*, 251–267. https://doi.org/10.1027/1901-2276.59.3.251

Jensen, A. R. (2002). Psychomnetric g: Definition and substantiation. In R. J. Sternberg & E. L. Grigorenko (Eds.), *The general factor of intelligence: How general is it?*

Jonassen, D. H. (1997). Instructional design models for well-structured and III-structured problem-solving learning outcomes. *Educational Technology Research and Development*, *45*(1), 65–94. https://doi.org/10.1007/BF02299613

Jonassen, D. H. (2000). Toward a design theory of problem solving. *Educational Technology Research and Development*, *48*(4), 63–85. https://doi.org/10.1007
/BF02300500

Jonsson, A., & Svingby, G. (2007). The use of scoring rubrics: Reliability, validity and educational consequences. *Educational Research Review*, *2*, 130–144. https://doi.org/10.1016/j.edurev.2007.05.002

Kalelioğlu, F., Gülbahar, Y., & Kukul, V. (2016). A framework for computational thinking based on a systematic research review. *Baltic Journal of Modern Computing*, *4*, 583–596.

Kaufman, J. C., & Plucker, J. A. (2011). Intelligence and creativity. In R. J. Sternberg & S. B. Kaufman (Eds.), *The Cambridge handbook of intelligence* (pp. 771–783). Cambridge: Cambridge University Press. https://doi.org/10.1017/CBO9780511977244.039

Kazimoglu, C., Kiernan, M., Bacon, L., & Mackinnon, L. (2012). A serious game for developing computational thinking and learning introductory computer programming. *Procedia - Social and Behavioral Sciences*, *47*, 1991–1999. https://doi.org/10.1016/j.sbspro.2012.06.938

Kelly, M. E., Loughrey, D., Lawlor, B. A., Robertson, I. H., Walsh, C., & Brennan, S. (2014). The impact of cognitive training and mental stimulation on cognitive and everyday functioning of healthy older adults: A systematic review and meta-analysis. *Ageing Research Reviews*, *15*, 28–43. https://doi.org/10.1016/j.arr.2014.02.004

Kenny, D. A., Kashy, D. A., & Cook, W. L. (2006). *Dyadic data analysis*. *Methodology in the social sciences*. New York, NY: Guilford Press. Retrieved from http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10254823

Kilpatrick, J. (1987). Problem formulating: Where do good problem come from? In A. H. Schoenfeld (Ed.), *Cognitive science and mathematics education* (pp. 123–147). Hillsdale, NJ: Erlbaum.

Kim, H.-Y. (2013). Statistical notes for clinical researchers: Evaluation of measurement error 1: Using intraclass correlation coefficients. *Restorative Dentistry & Endodontics*, *38*(2), 98–102. https://doi.org/10.5395/rde.2013.38.2.98

Kitchener, K. S., Lynch, C. L., Fischer, K. W., & Wood, P. K. (1993). Developmental range of reflective judgment: The effect of contextual support and practice on developmental stage. *Developmental Psychology*, *29*(5), 893–906. https://doi.org/10.1037/0012-1649.29.5.893

Kluwe, R. H., Misiak, C., & Haider-Hasebrink, H. (1991). The control of complex systems and performance in intelligence tests. In H. A. H. Rowe (Ed.), Intelligence: Reconceptualization and measurement. (pp. 227–244). Lawrence Erlbaum Associates, Inc.

Knoblauch, H., Tuma, R., & Schnettler, B. (2013). Video Analysis and videography. In U. Flick (Ed.), *The SAGE handbook of qualitative data analysis* (pp. 335–449). London: Sage.

Korkmaz, Ö., & Bai, X. (2019). Adapting Computational Thinking Scale (CTS) for Chinese High School Students and Their Thinking Scale Skills Level. *Participatory Educational Research*, *6*(1), 10–26.

Korkmaz, Ö., Çakir, R., & Özden, M. Y. (2017). A validity and reliability study of the computational thinking scales (CTS). *Computers in Human Behavior*, *72*, 558–569. doi:10.1016/j.chb.2017.01.005

Kramer, J. (2007). Is abstraction the key to computing? *Communications of the ACM*, *50*(4), 36–42. https://doi.org/10.1145/1232743.1232745

Kvist, A. V., & Gustafsson, J.‐E. (2008). The relation between fluid intelligence and the general factor as a function of cultural background: A test of Cattell's investment theory. *Intelligence*, *36*, 422–436. https://doi.org/10.1016/j.intell.2007.08.004

Lamprou, A., & Repenning, A. (2018). Teaching how to teach computational thinking. In I. Polycarpou, J. C. Read, P. Andreou, & M. Armoni (Eds.), *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education - ITiCSE 2018* (pp. 69–74). New York, New York, USA: ACM Press. https://doi.org/10.1145/3197091.3197120

Landis, R., & Koch, G. G. (1977). The measurement of observer agreement for categorial data. *Biometrics*, *33*, 159–174.

Lee, F. J., & Anderson, J. R. (2001). Does learning a complex task have to be complex? A study in learning decomposition. *Cognitive Psychology*, *42*, 267–316. https://doi.org/10.1006/cogp.2000.0747

Lee, G., Lin, Y. T., & Lin, J. (Eds.) 2014. *Assessment of computational thinking skill among high school and vocational school students in Taiwan*. In J. Viteli and M. Leikomaa (Eds.), Proceedings of EdMedia 2014-World Conference on Education Media and Technology, pages 173-180. Tampere, Finland: Association for the Advancement of Computing in Education (AACE).

Lim, W., Plucker, J. A., & Im, K. (2002). We are more alike than we think we are. *Intelligence*, *30*, 185–208. https://doi.org/10.1016/S0160-2896(01)00097-6

Liljedahl, P., Santos-Trigo, M., Malaspina, U., & Bruder, R. (2016). *Problem Solving in Mathematics Education*. Springer Nature. https://doi.org/10.1007/978-3-319-40730-2

Li, Y., & Schoenfeld, A. H. (2019). Problematizing teaching and learning mathematics as “given” in STEM education. *International Journal of STEM Education, 6*(44). https://doi.org/10.1186/s40594-019-0197-9.

Lockwood, J., & Mooney, A. (2018a). Computational thinking in secondary education: Where does it fit? A systematic literary review. *International Journal of Computer Science Education in Schools*, *2*(1). https://doi.org/10.21585/ijcses.v2i1.26

Lockwood, J., & Mooney, A. (2018b). Developing a computational thinking test using Bebras problems. In A. Piotrkowicz, R. Dent-Spargo, S. Dennerlein, I. Koren, P. Antoniou, P. Bailey, . . . C. Pahl (Chairs), *European Conference on Technology Enhanced Learning 2018,* Leeds, United Kingdom.

Lohman, D. F., & Lakin, J. M. (2011). Intelligence and reasoning. In R. J. Sternberg & S. B. Kaufman (Eds.), *The Cambridge Handbook of Intelligence* (pp. 419–441).

Cambridge: Cambridge University Press. https://doi.org/10.1017/CBO9780511977244.022

Lotz, M., Gabriel, K., & Lipowsky, F. (2013). Niedrig und hoch inferente Verfahren der Unterrichtsbeobachtung: Analysen zu deren gegenseitiger Validierung. *Zeitschrift für Pädagogik*, *59*, 357–380. Retrieved from http://nbn-resolving.de/urn:nbn:de:0111-pedocs-119425

Lourenço, O., & Machado, A. (1996). In defense of Piaget's theory: A reply to 10 common criticisms. *Psychological Review*, *103*(1), 143–164. https://doi.org/10.1037/0033-295X.103.1.143

Lu, J. J., & Fletcher, G. H. L. (2009). Thinking about computational thinking. In S. Fitzgerald (Ed.), *Proceedings of the 40th ACM technical symposium on Computer science education.* New York, NY: ACM.

Lübbers T., & Jansen, M. (2018). Application of microcontrollers for fostering computational thinking by using the calliope system in school. In J. C. Yang, M. Chang, L.-H. Wong, & M. M. T. Rodrigo (Eds.), *Proceedings of the 26th International Conference on Computers in Education. 2018* (pp. 500–505). Taoyuan County, Taiwan: Asia-Pacific Society for Computers in Education (APSCE).

Lubinski, D. (2004). Obituary, Lloyd G. Humphreys: Quintessential Scientist (1913?2003). *Intelligence*, *32*(3), 221–226. https://doi.org/10.1016/j.intell.2004.01.002

Lutz, C., Berges, M., Hafemann, J., & Sticha, C. (2019). Piaget’s cognitive development in Bebras tasks - A descriptive analysis by age groups. In S. N. Pozdniakov & V. Dagienė (Eds.), *Lecture notes in computer science. Informatics in schools. Fundamentals of computer science and software* (Vol. 11169, pp. 259–270). [Place of publication not identified]: Springer. https://doi.org/10.1007/978-3-030-02750-6_20

Luxton-Reilly, A., Whalley, J., Becker, B. A., Cao, Y., McDermott, R., Mirolo, C., . . . Simon (2017). Developing assessments to determine mastery of programming fundamentals. In J. Sheard & Education, ACM Special Interest Group on Computer Science (Eds.), *Proceedings of the 2017 ITiCSE Conference on Working Group Reports* (pp. 47–69). [S.l.]: ACM. https://doi.org/10.1145/3174781.3174784

Lister, R., & Leaney, J. (2003). Introductory Programming, Criterion-Referencing, and Bloom. *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, 143–147. https://doi.org/10.1145/611892.611954

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

Mackintosh, N. J. (2011). History of theories and measurement of intelligence. In R. J. Sternberg & S. B. Kaufman (Eds.), *The Cambridge handbook of intelligence* (pp. 3–19). Cambridge: Cambridge University Press. https://doi.org/10.1017/CBO9780511977244.002

Mangold. (2018). INTERACT - User guide: Mangold International GmbH (ed.). Retrieved from www.mangold-international.com

Martin, R. C. (2009). *Clean code: A handbook of agile software craftsmanship / Robert C. Martin ... [et al.]*. Indianapolis, IN.: Prentice Hall.

Mayer, J. d., Salovey, P., Caruso, D. R., & Cherkasskiy, L. (2011). Emotional intelligence. In R. J. Sternberg & S. B. Kaufman (Eds.), *The Cambridge handbook of intelligence* (pp. 528–549). Cambridge, UK: Cambridge University Press. https://doi.org/10.1017/CBO9780511977244.027

McCabe, T. J. (1976). A complexity measure. In *ICSE '76, Proceedings of the 2nd International Conference on Software Engineering* (407-). Los Alamitos, CA: IEEE Computer Society Press. Retrieved from http://dl.acm.org/citation.cfm?id =800253.807712

McDonald, C. (2018). Why Is teaching programming difficult? In J. Carter, M. O'Grady, & C. Rosen (Eds.), *Higher education computer science: A manual of practical approaches* (pp. 75–93). Cham: Springer. https://doi.org/10.1007/978-3-319-98590-9_6

McFadden, C. (2018, September 13). The origin of the term 'computer bug' [Blog post]. Retrieved from https://interestingengineering.com/the-origin-of-the-term-computer-bug

McGrew, K. S. (2009). CHC theory and the human cognitive abilities project: Standing on the shoulders of the giants of psychometric intelligence research. Intelligence, 37(1), 1-10.

McGrew, K. S. (2005). The Cattell-Horn-Carroll theory of cognitive abilities: Past, present, and future. In D. P. Flanagan & P. L. Harrison (Eds.), *Contemporary intellectual assessment: Theories, tests, and issues* (pp. 136–181). New York, NY: The Guilford Press.

MDESE. (2016). *Massachusetts digital literacy and computer science*. Malden, MA.

Menéndez-Varela, J.-L., & Gregori-Giralt, E. (2016). The contribution of rubrics to the validity of performance assessment: a study of the conservation–restoration and design undergraduate degrees. *Assessment & Evaluation in Higher Education*, *41*(2), 228–244. https://doi.org/10.1080/02602938.2014.998169

Mensing, K., Mak, J., Bird, M., & Billings, J. (2013). Computational, model thinking and computer coding for U.S. Common Core Standards with 6 to 12 year old students. In A. Szakál (Ed.), *2013 IEEE 11th International Conference on Emerging eLearning Technologies and Applications (ICETA): 24-25 Oct. 2013, Stary Smokovec, the High Tatras, Slovaki* (pp. 17–22). Piscataway, NJ: IEEE. https://doi.org/10.1109
/ICETA.2013.6674397

Michaels, G., Natraj, A., & van Reenen, J. (2014). Has ICT polarized skill demand?: Evidence from eleven countries over twenty-five years. *Review of Economics and Statistics*, *96*(1), 60–77. https://doi.org/10.1162/REST_a_00366

Miller, G. A. (1956). The magical number seven plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, *63*(2), 81–97.

Miller, G. A. (2003). The cognitive revolution: A historical perspective. *Trends in Cognitive Sciences*, *7*(3), 141–144. https://doi.org/10.1016/S1364-6613(03)00029-9

Miller, G. A., Galanter, E., & Pribram, K. H. (1960). *Plans and the structure of behavior*: Holt, Rinehart and Winston, Inc.

Moessinger, P., & Poulin-Dubois, D. (1981). Piaget on abstraction. *Human Development*, *24*, 347–353. https://doi.org/10.1159/000272712

Moreno-León, J., & Robles, G. (2014). Automatic detection of bad programming habits in scratch: A preliminary study. In *IEEE Frontiers in Education Conference (FIE), 2014: 22 - 25 Oct. 2014, Madrid, Spain* (pp. 1–4). Piscataway, NJ: IEEE. https://doi.org/10.1109/FIE.2014.7044055

Moreno-León, J., & Robles, G. (2015). Dr. Scratch: A web tool to automatically evaluate Scratch projects. In J. Gal-Ezer, S. Sentance, & J. Vahrenhold (Eds.), *Proceedings of the Workshop in Primary and Secondary Computing Education, London, United Kingdom, November 09 - 11, 2015* (pp. 132–133). New York, NY: ACM. https://doi.org/10.1145/2818314.2818338

Moreno-León, J., Robles, G., & Román-González, M. (2015). Dr. Scratch: Automatic analysis of Scratch projects to assess and foster computational thinking. *RED-Revista de Educación a Distancia*.

Moreno-León, J., Robles, G., & Román-González, M. (2016). Comparing computational thinking development assessment scores with software complexity metrics. In *Proceedings of 2016 IEEE Global Engineering Education Conference (EDUCON): Date and venue: 10-13 April 2016, Abu Dhabi, UAE* (pp. 1040–1045). Piscataway, NJ: IEEE. https://doi.org/10.1109/EDUCON.2016.7474681

Moreno-León, J., Robles, G., & Román-González, M. (2017). Towards data-driven learning paths to develop computational thinking with Scratch. *IEEE Transactions on Emerging Topics in Computing*, 1. https://doi.org/10.1109/TETC.2017.2734818

Moreno-León, J., Román-González, M., Harteveld, C., & Robles, G. (2017). On the automatic assessment of computational thinking skills. In G. Mark, S. Fussell, C. Lampe, M. C. Schraefel, J. P. Hourcade, C. Appert, & D. Wigdor (Eds.), *CHI'17: Extended abstracts: Proceedings of the 2017 ACM SIGCHI Conference on Human Factors in Computing Systems : May 6-11, 2017, Denver, CO* (pp. 2788–2795). New York, NY: The Association for Computing Machinery. https://doi.org/10.1145/3027063.3053216

Najafi, A., Niu, N., & Najafi, F. (2011). Multi-level decomposition approach for problem solving and design in software engineering. In K. Hoganson (Ed.), *ACM Digital Library, Proceedings of the 49th Annual Southeast Regional Conference* (p. 249). New York, NY: ACM. https://doi.org/10.1145/2016039.2016104

Newell, A., Shaw, J. C., & Simon, H. A. (1958). Elements of a theory of human problem solving. *Psychological Review*, *65*, 151–166. https://doi.org/10.1037/h0048495

Newman, I., Lim, J., & Pineda, F. (2013). Content validity using a mixed methods approach. *Journal of Mixed Methods Research*, *7*, 243–260. https://doi.org/10.1177/1558689813476922

Nickerson, R. S. (2011). Developing intelligence through instruction. In R. J. Sternberg & S. B. Kaufman (Eds.), *The Cambridge Handbook of Intelligence* (pp. 107–129). Cambridge: Cambridge University Press. https://doi.org/10.1017/CBO9780511977244.007

National Research Council (NRC). (2002). Helping children learn mathematics. Washington, DC: The National Academies Press. https://doi.org/10.17226/1043.

National Research Council (NRC). (2010). *Report of a workshop on the scope and nature of computational thinking*. National Research Council. Washington, D.C: National Academies Press.
Retrieved from http://site.ebrary.com/lib/academiccompletetitles/home.action

Nwadinigwe, I., & Naibi, L. (2013). The number of options in a multiple-choice test item and the psychometric characteristics. *Journal of Education and Practice, 4*(28).

O'Dell, D. H. (2017). The debugging mind-set. *Communications of the ACM*, *60*(6), 40–45. https://doi.org/10.1145/3052939

OECD (2016a). New skills for the digital economy. *OECD Digital Economy Papers*, *258.* https://doi.org/10.1787/5jlwnkm2fc9x-en

OECD (2016b). Skills for a digital world: 2016 Ministerial Meeting on the Digital Economy Background Report. *OECD Digital Economy Papers*, *250.* https://doi.org/10.1787/5jlwz83z3wnw-en

OECD (Ed.) 2017. *Key issues for digital transformation in the G20: Report prepared for a joint G20 German Presidency/ OECD conference*. Berlin, Germany.

Olsen, A. (2005). Using pseudocode to teach problem solving. *Journal of Computing Sciences in Colleges*, *21*.

Palts, T., & Pedaste, M. (2020). A Model for Developing Computational Thinking Skills. *Informatics in Education*, *19*, 113–128. https://doi.org/10.15388/infedu.2020.06

Panadero, E., & Jonsson, A. (2020). A critical review of the arguments against the use of rubrics. *Educational Research Review*, *30*, 100329. https://doi.org/https://doi.org/10.1016/j.edurev.2020.100329

Papert, S. (1980). *Mindstorm: Children, computers, and powerful ideas*. New York, NY: Basic Books.

Papert, S. (1996). An exploration in the space of mathematics educations. *International Journal of Computers for Mathematical Learning*, *1*(1). https://doi.org/10.1007/BF00191473

Papert, S., & Harel, I. (1991). Situating constructionism. In I. Harel & S. Papert (Eds.), *Constructionism: Research reports and essays, 1985-1990*. Norwood, NJ: Ablex.

Pauli, C., & Reusser, K. (2006). Von international vergleichenden Video Surveys zur videobasierten Unterrichtsforschung und -entwicklung. *Zeitschrift für Padagogik*, *52*.

Pease, A., Smaill, A., & Guhe, M. (2009). Abstract or not abstract? Well, it depends … *Behavioral and Brain Sciences*, *32*(3-4), 345–346. https://doi.org/10.1017/S0140525X09991063

Perković, L., Settle, A., Hwang, S., & Jones, J. (2010). A framework for computational thinking across the curriculum. In R. Ayfer, J. Impagliazzo, & C. Laxer (Eds.), *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education - ITiCSE '10* (p. 123). New York, NY: ACM Press. https://doi.org/10.1145/1822090.1822126

Piaget, J. (1952). *The origins of intelligence in children*. New York, NY: International Universities Press, Inc. https://doi.org/10.1037/11494-000

Piaget, J. (1960). *The Psychology of intelligence*. New York, NY: Littlefield, Adams & Co.

Polson, P., & Jeffries, R. (1985). Instruction in problem solving skills: An analysis of four approaches. In J. W. Segal (Ed.), *Thinking and learning skills* (pp. 417–455). Hillsdale, NJ: Erlbaum.

Portelance, D. J., & Bers, M. U. (2015). Code and tell. In M. U. Bers & G. Revelle (Eds.), *IDC 2015: ACM SIGCHI Interaction Design and Children : Tufts University, Boston, MA, USA, June 21-24, 2015* (pp. 271–274). New York, NY: ACM. https://doi.org/10.1145/2771839.2771894

Posner, M. I. (1969). Abstraction and the process of recognition. In G. H. Bower, K. W. Spence, J. T. Spence, & D. L. Medin (Eds.), *The psychology of learning and motivation: Advances in research and theory* (Vol. 3, pp. 43–100). New York, NY: Academic Press. https://doi.org/10.1016/S0079-7421(08)60397-7

Posner, M. I., & Keele, S. W. (1968). On the genesis of abstract ideas. *Journal of Experimental Psychology*, *77*, 353–363. https://doi.org/10.1037/h0025953

President's Information Technology Advisory Committee (PITAC) (2005). *Computational science: Ensuring America's competitiveness* (Report to the President, June 2005). Washington, DC: National Coordination Office for Information Technology Research and Development (NCO/IT R&D).

Pretz, J. E., Naples, A. J., & Sternberg, R. J. (2003). Recognizing, defining, and representing problems. In J. E. Davidson & R. J. Sternberg (Eds.), *The Psychology of problem solving* (pp. 3–30). Cambridge, UK: Cambridge University Press. https://doi.org/10.1017/CBO9780511615771.002

Priami, C. (Ed.). (2007). *Journal subline: 4780 : Lecture notes in bioinformatics*. *Transactions on computational systems biology*. Berlin: Springer. https://doi.org/10.1007/978-3-540-76639-1

R Core Team. (2017). R. Vienna, Austria. Retrieved from https://www.R-project.org/

Repenning, A. (2015). *Computational thinking in der Lehrerbildung*. Bern, Schweiz: Hasler-Stiftung.

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

Rigas, G., & Brehmer, B. (1999). Mental processes in intelligence tests and dynamic decision making tasks. In P. Juslin & H. Montgomery (Eds.), *Judgement and decision making: NeoBrunswikian and process-tracing approaches* (pp. 45-65). Hillsdale, NJ: Lawrence Erlbaum.

Rodrigo, Tabanao, Lahoz, Jadud (2009). Analyzing online protocols to characterize novice Java programmers. *Philippine Journal of Science*, *138*, 177–190.

Rodriguez, B., Kennicutt, S., Rader, C., & Camp, T. (2017). Assessing computational thinking in CS unplugged activities. In M. E. Caspersen, S. H. Edwards, T. Barnes, & D. D. Garcia (Eds.), *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education - SIGCSE '17* (pp. 501–506). New York, NY: ACM Press. https://doi.org/10.1145/3017680.3017779

Román-González, M., Moreno-León, J., & Robles, G. (2017). Complementary tools for computational thinking assessment. In S.-C. Kong, J. Sheldon, & R. K.-y. Li (Eds.), *Proceedings of the 2017 International Conference on Computational Thinking Education* (154-158). Hong Kong, China: The Education University of Hong Kong.

Román-González, M., Pérez-González, J.‐C., & Jiménez-Fernández, C. (2017). Which cognitive abilities underlie computational thinking?: Criterion validity of the Computational Thinking Test. *Computers in Human Behavior*, *72*, 678–691. https://doi.org/10.1016/j.chb.2016.08.047

Rosch, E. (1978). Principles of categorization. In E. Rosch & B. B. Lloyd (Eds.), *Cognition and categorization* (pp. 27–48). Hillsdale, NJ: Lawrence Erlbaum.

Rosch, E., Mervis, C. B., Gray, W. D., Johnson, D. M., & Boyes-Braem, P. (1976). Basic objects in natural categories. *Cognitive Psychology*, *8*, 382–439. https://doi.org/10.1016/0010-0285(76)90013-X

Ross, B. H. (1984). Remindings and their effects in learning a cognitive skill. *Cognitive Psychology*, *16*, 371–416. https://doi.org/10.1016/0010-0285(84)90014-8

Roy, G. G. (2006). Designing and explaining programs with a literate pseudocode. *Journal on Educational Resources in Computing*, *6*(1), 1-es. https://doi.org/10.1145/1217862.1217863

Royston, P. (1993). A pocket-calculator algorithm for the Shapiro-Francia test for non-normality: An application to medicine. *Statistics in Medicine*, *12*, 181–184. https://doi.org/10.1002/sim.4780120209

Schank, R. C., & Abelson, R. P. (1977). *Scripts, plans, goals and understanding: An inquiry into human knowledge structures*. *Artificial intelligence series*. Hillsdale, NJ: L. Erlbaum Associates.

Schulz, K., & Hobson, S. (2015). *Bebras Australia Computational Thinking Challenge Tasks and Solutions 2014*. Brisbane, Australia: Digital Careers.

Schulz, K., Hobson, S., & Zagami, J. (2016). *Bebras Australia Computational Thinking Challenge - Tasks and Solution 2016*. Brisbane, Australia: Digital Careers.

Seiter, L., & Foreman, B. (2013). Modeling the learning progressions of computational thinking of primary grade students. In B. Simon (Ed.), *Proceedings of the ninth annual international ACM conference on International computing education research* (p. 59). New York, NY: ACM. https://doi.org/10.1145/2493394.2493403

Selby, C. (2015). Relationships: Computational thinking, pedagogy of programming, and Bloom's Taxonomy. In J. Gal-Ezer, S. Sentance, & J. Vahrenhold (Eds.), *Proceedings of the Workshop in Primary and Secondary Computing Education, London, United Kingdom, November 09 - 11, 2015* (pp. 80–87). New York, NY: ACM. https://doi.org/10.1145/2818314.2818315

Selby, C., & Woollard, J. (2014). Refining an understanding of computational thinking. *Author's original*. Retrieved from http://eprints.soton.ac.uk/372410/

Sentance, S., Waite, J., Hodges, S., MacLeod, E., & Yeomans, L. (2017). Creating cool stuff. In M. E. Caspersen, S. H. Edwards, T. Barnes, & D. D. Garcia (Eds.), *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education - SIGCSE '17* (pp. 531–536). New York, NY: ACM Press. https://doi.org/10.1145/3017680.3017749

Shapiro, S. S., & Francia, R. S. (1972). An approximate analysis of variance test for normality. *Journal of the American Statistical Association*, *67*(337), 215–216. https://doi.org/10.1080/01621459.1972.10481232

Shi, W., Liu, M., & Hendler, P. (2014). Computational features of the thinking and the thinking attributes of computing: On computational thinking. *Journal of Software*, *9*(10). https://doi.org/10.4304/jsw.9.10.2507-2513

Shivhare, R., & Kumar, C. A. (2016). On the cognitive process of abstraction. *Procedia Computer Science*, *89*, 243–252. https://doi.org/10.1016/j.procs.2016.06.051

Shoemate, B. (2008, November 30). Einstein never said that … [Blog post]. Retrieved from http://www.benshoemate.com/2008/11/30/einstein-never-said-that/

Shrout, P. E., & Fleiss, J. L. (1979). Intraclass correlations: Uses in assessing rater reliability. *Psychological Bulletin*, *86*, 420–428. https://doi.org/10.1037/0033-2909.86.2.420

Shute, V. J., Sun, C., & Asbell-Clarke, J. (2017). Demystifying computational thinking. *Educational Research Review*, *22*, 142–158. https://doi.org/10.1016/j.edurev.2017.09.003

Siegler, R. S., DeLoache, J. S., & Eisenberg, N. (2014). *How children develop* (4th ed.). New York, NY: Worth.

Silverman, D. (2013). *A very short, fairly interesting and reasonably cheap book about qualitative research* (2nd ed.). London, UK: Sage.

Simmons, R. (1988). A theory of debugging plans and interpretations. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)* (pp. 94–99).

Simon, H.A. (1973). The structure of ill structured problems. *Artificial Intelligence 4*(3-4), 181–201.

Simon, H. A., & Newell, A. (1971). Human problem solving: The state of the theory in 1970. *American Psychologist*, *26*, 145–159. https://doi.org/10.1037/h0030806

Sipser, M. (2013). *Introduction to the theory of computation* (3rd ed.). Boston, MA: Thomson Course Technology.

Sonnleitner, P., Brunner, M., Greiff, S., Funke, J., Keller, U., Martin, R., Hazotte, C., Mayer, H., & Latour, T. (2012). The Genetics Lab. Acceptance and psychometric characteristics of a computer-based microworld to assess complex problem solving. *Psychological Test and Assessment Modeling, 54*(1), 54-72.

Sowder, J. (1992). Estimation and number sense. In D. Grouws (Ed.), *Handbook for research on mathematics teaching and learning* (pp. 371–389). New York: MacMillan.

Spearman, C. (1904). "General intelligence," Objectively Determined and Measured. *The American Journal of Psychology*, *15*(2), 201. https://doi.org/10.2307/1412107

Stadler, M., Becker, N., Gödker, M., Leutner, D., & Greiff, S. (2015). Complex problem solving and intelligence: A meta-analysis. *Intelligence*, 53, 92-101.

Stein, L. A. (2002). *Introduction to interactive programming in Java*: Morgan Kaufmann.

Sternberg, R. J. (1985). Implicit theories of intelligence, creativity, and wisdom. *Journal of Personality and Social Psychology*, *49*, 607–627. https://doi.org/10.1037/0022
-3514.49.3.607

Sternberg, R. J. (2004). Culture and intelligence. *American Psychologist*, *59*, 325–338. https://doi.org/10.1037/0003-066X.59.5.325

Sternberg, R. J. (2017). Human intelligence. *Encyclopaedia Britannica*. Retrieved from https://www.britannica.com/topic/human-intelligence-psychology/Development-of
-intelligence#ref13354

Sternberg, R. J., Conway, B. E., Ketron, J. L., & Bernstein, M. (1981). People's conceptions of intelligence. *Journal of Personality and Social Psychology*, *41*(1), 37–55. https://doi.org/10.1037/0022-3514.41.1.37

Sternberg, R. J., & Berg, C. A. (Eds.). (1992). *Intellectual development*. Cambridge: Cambridge University Press.

Stojanoski, B., Lyons, K. M., Pearce, A. A. A., & Owen, A. M. (2018). Targeted training: Converging evidence against the transferable benefits of online brain training on cognitive function. *Neuropsychologia*, *117*, 541–550. https://doi.org/10.1016
/j.neuropsychologia.2018.07.013

Swaid, S. I. (2015). Bringing computational thinking to STEM education. *Procedia Manufacturing*, *3*, 3657–3662. https://doi.org/10.1016/j.promfg.2015.07.761

Tabanao, E. S., Rodrigo, M. M. T., & Jadud, M. C. (2011). Predicting at-risk novice Java programmers through the analysis of online protocols. In K. Sanders (Ed.), *Proceedings of the seventh international workshop on Computing education research* (p. 85). New York, NY: ACM. https://doi.org/10.1145/2016911.2016930

Tabesh, Y. (2017). Computational thinking: A 21st Century skill. *Olympiads in Informatics*, *11*(2), 65–70. https://doi.org/10.15388/ioi.2017.special.10

Teague, D., & Lister, R. (2014). Longitudinal think aloud study of a novice programmer. In J. Whalley (Ed.), *Proceedings of the Sixteenth Australasian Computing Education Conference - Volume 148*. Darlinghurst, Australia: Australian Computer Society, Inc.

Terman, L. M. (1921). Intelligence and its measurement: A symposium--II. *Journal of Educational Psychology*, *12*(3), 127–133. https://doi.org/10.1037/h0064940

Thalheim, B. (2009). Abstraction. In L. Liu & M. T. Özsu (Eds.), *Springer reference. Encyclopedia of database systems*. New York, NY: Springer.

Thies, R., & Vahrenhold, J. (2013). On plugging "unplugged" into CS classes. In R. McCauley (Ed.), *Sigcse'13: Proceedings of the 44th ACM Technical Symposium on Computer Science Education; March 6 - 9, 2013, Denver, Colorado, USA*. New York, NY: ACM. https://doi.org/10.1145/2445196.2445303

Thompson, N. (2017). What is classical item difficulty (P value)? Retrieved from http://www.assess.com/classical-item-difficulty-p-value/

Thurstone, L. L. (1938). *Primary mental abilities*. Chicago, IL: University of Chicago Press.

Toga, A. W., & Thompson, P. M. (2005). Genetics of brain structure and intelligence. *Annual Review of Neuroscience*, *28*, 1–23. https://doi.org/10.1146/annurev.neuro.28.061604.135655

Touretzky, D. S., Marghitu, D., Ludi, S., Bernstein, D., & Ni, L. (2013). Accelerating K-12 computational thinking using scaffolding, staging, and abstraction. In T. Camp, P. Tymann, J. D. Dougherty, & K. Nagel (Chairs), *Proceeding of the 44th ACM technical symposium,* Denver, Colorado, USA.

Trachtenberg, J. (1960). *The Trachtenberg Speed System of Basic Mathematics*. Garden City, NY: Doubleday and Company, Inc.

Trevethan, R. (2017). Intraclass correlation coefficients: Clearing the air, extending some cautions, and making some requests. *Health Services and Outcomes Research Methodology*, *17*(2), 127–143. https://doi.org/10.1007/s10742-016-0156-6

Urbina, S. (2011). Tests of intelligence. In R. J. Sternberg & S. B. Kaufman (Eds.), *The Cambridge handbook of intelligence* (pp. 20–38). Cambridge, UK: Cambridge University Press. https://doi.org/10.1017/CBO9780511977244.003

Van Dyne, M., & Braun, J. (2014). Effectiveness of a computational thinking (CS0) course on student analytical skills. In J. D. Dougherty, K. Nagel, A. Decker, & K. Eiselt (Chairs), *SIGCSE '14 Proceedings of the 45th ACM technical symposium on Computer science education,* Atlanta, Georgia, USA.

Vaníček, J. (2014). Bebras Informatics Contest: Criteria for Good Tasks Revised. In Y. Gülbahar & E. Karataş (Eds.), *Informatics in Schools. Teaching and Learning Perspectives: 7th International Conference on Informatics in Schools: Situation, Evolution, and Perspectives, ISSEP 2014, Istanbul, Turkey, September 22-25, 2014. Proceedings* (pp. 17–28). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-09958-3_3

Voogt, J., Fisser, P., Good, J., Mishra, P., & Yadav, A. (2015). Computational thinking in compulsory education: Towards an agenda for research and practice. *Education and Information Technologies*, *20*, 715–728. https://doi.org/10.1007/s10639-015-9412-6

Vuorikari, R., Punie, Y., Carretero, S., & van den Brande, L. (2016). *DigComp 2.0: The digital competence framework for citizens*. *EUR, Scientific and technical research series: Vol. 27948*. Luxembourg: Publications Office.

Wang, X., & Zhou, Z. The research of situational teaching mode of programming in high school with Scratch. In *2011 6th IEEE Joint International Information Technology and Artificial Intelligence Conference (ITAIC),* Chongqing, China.

Ward, M. (1995). A definition of abstraction. *Journal of Software Maintenance: Research and Practice*, *7*, 443–450. https://doi.org/10.1002/smr.4360070606

Watt, D. A., & Findlay, W. (2004). *Programming language design concepts*. Chichester, UK: John Wiley & Sons, Ltd.

Wechsler, D. (1958). *The measurement and appraisal of adult intelligence* (4th ed.). Baltimore, MD: Williams & Wilkins Co. https://doi.org/10.1037/11167-000

Weintrop, D., Beheshti, E., Horn, M., Orton, K., Jona, K., Trouille, L., & Wilensky, U. (2016). Defining computational thinking for mathematics and science classrooms. *Journal of Science Education and Technology*, *25*(1), 127–147. https://doi.org/10.1007/s10956-015-9581-5

Weintrop, D., & Wilensky, U. (2018). How block-based, text-based, and hybrid block/text modalities shape novice programming practices. *International Journal of Child-Computer Interaction.* Advance online publication. https://doi.org/10.1016/j.ijcci.2018.04.005

Wenke, D., & Frensch, P. A. (2003). Is success or failure at solving complex problems related to intellectual ability? In J. E. Davidson & R. J. Sternberg (Eds.), *The Psychology of Problem Solving* (pp. 87–126). Cambridge, UK: Cambridge University Press. https://doi.org/10.1017/CBO9780511615771.004

Wentworth, P., Elkner, J., Downey, A. B., & Meyers, C. (2012). *How to think like a computer scientist: Learning with Python 3.*

Werner, L., Denner, J., & Campe, S. (2012). The Fairy Performance Assessment: Measuring computational thinking in middle school. In Sigcse Conference Committee (Ed.), *Sigcse 12 Proceedings of the 43rd Acm Technical Symposium on Computer Science Education.* New York, NY: Association for Computing Machinery.

Werner, L. & Denning, J. (2009). Pair Programming in Middle School. *Journal of Research on Technology in Education 42*(1), 29–49.

White House (2017). President Trump signs memorandum for STEM education funding. https://www.whitehouse.gov/articles/president-trump-signs-memorandum-stem-education-funding/

Wilhelm, O. (2005). Measuring reasoning ability. In O. Wilhelm & R. W. Engle (Eds.), *Handbook of understanding and measuring intelligence* (pp. 373–392). Thousand Oaks, CA: Sage. https://doi.org/10.4135/9781452233529.n21

Williamson, B. (2016). Political computational thinking: Policy networks, digital governance and 'learning to code'. *Critical Policy Studies*, *10*(1), 39–58. https://doi.org/10.1080/19460171.2015.1052003

Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, *49*(3), 33–35. https://doi.org/10.1145/1118178.1118215

Wing, J. M. (2008). Computational thinking and thinking about computing. *Philosophical transactions of the Royal Society A: Mathematical, physical and engineering sciences*, *366*(1881), 3717–3725. https://doi.org/10.1098/rsta.2008.0118

Wing, J. M. (2011). Research notebook: Computational thinking—What and why? Retrieved from https://www.cs.cmu.edu/link/research-notebook-computational-thinking-what-and-why

Wirth, J., & Klieme, E. (2003). Computer-based assessment of problem solving competence. *Assessment in Education: Principles, Policy, & Practice, 10*, 329-345.

Wittmann, W. W., & Hattrup, K. (2004). The relationship between performance in dynamic systems and intelligence. *Systems Research and Behavioral Science, 21*(4), 393-409.

Wittmann, W., & Suß, H.-M. (1999). Investigating the paths between working memory, intelligence, knowledge, and complex problem-solving performances via Brunswik symmetry. In P. L. Ackerman, P. C. Kyllonen, & R. D. Roberts (Eds.), *Learning and individual differences: Process, traits, and content determinants* (pp. 77-108). Washington, DC: APA.

Wu, B., Hu, Y., Ruis, A. R., & Wang, M. (2019). Analysing computational thinking in collaborative programming: A quantitative ethnography approach. *Journal of Computer Assisted Learning*, *35*, 421–434. https://doi.org/10.1111/jcal.12348

Wüstenberg, S., Stadler, M., Hautamäki, J., & Greiff, S. (2014). The role of strategy knowledge for the application of strategies in complex problem solving tasks. *Technology, Knowledge and Learning, 19*, 127-146.

Wüstenberg, S., Greiff, S., & Funke, J. (2012). Complex problem solving – more than reasoning? *Intelligence, 40*, 1-14.

Xu, Z., Ritzhaupt, A. D., Tian, F., & Umapathy, K. (2019). Block-based versus text-based programming environments on novice student learning outcomes: A meta-analysis study. *Computer Science Education*, *29*(2–3), 177–204. https://doi.org/10.1080/08993408.2019.1565233

Yap, B. W., & Sim, C. H. (2011). Comparisons of various types of normality tests. *Journal of Statistical Computation and Simulation*, *81*, 2141–2155. https://doi.org/10.1080/00949655.2010.520163

Zamanzadeh, V., Ghahramanian, A., Rassouli, M., Abbaszadeh, A., Alavi-Majd, H., & Nikanfar, A.‑R. (2015). Design and implementation content validity study: Development of an instrument for measuring patient-centered communication. *Journal of Caring Sciences*, *4*, 165–178. https://doi.org/10.15171/jcs.2015.017

7 APPENDIX

Appendix A: Summary of literature analysis to define CT

	Problem solving	Decomposing	Abstracting	Algorithmic design
	Works based on experts' opinions like workshops and online surveys			
NRC (2010)	CT as mental tool to reformulate problems to solve it (p. 3);	Problem Decomposition and modularisation (p. 3);	Problem abstraction (p. 3); CT as an abstract thinking tool to handle complexity (p. 11); Abstraction as core for CT (p. 12); CT is the creation and managing of abstraction (p. 16);	Understanding of the complexity of algorithms (p. 3); Knowing specific algorithms (p. 8); "The processes by which these algorithms are developed and tested involve computational thinking." (p. 37); The solution they described are

				being often "algorithmic" (e.g., p. 3 and 26)
Corradini et al. (2017)	Identified as "absolutely necessary in any definition of CT"	Identified as "important category" for CT	Identified as "important category" for CT"	Algorithmic thinking and automation are both identified as "absolutely necessary in any definition of CT" Logical thinking is identified as "important for a definition"
Barr et al. (2011)	Highlighted CT as problem solving for complex problems	See table: one of the core concepts and capabilities of CT; Declared decomposition as one working strategy in sense of CT;	See table: one of the core concepts and capabilities of CT; Described as using abstraction to design solutions to problems as a core concept of CT as well as being able to "move between levels of abstraction";	See table: one of the core concepts and capabilities of CT (with automation; parallelisation) Understanding of algorithmic processes as vital for CT; Declared that CT is highly associated with algorithmic

			Using abstraction in sense of repeated commands and iterations, in general, being able to generalise solutions for different problems and situations	thinking; Creating algorithms as problems; Mentioned testing and debugging;
ISTE and CSTA (2011)	Declared CT as problem-solving process; "Reformulating problems in a way that computers can help to solve them"		Using abstraction in order to handle data	Being able to use algorithmic thinking to create automating solutions

	Systematic literature reviews			
Selby et al. (2014)	Emphasised that the community mainly accepted CT as a thought process to deal with problems; They concluded that there is a consensus that CT is a type of problem solving but also pointed out that the term is not sufficiently defined.	CT is about transforming difficult problems into ones that can be solved more easily and concludes that a definition of CT should include the concept of decomposition	As they pointed out that many authors declared abstraction as a key competence they concluded that a definition of CT should include the concept of abstraction	They conclude that there appears to be a consensus that CT incorporates aspects of the creation and use of algorithms; The idea of algorithm, incorporating the design process, is represented consistently in literature. They further conclude that a definition of CT should include something in a sense of algorithm design;
Bocconi et al. (2016)	Concludes that CT describes thought process which (re)-formulate problems in order to solve it computationally.	Identifies it as core skills of CT (p. 18)	Abstraction & generalization as core skills of CT (p. 18)	Algorithmic thinking & automation & debugging as core skills of CT
Kalelioğlu et al. (2016)	Concluded that most definitions in the literature dwell on problem solving, understanding problems or formulating problems; 35 % of 125 papers mentioned it	6 % of 125 papers mentioned it	49 % of 125 papers mentioned it	28 % of 125 papers mentioned it
Shute et al. (2017)	“Conceptual foundation required to solve problems effectively and efficiently”;	Identify decomposition as one of the most often components of CT;	Identify abstraction as one of the most often components of CT;	They state that CT means solving problems algorithmically and also state

	CT means approaching problem in a systematic way	state as one of the main facet of CT	abstraction means finding patterns within problems and solutions and therefore being able to generalise solutions to similar problems. State as one of the main facet of CT They also state generalisation as one of the main facet of CT but this is considered as part of abstraction in this thesis.	that algorithms and debugging are concepts more often associated with CT than others; Solutions are designs algorithmically. State as one of the main facet of CT. They also described debugging as main facet which is considered as part of algorithms here in this thesis.

Appendix B: List of complete set of revised Bebras tasks

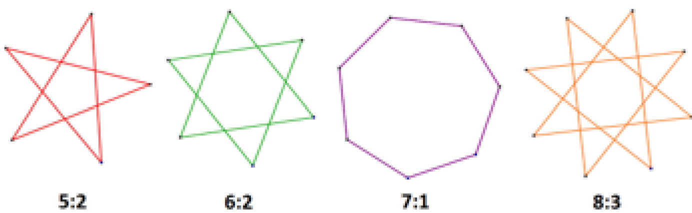
Task #1 – #115A1

Age Group: 11 + 12

Difficulty: A

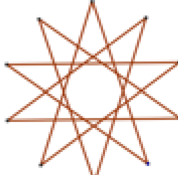
Introduction / Presentation

In the following you see several images of stars. There is a specific system for labelling the stars according to their shape. Two numbers are needed for labelling. A number of dots for the star. A number indicating if a line from a dot is drawn to the nearest dot (number is 1), the second closest dot (the number is 2), etc. Here are four examples for this labelling system:



Question & Answer

According to this specific labelling system, how would you label the following star?



9:3	
9:4	
10:4	X
10:5	

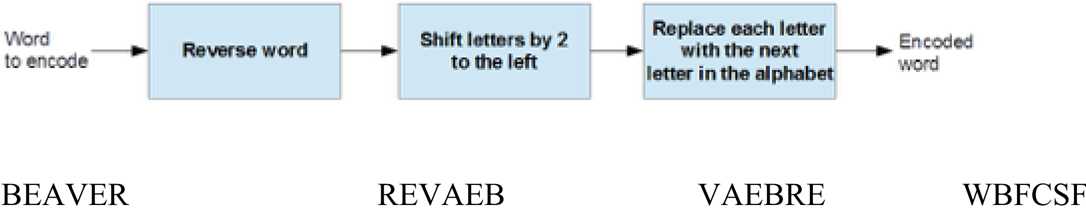
Task #2 – #215A2

Age Group: 11 + 12

Difficulty: A

Introduction / Presentation

An encryption machine transforms messages according to the following rule:



BEAVER REVAEB VAEBRE WBFCSF

Question & Answer

What is the actual message of **PMGEP**?

RIVER	
KNOCK	
FLOOD	X
LODGE	

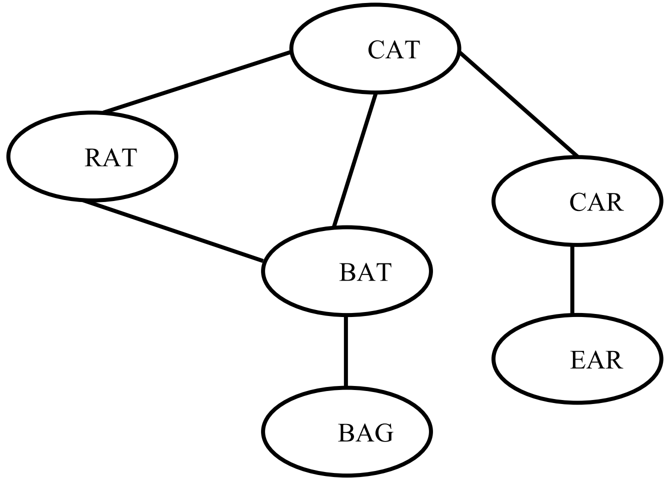
Task #3 – #315A3

Age Group: 11 + 12

Difficulty: A

Introduction / Presentation

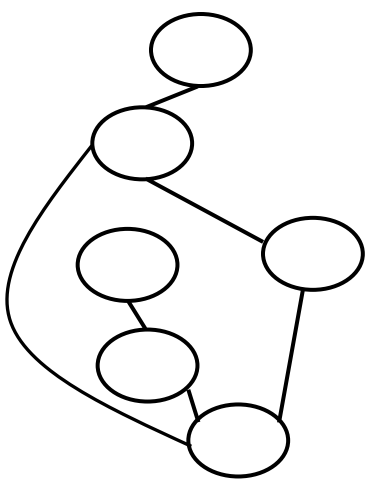
In the following picture you can see how different words are connected.

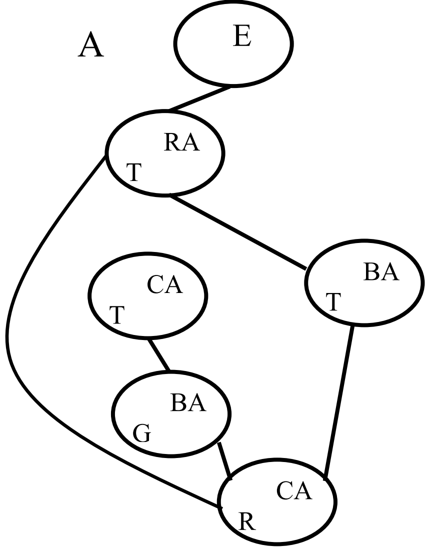


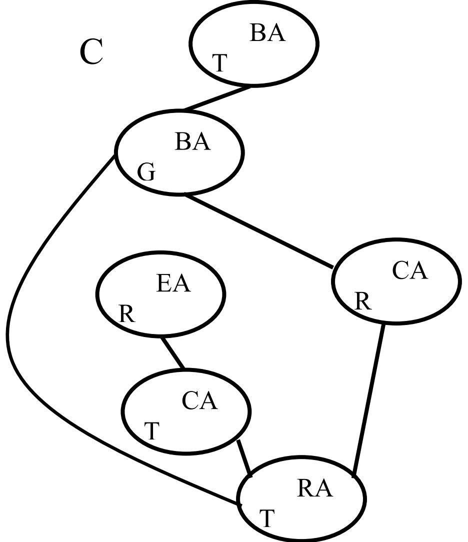
The rule for the connection is that any two words are connected that differ by exactly one letter.

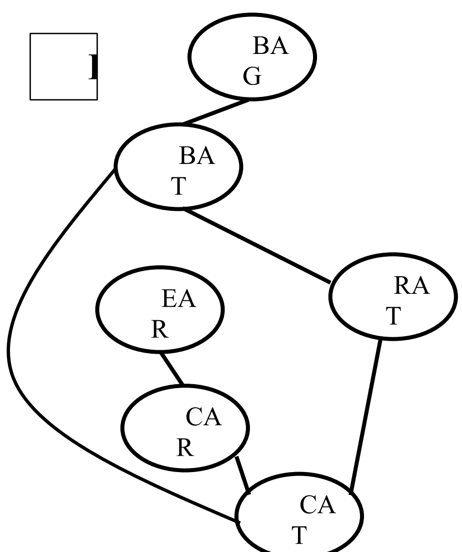
Question & Answer

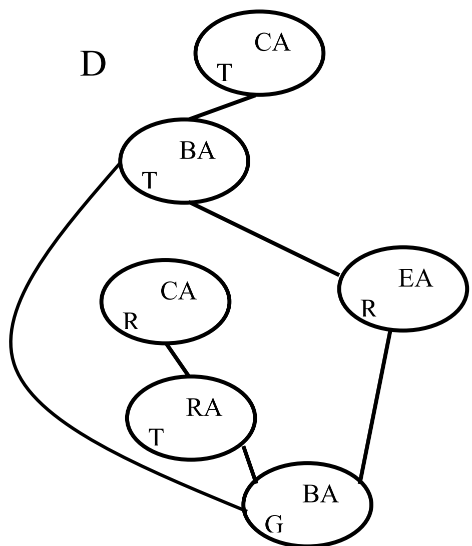
According to this rule, how do you have to rearrange the words that they fit in this rearrange picture?




A
E
RA
T
BA
T
CA
T
BA
G
CA
R


C
BA
T
BA
G
CA
R
EA
R
CA
T
RA
T


BA
G
BA
T
RA
T
EA
R
CA
R
CA
T


D
CA
T
BA
T
EA
R
CA
R
RA
T
BA
G

Task #4 - #415A4

Age Group: 11 + 12

Difficulty: A

Introduction / Presentation

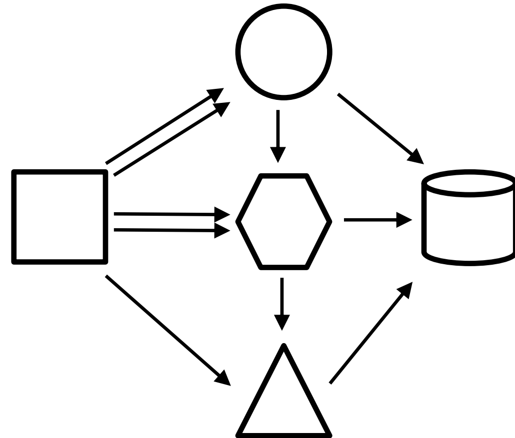
In the following can you see how objects convert in other objects. The rule is:

two squares convert into one circle

One circle and two squares convert into one hexagon

One hexagon and one square convert into one triangle

One circle, one hexagon, and one triangle convert into one cone



Question & Answer

How many squares do you need to create one cone?

A	5
B	10
C	11
D	12

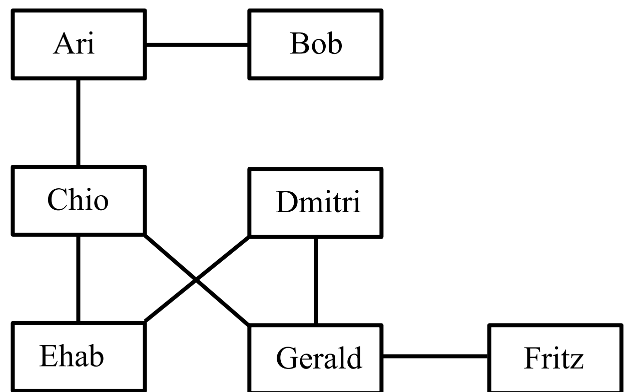
Task #5– #515B1

Age Group: 11 + 12

Difficulty: B

Introduction / Presentation

Imagine seven people who are active in an online social network called Selfiegram. Selfiegram only allows them to see the photos on their own and their friends' pages. In the following diagram, if two persons are friends they are joined by a line.



After a while everybody posts a picture of themselves on all of their friends' pages.

Question & Answer

Which persons' picture will be seen the most?

A	Ari
B	Bob
C	Chio
D	Dmitri
E	Ehab
F	Fritz
G	Gerald

Task #6 – #615B2

Age Group: 11 + 12

Difficulty: B

Introduction / Presentation

The Stack Computer is loaded with boxes from a conveyer belt. The boxes are marked with a number or an operator, that is +, -, * or /.

The computer is loaded until the top box is a box marked with an operator. This operator is then used on the two boxes below it. The three boxes are then fused into one single box and marked with the outcome of the calculation.

In the stack Computer, calculations are entered in a different way to normal calculator.

Examples:

2 + 3 must be entered as 2 3 +

10 - 2 must be entered as 10 2 –

5 * 2 + 3 must be entered as 5 2 * 3 +

5 + 2 * 3 must be entered as 5 2 3 * +

(8 - 2) * (3 + 4) must be entered as 8 2 - 3 4 + *

Question & Answers

How should the following computation be entered: 4 * (8 + 3) - 2?

Answer: 4 8 3 + * 2 -

However, the following answers are also acceptable as they all produce the correct output:

4 3 8 + * 2 -

8 3 + 4 * 2 -

3 8 + 4 * 2 -

These inputs all lead to the same result, even though the order of the operators and operations are not the same as intended in the given expression.

Task #7 – #715B3

Age Group: 11 + 12

Difficulty: B

Introduction / Presentation

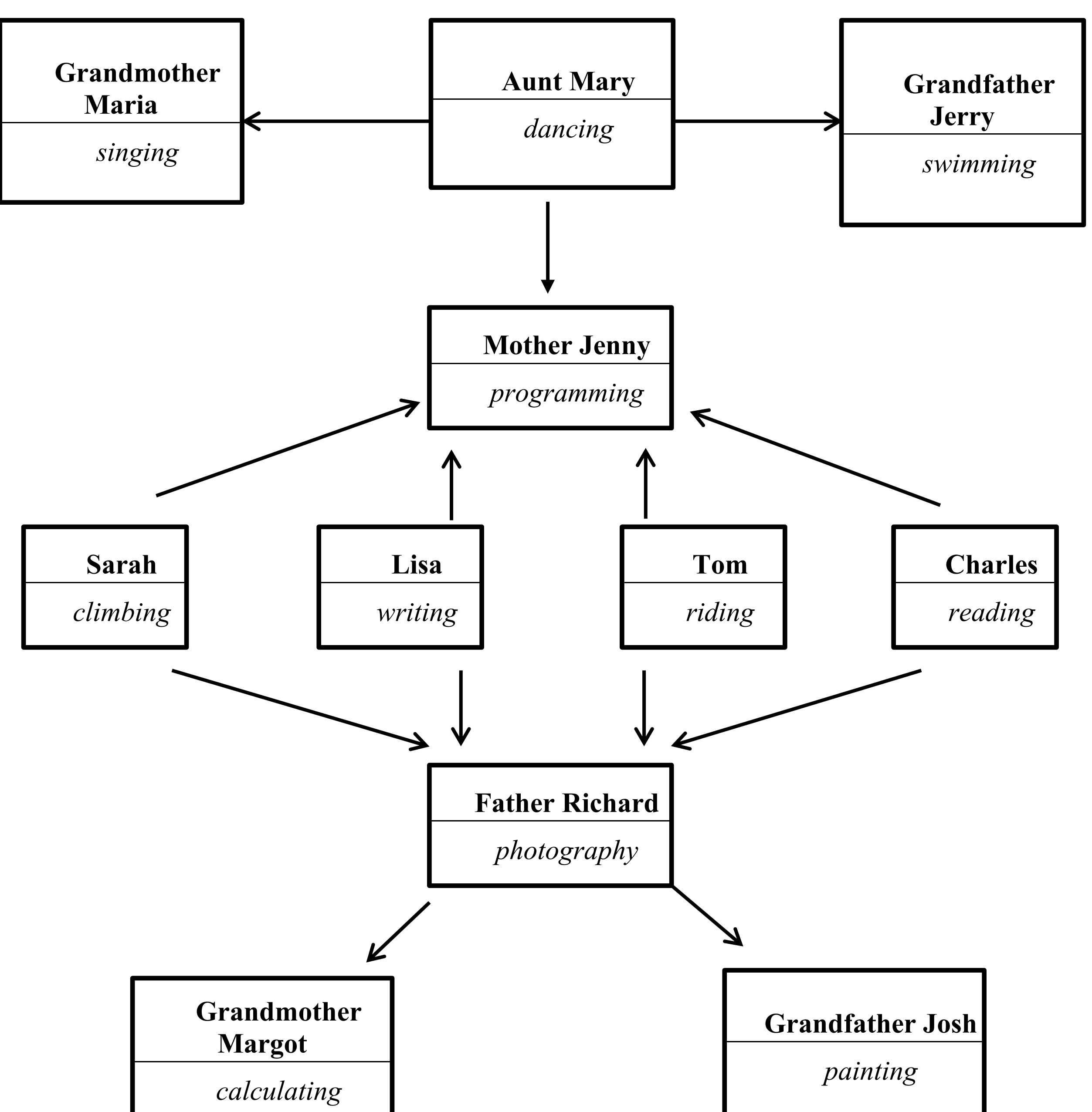
All members of a family have abilities. Imagine there is a rule for heritage as follows:

A daughter inherits all her abilities from her mother

A son inherits all his abilities from his father

Each family member also has one extra ability

The diagram below shows exemplary the relationship between family members. It also shows the extra ability for each person.



Examples:

Mother Jennifer has inherited the ability to sing from grandmother Maria, and she also has the ability to program.

Lisa inherits two abilities from her mother and also has the ability of writing. This means she can write, program and sing.

Question & Answer

Look at the diagram above. Which of these answers is true?

A	A. Tom's abilities are riding, painting and photography
B	B. Sarah has abilities in reading, programming and singing
C	C. Tom inherits from Grandmother Margot the ability to calculate
D	D. Aunt Mary has abilities in dancing and swimming

Task #8 – #815B4

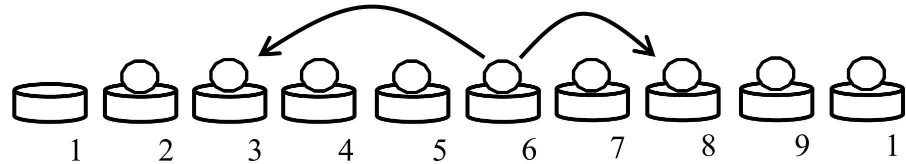
Age Group: 11 + 12

Difficulty: B

Introduction / Presentation

There are 10 plates in a row. There is one coin on each plate.

The aim is to collect all coins. You start at plate 1 and take the coin. After each single coin you take, you either go two plates forward, or backwards three plates (see figure as an example). You are not allowed to go back on an already empty plate.



Question & Answers

If you collect all 10 coins, which coin do you collect last?

2	3	4	5	6	7	8	9	10
							X	

Task #9 – #915C1

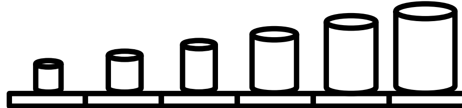
Age Group: 11 + 12

Difficulty: C

Introduction / Presentation

A factory produces sets of 6 bowls of different sizes. A long conveyor belt moves the bowls one by one, from left to right.

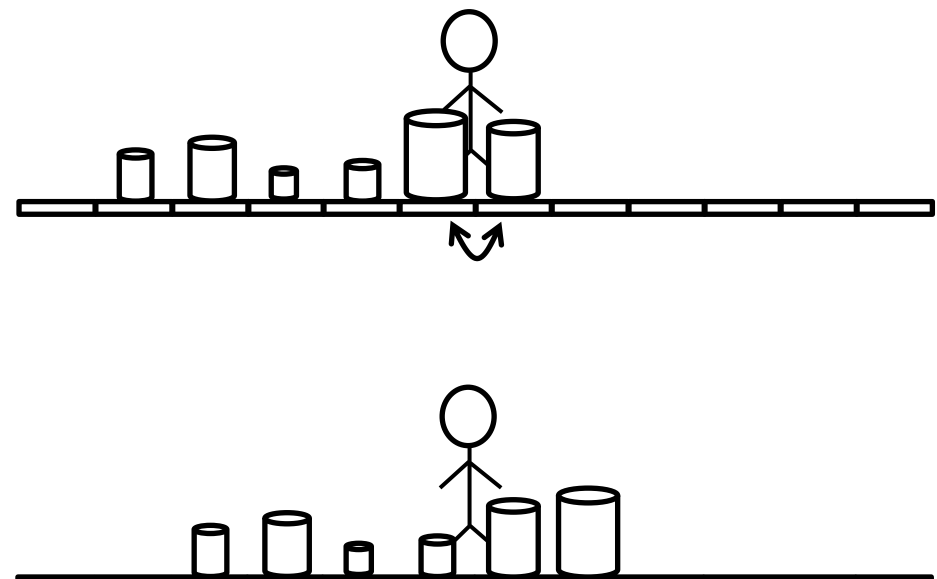
Bowl production places the 6 bowls of each set onto the conveyor belt in a random order. Before packing the bowls, they need to be sorted to look like this:

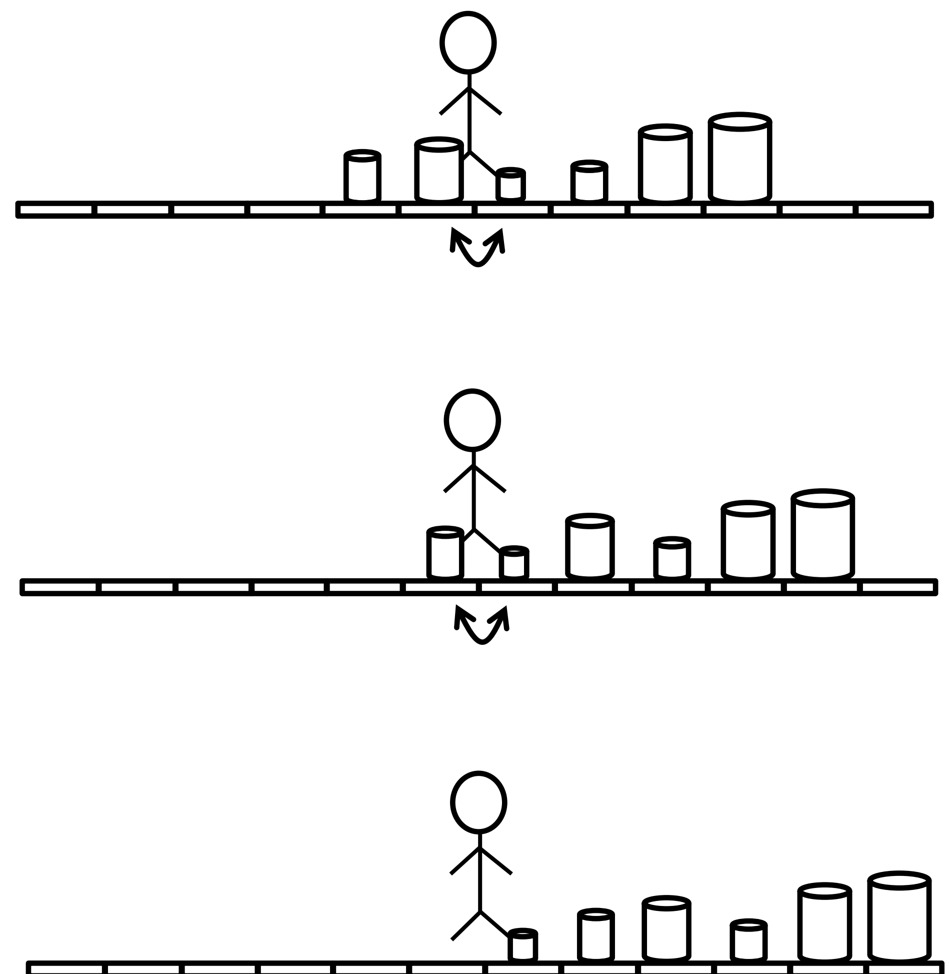


To help with the sorting, the factory places workers along the conveyor belt.

When a set of bowls passes a worker, they will swap any two neighbouring bowls which are in the wrong order. The worker will keep doing this until the set of 6 bowls has finished passing.

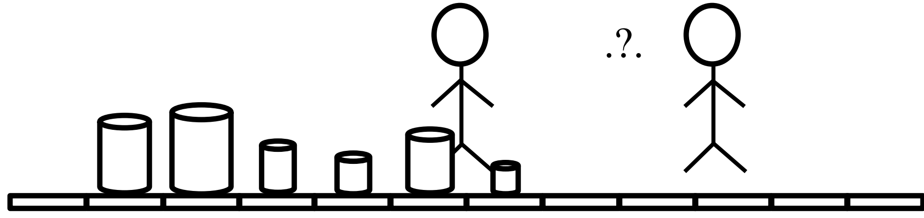
See how the order of a set of bowls changes as it passes one worker:





Question & Answer

How many workers should be put along the line to sort the set of bowls on the right?



Answer: 4

Task #10 – #1015C2

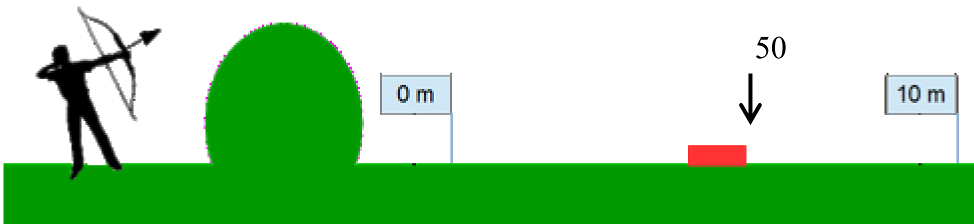
Age Group: 11 + 12

Difficulty: C

Introduction / Presentation

Arnaud would like to reach a target with his arrow. He can adjust the arc to shoot an arrow in a range between 0 m and 10 m.

The position of the target is unknown, but after each shoot, his friend Marc tells Arnaud whether the arrow reached the ground before or after the target.



Question & Answer

Given that the target has width of 50 cm, what is the minimal number of arrows needed to be sure to hit the target, no matter where it is located?

A	3
B	4
C	5
D	6

Task #11 – #1115C3

Age Group: 11 + 12

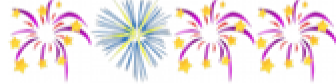
Difficulty: C

Introduction / Presentation

Imagine the Sydney Friday firework has a special meaning. Furthermore, there are two kinds of rockets and every composition of sequence of both rockets stands for a different word. In the following picture you see the meaning of five different sequences of rockets.

Word	Code
Log	
Tree	
Rock	
River	
Food	

For example, to send the message “food, log, food”, you have to shoot:



Question & Answer

How many different meanings can the following sequence of fireworks have?

Answer: 4

Task #12 – #1215C4

Age Group: 11 + 12

Difficulty: C

Introduction / Presentation

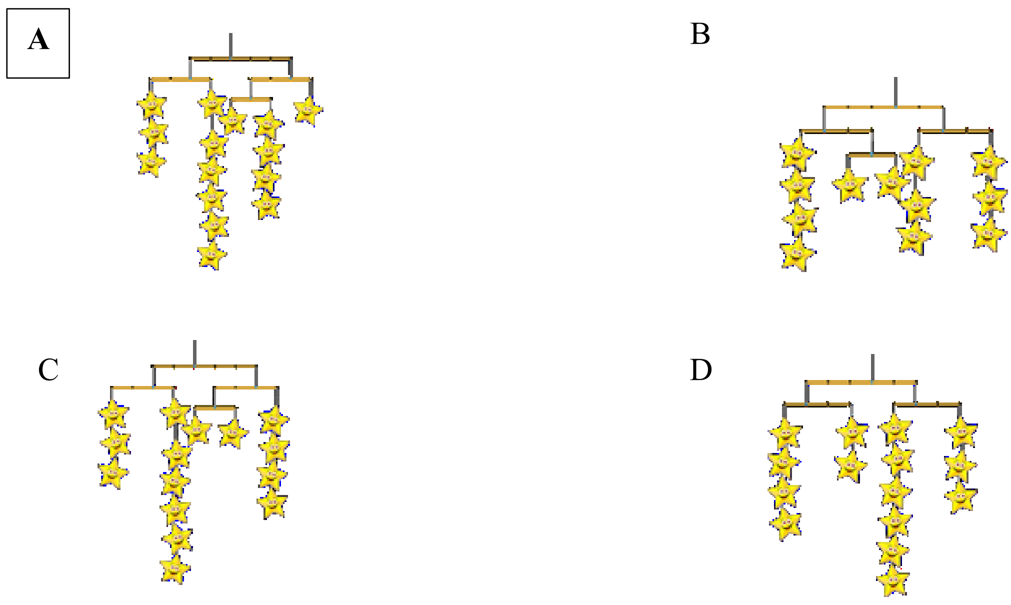
A mobile is a piece of art that hangs from the ceiling. You may remember one hanging from the ceiling in your room. A mobile consists of sticks and figures. Each stick has a few points to which figures or other sticks may be attached. Also, each stick has a hanging point, from which it is attached to a stick further above (or to the ceiling). The following example mobile can be described using these numbers and brackets:

(-3 (-1 1) (1 1)) (2 3)

Question & Answer

Which of the following mobiles could be constructed using these instructions?

(-3 (-1 4) (2 (-1 1) (1 1))) (2 (-1 6) (2 3))



Task #13 – #1315C5

Age Group: 11 + 12

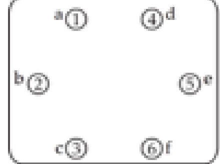
Difficulty: C

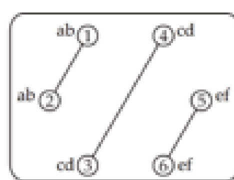
Introduction / Presentation

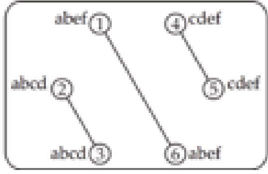
Every Friday, six spies exchange all the information they have gathered during the week. A spy can never be seen with more than one other spy at the same time. So, they have to have several rounds of meetings where they meet up in pairs and share all the information they have at that point.

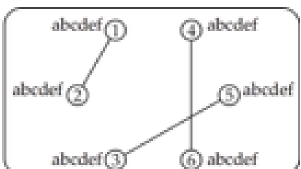
The group of six spies needs only three rounds to distribute all their secrets:

Before the meeting each spy holds a single piece of information. Spy 1 knows 'a', spy 2 knows 'b', etc. In the first round spies 1 and 2 meet and exchange information so now both know 'ab'. The diagrams show which spies meet in each round with a line. It also shows which pieces of information they all have. After three rounds all information has been distributed.





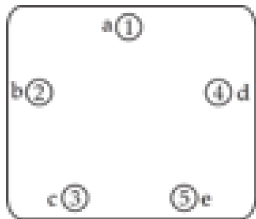




Question & Answer

Which of the following statement is true?

After an international incident one spy has stopped attending the meetings. What is the minim number of rounds needed for the five remaining spies to exchange all information?



Answer: 4

Task #14 – #1414A1

Age Group: 11 + 12

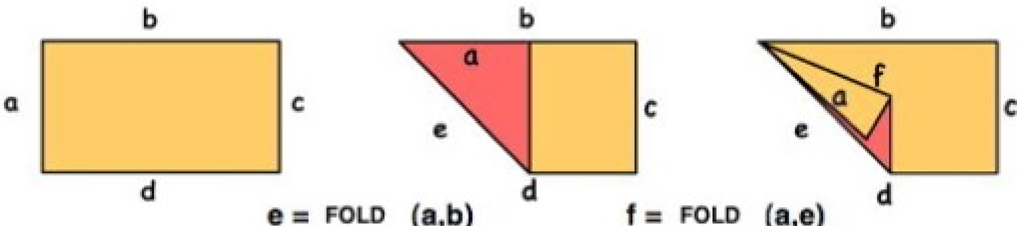
Difficulty: A

Introduction / Presentation

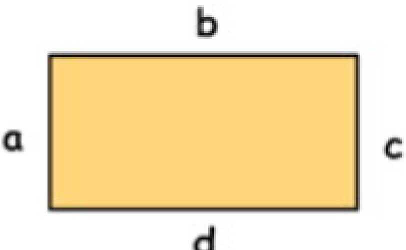
Imagine someone have come up with a language for describing how a piece of paper should be folded. The commands in this language are called FOLD.

z = FOLD(x,y) for example means:
Fold the piece of paper in such a way that side x and side y overlap. This way, a new side is created. We call this side z.

An example with two consecutive commands:



Imagine a rectangle-shaped piece of paper of which side b is twice as long as side a.



You are allowed to turn the piece of paper over.

The following sequence of commands is executed:

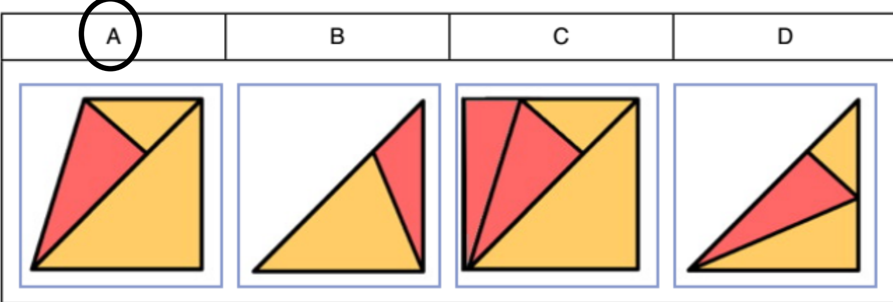
e = FOLD(c,a)

f = FOLD(c,d)

g = FOLD(a,f)

Question & Answer

What will the piece of paper look like afterwards?



Task #15 – #1514A2

Age Group: 11 + 12

Difficulty: A

Introduction / Presentation

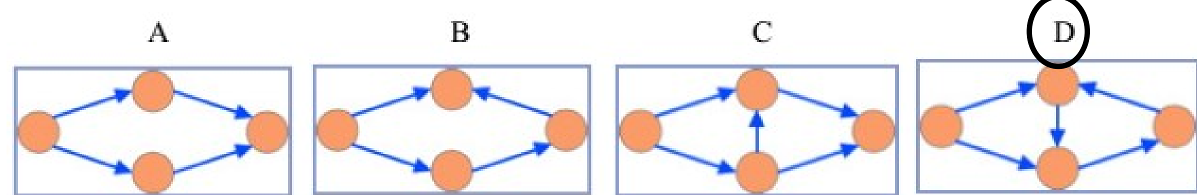
For a group assignment a class is split up into four groups. Each group divides the different tasks between the group members. Three groups manage to finish the complete assignment but one group fails to do so.

What happened?

The most group members have to wait for other members before they can start with their own tasks. You see below a diagram for each group to show the dependencies between students in each group. A circle represents a student. An arrow from student 1 to student 2 means that student 2 has to wait for student 1 to finish their tasks.

Question & Answer

Which diagram represents the group that did not finish the assignment?



Task #16 – #1614A3

Age Group: 11 + 12

Difficulty: A

Introduction / Presentation

A group a scientists have come up with a secret code for encrypting messages, no nobody else can read them.

In their secret code, the vowels (A, E, I, O, U) and the punctuation remain unchanged. The consonants are replaced by the next consonant in the alphabet where Z becomes B.

Question & Answer

How would you write "GIVE ME A CALL" as a secret code?

A	GOVE MI E CELL
B	FITE LE A BAKK
C	HOWE NI E DEMM
D	HIWE NE A DAMM

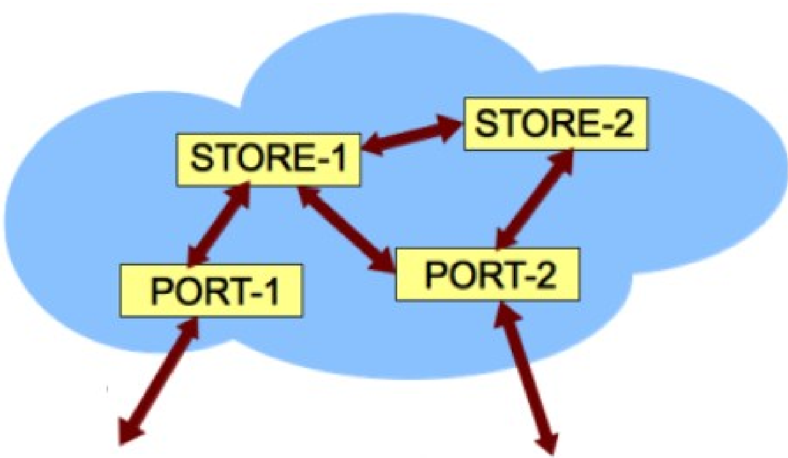
Task #17 – #1714B1

Age Group: 11 + 12

Difficulty: B

Introduction / Presentation

Imagine data are stored in a cloud containing four servers. The image shows all the connections between the servers.



To lower the risk of losing data, all data are stored on both STORE-1 and STORE-2. To increase the accessibility, all data are available through both PORT-1 and PORT-2.

No data is stored on PORT-1 and PORT-2.

Question & Answer

Which statement is FALSE?

A	If STORE-1 and PORT-2 crash, all data become inaccessible.
B	If PORT-1 and PORT-2 crash, all data become inaccessible.
C	If STORE-1 and STORE-2 crash, all data are destroyed.
D	If PORT-1 and PORT-2 crash, all data are destroyed.

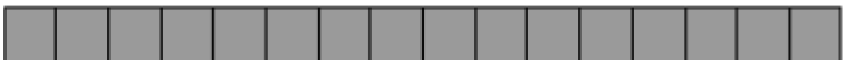
Task #18 – #1814B2

Age Group: 11 + 12

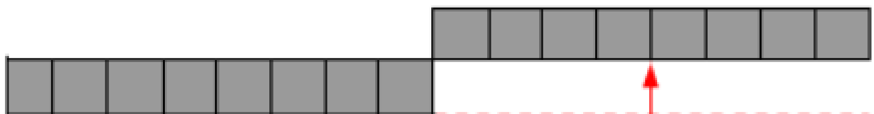
Difficulty: B

Introduction / Presentation

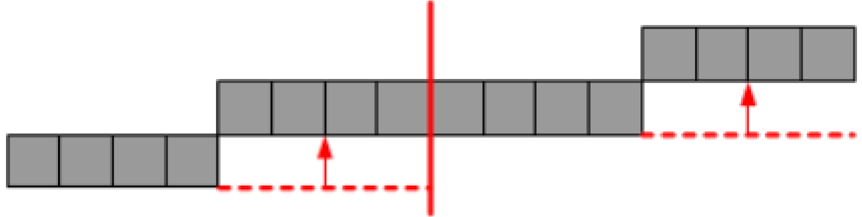
A paper strip is divided into 16 equal pieces:



Such a strip be used for “half-sliding”. This is one by spotting the strip half and sliding the right half up:



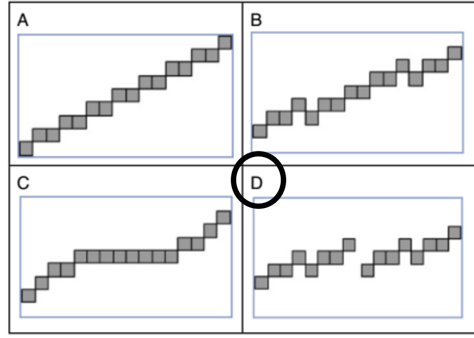
The two halves are also split in half and again, both right halves are slid up.
This would be look like this:



We do this again with the four-piece strips and, after that, with two piece strips.

Question & Answer

What will the final result look like?



Task #19 – #1914B3

Age Group: 11 + 12

Difficulty: B

Introduction / Presentation

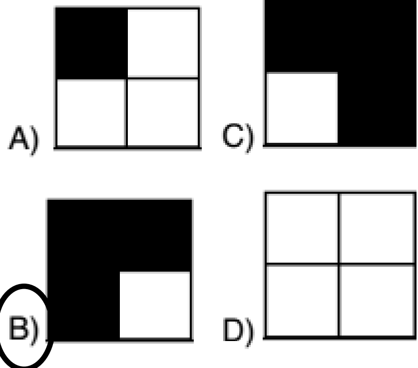
Imagine you receive a message sent on a 6 x 6 grid. Unfortunately, part of the message has been destroyed (the red coloured squares).

	1	2	3	4	5	6
1						
2						
3						
4						
5						
6						

However, the additional squares help to determine the message. Each square in column 6 is coloured such that the number of black squares in each row is even. Similarly, each square in row 6 is coloured such that the number of black squares in each column is even.

Question & Answer

Which of the following images could be the pattern underneath the red squares?



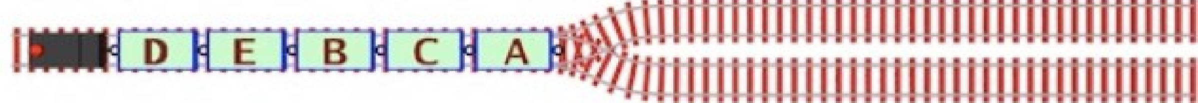
Task #20 – #2014C1

Age Group: 11 + 12

Difficulty: C

Introduction / Presentation

The wagons of the freight train from the Railroad company are placed in the order D-B-C-A:



The locomotive can move forwards and backwards and is able to pull and push an unlimited number of wagons. Connecting or de-connecting a wagon is called ONE railroad operation. Moving alone is *not* considered as a railroad operation.

Question & Answer

How many railroad operations are necessary to put the wagons in the order A-B-C-D-E?

A	6
B	8
C	10
D	12

Task #21 – #2114C2

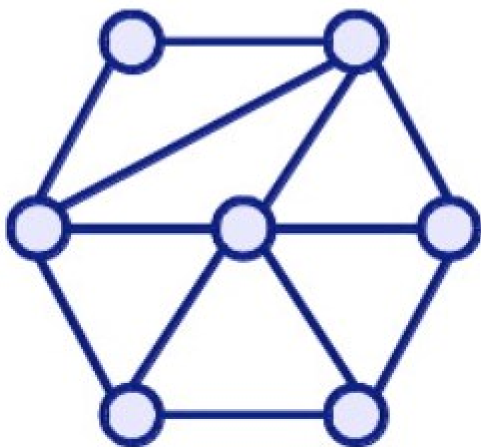
Age Group: 11 + 12

Difficulty: C

Introduction / Presentation

Neighbourhoods in areas on maps can be presented as a diagram. In such a neighbourhood diagram each neighbourhood is represented by a node.

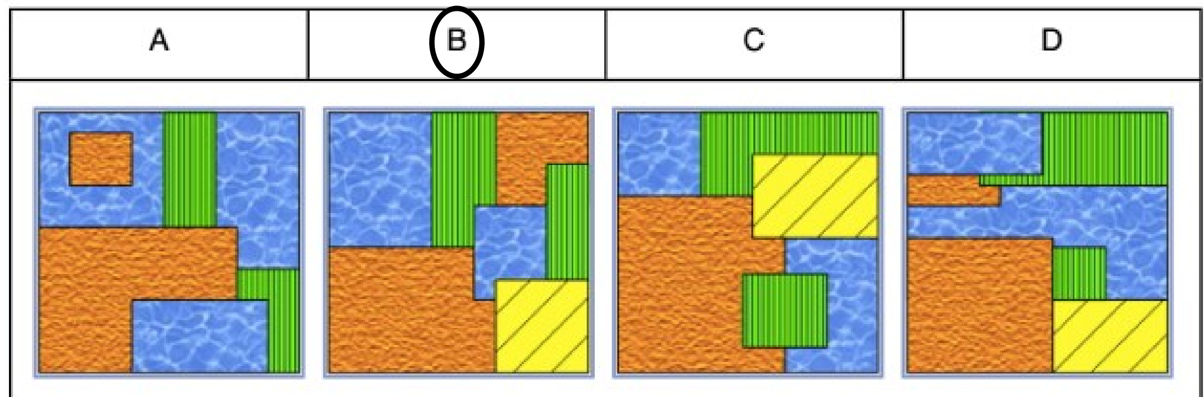
A line between two nodes means that the two neighbourhoods share one or more borders.



The diagram on the right shows the connections between seven neighbourhoods in a certain area.

Question & Answer

Which of the following maps is described by the diagram?



Task #22 – #2214A4

Age Group: 11 + 12

Difficulty: A

Introduction / Presentation

This picture shows an ancient ocarina in duck-shape. This duck-ocarina is a special musical instrument and has only six different tones.

And: after a tone is played, only the same tone or the tone directly above or below it can be played. Therefore, a melody for the ancient ocarina can be written with only three different symbols:

0 means “play the same tone again”.

+ means “play the next tone above it”.

- means “play the next tone blow it”.

Question & Answer

Which of these melodies can NOT be played with this ocarina?

A	(+000+000+000+000+)
B	(-----0+++++0-----)
C	(---0+-0--000+)
(D)	(--+--+--0-+--)

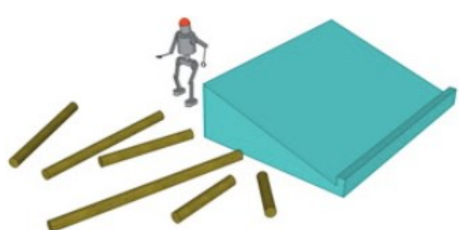
Task #23 – #2314B4

Age Group: 11 + 12

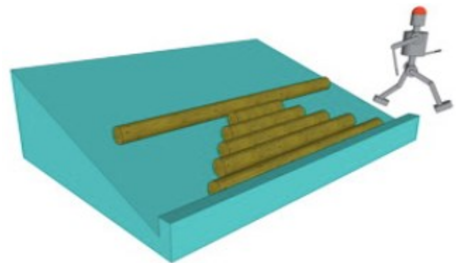
Difficulty: B

Introduction / Presentation

There is a robot to sort tree trunks. On the ground, there are several tree trunks of different lengths.



The robot chooses a tree trunk following certain formula, lays it on top of the ramp and lets it roll down. He repeats this, until there are no more tree trunks on the ground as you can see in the following picture:



Question & Answer

Which formula does the robot use to decide in which order the tree trunks have to be placed on the ramp?

A	Take the longest tree trunk.
B	Take the shortest tree trunk.
C	Take the second longest tree trunk. If only one trunk remains, take that one.
D	Take the second shortest tree trunk. If only one trunk remains, take that one.
E	Take the longest tree trunk first and the shortest tree trunk last.
F	Take the shortest tree trunk first and the longest tree trunk last.

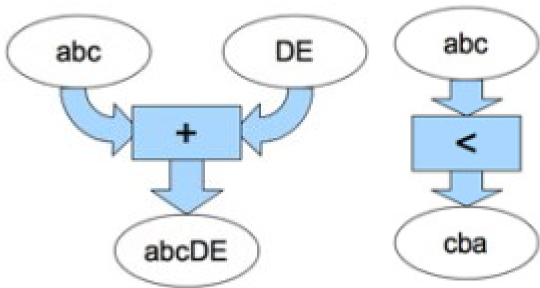
Task #24 – #2414B5

Age Group: 11 + 12

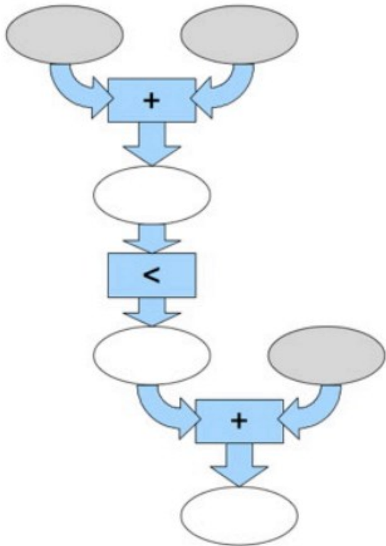
Difficulty: B

Introduction / Presentation

Imagine there are two types of text machines:



A + machine (left) takes two pieces of text and joins them. A < machine (right) takes one piece of text and reverses it. By linking both machines to each other we get a more complex text machine (below). It needs three pieces of text (in grey ellipses) and writes text in the white ellipses.



This complex machine needs three texts to work on (grey ellipses), processes them, and gives one text as the result of its work in the bottommost ellipse.

Question & Answer

Which three text pieces do you need to put in this text machine in order to get the text INFORMATION in the lowest ellipse?

A	AMR OFNI TION
B	INF ORMA TION
C	AMR OFNI NOIT
D	FNI AMRO NOIT

Task #25 – #2514C3

Age Group: 11 + 12

Difficulty: C

Introduction / Presentation

The medical records of patients contain personal data, which should not be made public. For the publication of a research project, the hospital has made some data public, without mentioning the names of the patients. The table on the left shows a part of this list.

Because of the upcoming elections, the city with postcode M1 1AA publishes a list with all constituents at the same time. This table on the right shows the constituents who are born on January 1st.

Date of birth	Gender	Postcode	Case		Date of birth	Gender	Name
01/01/1974	male	B33 8TH	Diabetes		01/01/1958	female	Melanie Meyer
01/01/1976	male	M1 1AA	Lung cancer		01/01/1976	male	George Smith
01/01/1976	female	CR2 6XH	Breast cancer		01/01/1976	male	Robert Jones
01/01/1976	female	AB1A 1GD	Miscarriage		01/01/1984	female	Catharine Free
01/01/1984	female	DN55 1PT	Heart attack		01/01/1984	female	Eve Miller
01/01/1985	female	W1A 1HQ	Breast cancer		01/01/1988	female	Anne Beaver
01/01/1987	female	EC1A 1BB	Skin cancer		01/01/1988	male	Roman Peterson
01/01/1988	male	M1 1AA	Diabetes		01/01/1988	female	Isabelle Bourne
01/01/1988	female	M1 1AA	Influenza		01/01/1989	male	Martin Klaus

Thanks to these two tables, you know for sure that one of the persons on the right has a disease and you also know which disease it is.

Question & Answer

What is the name of this person?

A	George Smith
B	Roman Peterson
C	Eve Miller
D	Isabelle Bourne

Task #26 – #2614C4

Age Group: 11 + 12

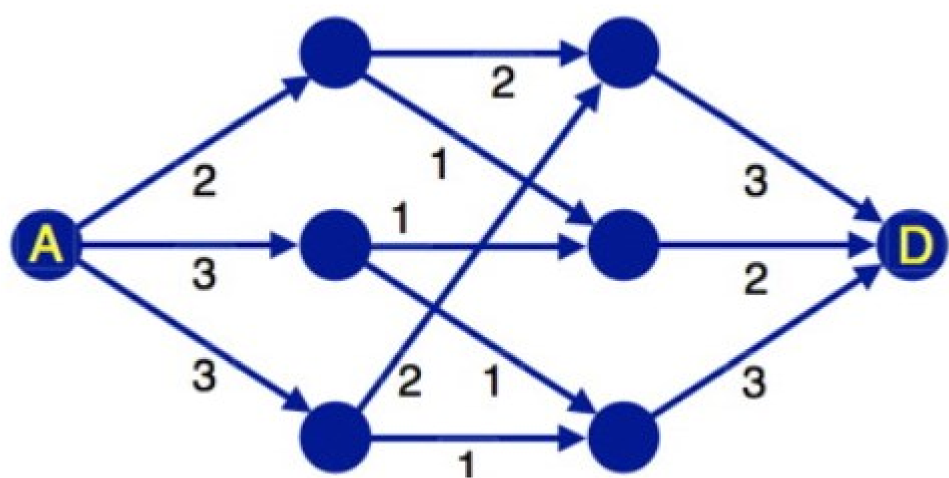
Difficulty: C

Introduction / Presentation

In forest (A) is an area where beavers fell trees for their dams. They transport the tree trunks to their new dam (D) - through an infrastructure of channels.

The arrows represent the channels; the dots are points where the water splits up or comes together.

Every channel has a restricted capacity. The numbers next to the channels show how many tree trunks can be transported through the channels in one minute, as you can see in the picture below.



Question & Answer

How many tree trunks can be transported from A to D at most in one minute?

Answer: 7

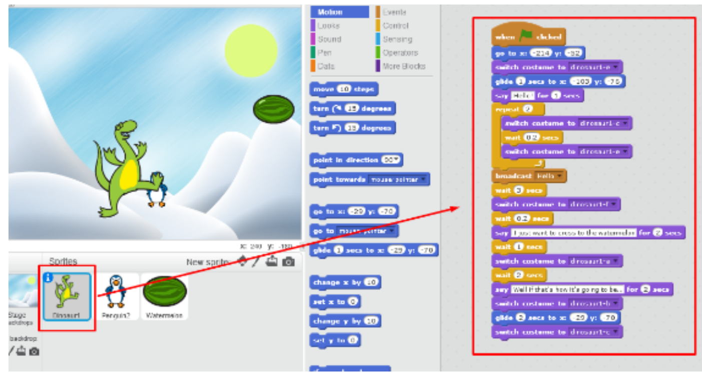
Appendix C: Programming rubric scheme

Programming Concept	Not evident 0	Poor 1	Satisfactory 2	Good 3	Excellent 4
Extent / Richness (1.0 = 20 %) *"What and how much is happening in the Scratch product?"*	• Overall picture: nothing is happening	• Overall picture: There is something happening/ working, but it is not clear what • There are only up to a couple sprites and they have only up to a couple code chunks or do not have any • The program has barely moving, or changing, or counting, switching, or sound making elements	• Overall picture: **1 thing** is happening • There are only up to a couple and they have a few code chunks • The elements of the program are **mainly** moving, or changing, or counting, switching, or sound making elements	• Overall picture: mainly **2 things** are happening • There are quite many sprites and the most of them have a substantial code chunks • The elements of the program do **mainly** 2 things, for instance: → moving and changing, or → moving and counting, or → making sound and changing	• Overall picture: mainly **> 2 things** are happening • There are a quite many sprites and all have a decent amount of code chunks • The elements of the program do > 2 things, for instance: → moving and changing and counting, or → moving and counting and making sound, or → moving and switching and counting and making sound
Variety / Scratch or Coding Usage (1.0 = 20%) *"How much of Scratch do they use?"*	• No evident useful use of any code chunks	• The program has up to a couple code chunks • Only low-level code chunks • No high-level code chunks	• The program has a few code chunks • Many low-level code chunks but essentially **of the same kind**; not many different code chunks • Up to a couple high-level code chunks • Reasonable usage of Scratch's possibilities	• Overall picture: the program has several code chunks • Many **different** low-level code chunks → A few motion and a few looks code chunks → A few motion and a few looks and a few sound code chunks • A few more high-level code chunks but **of the same kind**; not many different code chunks • Reasonable usage of Scratch's	• Overall picture: the program has many code chunks • Many different low-level code chunks • Many different high-level code chunks • All main code chunks (motion, looks, sound, data, events, control, sensing, operators) are used

				possibilities	
Programming Concept	Not evident 0	Poor 1	Satisfactory 2	Good 3	Excellent 4
Organisation (0.5 = 10%) *"How messy or clean does the work space look?"*	• It looks very messy • No kind of organisation is evident • Many dead listings (relatively speaking to the total amount of listings)	• It looks somehow messy • Up to a couple of listings are ordered • Listings are not in any meaningful order • Many dead listings (relatively speaking to the total amount of listings)	• It looks somehow tidy • The most listings are ordered but not necessarily meaningfully • Several dead listings (relatively speaking to the total amount of listings)	• It looks pretty tidy • All listings are in a meaningful order • Up to a couple or a few dead listings (relatively speaking to the total amount of listings)	• It looks very organised • All listings are in meaningful order AND aligned with screen • No or only up to a couple dead listings (relatively speaking to the total amount of listings)
Functionality (1.0 = 20%) *"How <u>well</u> do they use Scratch?"*	• The intention is unclear • Nothing works	• The intention is clear, but the program does not work as intended, for instance: → Sprites do not move correctly → If it is a game, it is unplayable; if it is a story, the plot is unclear	• The intention is clear; in general, the program works as intended but has some problems, for instance: → You have to set some sprites manually → Sprites which are supposed to be hidden are not hidden	• The intention is clear; in general, the program works as intended but with some minor flaws, for instance: → Sprites which are supposed to move can move but are moving too fast or too slow → Appearing text is too fast; not enough time to read a text	• The intention is clear; it works as intended without any flaws; for instance: → The speed of moving sprites is reasonable → There is enough time to read text → Moving and shooting works as intended
Efficiency (1.5 = 30%) *"How well developed is their control flow?"*	• No evident use of efficient use of control code chunks	• Many code chunks and listings are essentially copy-pasted (not only the similar kind but actually copy-pasted) • No or up to a couple control code chunks	• A few code chunks and listings are copy-pasted • A few control code chunks	• Up to a couple code chunks and listings are copy-pasted • Several control code chunks; reasonable usage of control chunks	• No *unnecessary* duplications • Comprehensive and complete use of control code chunks • *(If there are many iterations f chunks it can't be "excellent")*

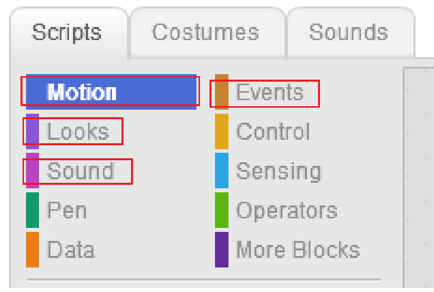
<table>
<tr><th colspan="2">Glossary</th></tr>
<tr>
<td>

Scratch product: All sequences from all sprites together form the Scratch product. Students either programmed a story or a game.

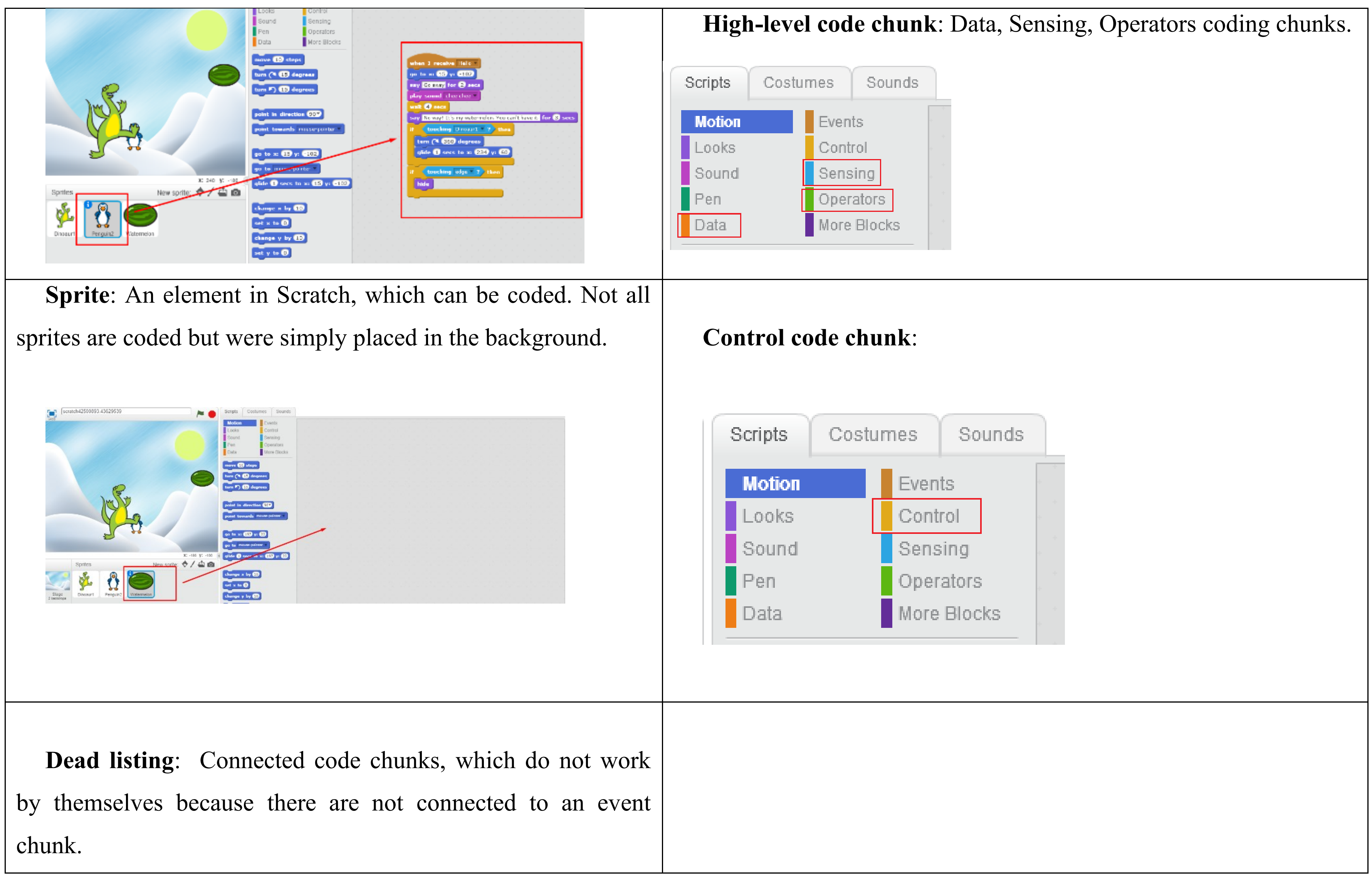


</td>
<td>

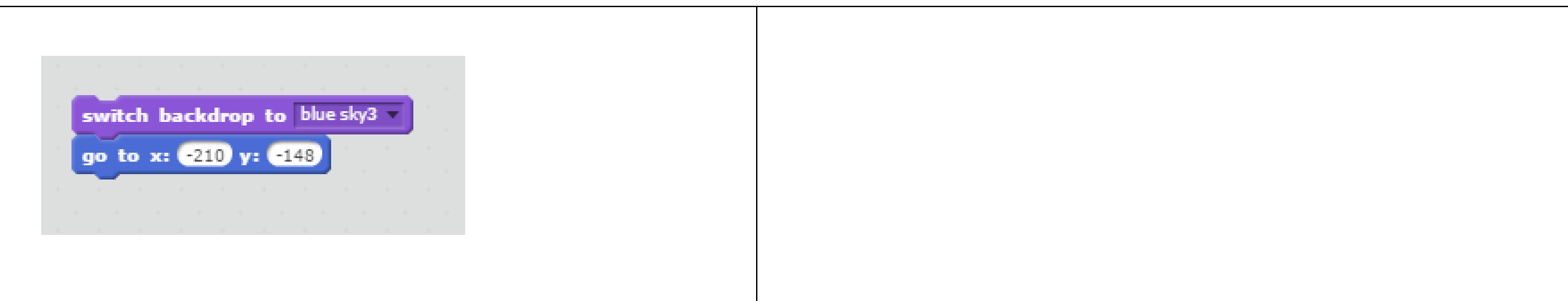
Low-level code chunk: Motion, Events, Looks, Sound coding chunks



</td>
</tr>
</table>



	High-level code chunk: Data, Sensing, Operators coding chunks.
Sprite: An element in Scratch, which can be coded. Not all sprites are coded but were simply placed in the background.	**Control code chunk**:
Dead listing: Connected code chunks, which do not work by themselves because there are not connected to an event chunk.	


switch backdrop to blue sky3
go to x: -210 y: -148

Appendix D: Explanation of ICC for within-variables

Based on formula in Kenny et al. (2006, p. 34):

The measurement of two members of pair *i* is donated as X_{1i} and X_{2i}. There are a total of *k* pairs. The overall average is donated as *M*. Let

$$d_i = X_{1i} - X_{2i}$$

and

$$m_i = \frac{X_{1i} + X_{2i}}{2}$$

Thus, *d* represents the difference and *m* the average of both measurements per pair. The mean square between pairs is defined as

$$MS_B = \frac{2\sum(m_i - M)^2}{k - 1}$$

and the mean square within pairs is defined as

$$MS_W = \frac{\sum d_i^2}{2k}$$

The ICC as used in this study is defined as

$$ICC = \frac{MS_B - MS_W}{MS_B + MS_W}$$

Appendix E: Formula for Z-score for transition probability

Based on Bakeman et al. (2011, p. 105):

R	number of rows (given)
C	number of columns (targets)
x_{rc}	observed joint frequency for cell in r-th row and c-th column of a $R \times C$ table
$x_{\cdot c}$	sum of the counts in the c-th column
$x_{r\cdot}$	sum of the counts in the r-th row
$N = x_{\cdot\cdot}$	number of counts total for a $R \times C$ table
$p_c = \frac{x_{\cdot c}}{N}$	probability for the c-th column
$p_r = \frac{x_{r\cdot}}{N}$	probability for the r-th row
$e_{rc} = p_c \times x_{r\cdot}$	expected frequency by chance
g_r	code for the r-th row (the given)
t_c	code for the c-th column (the target)
$P(t_c \vert g_r) = \frac{x_{rc}}{x_{r\cdot}}$	conditional probability of t_c given g_r

$$Z_{rc} = adjusted\ residuals = \frac{x_{rc} - e_{rc}}{\sqrt{e_{rc}(1 - p_c)(1 - p_r)}}$$

The adjusted residuals follow a normal distribution and are, therefore, referred as Z-scores.

Appendix F: Distributions of variables for visual inspection

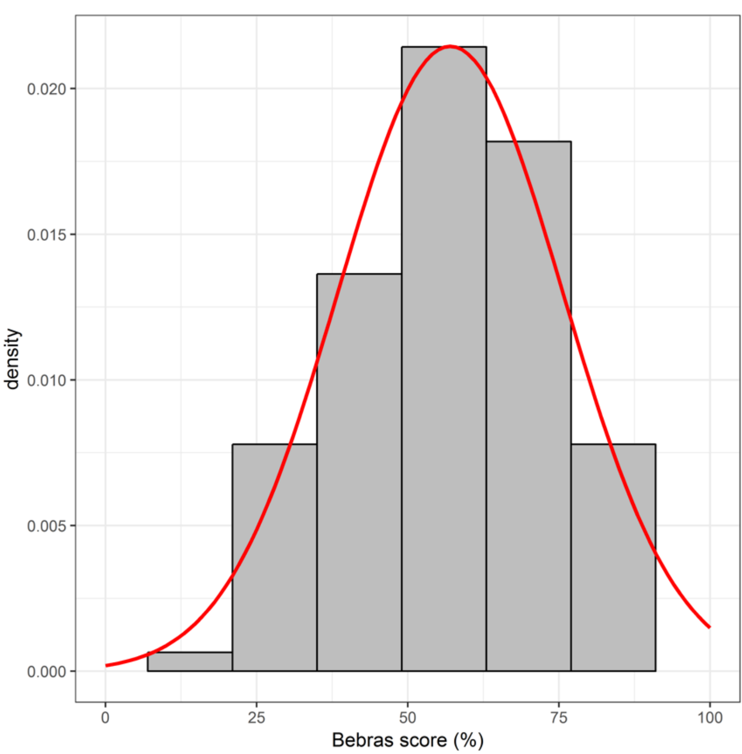


Figure F.1. Distribution of percentage of achieved scores in the Bebras tasks (units are individuals)

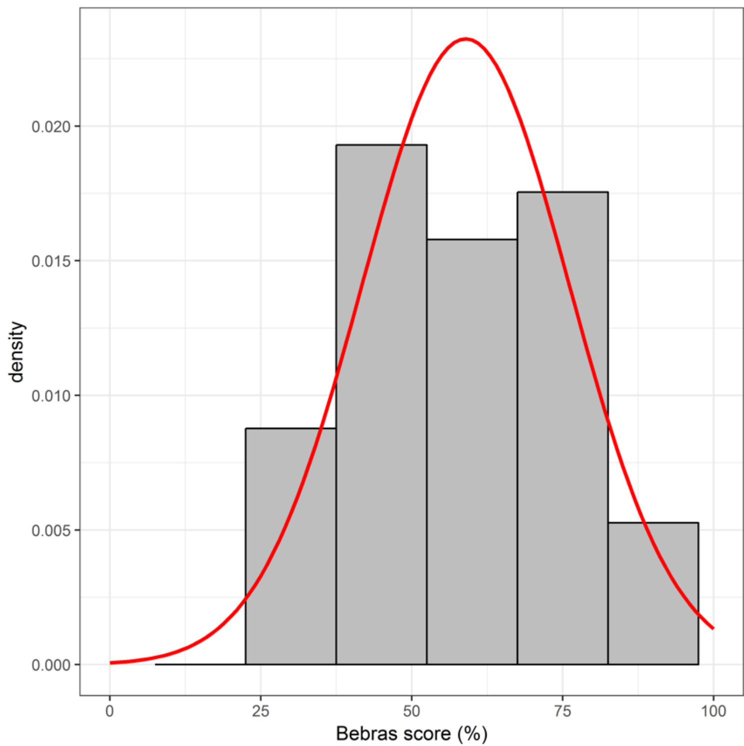


Figure F.2. Distribution of percentage of achieved scores in the Bebras tasks (units are pairs)

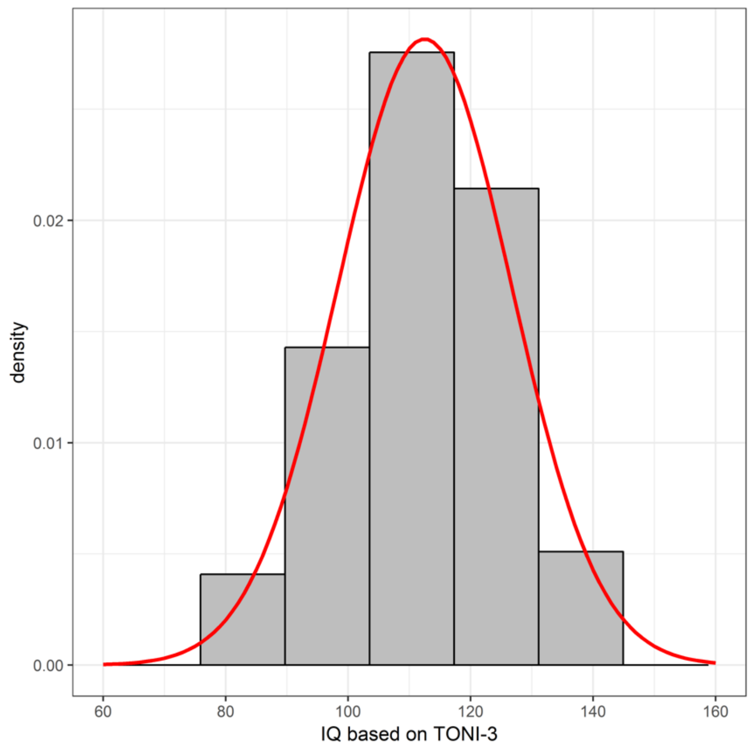


Figure F.3. Distribution of IQ based on TONI-3 (units are individuals)

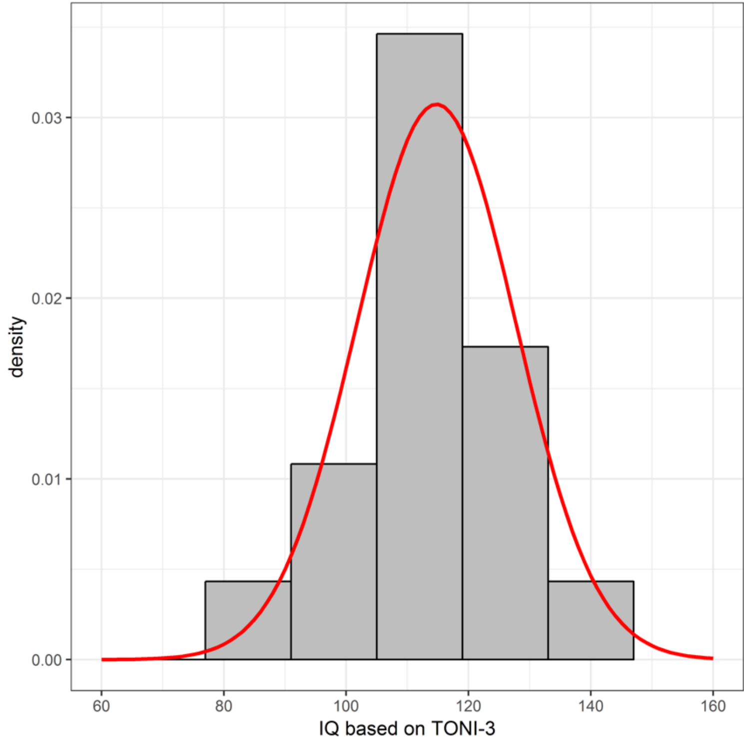


Figure F.4. Distribution of IQ based on TONI-3 (units are pairs)

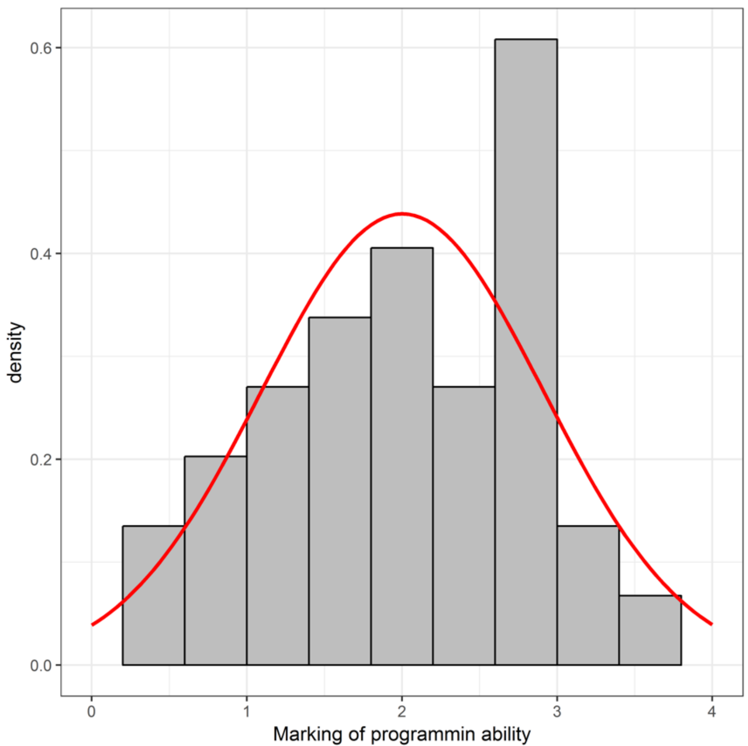


Figure F.5. Distribution of weighted means of programming quality rubric scheme (units are pair)

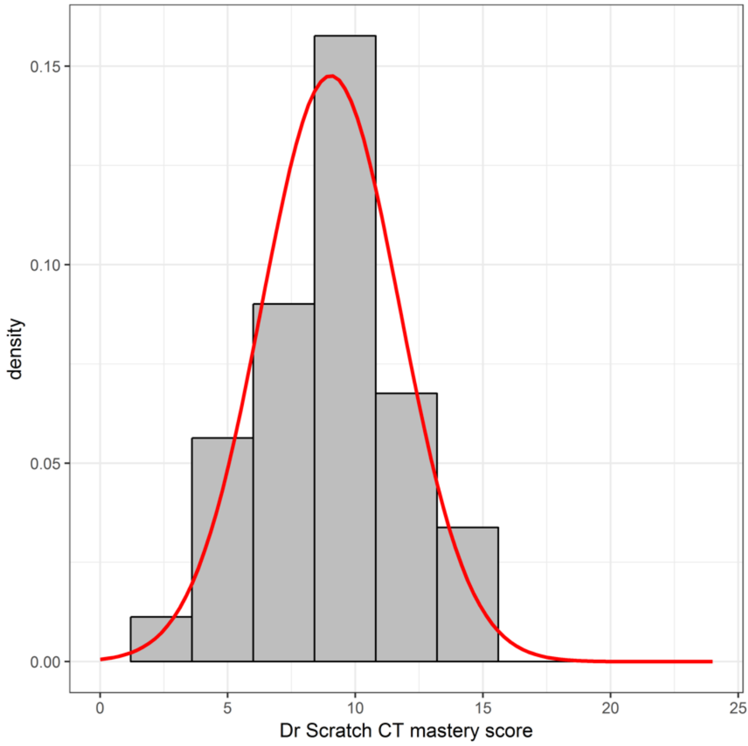


Figure F.6. Distribution of achieved CT mastery score based on Dr Scratch (units are pairs)

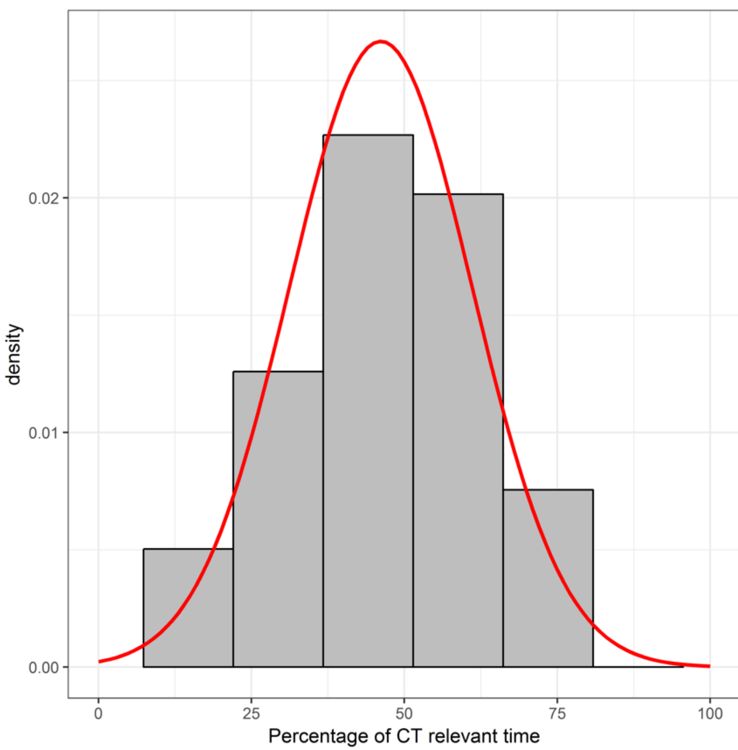


Figure F.7. Distribution of percentage actions and saying linked to CT (units are pairs)

Appendix G: Detailed item analysis for task difficulty

The average Bebras score did not raise any concerns that the test was too easy or too difficult, although it was slightly higher than 50 %. To check whether the Bebras tasks were indeed able to cover the whole spectrum of CT, the item difficulty index (*D*) for each Bebras task was examined. The item difficulty index shows the proportion of participants who managed solve the task and ranges between 0.00 and 1.00. According to Nwadinigwe and Naibi (2013) and Thompson (2017), an as hard labelled item has an expected probability to be solved (Δ) between .00 to .29, a medium-difficulty item lies between .39 and .69, and easy labelled items between .70 to 1.00. A large number of very easy items or a large number of very difficult items indicate floor and ceiling effects, respectively. Ceiling and floor effects reflect that the instrument did not cover the whole spectrum of the measured construct, which causes problems for further analyses.

Table G.1 contains an overview of all used Bebras tasks and the theoretically expected probabilities for solving items labelled as easy, medium, and hard, and the actual frequency of participants who were able to solve these tasks. Comparing the expected probabilities and their empirical counterparts reveals that four tasks were solved by fewer participants than expected (i.e., these tasks were unexpectedly more difficult) and eight tasks were solved by more participants than expected (tasks were unexpectedly easier). Eight tasks were solved by an expected number of participants. The range of solved items lies between 36 % and 90 %. Overall, the Bebras tasks were slightly easier than expected. This is true especially for tasks labelled as hard. Nevertheless, the effect is not big enough to identify serious ceiling effects for any items. In conclusion, the whole spectrum of CT was sufficiently covered by the Bebras tasks for this sample.

Table G.1

Overview of Solved Bebras Tasks

#	Name of Bebras task	Δ	*D*	Δ - *D*
	easy			
1	Drawing stars		.81	0
3	Word chains		.90	0
6	You won't find it		.90	0
11	Folding paper	.70 – 1.00	.36	+
13	Group assignment		.56	+
14	Beavers secret code		.82	0
18	Bebrocarina		.48	+
	medium			
4	Beaver the alchemist		.36	+
7	Stack computer		.72	-
8	Super power family (hard)		.84	-
9	Kangaroo		.59	0
10	Beaver cloud	.39 – .69	.52	0
19	Sorting tree trunks		.78	-
17	Popularity		.57	0
20	Text machine		.62	0
	hard			
2	Spies		.49	-
5	Reaching the target		.45	-
12	Fireworks (hard)	.00 – .29	.39	-
15	Freight train		.37	-
16	Neighbourhoods		.42	-

Note. Δ = theoretical probability to solve this item. *D* = frequency how often this item was solved. A Difference of 0 indicates that this task was solved by expected number of participants; + less often solved than expected; - more often solved than expected.

Appendix G: Scatterplots for programming quality and different measures

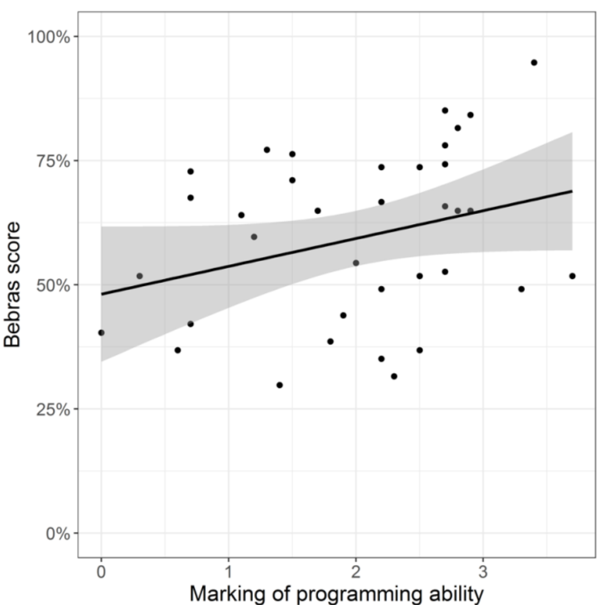


Figure H.8. Scatterplot for programming quality and achieved Bebras score (within variable) with regression line and confidence band.

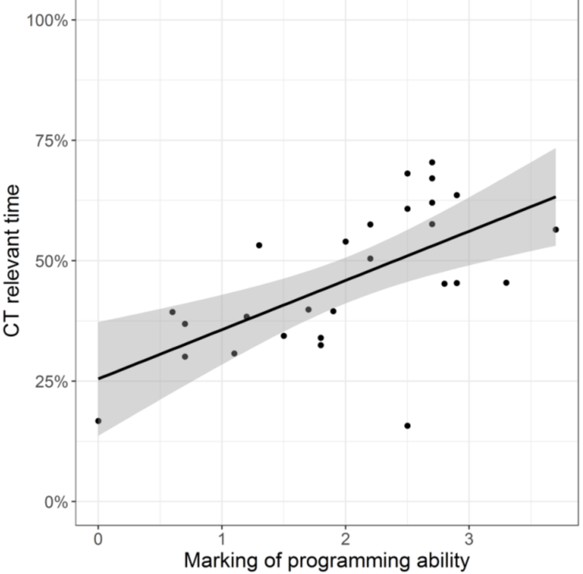


Figure H.9. Scatterplot for programming quality and percentage of CT-relevant time with regression line and confidence band.

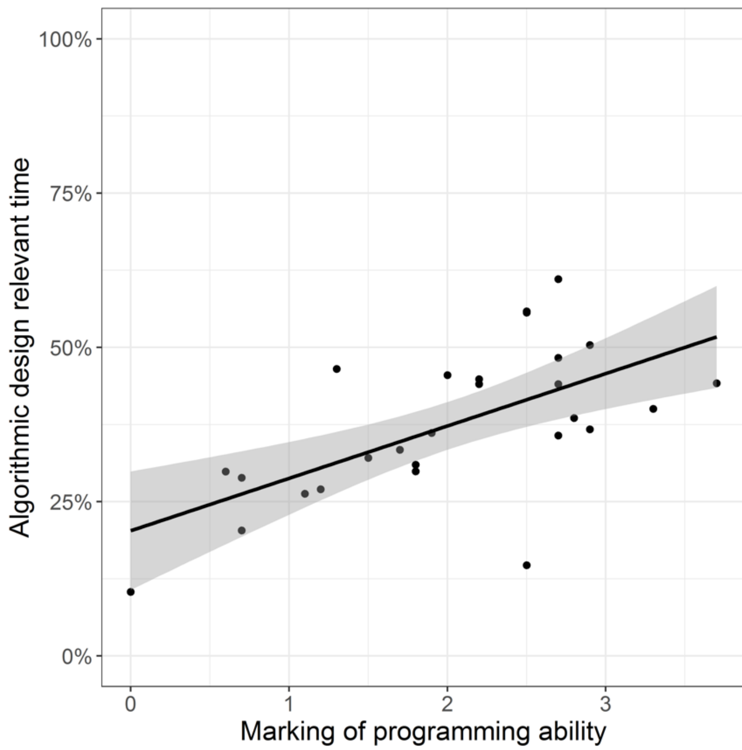


Figure H.10. Scatterplot for programming quality and percentage of algorithmic design relevant time with regression line and confidence band.

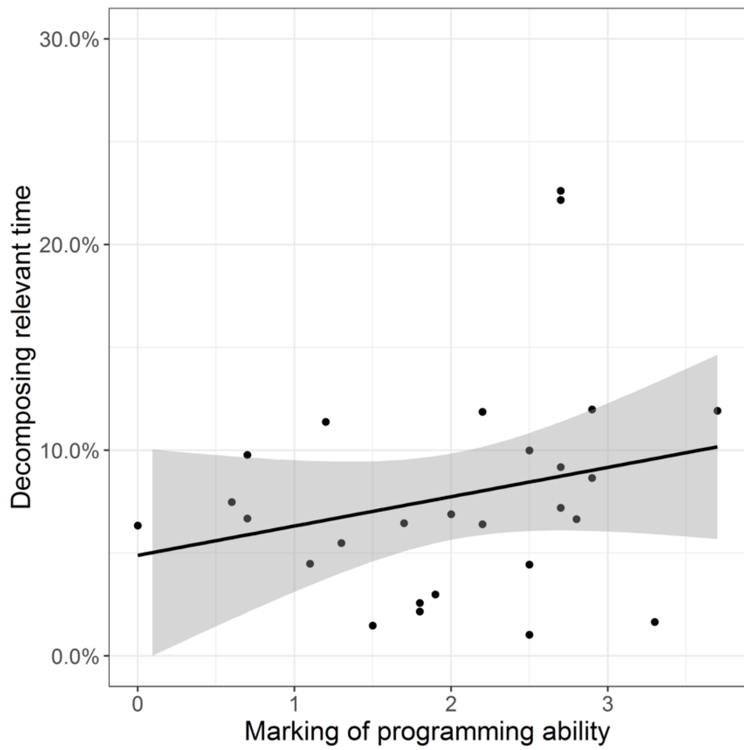


Figure H.11. Scatterplot for programming quality and percentage of decomposing relevant time with regression line and confidence band.

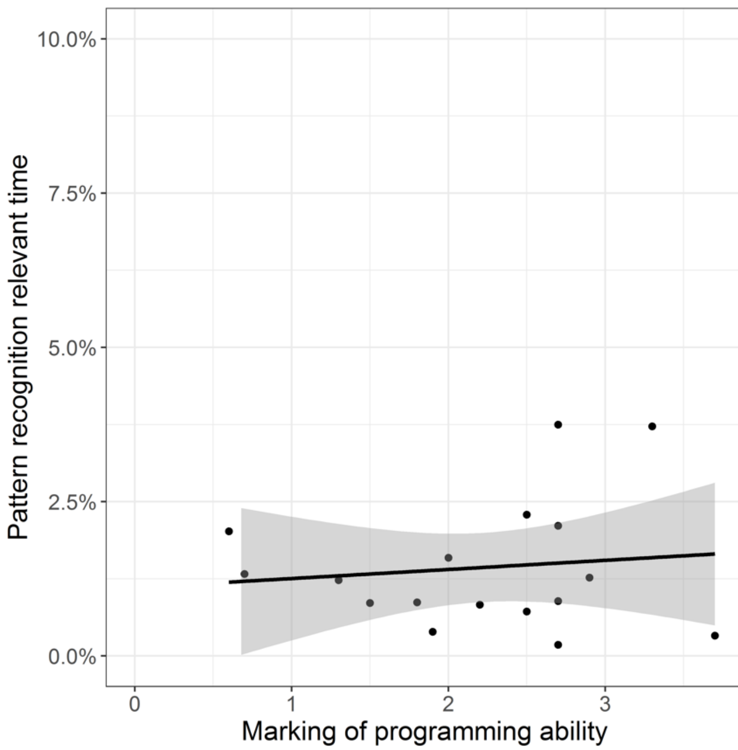


Figure H.12. Scatterplot for programming quality and percentage of pattern recognition relevant time with regression line and confidence band.

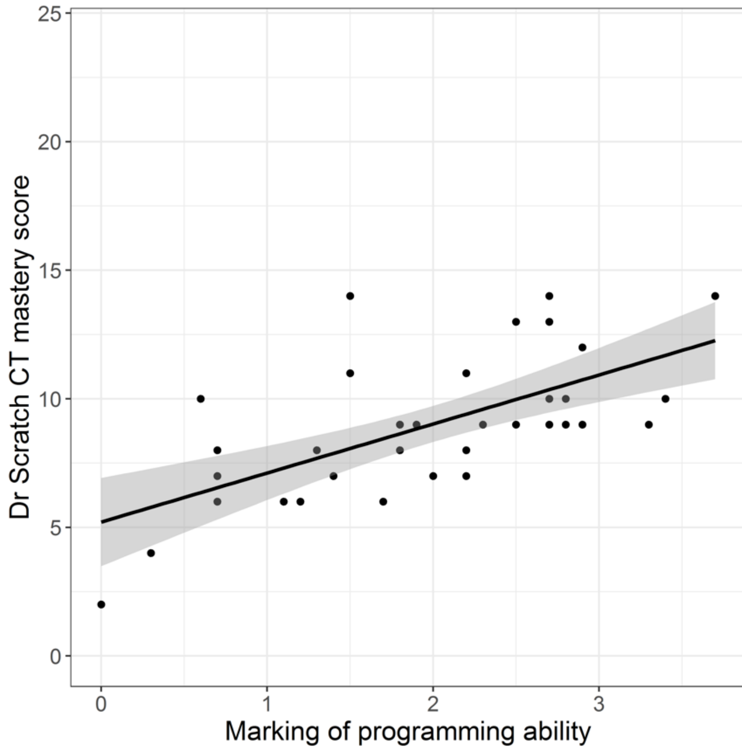


Figure H.13. Scatterplot for programming quality and Dr Scratch CT mastery score with regression line and confidence band.

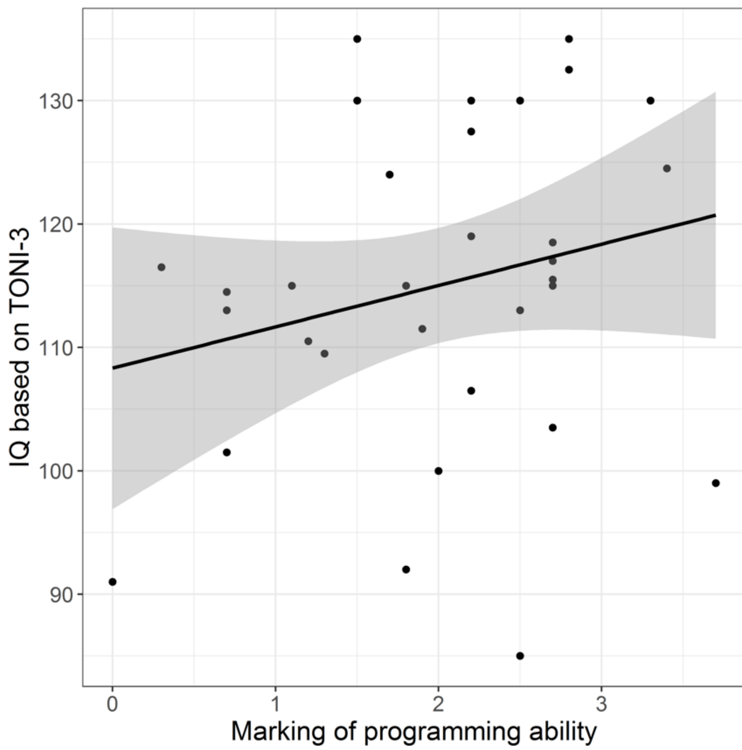


Figure 0.14. Scatterplot for programming quality and IQ based on TONI-3 with regression line and confidence band

Appendix I: Non-parametric analysis

Table I.1

Spearman's ρ *Correlations Between Programming Quality, Dr Scratch and Different Measures*

	Programming quality		Dr Scratch mastery score		*N* (pairs)
	ρ	*p*	ρ	*p*	
Bebras score	.32	.027	.29	.041	37
Time of CT-relevant behaviour (overall)	.65	<.001	.57	.001	27
Time of decomposing	.28	.077	.27	.089	27
Time of pattern recognition	.02	.951	-.42	.095	17
Time of algorithmic design	.63	<.001	.52	<.001	27
IQ based on TONI-3	.29	.055	.19	.149	32

Note. one-sided *p*-values.

$\rho_{prog.qual.\&DrScr} = .61$, $p < .001$

Appendix J: Assumptions for regression models

1) Mean of the residuals is 0:

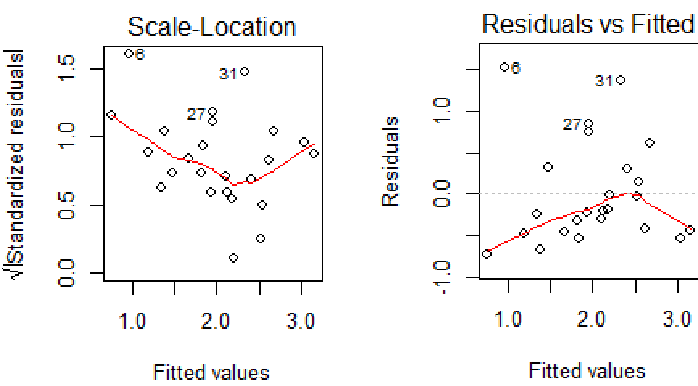
$$E(Y = programming\ quality|\hat{e}) = 4.278533e - 17 \approx 0$$

$$E(Y = Dr\ Scratch\ mastery|\hat{e}) = -1.150316e - 16 \approx 0$$

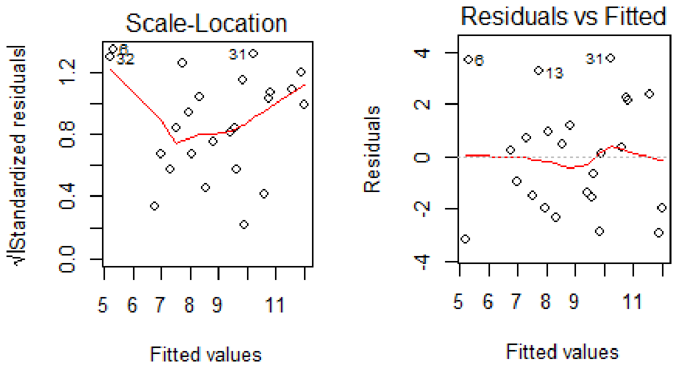
Conclusion: assumption met

2) Homoscedasticity of residuals:

Y = programming quality



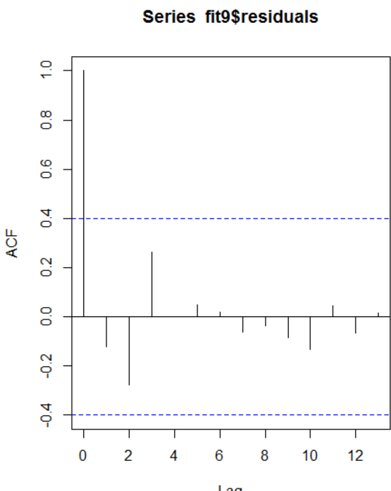
Y = Dr Scratch mastery score



Conclusion: No real pattern for "Res vs Fitted" and "sqrt(STres) vs Fitted"; no heteroscedasticity; assumption met

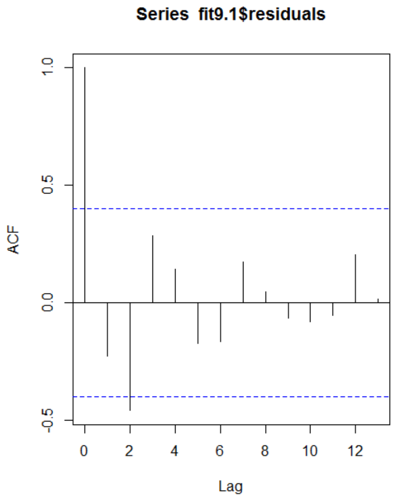
3) Residuals are not correlated (no autocorrelation):

Y = programming quality



Conclusion: No real pattern

Y = Dr Scratch mastery score



Conclusion: No real pattern

In addition, Bartels (1982) test for randomness was performed. The null hypothesis stands for randomness.

Bartels Statistics (Y = programming quality) = 0.04, p = .964

Bartels Statistics (Y = Dr Scratch mastery score) = 0.88, p = .377

Conclusion: Null hypothesis should be kept for both models; no autocorrelation; assumption met

4) Residuals and predictors are not correlated:

	Residuals	
	r	*p*
Bebras score	.06	.715
Time of CT-relevant behaviour (overall)	.05	.793
TONI-3 IQ	.28	.16

Note. Y = programming quality

	Residuals	
	r	*p*
Bebras score	-.09	.597
Time of CT-relevant behaviour (overall)	-.11	.587
TONI-3 IQ	.17	.356

Note. Y = Dr Scratch mastery score

Conclusion: no significant correlations between residuals and any predictors; assumption met

5) Number of observations ≥ numbers of predictors:

N = 24; predictors = 3 for each model

Conclusion: N > k ; assumption met

6) Variance of all predictors > 0:

Var(Bebras score) = .03 > 0

Var(Time of CT-relevant behaviour) = 223.66 > 0

Var(TONI-3 IQ) = 168.47 > 0

Conclusion: assumption met

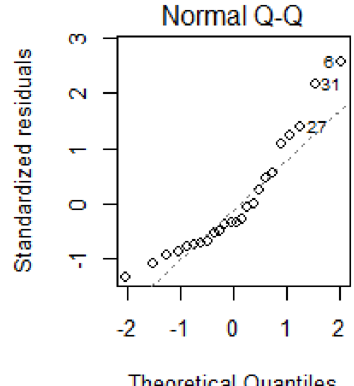
7) No (perfect) multicollinearity (VIF < 4):

	Bebras score	Time of CT-relevant behaviour	TONI-3 IQ
VIF	1.76	1.16	1.56

Conclusion: no serious multicollinearity; assumption met

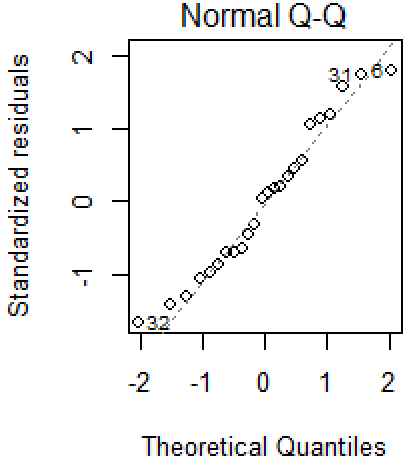
8) Residuals are normally distributed

Y = programming quality



Conclusion: data approximately normally distributed

Y = Dr Scratch mastery score



Conclusion: data approximately normally distributed (higher deviation at end poles)

In addition, Shapiro-Francia test was performed. The null hypothesis stands for normal distribution.

W'(Y = programming quality) = 0.88, p = .011

W'(Y = Dr Scratch mastery score) = 0.96, p = .489

Conclusion: residuals for model(Y=programming quality) significant different from normal distribution. Assumption fully met for model(Y=Dr Scratch mastery score) but only partially met for model(Y=programming quality)