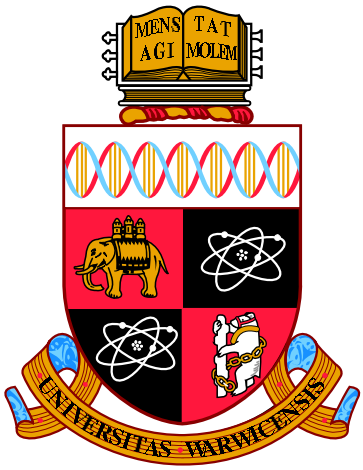


Towards a Model of Giftedness in Programming: An Investigation of Programming Characteristics of Gifted Students at University of Warwick

by

Ayman Qahmash

Thesis

Submitted to the University of Warwick

for the degree of

Doctor of Philosophy

Department of Computer Science

January 2018

THE UNIVERSITY OF WARWICK

Contents

List of Figures

Page

List of Tables

Page

Acknowledgments

The completion of my thesis has been the most rewarding journey in my life thus far, and all praise is due to God.

I wish to extend my immense gratitude to my supervisors, Dr Mike Joy and Dr Adam Boddison, for their endless support, patience and guidance throughout my study.

I would like to dedicate this thesis to my parents' souls, which have inspired me to achieve what they would have liked to witness. To my brothers, Mohammed and Adel, your support allowed me to finish my thesis, and your encouragement kept me moving forward.

To my wife, Wadia, if it were not for your support and patience, achieving my goals would have been impossible. Nothing can describe your endless encouragements expressing your belief in me, which motivated me to finish my thesis. Thank you, my dear wife, for everything.

The study was financially supported by King Khalid University, for which I am thankful.

Declarations

I certify that the material included in this thesis is my own work. I confirm that no part of this thesis has been either published in another form or submitted for a degree at another university.

Ayman Qahmash

Abstract

This study investigates characteristics related to learning programming for gifted first-year computer science students. These characteristics include mental representations, knowledge representations, coding strategies, and attitudes and personality traits. This study was motivated by developing a theoretical framework to define giftedness in programming. In doing so, it aims to close the gap between gifted education and computer science education, allowing gifted programmers to be supported. Previous studies indicated a lack of theoretical foundation of gifted education in computer science, especially for identifying gifted programmers, which may have resulted in identification process concerns and/or inappropriate support.

The study starts by investigating the relationship between mathematics and programming. We collected *3060* records of raw data of students' grades from 1996 to 2015. Descriptive statistics and the Pearson product-moment correlation test were used for the analysis. The results indicate a statistically significant positive correlation between mathematics and programming in general and between specific mathematics and programming modules.

The study evolves to investigate other programming-related characteristics using case study methodology and collecting quantitative and qualitative data. A sample of $n=9$ cases of gifted students was selected and was interviewed. In addition, we collected the students' grades, code-writing problems and project (Witter) source codes and analysed these data using specific analysis procedures according to each method. The results indicate that gifted student programmers might possess a single or multiple characteristics that have large overlaps. We introduced a model to define giftedness in programming that consists of three profiles: mathematical ability, creativity and personal traits, and each profile consists of sub-characteristics.

CHAPTER 1. Introduction

Preface

Novice programmers have encountered a wide range of difficulties learning programming concepts, cognitive abilities, and programming languages. A great deal of research has investigated these characteristics to understand how novice programmers acquire programming knowledge and how the cognitive abilities of novice programmers can be developed. Consequently, computer science (CS) educators have been able to identify students who manifest difficulties in learning programming based on the specific characteristics investigated earlier. However, students who might have a fast pace of acquiring programming and/or might have programming experience that allows them to skip the basic programming module might have not been sufficiently supported by CS educators (Carter et al., 2010).

One factor that led to this concern is that there has been little research about gifted student programmers to understand certain characteristics related to how do they learn programming. Consequently, these students cannot be identified. Therefore, appropriate support might not be provided. To provide support for a specific group of students, we need to understand their needs within a specific context, which is programming in this case. To achieve that, gifted education outlines the concepts of the identification process, identification method, and enrichment and acceleration programmes, which should be considered first.

There is no doubt that all students must be provided with support within any educational institute. However, in some situations, it becomes very difficult to provide support for a large class size, especially in programming modules.

I have had two teaching experiences. I previously taught introductory programming and web technology modules at King Khalid University in Saudi Arabia. Class sizes were enormous, and there was no doubt that the abilities of students varied, which made the situation more difficult to meet all student needs. In Saudi Arabia, CS modules are not introduced properly during high school, where curricula mostly cover basic information and communications technology (ICT) topics. The consequence can be that some students arrive at university with limited or no programming experience. In cases where university teaching staff need to include prerequisite programming content, this can be at the expense of the content of the module being taught. During computer lab sessions, many students were struggling to retrieve information from the server side using PHP server scripting language; yet, two students finished the tasks in a very short time. Those two students achieved high grades for the web technologies module, but it was not clear whether their expectations had been met.

The second teaching experience was at the University of Warwick teaching Design of Information Structures and Professional Skills modules. The lab session consisted of roughly 25 students with 4 to 6 lab tutors. The lab manual was clearly designed to start with tasks aimed to provide the basics of certain programming concepts, and the task-level depth and difficulty increased. The tasks were marked once students had completed them all. If the student could not finish, the remaining tasks would be marked in the next lab session.

If a gifted student can be identified at an early stage, an instructor can adapt their practices earlier. However, we need to define how a gifted student programmer can be identified, based on a specific giftedness theory. Moreover, we need to understand what programming means in certain contexts. Thus, investigating characteristics related to programming and determining what sort of characteristics gifted student programmers might possess would be a first step to help CS educators address such questions. The issue is how can we achieve that when we lack a theoretical base in CS education (CSE) research?

The starting point for my research was to understand the philosophy behind giftedness, addressing the ontological and epistemological perspectives. Then, we need to understand multiple aspects of what programming means, how programming can be learnt, and what cognitive abilities are required to accrue programming knowledge. Once we addressed the concerns above, certain characteristics were suggested based on a literature review derived from CSE and the profession of computer programming through gifted student programmers. As mathematical ability has previously been identified as a common characteristic, we started by investigating the statistical correlation between programming and mathematical ability. Then, we investigated other characteristics using case study methodology, allowing us to gather both quantitative and qualitative data.

This study investigates characteristics related to programming for first-year gifted students in the Department of Computer Science at the University of Warwick. Identifying specific characteristics that gifted student programmers might possess helps define giftedness in programming, which would assist CS educators in providing support for gifted student programmers based on a theoretical foundation. Gifted education principles, the identification process, and enrichment and acceleration programmes should be derived from a theoretical foundation for a specific context, which is programming in this study.

As educational institutes should meet all student needs and expectations, CSE researchers have investigated multiple characteristics related to programming to support novice programmers who may be struggling to learn programming. However, the literature defining the characteristics of individual gifted student programmers is sparse. Thus, this study aimed to identify characteristics of gifted student programmers to develop a theoretical foundation to define giftedness in programming. The theoretical foundation can be used to derive methods to identify gifted student programmers and support all students in developing certain characteristics to promote learning programming.

This study started by investigating the literature to reveal the most common student characteristics related to programming. Second, the relationship between mathematical ability and programming was investigated, and the results indicated a significant statistical positive correlation. Then, a case study methodology was used to investigate specific characteristics, including mental representations, knowledge representations, coding strategies, and attitudes and personality traits. The findings from this study were synthesised to produce a model of three profiles: mathematical ability, creativity, and personal traits.

1.1 Background

Programming is a human activity that involves a complex incorporation of certain cognitive abilities and skills to solve a problem using a language that can be understood by a computer. The complexity of incorporating logic, problem solving, abstraction, and computer programming language can make the process of learning programming a notoriously difficult task for students.

The rapidly changing field of CS means CSE must constantly adopt new curricula and pedagogies. As a result, failure and dropout rates among CS students have been controversial research topics among CSE researchers (Bennedsen & Caspersen, 2007; Petersen et al., 2016; Watson & Li, 2014). In an attempt to improve the teaching of programming, CSE researchers have devoted a great deal of effort to supporting novice programmers by enhancing curricula and pedagogies. However, *gifted student programmers* have different needs than other students. Gifted student programmers who may excel in knowledge acquisition and problem-solving ability require a challenging curriculum to maintain a steady level of engagement during the course (Eyre, 1997), rather than starting by teaching them the basic 'Hello World' program in the first week. Unchallenging content may result in disengagement, boredom, and low achievement, as students will not put more effort into learning something they already know (Carter et al., 2010). In addition, gifted student programmers, who might learn programming at a faster pace than their peers, may not learn enough from a curriculum

that typically lacks challenging tasks and advanced knowledge. In programming, the natural sequence of programming concepts makes the pace of learning an important matter, and if motivation is lost, the learner may not be developed (Johnson et al., 2000).

Research has shown that some gifted students do not receive tailored support to enrich their learning; for example, one survey indicated that 34% of CS academics do nothing to support gifted student programmers (Carter et al., 2010). The limited types of support that have been offered to gifted students include competitions and differentiated teaching to provide a specific study pathway that suits the needs of gifted students.

It is unlikely that a university computer programming class would comprise only novice students. It is more likely that some students might have programming experience, some students might have work experience or training, and some might be gifted. Those students might be sufficiently advanced to be accelerated by skipping introductory programming courses (generally known as Computer Science 101 or CS1). However, most CS undergraduate degrees consider CS1 to be a core module and a prerequisite for advanced modules. Thus, CS1 cannot be skipped by gifted students and acceleration is not always an option.

Despite the old perspective of gifted education that focuses on the minority of students who must meet restricted criteria, the new perspective is to provide a learning environment that supports gifted students to reach their full potential. The environment should be inclusive, allowing more students to be developed through revolving stages of development. Various concepts and definitions of giftedness have been proposed. These definitions have been analysed according to the level of restrictiveness proposed from two different points of view. Society has the view that a small number of people have outstanding cognitive abilities, which the rest of us do not possess. However, some studies have suggested that what individuals think is outstanding could appear to be unexceptional. Moreover, human ability could be pushed to reach the highest level of performance if we try to provide individuals with environments that

maximise their potential and performance (Eyre, 2011).

Identifying gifted students is a vital process so that students can be triaged into differentiated teaching tracks or accelerated programmes. Several methods of identifying students have been implemented in gifted education, including assessing IQ or academic performance, and teacher nominations. Professionals working in the field of gifted education are yet to have a consistent approach to identification of gifted students (Eyre, 2011). Giftedness is often considered an exclusive concept in education. Some suggest that giftedness is a unique measurable phenomenon; for example, giftedness can be exhibited by someone who achieves a certain score on a scale.

Other perspectives favour defining giftedness to be as inclusive as possible and to focus on providing enrichment that can cater for different levels of gifted students. For example, identifying the top 25% of students and nurturing their talents is arguably better than restricting the access of a special education programme to be only for the top 5% of students. It has been argued that acceleration, enrichment, and differentiated programmes not only have a positive influence on gifted students, even average students can increase achievement in these programmes (Reis & Renzulli, 2010). Thus, identifying gifted students is an entry point for a long process of development and maximising the potential of gifted students rather than being the end product in and of itself.

Some would argue that labelling certain students as 'gifted' could be controversial and humiliating for others. However, using a more liberal definition of giftedness to be more inclusive in identifying gifted students may decrease the focus on the literal meaning of the term 'gifted' and switch the emphasis to the ultimate purpose of gifted education. In an educational context, certain adjectives are inevitably used to describe certain groups of students; yet, it is important that no emotional harm is caused by using these adjectives. Thus, the term 'gifted' should imply a positive description and characteristic.

Some CSE researchers have attempted to support gifted students through various methods, including differentiated tracks, accelerated classes, and competitions. However, the ways in which gifted student programmers were identified for these programmes and the theoretical foundation of defining giftedness in programming were not clear. We must ensure equity when identifying gifted students, as using too broad or a wrong identification method might increase the chances of inadvertently excluding a gifted student. It has been recommended that the identification process should be derived from existing giftedness theory (Colangelo & Davis, 2002). If we assume that giftedness in programming is defined to be a phenomenon that can be quantified, then the identification process would involve measuring a student's programming ability based on the intelligence quotient (IQ) or the Programming Aptitude Test (PAT) score. If a student must achieve a specific score to be identified as gifted, can we include a student who missed the cut-off score by one point?

Tests of this nature have also been criticised for validity and reliability. A controversial claim suggested that the PAT designed by Dehnadi and Bornat (2006) can measure student programming abilities and can distinguish between a student who was 'born to be a programmer' from one who was not. In 2014, the co-author Richard Bornat wrote a personal communication to retract his claim. Other researchers who replicated the study produced different results that contradicted their claims (Bornat, 2014).

The information technology (IT) industry including International Business Machines (IBM) introduced PAT to evaluate student reasoning, logic, and mathematical abilities. However, the test failed to maintain significant statistical results to indicate whether students can be successful in learning programming (Boesch & Steppe, 2011) Thus, the PAT score failed to measure programming abilities. Researchers now consider how multiple characteristics can affect success in programming, including age, gender, programming experience, problem solving, abstraction, and mathematical abilities.

This reveals a gap in the literature on giftedness in CSE; research is limited to a few researchers who touch on important aspects of gifted education but without any

theoretical foundation. The effort of CSE researchers was a short-term intervention strategy that lacked a definition of giftedness based on CS and lacked robust identification methods. If the PAT is not a valid tool to measure programming ability, are there any identification methods to help CS educators? Can PAT be incorporated into other methods?

1.2 Research Questions

This is an inquiry into gifted education within the context of programming to establish a foundation for CS educators. The inquiry seeks to provide a definition of giftedness in programming and to explore methods for the identification of gifted programmers. In particular, the inquiry considers whether there are characteristics or traits that may support the identification of gifted programmers. The characteristics include mathematical ability, mental representation, knowledge organisation, coding strategies, and attitudes and personal traits.

The mathematical ability has been an important factor in learning programming. There can sometimes be an assumption that excelling in programming requires excelling in mathematics which implies a relationship between mathematics and programming. However, there is a lack of statistical evidence to prove the relationship. Thus, research question 1 of this inquiry will investigate the relationship between mathematical ability and programming. Previous research suggests (Henderson & Stavely, 2014) that certain mathematical concept is a prerequisite for learning certain programming concepts. Thus, research question 2 will investigate a specific correlation between discrete mathematics, calculus and programming modules.

Mental representation includes approaches to problem-solving and abstraction ability. Previous research (Blackwell, 2002; Teague & Lister, 2014; McKeithen et al., 1981; Joseph, 2015) suggests that expert programmers may possess unique problem-solving strategies and abstraction abilities. Research question 3 of this inquiry will build on the existing literature by exploring the mental representation of first-year gifted student programmers.

There can sometimes be an assumption that expert programmers may demonstrate advanced knowledge of specific mathematical concepts and advanced data structures. Research question 4 will test this assumption in the context of gifted student programmers.

Coding strategies, attitudes and personal traits are important characteristics of effective programmers (Joseph, 2015; McConnell, 2004; Lammers, 1986). An expert programmer should possess a specific coding strategy, communication skills and co-operative skills. Research questions 5 and 6 of this inquiry will investigate these characteristics in the context of education and will seek to establish whether or not gifted student programmers possess specific coding strategies, attitudes and personal traits.

RQ: 1 To what extent does mathematical ability correlate with programming ability in general?

- RQ: 1.1 What is the correlation between student performance in discrete mathematics and programming modules?
- RQ: 1.2 What is the correlation between student performance in calculus and programming modules?

RQ: 2 What are student perceptions about the relationship between mathematical ability and programming ability?

RQ: 3 What mental representation strategies do gifted students tend to use?

RQ: 4 What mathematics and programming knowledge do gifted students tend to have?

RQ: 5 What coding strategies do gifted students tend to use?

RQ: 6 What attitudes and personality traits do gifted students tend to possess?

1.3 Context of the Study

The study was conducted in the Department of Computer Science at Warwick University, which hosts several undergraduate degrees, including CS and discrete mathematics (DM). The difference between the two degrees is that the DM degree includes more mathematics modules incorporated with the CS modules. Both degrees share first-year core programming modules.

As Warwick University has high standards of admission requirements, the CS degree requires that students must obtain three advanced levels (A-levels), which is equivalent to a high school leaving qualification. The student must achieve A grades for A-levels, including in mathematics. Further, DM degree students must obtain three A-levels with at least one A* grade in either mathematics or further mathematics. The admission process is thus highly competitive.

The context of this study provides us with a unique case study of a group of students who manifested high performance in modules such as computing and mathematics during their early education. Warwick University students in the CS degree programme are believed to be among the top CS students in England. In addition, participants for this study were selected based on their high performance for the first-year programming modules relative to their peers in the course. Thus, all study participants were classified as gifted programmers.

1.4 Significance of the Study

Addressing the current shortage of literature on giftedness in CS is the main motivation for this research. That can be accomplished by understanding the ontology and epistemology for both giftedness and programming within the educational context. Closing the gap between the disciplines of CS and gifted education can improve the efforts of CSE educators to design curricula, pedagogies, and activities. To be more specific, this study contributes to the field of CSE by establishing a theoretical framework for defining giftedness in programming. Many aspects of gifted education, such as identifi-

cation processes, identification methods, and enrichment and acceleration programmes can then be derived from this theoretical base.

With increasing emphasis on innovations and entrepreneurial job opportunities within the private sector, students programmers could become net contributors to the economy. Investing the time and money in an appropriate curriculum for gifted student programmers could improve the economy in the long term, as these individuals excel in their respective careers.

There is a well-established process in the education of gifted students, from early education, beginning with gifted student identification, curriculum enrichment, and acceleration programmes. Whilst some argue that early identification and appropriate provision of gifted students can improve their higher education outcomes, others believe this is rarely the case(Robinson, 1997; Albon & Jewels, 2008). This study contributes to the literature about support student programmers at university level.

1.5 Structure of the Thesis

This thesis consists of seven chapters. This chapter provided an overview of the study background, motivations, questions, context, and significance. The second chapter provides an overview of the literature on theories of giftedness, psychology of programming, and empirical characteristics, including ability in mathematics, problem solving, and abstraction, personal traits, and programming profession related characteristics, such as coding strategies. Various educational taxonomies are also discussed in this chapter. The third chapter outlines the research paradigm, study methodologies, data collection procedures, and analyses. In addition, this chapter addresses the study's validity, reliability, and ethical considerations. The fourth chapter provides details of the numerical analysis procedures followed by statistical results from investigating the correlation between programming ability and mathematics ability in general, plus specific investigations of the correlation between discrete mathematics, calculus, and programming. The fifth chapter presents the data analysis and findings. This chapter investigates the characteristics of gifted student programmers, including multiple data

collection for each participant. The sixth chapter synthesises results from all phases of the study into a potential model for identifying gifted student programmers. The model consists of three profiles: mathematical ability, creativity, and personal traits. Each profile consists of sub-characteristics. The final chapter summarises this study, including the main findings derived from answering the research questions and sub-questions. Study strengths and limitations are outlined, followed by key considerations and implications.

CHAPTER 2. Review of the Literature

Preface

The purpose of this chapter is to review the relevant research on gifted education and programming, exploring theoretical foundations and backgrounds that relate to our research. As our research context is based on CSE, we will explore educational theories that can be considered by CS educators. It is essential that CS educators consider educational theories when investigating issues related to pedagogy and student learning where well-developed theories can be implemented. In addition, interdisciplinary research provides unique aspects derived from two different contexts, which could benefit either discipline.

The literature was accessed through the University of Warwick online databases using specific terms and words to obtain journal articles and conference papers related to gifted education and programming. The literature on gifted education was gathered from journals, such as *Gifted Child Quarterly*, *Learning and Individual Differences*, and *Journal for the Education of the Gifted*. The terms that were used included giftedness theories, intelligence theories, gifted education principles, identification, enrichment, and acceleration. Literature on programming was collected through journals such as *Computer Science Education*, *Association for Computing Machinery (ACM) Inroads*, and *Special Interest Group on Computer Science Education*. The searched terms included expertise in programming, high-performing students, learning programming, problem-solving strategy, abstraction ability, and mathematics and programming. As the term *giftedness* was absent in the context of CSE, I used broad search terms to find articles that could be related to giftedness in programming or CS. The literature was

organised and categorised into sub-databases using JabRef software. The categories included gifted education, programming characteristics, expert programmers, and programming and mathematics. The general approach was to first read an article and then to follow up on the references extracted from that article.

In general, CSE research is about better understanding pedagogy in CS. Most CSE research has focused on intervention during the learning process, be it the type of intervention, including technology implementation or the adoption of different teaching strategies. However, it has been argued that CSE research lacks the educational theoretical foundation to support teaching activities, pedagogy, and interventions. The literature covering gifted education in the context of CS in sparse. As a result, few studies have considered different forms of support, including competitions and differentiated study paths (Carter et al., 2007; Davis et al., 2001; Jenkins & Davy, 2000). Without questioning whether the provided support was appropriate and derived from the interests of students or not, the lack of a methodological approach to identifying gifted students was a concern raised in previous research. In the context of CS, educators are aware of the challenges of teaching groups of students with a diverse range of knowledge, skills and experience. However, it was clear from the literature that giftedness might be considered a controversial concept, as the previous study referred to students as *over-performing* students and *rocket scientists*. That might happen because of certain aspects, such as a lack of understating of the recent giftedness paradigm, which emphasises the provision of an environment that could help a wide range of students who have the potential to be developed and supported through different stages. Another aspect could relate to the fair allocation of recourses. It could be argued that institutions provide financial and emotional support for some students with special needs, but what if their needs are different? Thus, CSE literature lacks research on applying gifted education principles, which first need to be derived from a specific definition of giftedness in programming. In addition, there has been little research on how to identify gifted student programmers and what the identification methods should be, based on student interests and what is appropriate to measure certain characteristics in pro-

gramming. As a result, enrichment and acceleration programmes should be designed based on the definition of programming and the characteristics that are manifested and can therefore be developed. The literature indicates that the lack of challenging tasks in programming modules, especially for students who excel in learning programming, has been considered by CS educators (Carter et al., 2007). However, there is insufficient research to investigate whether providing challenging tasks on student performance has an effect. That could be an area for investigation within CSE.

Moreover, CSE addresses concerns related to many educational aspects within the field of CS. CSE is growing and is still a relatively young field (Robins, 2015). The CSE research has been conducted by computer scientists who are capable of understanding CS and the necessary student characteristics and abilities for learning programming. However, CS academics might not be fully suited or dedicated to education research, which raises concerns around the methodological and theoretical rigour of some CSE research. It has been argued by Almstrum et al. (2005) that CSE research disregards long established education, cognitive, and learning theories. Over a decade later, Robins (2015) argued that CSE research is still facing multiple challenges related to the lack of theoretical and methodological foundations.

Fincher and Petre (2004) addressed major principles for CSE research, including linking CSE to relevant previous research and using appropriate methods to directly investigate research questions. In addition, they emphasise that CSE researchers should provide chains of evidence supporting research findings, allowing their studies to be replicated and therefore allowing results to be generalised. Moreover, Lister (2016) postulated that educational theories derived from the research of Jean Piaget and subsequent neo-Piagetian researchers provide perspectives on how programming can be learnt. Based on Lister's experiences and findings on conducting CSE, he proposed a concept of computer programming epistemology in his attempts to close the gap between CS as knowledge versus how CS can be learnt. Thus, this chapter aims to serve the purpose of shaping our research based on existing theoretical foundations in gifted education and learning programming as well as to link the two fields of computer

science and education.

This chapter is divided into five sections. The first explores gifted education, including the historical background on giftedness and its definitions, methods of identifying gifted students, and enrichment and acceleration programmes. In the second section, an epistemological view of programming will be discussed to define what programming is. The third section explores characteristics are needed to learn programming. In the fourth section, educational theories related to learning and assessing programming are explored. The chapter concludes with a refliction of the literature review.

2.1 Theories of Giftedness

Historically, gifted education has fallen into three categories (Eyre, 2011). First, the *unique individual paradigm* was introduced in the early to mid-twentieth century to concentrate on a minority of unique individuals. These people were considered to be 'geniuses' with unique characteristics but were considered freakish and irrelevant to the education system because they had special needs that could not be addressed by a school.

From the mid-to-late twentieth century, the *cohort paradigm* shifted the focus to a group of individuals selected from schools who were considered gifted. In addition, theories such as Galton's had been developed to define human cognitive abilities and human traits that can be inherited and measured (Jensen, 2002). The Stanford-Binet Intelligence Scale has been used to identify gifted students who score above 140 in order to provide them with different education (Eyre, 2011).

However, concerns have been raised. For example, high-achieving adults, such as Nobel Prize winners, may have been unexceptional children, so early gifted identification may not be valuable (Bloom, 1982). Another issue is that developed intelligence theories provide a more complex definition of giftedness. Moreover, using IQ tests may not be relevant to measure different aspects of intelligence. The idea of testing a complex phenomenon shaped by a combination of nature and nurtured aspects may

also be challenging to identify gifted students based on cognitive assessments (Eyre, 2011). Thus, the main focus of this paradigm is to define the characteristics that signify giftedness rather than to develop education systems that can deal with these sorts of unique abilities.

In the twenty-first century, the *human capital paradigm* has emerged, emphasising how giftedness can be developed, rather than identifying gifted students. Therefore, the new paradigm for gifted education focus is on developing educational systems that can fulfil student needs and enhance their potential so they become high performers in the future. In addition, this approach tends to eliminate the fact that gifted adults are considered a homogenous cohort with common needs, rather than individuals who have great potential to perform highly in specific domains with different learning styles and different learning speeds. Another aspect of this paradigm is to describe the behaviour and learning techniques that have been exhibited by high-ranking performers in specific fields. Although the term 'giftedness' is still used by some advocates of the human capital paradigm, it is employed to describe the final point, not the starting point. Therefore, some prefer to use the term 'high performance' to make it clear that this approach favours developing a potential high performance (state) rather than identifying an innate gift.

Indeed, discussion on determining whether a child is born with an innate gift or whether the gift can be developed has never been concluded. Scientists, such as Andres Ericsson, have eliminated the idea of an innate gift and have focused on high performance, which can be acquired and developed by hardworking individuals.

However, it is a fact that some genetic aspects could be inherited from parents and may affect newborns; thus, genetics can play a possible role in shaping human abilities. Children who have a quality of giftedness that naturally manifests itself can enhance that quality and make more effort to attain outstanding accomplishments. For example, it is no coincidence that Mozart's parents were music teachers (Eyre, 2011).

Despite the two extreme views on the nature of giftedness, it is important to go beyond that by accepting the fact that giftedness is a phenomenon shaped by multiple

factors, which may be innate and/or nurtured.

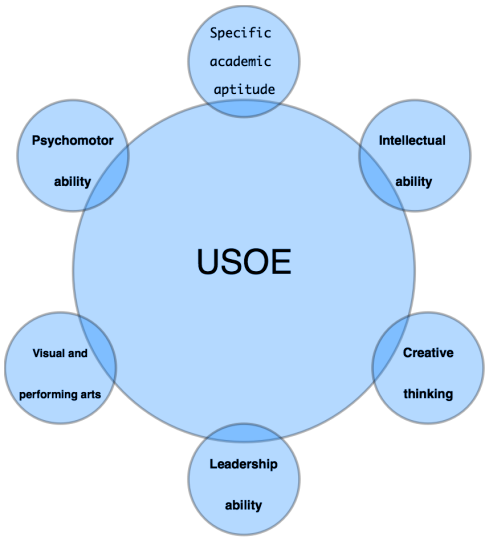


Figure 2.1: USOE definition of gifted and talented students.

Defining giftedness in the educational context has been important to develop school policies to identify gifted students for inclusion in enriched or accelerated education programmes. The United States Office of Education (USOE) defined gifted and talented students as students who exhibit outstanding abilities manifested through high performance in intelligence, particular academic aptitude, creative thinking, leadership, visual and arts performance, and psychomotor ability (Marland, 1971). The USOE states that these students should be provided with different educational programmes addressing their potential.

Although the USOE definition includes several human abilities in which a large spectrum of students could be identified as gifted, some issues arise out of the USOE definition. First, the definition does not include motivational aspects. Second, there is an issue with the non-parallel nature of the six categories that have been identified. Categories such as specific academic aptitude and visual and performing arts aptitude may be considered general human performance, whereas other categories can be applied only in some performance areas; for example, a gifted student may show creativity within a specific aptitude, such as programming. Third, teachers may misinterpret the USOE definition and/or may not be able to develop identification methods based on the six categories.

The Department of Education in the United Kingdom defines the group of students who are eligible for gifted programme membership as 'children and young people with one or more abilities developed to a level significantly ahead of their year group (or with the potential to develop those abilities)' (DFE, 2008). This definition means that, along with students who achieve in the top 5% nationally, students who manifest abilities yet are underachievers can also be identified, and abilities such as sport and art can be considered during the identification process. Different identification methods are suggested including tests, assessments, and teacher, peer, and parent nominations. Psychologists and educationalists have developed multiple theories for giftedness along with methods of student identification and suggested models for providing special educational programmes.

2.1.1 Multiple Intelligences Theory

In the early 1980s, Gardner introduced the multiple intelligences (MI) theory, believing that human cognition can be described as multiple abilities and talents that could be called intelligences (Gardner, 2006). The MI theory suggests that human cognitive ability consists of different categories of intelligences, including linguistic, logical mathematical, spatial, musical, body kinaesthetic, interpersonal, intrapersonal, and neutralist (Gardner, 1987).

Gardner's theory opposes the view that suggests that human intelligence can be quantified and measured based on intelligence tests. The IQ test has been used to measure student intelligence but could be criticised for suggesting that human cognition is only one dimensional where people can be categorised based on quantifying their intelligence. The implications of this view on gifted education have been questioned. For example, using tests to identify gifted students can be a requirement for including students in enriched or accelerated programmes. In some cases, students must achieve a score of 130 based on an IQ test to be eligible for a gifted programme; yet, if the score is 129, a student might not be included (Gardner, 2006).

It has been argued that Western societies consider only linguistic and logical

mathematical intelligences in their curriculum. However, the MI theory suggests that educational curricula need to fulfil student needs by considering their individual differences to maximise their intellectual gains (Fasko, 2001). However, Gardner rebutted that the MI theory has been misunderstood by some educators, suggesting that MI implications for education have been, in some cases, mixed with other intelligence theories (Gardner, 2003). In addition, Gardner emphasised that MI should not be an educational goal itself; rather, it should be used as a theoretical foundation that could be helpful in achieving educational goals. These educational goals should be derived from teachers, curricula, or student beliefs and not from the MI theory itself.

2.1.2 Three-Ring Conception of Giftedness

A person who has been recognised for his or her outstanding creative or productive achievements may exhibit a combination of three ingredients: above-average ability, task commitment, and creativity (Renzulli, 1978). This is known as the three-ring conception of giftedness, shown in Figure 2.2, and was later expanded to include co-cognitive traits promoting social capital (Renzulli, 2002). Renzulli emphasised that possession of a single trait does not make giftedness; giftedness can be manifested when all clusters interact with each other equally. It is important that all three clusters must be equally considered for identification purposes and that one cluster cannot dominate. Renzulli noted giftedness definitions lie on a spectrum where some conservative definitions exclude certain fields, such as art and music, and define giftedness based solely on intelligence (Renzulli, 1978).

Other definitions of giftedness provide wider inclusion of many fields of performance, allowing more students to be included within the identification process. In addition, Renzulli stated that above-average ability might not necessarily be superior ability, as research indicates that creative accomplishment might not correlate with tested intelligence (Renzulli, 1978). The task commitment cluster is defined as creative and pro-

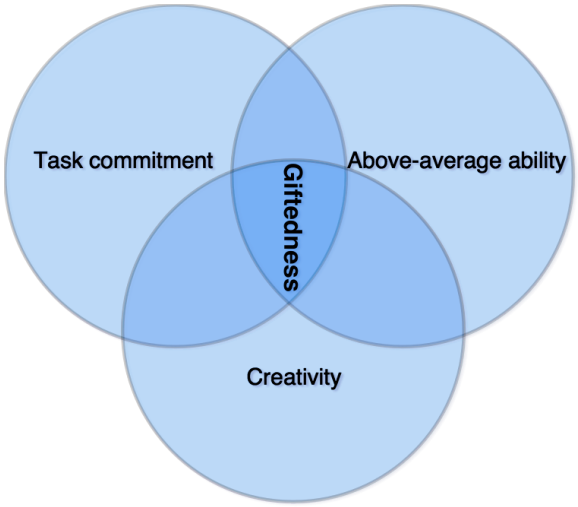


Figure 2.2: Three-ring conception of giftedness.

ductive energy to accomplish a specific task or specific area of performance (Renzulli, 1978).

Renzulli's argument of including non-intellectual traits in giftedness is that previous research suggested that one crucial factor observed in gifted individuals is the ability to be fully dedicated to accomplishing a task over a long period. An individual who strives to accomplish great achievements is often highly motivated by both intrinsic and extrinsic motivation (Renzulli, 1984a). Deci and Ryan (1985, p. 234) defined intrinsic motivation as a 'natural ongoing state of the organism unless it is interrupted' in which this innate ability can be a motivating factor, leading individuals to accomplish tasks. However, extrinsic motivation considers other motivating factors, such as money and rewards.

Renzulli addressed the issue of measuring creativity. Individuals can be defined as gifted based on their creative accomplishments, but Renzulli believed the issue was more about how a creative person could be identified. Divergent thinking tests have been used to measure creativity, raising concern as to whether these tests truly measure creativity. Procedures to measure creativity should be designed based on specific areas of performance, for example, identifying creative architects should be based on criteria derived from the field of architecture.

2.1.3 Differentiated Model of Giftedness and Talent (DMGT)

Gagné (1985) claimed that previous giftedness definitions, including those by Renzulli, did not distinguish between the terms 'giftedness' and 'talent', which could cause confusion and ambiguity. Therefore, he introduced a model to differentiate between the two, distinguishing between ability and performance. A gifted student might possess at least one superior innate ability, which may be untrained and raw, allowing the gifted student to be in the top 10% among his or her age group. Conversely, a talented student may exhibit superior developed abilities (or skills) and knowledge in a particular field of human activity and be among the top 10% of his or her age group.

The differentiated model of giftedness and talent (DMGT) consists of three components (Gagné, 1985; Gagné, 2000). The first consists of giftedness ability domains, which could be generic or specific domains, including intellectual, creative, socio-emotional, sensorimotor, and others. The second component is formed by catalysts that could be interpersonal or derived from physical or psychological aspects, which could be partially under the influence of genetic effects, as well as motivation, which could be an important psychological catalyst. Another type of catalyst could relate to environmental aspects that surround gifted students, including parents, teachers, and gifted education programmes, which can play a positive role in development. In addition, chance can be a causal aspect related to both environmental or genetic catalysts. The third component of the DMGT model consists of specific fields of talent, which can be important components of both generic and specific fields of giftedness. Skills in certain human activities should not be ignored and should be considered talents.

Gagné referred to CS as a field that requires various talents or skills, including hardware design and programming. However, programming in this context refers to the ability of coding rather than being a sophisticated, cognitive activity that includes reasoning, problem solving, abstraction, and mathematical abilities. Thus, Gagné's

view of programming might not be accurate, leading him to consider programming to be a field of talent but not a field of giftedness.

Gagné (1985) criticised the three-ring conception for ignoring the underachieving gifted student who might score a high grade in an IQ test, yet might lack motivation and/or task commitment. However, Renzulli's giftedness theory is less restrictive in identifying gifted students, where different methods of identification can be used rather than exclusively an IQ test. Further, Renzulli introduced the revolving door identification model (RDIM), which will be discussed later in this section, to overcome restrictions in identifying gifted students and to be more inclusive (Renzulli, 1984b).

2.1.4 Enrichment and Acceleration

It is unlikely that students in a class will all have the same learning styles, abilities, or pace. Some students learn at a faster pace or want to explore more. However, some education systems cannot meet the needs of these students.

Acceleration enables gifted students to move through curricula more quickly than the rest of the class. Types of acceleration could include single module acceleration, grade skipping, early school enrolment, and advance placement courses (AP) (Colangelo et al., 2004). Acceleration does not mean pushing students beyond their limits to learn advanced knowledge, but rather providing a flexible educational track to meet the needs of different individuals.

The term *enrichment* refers to curriculum changes or developments to provide richer content and a variety of learning experiences. In addition, an enriched curriculum should be planned carefully, based on gifted learner characteristics to provide them with challenging content, which is more advanced than the normal curriculum (Colangelo & Davis, 2002).

Gifted education aims to enrich students with more learning materials, learning experiences, teaching methods, and guidance to fulfil their individual learning pace needs. It is unlikely, for example, to have a class full of students who can learn at the same level, and certain curricula cannot cater to different learning paces (Renzulli,

1976). Gifted students may enjoy being selected for enrichment programmes so they can have the freedom to choose from a variety of resources to be more engaged.

2.1.4.1 Enrichment Triad Model

Renzulli (1976) suggested three related types of enrichment: (1) general exploratory activities, (2) group training activities, and (3) individual and small group investigation of real problems. The first two approaches, which are generally necessary for any enrichment program, aim to develop student thinking abilities as well as to extend student interest. The third approach is considered to be an important area for gifted students.

The enrichment approach can provide a wide range of topics, disciplines, hobbies, and/or events added to the regular curriculum. These additional activities can be used to train gifted students to develop a wide range of characteristics and skills and may include creative thinking, critical thinking, problem solving, communication, and skills that can be learnt from advanced reference materials. The enrichment focuses on gifted students who manifest interest in specific areas and are keen to spend more time training and acquiring more knowledge and skills.

Renzulli (1976) explained the aims of enrichment as follows:

1. Focus on self-selected specific areas of study and enable gifted students to acquire knowledge and to produce creative ideas as well as manifest task commitment;

2. Increase the knowledge of gifted students and their thinking processes related to a specific field of study;

3. Develop genuine products that can benefit a particular targeted audience;

4. Provide gifted students with self-learning skills to enable them to plan, organise, utilise resources, improve time management, and evaluate their performance;

5. Develop task commitment and confidence in gifted students.

2.1.5 Identification Methods

Research into giftedness has primarily focused on identifying gifted students by various processes, such as IQ tests, teacher observations, and academic achievement, to provide the gifted student with an enriched curriculum or acceleration programme. The purpose of giftedness identification is not only to find gifted students but also to provide for their educational needs (Colangelo & Davis, 2002).

2.1.5.1 Principles of Giftedness Identification

Six giftedness identification principles were derived from a national panel of experts (Colangelo & Davis, 2002).

1. Defensibility: identification tools must be derived from the best research results and recommendations;
2. Advocacy: identification tools should be designed based on student interest and should not harm students;
3. Equity:
   (a) The identification process should not overlook any student. All students from different groups should be represented according to their demographic characteristics;
   (b) Students should have the civil right to have equal access to designed programmes;
   (c) To identify disadvantaged gifted students, strategies should be clearly specified;
   (d) Cut-off scores should be avoided, as they can disadvantage students;
4. Pluralism: 'the broadest defensible definition of giftedness must be used';
5. Comprehensiveness: as many gifted students as possible should be identified;

6. Pragmatism: 'procedures should allow for cost-effective modification of available instruments and personnel'.

These principles underpin identification in general. Various researchers have suggested different methods to identify giftedness. These methods may include test scores, grades, interview performance tasks, and recommendations (Coleman, 2003). Implementation of these methods may vary based on the different contexts in which single or multiple methods can be used.

However, some argue that essential traditional assessments, such as IQ tests, teacher recommendations, and parent questionnaires are inadequate when identifying gifted low-income students (Passow & Frasier, 1996). Cost-effective methods, such as portfolios, nominations, and non-traditional standardised tests may help this type of student to be identified as gifted.

Another type of identification method is the non-verbal intelligence test, which aims to reduce possible language difficulties among different ethnic and socioeconomic groups. In addition, there is a need to introduce specific methods to help detect cognitive abilities that do not appear to be considered within existing tests. Dynamic assessment is a process of test, intervention, and retest to monitor student improvement after an intervention (Bracken & McCallum, 1998). Additional assessment, which may help to assist cognitive ability, includes performance tasks in which open-ended challenging questions can be designed to understand a student's cognitive process rather than simply getting the right answer (Bracken & McCallum, 1998).

2.1.5.2 Revolving Door Identification Model

Although various identification methods have been developed to help a teacher identify gifted students to be a part of gifted education programmes, it is very important that each identification method is derived from a particular giftedness definition (Renzulli, 1984b). For example, if giftedness is defined as an outstanding score on a linguistic test, an enrichment programme could be designed to enhance linguistic ability. Accordingly, an identification method must be relevant to specific characteristics that have been

defined. The RDIM is based on the enrichment triad model in which both models are derived from Renzulli's three-ring conception of giftedness. The RDIM aims to overcome various obstacles to gifted student identification.

2.1.5.2.1 Main goals of RDIM

The goals of RDIM were identified by Renzulli (1984b). The goals were summarised as follows:

1. Provide multiple levels of enrichment to a wide range of students, not only the top 3% to 5% of the student population;
2. Create a cooperative learning environment by integrating the enrichment programme into classrooms;
3. Reduce concerns regarding elitism and negative behaviour expressed towards gifted students for being part of an enrichment programme;
4. Improve the quality of enrichment programmes for a wider spectrum of students.

2.1.5.2.2 Principles

Renzulli (1984b) outlined principles for RDIM identification.

1. Various techniques should be used to identify gifted students because:
   (a) Giftedness may be manifested in several ways;
   (b) Giftedness may emerge at certain times within specific circumstances;
2. Gifted identification needs to be based on individual knowledge, the cultural environment, and a specific area of study;
3. The identification method should consider self-nomination;
4. The identification method should consider reasonable freedom to reflect the method;
5. The identification method should be adequate to be re-evaluated on a regular basis and that can include;

(a) Following up on the selected student and others;

(b) Modifying, adding, and/or deleting specific identification tools;

(c) Reconsidering a student who has not been nominated to be a part of the enrichment programme;

6. Evaluation of identification methods should provide feedback to enhance the programme.

2.1.5.2.3 RDIM methodology

The first step to implement RDIM is to form a 'talent pool' group by selecting students who are in the top 15% to 20% of the student population in a general and/or particular field of study. The RDIM provides identification methods to include a wide spectrum of students. There are three reasons to include the top 15% to 20% in the talent pool. First, it has been shown that students who manifest signs of developing task commitment and creativity and who have above-average ability but may not be outstanding have a high chance of gifted behaviour (Renzulli, 1984b). Second, activities provided to the top 3% of students can be suitable for a larger student population. It could be argued that high performing students will still achieve good outcomes when working to an enriched curriculum. The RDIM method to identify a talent pool relies on four basic items of information which are (Renzulli, 1984b):

1. Psychometric information obtained from creativity, aptitude, achievement, and a typical intelligence test;

2. Developmental information gathered from family, teacher, self-nomination, and rating scales;

3. Sociometric information collected from rating and peer nominations;

4. Performance information gathered from previous studies.

It is important to mention that RDIM provides sufficient flexibility to include other criteria or tools that have been designed by teachers or schools. In addition, if any

student has been overlooked during the identification process, a special nomination can be used to overcome this situation.

The second identification step is involving students in an advanced level enrichment and acceleration experience. At this stage, enrichment aims to provide two important factors, which are to involve gifted students in advanced enriched experiences and to provide a method to move students from the talent pool to be a part of the third type of triad enrichment model that is concerned with individual and small group investigations of a real problem.

The implementation of this step is based on the concept of action information, which refers to all dynamic interactions that can be observed when a gifted student manifests tremendous interest in a certain field of study, topic, or specific problem.

In terms of gifted education and gifted identification in CS in general, and in programming in particular, the concept of action information has not been extensively explored, resulting in a gap in the literature. However, there is a considerable amount of literature that explores high-achieving programming students and the need for an enhanced curriculum, an innovative teaching methodology, the provision of competition, and additional activities to extend student abilities (Han & Beheshti, 2010; Beck & Chizhik, 2008; Carter et al., 2007).

However, few studies address how to identify gifted students in programming and how to provide enriched programmes to keep them motivated. It could be argued that those gifted students who may perform well on their own do not require as much attention as struggling students. However, it is not fair for a gifted student who may have paid tuition fees expecting to gain advanced knowledge and outstanding skills to become demotivated because it is thought that helping struggling students is more important.

In previous work relating to high achieving students in programming, the teaching over-performing students (TOPS) competition aimed to motivate and develop first-year CS students who showed high programming ability in classes across different universities (Carter et al., 2007). Although the programming competition may have

motivated high-achieving students, the researchers did not explain precisely how they identified high-ability programming students to be competitors, and there was no definition of what they meant by 'programming ability' (i.e. problem solving versus coding versus ability to understand programming abstractions). Another concern regarding the way in which the competitors were identified relates to equity, where all students should have the chance to be identified to participate in such an enrichment program as mentioned by Carter et al. (2007).

The study by Davis et al. (2001) aimed to provide different programming sessions based on the different programming experiences of first-year CS students. The author classified students according to the answers they gave to an initial skills questionnaire to determine whether they had the necessary experience to join enriched 'space cadet' sessions. Students with only fair programming knowledge had to attend normal class and results showed that 40% of the 161 participants were classified as 'A-Level Computer Science'.

These types of classification questionnaires raise the issue of accuracy. The authors of that study focused on providing different content based on previous student experience, whereas it could be argued that experience might not be a key factor when identifying a gifted programming student.

Jenkins and colleagues proposed a similar approach to deal with a variety of student experiences in programming (Jenkins & Davy, 2000). Based on the authors' experience at Leeds University, they suggested four categories of students colloquially named: Rocket Scientists, Strugglers, Serious Strugglers, and Averages. They categorised students based on a self-assessment of their experience of an interview with a faculty member during the early enrolment period. A second, more accurate attempt to categorise students was to test the students' skills to overcome problems related to the first attempt, where variables, such as pressure and over-confidence resulted in inaccurate student self-classification. In the final stage of their method, each group was provided a specific study path.

In this study, authors realised that students needed to be categorised to provide

them with a specific study path according to the level of their experience and knowledge. However, a lack of understanding of programming characteristics and identification methods resulted in a lack of accuracy of student categorisation. Therefore, two attempts were conducted to enhance the categorisation processes. The first attempt used self-assessment, whereas the second attempt used self-classification. This is an example of how the lack of a theory of giftedness in programming leads to uncertainty of how to identify gifted student programmers.

Throughout the literature, a lack of theoretical foundation in defining giftedness in programming and gifted student identification have resulted in controversial ways to address gifted student learning. In addition to issues related to the term *gifted* and what it takes to be gifted in programming, previous work used various terms to address 'gifted' students, namely A-level computer science, rocket scientists, and over-performing students based on their initial programming experience (Jenkins & Davy, 2000; Davis et al., 2001); however, there is no clear identification method (Carter et al., 2007). Thus, can we call the above students 'gifted in programming'? This brings us again to the point where there is no precise definition of gifted programming students to assist us to identify them in a proper way, as gifted educationalists have pointed out (Colangelo & Davis, 2002). Thus, it is important to understand the nature of programming to highlight characteristics of gifted student programmers and how these characteristics can be developed in an educational context.

This section of the literature review explored the territory of gifted education, discussing the main principles of the field including giftedness theories, enrichment, acceleration, and methods of student identification. We now need to review aspects of programming psychology, epistemology, and pedagogy to understand how programming knowledge can be acquired by students.

2.2 Psychology of Programming

To understand the characteristics and cognitive abilities that are required to learn programming, we need to review programming pedagogy. Early programming defini-

tions considered that programs describe calculations that represent a sequence of state changes, using languages such as Fortran. Another approach to programming then emerged, focusing on function. The functional programming paradigm originated from pure mathematical theory, such as lambda calculus. Function notations can be represented as mathematical expressions to be evaluated based on rules where the value of an expression can be computed based on its arguments (Aaby, 2004). Another programming paradigm, which incorporates both approaches, describing calculations and functional programming, is object-oriented programming (OOP) that involves more data processing, such as building functions, classes, and objects that have attributes to describe objects. The last paradigm is the logic programming paradigm, which relies on logical relations derived from a problem statement that consists of rules, facts, and a goal statement. The rules and facts are written in logical expressions and can be evaluated to prove the goal statement(Aaby, 2004).

In terms of software development, programming is a process that fulfils requirements derived from analysis to achieve desired goals. However, another view of programming suggests two important aspects: programming and coding. Programming involves drawing the schedule of situational instructions for each programming task, while coding is considered a complicated task, which originally used assembly languages, known as low-level programming languages to denote low abstraction between the language and the machine. In contrast, high-level programming languages have been developed to establish human-based languages to communicate naturally with computers.

Regardless of the type of programming language, it must be considered a tool that can help to transform a high abstraction problem into a low-level machine process. For example, a study of expert programmers who solved a problem using multiple languages showed that the chosen languages had a minor influence on the solution (Petre, 1990).

Programming languages have limitations when attempting to simulate natural languages. In addition, they could have a high level of abstraction. For instance, a

novice may encounter issues trying to transfer a natural language into programming language semantics and logical expressions (Hoc & Nguyen-Xuan, 1990).

Another programming characteristic that can be derived from programming languages is the complexity of mapping the abstraction of programming concepts to be more humanly understandable. Thus, programming can be defined as a complicated task to find a mathematical solution using a convenient problem-oriented language (Blackwell, 2002).

2.2.1 Why Programming Is Difficult

Psychologically, the human cognition process includes several abilities, such as attention, perception, memory, language production, problem solving, and reasoning. Likewise, computer programming is considered a sophisticated human activity that involves several programmer skills, including analysis, design, implementation (coding), and evaluation, which are considered cognitive abilities. In addition, crucial social skills, such as communication, are essential factors, as a programmer must identify and analyse specifications that can result from design meetings.

One reason that programming is a sophisticated human task is that these cognitive abilities may need to be simultaneously incorporated with each other during programming (Hoc, 2014). As programming is a human activity, it is important to identify similarities between programming and human activity to conduct more effective research. Programming activities can be derived from professional contexts or from the study of CS and software engineering. Studying these activities in depth may allow more effective development of human programming tools, such as those used for designing and modelling.

One similarity that can be derived from both programming and human activities is the cognitive process, which could be a major cause of programming difficulty. It has been suggested that common difficult aspects of programming tools may be the loss of direct manipulation and introducing abstract representations (Blackwell, 2002).

From a cognitive science perspective, loss of direct manipulation and image

representation can reduce the effects of framing a problem. In other words, when a person tries to draw a mental representation for a specific changeable situation, the representation will depend on simulating changes that may affect the situation. In this case, the representation may be affected by an unlimited range of factors that may affect the situation and consequently how it is represented. If all factors that may affect the situation are considered during the cognitive representation process, direct manipulation may result in a more accurate representation. The benefits of cognitive direct manipulation result in more control where the user can continuously be able to represent the current situation, as a single action must have a unique visible influence on representations (Blackwell, 2002).

However, in the case of programming, direct manipulation may not be applicable, as a programmer cannot have control of unexpected changes of such a program. For example, a program may need to be run in a certain time where a certain rule is satisfied or the program may need to deal with a huge amount of data, which is not available currently for the programmers. Consequently, abstraction can be an obstacle in programming, as we no longer have direct manipulation of all factors that may affect program implementation over time and/or space. Blackwell (2002) described abstraction as a 'results from forming some representation of the state of the world either a mental representation, a linguistic representation or some other representational system'. Abstraction in programming will be discussed in more detail in Section 2.3.

Another aspect suggests that notations have been introduced to minimise the abstraction level and help the programmer to complete programming tasks. However, Green (1989) found that notations have limitations describing important constraints, as notation designers try to mitigate the effects of abstractions.

Another way to help programmers overcome abstraction factors, such as the loss of direct manipulation and use of notations, is to use programming development environments, often called integrated development environments (IDEs). Basic programming tasks to program, for example a microwave, allow the user to use predefined functions and simple notations, whereas sophisticated programming tools provide pro-

grammers with a way to build functions and to define new notations, which can lead to increased complexity (Lahtinen et al., 2005).

Programming may be considered a complex human activity that involves multiple cognitive abilities. These may be incorporated synchronously to solve a problem according to a set of computation rules, such as languages and logics, which may lead to abstraction. It is essential to account for the programming context in which our research focuses (i.e. on education aspects, which may lead students to professional domains).

In a professional context, the problem can be divided into several tasks that can be tackled by teams of designers, analysts, and programmers. However, that is not possible in the educational domain where other important factors, such as knowledge acquisition and problem-solving skills must be considered when defining programming (Blackwell, 2002). Thus, the definition of programming should consider the experience of the student programmer to use cognitive abilities devoted to problem solving.

2.3 What Is Needed to Be a Good Programmer?

In this section, the programmer characteristics derived from educational and industrial contexts will be discussed.

2.3.1 Mathematical Ability

> Computer Science and Engineering is a field that attracts a different kind of thinker a natural computer scientist thinks algorithmically. Such people are especially good at dealing with situations where different rules apply in different cases; they are individuals who can rapidly change levels of abstraction, simultaneously seeing things 'in the large' and 'in the small' (Knuth, 1992).

The role of mathematics in learning programming is still a debated issue among CS educators. Through the history of the epistemological development of CS, mathe-

matics has been the core knowledge and skill for computer scientists. In the early 1970s, the rapid growth of the study of computational complexity and formal verification was included in CS curricula, resulting in seven mathematical modules being incorporated into curricula (Ralston & Shaw, 1980). Some argue that the discipline of CS is simply another paradigm of mathematics (Hartmanis, 1993) but with the recent and rapid development in CS, these claims could be questioned. The rapid switch in programming paradigms from imperative, logical, object-oriented, and functional to multi-paradigm programming languages might influence how many mathematical concepts need to be taught. For instance, Python is a popular functional programming language that has gained popularity because it is simple to learn and has a certain degree of abstraction, as the computation process is done based on calling functions. These functions can be built as open source, which allows other programmers to implement functions without understanding the computations behind the functions. ln contrast, a logic-based programming language might include many computations, requiring knowledge of mathematical logic and proof.

Mathematical ability might be considered an indicator of programming aptitude, as mathematics provides a variety of skills, such as reasoning and problem solving, that are required during computational thinking. Mathematics was considered a significant factor when predicting student success in programming (Pacheco et al., 2008; Watson & Li, 2014) as well as designing PATs based on mathematical ability. However, Henderson and Stavely (2014) claimed that most mathematical concepts have not been used effectively in programming modules and that the important relations between mathematical concepts and CS fundamentals have not been made apparent to students. For instance, it is essential that students must acquire basic algebra and logic to help them learn programming fundamentals, whereas it might be confusing for students to understand the role of calculus in an introductory programming course.

However, introducing DM basics may help first-year CS students grasp the basics of programming and could be a much more valuable foundation for the third and fourth years of instruction, when advanced computing courses are introduced. A study

conducted by Pioro (2006) suggested a positive correlation between student grades in a computer programming course and the averages of two grades obtained in DM and Calculus. However, if the two modules were instead Calculus 1 and Calculus 2, the correlation between the mathematics grades and the programming grades was insignificant.

Only some CSE researchers have examined the role of mathematics in learning programming. Research has more commonly focused on how mathematical ability could affect student performance in programming (Pioro, 2006; Wilson & Shrock, 2001; Gomes & Mendes, 2008). Mathematical abilities, such as problem solving, reasoning, and abstraction, have been predictors of student success in programming (Simon et al., 2006; Tukiainen & Mönkkönen, 2002; Power et al., 2011) and are important factors for designing PATs.

Several studies have investigated the relationship between a mathematics background and programming aptitude in students. Pacheco et al. (2008) investigated the assumption that a lack of problem-solving ability could lead to difficulties in learning programming. The grades of two cohorts of first-year university CS students at different institutions were analysed to identify correlations with different learner characteristics, such as programming background, problem-solving ability, motivation, and learning style. The results shown in Table 2.1 indicated a positive correlation between programming and calculus ability as well as a relationship between mathematics grades in secondary education and performance in programming.

Study	Variable 1	Variable 2	N	Correlation
(Pacheco et al., 2008)	Calculus	Programming with C	59	0.49
(Pacheco et al., 2008)	Calculus	Programming with Python	36	0.41
(Pacheco et al., 2008)	Math	Programming with Python	36	0.37
(Harris, 2014)	Math SAT	PAT	16	0.54
(Tukiainen & Mönkkönen, 2002)	Pre-uni math	CS1 programming	33	-0.28
(Bennedsen & Caspersen, 2005)	Pre-uni math	CS1 programming	20	0.39
(Bergin & Reilly, 2005b)	Pre-uni math	CS1 programming	30	0.46

Table 2.1: Previous research correlation results for mathematics and programming.

Another study that predicted a variety of success factors in programming also concluded that mathematics grades from high school positively correlated with programming exam grades (Bennedsen & Caspersen, 2005). A further study found a pos-

itive correlation between standardised high school scores in mathematics and student performance in university-level programming (Harris, 2014). Bergin and Reilly (2005b) also showed a mathematics background plays a positive role in programming performance. However, research on using a PAT and mathematics grades from high school to predict university student programming outcomes is questionable as Tukiainen and Mönkkönen (2002) concluded with controversial findings indicating a statistically insignificant negative correlation, as shown in Table 2.1.

Issues related to the previous studies include sampling methodology and size. For instance, in Pacheco et al. (2008), the two sample cohorts were from different institutions, so multiple educational factors could confound their results. For example, teaching quality or curricula could positively or negatively influence student performance. A small sample size could also affect the statistical significance of the correlation. Thus, it was decided further to investigate the potential correlation between mathematical ability and programming, as the literature suggested that mathematical ability is an essential aspect of learning programming. Therefore, to determine whether mathematical ability is an important characteristic for identifying gifted student programmers, the relationship between mathematics and programming needs to be investigated.

Based on previous research, we can confirm that the relationship between mathematical ability and programming does exist. However, the relationship depends on different factors including different types of mathematics and different programming paradigms. In addition, mathematical ability is an important characteristic that can help students to be a good programmer and can be an indicator of student programming aptitude to some extent. However, mathematical ability is not the only characteristic that a good programmer possesses.

2.3.2 Problem-Solving Ability

As problem-solving ability is a part of the human cognitive process and is an important ability required during programming, we find it helpful to understand problem-solving

strategies in general to help us teach programming according to these strategies. A problem consists of four aspects: initial state, operators, restriction, and goal state. Operators may have restrictions and aim to provide a fuzzy path to solve a problem, starting from the initial state to the goal state within the problem space (Ormerod, 1990).

Programming is viewed as a problem-solving activity that simulates human problem-solving ability in which a programmer needs to find a solution that represents the goal state. Programming language semantics and syntactics, which may cause a language restriction, could be considered operators. Programmer cognitive ability also represents an operator and may affect the way in which the problem can be represented.

Mason et al. (2010) defined three stages for solving mathematical problems: entry, attack, and review. Another well-known problem-solving strategy introduced by Polya (2004) aims to break down mathematics problems into four phases, as follows:

1. Understand the problem;
2. Devise a plan to solve the problem;
3. Carry out the plan;
4. Look back.

This strategy is derived from the fact that programming is initially defined as a calculation description and could be applied to help students learn how to program. In programming, students may develop cumulative experience solving problems and other similar programming tasks. Following Polya (2004), a modification to the original strategy has been introduced by Vickers (2008), adding a fifth phase - a description of what students have learnt from the process - to put more emphasis on memorisation and problem understanding.

As a good programmer, developing a program needs to include documentation that could consist of internal comments to explain written code in a simple way so that others can understand it. A sixth phase was therefore added by Polya (2004) to explain results and to document written code. The purpose of this phase is to help

other programmers maintain written code by debugging some errors or updating some part of the program, which may have been costly or time-consuming.

2.3.2.1 Knowledge Organisation

In general, problem-solving ability depends on student knowledge organisation. McKeithen et al. (1981) suggested that knowledge organisation among experts differs from that of novices. The expert shows more ability to recall and organise related groups of meaningful information that may result in more accurate performance, rather than processing single information. Binstock (2012) stated that knowledge organisation allows a programmer to master multiple programming concepts, such as data structures and algorithms, and is what makes a good programmer who can apply these concepts to understand the wider picture. Joseph (2015) stated that knowledge of data structures and algorithms is one of the most important aspects of being competent in programming.

Table 2.2 summarises the differences between novice and expert programmers.

	Expert	Novice
Problem Representation	Algorithm	Application
Knowledge Organisation	Recall chunk of related information	Recall of general information
Coding Strategy	Backward (breaking the goal into small tasks) Frequent debugging Comments	Forward (line by line)
Programming Language	Semantic	Syntactic

Table 2.2: Novice vs expert programming characteristics.

2.3.3 Ability to Abstract

Computer science educators have investigated the role of abstraction ability in learning programming, unpacking different aspects on assessing student abstraction ability and understanding the influence of abstraction on cognitive abilities.

Teague and Lister (2014) conducted a longitudinal research to measure cognitive

development based on a neo-Piagetian theory defined for programming by Lister (2011). The theory consists of four levels:

1. Sensorimotor, where the student is not able to execute a piece of code nor able to determine a variable value (code trace);

2. Preoperational, where the student can determine a variable value but is not able to understand the big picture of the code;

3. Concrete operational, where the student can read, trace, and understand short pieces of code;

4. Formal operational, where the student is able to write a program solving a problem, exhibiting all aspects of problem-solving abilities, including identifying problem requirements from the description, decomposing the problem, solving sub-problems, recomposing sub-problem solutions, and evaluating the solution (Teague & Lister, 2014).

Their study aimed to observe student cognitive development over long periods, conducting a series of 'think-aloud' sessions to observe and record student behaviours and answers to programming questions, including code trace tasks. Although the sample size was only one student, the study provided qualitative results that suggested the cognitive ability of the student had developed. His answers for the first think-aloud task were categorised as sensorimotor, but after three semesters, the answers gradually improved, and he transitioned to higher levels of cognition, such as concrete operational. As the student progressed in his study, he was able to provide reasonable explanations of the iterations used for loops in more abstract terms rather than explaining the implementation details.

Koppelman and van Dijk (2010) investigated student abstraction abilities and the influence on learning programming by analysing three programming problems for an introductory programming course. The author emphasised the importance of abstraction ability in learning programming and argued that abstraction is a prerequisite ability for designing and writing a computer program. Moreover, abstraction is what

makes a good software engineer who can produce a clear software design and can write elegant code. In addition, a CS student requires abstraction ability, which can be innate or developed.

Although Teague and Lister's (2014) study sample was small, the results suggested that cognitive abilities including abstraction can be developed through a period of learning programming at the university level. However, individual differences in nurturing cognitive abilities still influence outcomes, and some students may struggle with learning abstract concepts in programming (Kramer, 2007). The study addressed student difficulties in learning abstract concepts in programming and addressed pedagogical concerns to improve the perspectives of CS educators on teaching abstraction. The method of the study was to analyse three programming problems (nested loop, nested SQL queries, and recursive functions), understanding the abstraction roles for these tasks, and how abstraction can be taught and assessed.

The first and second problems shared the same concepts of nested loops. For example, in the first problem, an array a of 100 integers i was given to find an element $a[i]$ that had the property that the sum of $a[i]$ equals 10.

The logical way to solve this problem would be to find the first element of the array and add the digits and check whether the sum equals 10. Otherwise, the function would keep looking to the next element of the array, repeating the same process until the problem was solved.

When a novice programmer writes code to solve this problem, nested loops are required, but abstraction cannot be recognised. Thus, teaching abstraction explicitly in this problem might help a novice programmer to recognise abstraction. In addition, Hazzan and Kramer's (2016) study suggested abstraction should be considered a 'fit for purpose' programming task where abstraction should be directly explained by instructors.

Recursion in programming is a function that is written to perform infinite computations by calling upon itself until a certain condition is met. Recursion is known to be a difficult concept for novice programmers to learn, as the nature of the problem

may require a certain degree of abstraction ability.

Koppelman and van Dijk (2010) discussed an example of the recursion problem where the student was required to count the number of digits in a given positive integer of base 10. A possible algorithm, such as algorithm 1, is to check whether the integer is less than 10; if so, the function will return the number of digits to one. Otherwise, the function will recursively and incrementally count the position of the digit in the given integer until the function reaches the last digit that can be determined, when the result of dividing the integer by 10 equals 0.

Algorithm 1 For counting the number of digits of integers.

```
1: n ← positive integers
2: procedure NumOfDigit(n)
3:     base ← 10
4:     if n <base then
5:         return 1
6:     else
7:         return NumOfDigit (n / base) + 1
```

Students tend to write all recursion calls and trace the process by hand when trying to understand the computation process. In addition, the literature on learning abstraction suggests that novice programmers tend to simulate what the computer processor does to solve recursion problems. Novice programmers focus only on recursion implementation, missing what is happening behind the scenes (Ginat, 2004; Sooriamurthi, 2001).

Students struggled to be abstract from the function implementation tracing the function calls, which was not the purpose of the recursive function. Thus, this study addressed concerns related to teaching programming concepts requiring abstraction ability that can be challenging for novice programmers. They recommended that instructors identify and explain abstract concepts related to specific programming problems to students. Further, abstraction concepts should be taught at early stages, emphasising their importance and implications in CS (Koppelman & van Dijk, 2010).

Hazzan and Kramer (2016) also attempted to understand how to assess abstraction ability. Unlike Kramer's (2007) study that focused on measuring student

abstraction through specific programming problems, Hazzan and Kramer (2016) developed 10 different patterns of questions to assess student abstraction abilities and asked 11 experts to answer three open-ended questions relating to abstraction presented in these patterns. These patterns, which were designed to be more generic than a specific problem, allowed experts to provide open opinions on identifying possible abstraction concepts and implementations from each pattern. The study indicated that the pattern with the highest expert agreement and the most suitable pattern to measure student abstraction abilities was pattern 5. In pattern 5, the student was asked to construct two levels of representations of a given system representation where the representations given by the students should be one more and one less abstracted than the given representation. In contrast, the pattern of measuring abstraction with the lowest expert agreement was pattern 2, where the student was asked to rank multiple representations of a system based on the abstraction level for each representation. Although the study suggested a unique method for measuring student abstraction abilities by constructing abstracted representations of systems, further investigations are needed to repeat the study to ensure constant results from different sets of participants.

Another study measuring abstraction ability was conducted by Bennedsen and Caspersen (2006), questioning the hypotheses related to the relationship between programming and mathematics, and cognitive development.

The study had two aims. The first aim was to investigate the correlation between student programming performance, measured by student grades in programming modules, and cognitive development as an abstraction ability. Student abstraction abilities were measured by reasoning tests developed by Piaget, where abstraction ability was categorised based on Adey and Shayer's cognitive development stages (Bennedsen & Caspersen, 2006).

The second aim was to investigate the relationship between mathematical ability, measured by student grades in high school, and abstraction as a cognitive ability. The results yielded unexpected conclusions, as the statistical Pearson correlation coefficient test on $n=145$ observations yielded a weak correlation between programming

and abstraction ability. The correlation test of *n=128* observations yielded a result of *r = 0.18*, indicating very weak correlation between mathematics and abstraction.

The results contradict the large body of literature discussed earlier that suggested abstraction is a key factor in learning and in success in programming. In addition, abstraction is one important component of mathematical ability, which suggested that the relationship between abstraction ability and mathematical achievement does exist. Thus, Bennedsen and Caspersen's results raised a number of concerns around the instrument that had been used to measure cognitive development and programming and mathematical performance.

2.3.4 Coding Strategies

A programmer should also be able to analyse multiple scenarios that might affect the implementation of software, considering, for example, a scenario of receiving null arguments. In addition, Warne (2014) emphasised the skill of naming different components of programming, such as functions, classes, and variables. Names should be clear to represent a programmer's thoughts, as the program becomes self-documented and can be readable and easy to understand.

Being consistent in naming variables, declaring functions, handling bugs, and commenting is a required programmer skill to reduce complexity. For example, when adding arguments in a function declaration statement, it is recommended to follow the same order of a database to help identify missing arguments.

Software developers should have the ability to learn new language features and existing code to allow new functionalities to be integrated. In addition, being a programmer requires being passionate and a 'life-long' learner to cope with the rapid changes in the field. Brooks (1995, p. 7) noted that a passion for learning is one important aspect that makes programming interesting.

Coding is one activity of the programming process where programmers convert an algorithm to be understood by a computer through a programming language. Different programmers have different coding strategies to express thoughts based on specific

algorithms that have been designed to solve a problem. We should bear in mind that certain programming languages have limitations preventing certain algorithms to be implemented and coded. As a result, each programming language has strict syntax.

There are certain coding strategies that are considered to be best practices in coding derived from experience, language rules, and language paradigms. Applying certain coding strategies allows the code to be readable and understandable by other programmers.

McConnell (2004) identified numerous coding strategies in his book *Code Complete*. Writing a high-quality function, which allows a programmer to encapsulate a number of programming statements to compute a specific problem, requires the programmer to consider a range of perspectives. A function must be written to avoid code duplication and to reduce complexity by breaking down a whole problem into smaller problems solved by writing small functions. In addition, functions can be used to hide details and to group codes that share the same behaviours and functions, simplifying code testing and debugging (Warne, 2013). A function must have a header, a declaration statement that includes the function name, and possible arguments.

Naming the function, which could apply to naming variables and classes, should represent what the function does as well as the function having a brief description of what it does and what parameters it should receive. The written description should be placed before the function header. Self-documented code should include a meaningful general description of the whole code in terms of the main purposes of the code, and the code should have in-text comments to explain the purpose of certain iterations, functions, or variables. Committing code that has been updated can keep track of all changes to be noted later by either programmers or other developers.

Code indentation is another best practice, allowing code to be readable, modifiable and understandable, especially when working with groups of programmers or when revisiting segments of code is needed. Constant code indentation has been recommended to, for example, distinguish between levels of iterations to make the body of a function more recognisable.

Programming errors may be syntactic or semantic. The syntactic error, which is detected by the programming language compiler, prevents code from being executed if a programmer did not follow the programming language grammatical rules. In contrast, semantic errors can be logical errors and could be hard to detect by a programmer. These errors may not prevent the code from being executed but may result in unexpected results. A programming language compiler would then display useless and/or ambiguous error messages.

Handling errors in debugging is a basic skill that any programmer should possess; yet, there are different techniques ranging from basic to advanced. For instance, returning neutral values, such as returning numerical values of numerical computations, is a basic debugging technique. A programmer who masters different debugging techniques, such as printing error messages indicating which line includes the error to reduce the amount of examined code, has more chances to discover and fix code bugs. In addition, the technique of exceptions allows a programmer to handle unexpected errors that have been caught during runtime (McConnell, 2004).

2.3.5 Attitudes and Personal Traits

McConnell (2004) devoted a full chapter to discussing programmers' personal traits and distinguishing between intelligence and personal traits. McConnell stated that intelligence, which might not be associated with good programming, can be an innate ability, whereas personal traits can be developed, allowing a programmer to be successful. Personal traits may include learning style, communication, cooperation skills, curiosity, and creativity. Learning style can be important in the educational context, which might affect the student learning process during lab sessions where pair programming is used. In addition, communication skills and cooperation skills might affect the learning process. In addition, personal traits are important in the IT professional context.

2.3.5.1 Learning Style

Research on investigating learning style in CS depends on quantitative studies using instruments, such as the Keirsey Temperament Sorter, the LSI and the Myers Briggs Type Indicator (MBTI) (Galpin et al., 2007; Seyal et al., 2015; Venkatesan & Sankar, 2014; Kanij et al., 2013). Galpin et al. (2007) investigated the personalities and learning styles of CS students using the Keirsey Temperament Sorter and the LSI tests. Responses were collected from 226 students who studied first-, second-, or third-year CS, suggesting that the majority were convergers or assimilators, where the number of convergers increased with student progression in the years of study, and the number of assimilators decreased. In addition, the majority of student learning styles were based on abstraction and understanding concepts. Some of the first-year students preferred to learn through active experimentation, but as they progressed into the final years of study, their learning became balanced between active experimentation and reflective observation. However, the study did not include relevant information indicating any correlation between learning styles and academic achievement.

Another study by Seyal et al. (2015), which aimed to investigate the relationship between CS student learning styles using the LSI, performance based on programming module results, and a demographic survey, indicated that learning style can affect student performance. The study results showed that 'convergers' and 'assimilators' were predominant (46% and 42%, respectively), which suggests behavioural differences between students. In addition, statistical results indicated that 39% of 'convergers' and 15% of 'assimilators' managed to successfully pass, whereas 100% of 'divergers' and 99% of 'accommodators' passed. Another interesting finding from that study was that gender affected learning style, so that female 'convergers' were predominant (31%) followed by 'divergers' (15%), whereas males were 'convergers' (23%) followed by 'assimilators' (15%).

2.3.5.2 Information Technology Profession Characteristics

We can gain further information about what good programmers can do by examining characteristics of IT professions in the industry context, the job of a software engineer is to work closely with a client to develop software where the software engineer addresses client requirements, software design, implementation, and evaluation.

A recent study conducted by Li et al. (2015) explored characteristics of good software engineers by conducting semi-structured interviews with 59 Microsoft employees working at different levels of positions, including experienced software developers. During the one-hour interview, participants were asked to identify attributes of good software engineers with whom they had worked. The researchers used a grounded theory approach to code interview transcripts and identified 53 attributes grouped into four themes: personal traits, decision making, teammates, and software products. A good software engineer should manifest certain personal traits, and they should be passionate, curious, hardworking, and self-taught.

While decision making means that a software engineer should have knowledge of the organisation and people, they should also be able to tackle complex issues related to technical problems, trying to understand issues underlying the complexity. The ability to abstract enables a software engineer to consider multiple factors that might affect producing software, including technical, organisational, and business needs.

The theme of teammates describes creating a shared environment that allows people to understand each others' views and is an attribute required in the industry to boost confidence in all members of the team. In addition, this involves creating and sharing success among teammates. Along with previous attributes, the teammate relationship should be based on respect and honesty. Another attribute of a good software engineer is to be questioning and to push the team to challenge situations that might trigger motivation.

The experienced software engineers in the study by Li and colleagues also identified important software product attributes. Participants described a good software as being elegant in terms of design and usability, which allowed the user to understand

and use the software in a simple way. Reducing complexity of software reduced code errors, and therefore avoided the cost of debugging complex issues.

Another important attribute raised by participants was creativity. It is important that developers understand problem requirements and the limitations of proposed or existing solutions, allowing innovative software to be produced. This study provided an extensive model of four types of programmer characteristics required by industry and detailed multiple attributes. The study aim was to identify a wide range of characteristics and attributes; however, other attributes, such as gender and background, were not explored. In addition, the model was derived from the context of one organisation that may have specific rules and work culture that may be different from other organisations. If the study had been conducted in another organisation, a different model of characteristics may have emerged.

Industry experts may also be useful in suggesting characteristics as competencies required for programmers seeking employment. Joseph (2015), who works as a programmer and IT director with 20 years of experience, developed a programmer competency matrix consisting of five dimensions - CS, software engineering, programming, experience, and knowledge - where each dimension consists of multiple attributes. In addition, the matrix specifies four levels of performance for each attribute.

The CS dimension specifies how a programmer should be competent in data structure, algorithms, and systems programming. For example, a highly competent programmer should have knowledge of various data structures, such as Adelson-Velskii and Landis (AVL) red black trees and lists, and a programmer should possess knowledge of different algorithms, such as dynamic programming, divide and conquer, and graphs.

The second dimension, software engineering, consists of attributes such as controlling different versions of source code. The programmer should know how to use a version control system and should also be able to build units to test codes.

The third dimension, programming, consists of multiple attributes related to problem and system decomposition, communication, code organisation, code optimisation, and error debugging. For instance, a programmer should be able to decompose

a problem using appropriate algorithms and data structures and by writing generic OOP code to cope with any possible changes of the problem requirements. In addition, a highly competent programmer should be able to communicate ideas, analysis, and design of such software.

The fourth dimension of competency for programmers is experience in using different programming paradigms. A programmer who understands imperative, logic, declarative, and OOP is highly competent. In addition, experience in programming for multiple operating systems is another attribute of competency for programmers.

The fifth dimension, knowledge, indicates programmers' knowledge of their tool using multiple IDEs including open source. In addition, a highly competent programmer should be self-taught and be able to keep up with the rapid changes in the field and be able to acquire new languages and technologies.

The matrix provides numerous attributes that can be manifested by different levels of performance of the programmer, explaining how novice, average, and highly competent programmers perform in each attribute, for example. Although the matrix explores multiple attributes about programming competency, the matrix is derived from personal experience, and there are some attributes, such as academic performance and mathematical ability, that have not been included.

2.3.5.3 Curiosity

Li et al. (2015) investigated multiple characteristics related to the software development profession. One of the themes that emerged from the study was personal characteristics, including curiosity. Curiosity related to knowing how and why certain codes have been implemented in such a way or how certain customer requirements would affect the algorithm design and code implementation. Curiosity is an important characteristic that allows the programmer to discover and learn new programming languages and concepts. New technologies evolve rapidly, which makes it difficult for a busy programmer to learn and adapt. However, the curiosity to be well-informed of the constant developments in the field gives the programmer a sense of future direction. In a very

competitive industry, there is little chance of survival for programmers who are not willing to adapt quickly to the current state of the industry. One aspect of being a curious programmer is to learn new languages and/or concepts by experimenting with how a new language works. Writing small programs might help to understand the new language syntax and to understand how to debug the code. McConnell (2004) included other strategies for learning new programming languages, such as learning from successful projects, reading programming language documentation, and affiliating with other professionals.

2.4 Taxonomies in Computer Science

Educational taxonomies have been implemented in many domains to enhance pedagogy, assessments, and teaching methods, all of which affect student learning, knowledge, and skills. There have been many attempts to apply different taxonomies, and these have been valuable in providing insight into CSE to understand different educational factors. Well-developed educational taxonomies, such as Bigg's structure of observed learning outcome (SOLO) taxonomy (Biggs & Collis, 1982), have been used to measure student outcomes and classify exam questions based on what they are supposed to measure. Although an educational taxonomy provides a generic framework that can be implemented in various disciplines, educators may not always come to an agreement on classifications (Fuller et al., 2007).

The SOLO taxonomy (Biggs & Collis, 1982) aims to distinguish students' cognitive levels, which are required during their learning process. The first level is Prestructural (P), where a student is provided with a new problem and seemingly irrelevant information. At this stage, the student has not understood the problem and tries to use simple information to solve it. The second level is Unistructural (U), as the student starts to focus on a single aspect that can be used to solve the problem. The third level is Multistructural (M), where the student starts to understand more than one factor that may help to solve the problem. The fourth level is Relational (R), which focuses on the qualitative development as the student starts to understand and identify rela-

tionships between several aspects. The fifth level is Extended Abstract (EA), where the student manifests the ability to hypothetically think about other new factors that may help to solve the problem. In addition, the student may be able to generalise, evaluate, and/or apply the knowledge to other problems. The SOLO taxonomy has been applied to CSE to assess student performance in a few specific aspects of programming (e.g. code comprehension, code writing, and algorithm design).

Programming may involve high levels of human cognition, such as problem solving, abstraction, and reasoning. Analysis of 734 course syllabi from science faculties showed that CS competencies fall into the higher levels of the SOLO taxonomy, more than natural science or mathematics competencies (Brabrand & Dahl, 2009). Specifically, programming skills such as analysis, design, implementation, and structure fall into the Relational level of the SOLO taxonomy and account for 15% of all CS skills. Teaching methods and assessments of programming competencies must therefore be tailored to this higher level. This is a complicated task, as diverse competencies need to be acquired by a range of students with varied individual differences and learning styles.

Lister et al. (2006) developed an interpretation of the SOLO taxonomy to apply to student answers to code comprehension problems using multiple-choice questions (MCQs). However, MCQs were not adequate to elicit responses at the Relational level. Therefore, the study was extended to analyse different types of questions. A total of 108 students were asked to provide written explanations of a segment of code, allowing student responses to be categorised (based on SOLO) by three academics. In addition, responses were also compared with those of eight expert academics across each SOLO level. Results showed that half the students provided Multistructural answers (i.e. students were only able to explain the code line by line without indicating the purpose of the code). In contrast, seven of the eight experts provided Relational answers.

Sheard et al. (2008) compared the 'explain in plain English' question responses for 120 students in an introductory programming course with postgraduate student responses to the same questions, in an attempt to reduce instructional ambiguity.

Again, most undergraduate students provided answers at a Multistructural level, while postgraduate students answered at a Relational level.

Later, Lister et al. (2010) applied SOLO to measure student performance in code writing, relying on Biggs (1999) verb descriptions for each level. In addition, Hattie and Purdie (1998) provided examples of how SOLO can be applied to language translation, categorising how certain phrases are interpreted rather than by translating words in isolation without understanding either the relation between the words or the context. For example, word-by-word translation, which is Unistructural, might provide meaningful translation that does not reflect the purpose of the original phrase. In the context of code-writing questions, a student may provide a direct translation of a certain program specification, which does not result in the correct code, whereas applying some changes to produce a translation that is close to a direct specification might result in valid code. Based on Hattie and Purdie's theoretical framework, SOLO categories for code writing were proposed as shown in Table 2.3.

Phase	SOLO category	Description
Qualitative	Extended Abstract - Extending [EA]	Uses constructs and concepts beyond those required in the exercise to provide an improved solution
	Relational - Encompassing [R]	Provides a valid well-structured program that removes all redundancy and has a clear logical structure. The specifications have been integrated to form a logical whole.
Quantitative	Multistructural - Refinement [M]	Represents a translation that is close to a direct translation. The code may have been reordered to make a valid solution.
	Unistructural - Direct Translation [U]	Represents a direct translation of the specifications. The code will be in the sequence of the specifications.
	Prestructural [P]	Substantially lacks knowledge of programming constructs or is unrelated to the question.

Table 2.3: SOLO categories for code-writing tasks (Lister et al., 2010).

Initial analyses of the code-writing answers of 30 students were conducted to develop the proposed taxonomy. Students were asked to write code involving three conditional statements in which providing a direct translation for sequenced conditional statements was considered Unistructural. However, when students considered removing

redundancy, solutions tended to increase on the SOLO scale, becoming Relational. The student responses were only Unistructural and Multistructural. However, a second analysis of a different, more difficult code-writing question was conducted for a further 59 students, with only two responses categorised as Relational.

Although the SOLO taxonomy provides a theoretical basis for analysing student approaches to answering code-writing questions, it is evident that the levels of the questions may limit student responses to certain SOLO categories. If a student is asked to write a program to assign a value to a variable and print out the value, it is clear that the student response will be Unistructural, as there will be no chance to provide a response at any upper level. Thus, this study needs to be replicated across different levels of code-writing questions (Lister et al., 2010).

To address issues around inconsistent mapping of the SOLO levels, Whalley et al. (2011) proposed a refined SOLO taxonomy. A grounded theory approach was used to analyse and map nearly 750 responses to three code-writing questions (discount problem, average calculation, and printing a box of asterisks). The mapping process started with developing empirical categories consisting of silent programming elements (SPEs) to extract program constructs, syntactical elements, and code features via constant comparative qualitative coding of the responses. The process allowed CS educators to identify SPEs, which could emerge from the code. Producing SPEs could be advantageous and is practical for different code-writing questions. The next stage was to extract broad features that reflected a general code quality that appeared in most code, such as redundancy and efficiency. The extracted features indicated the level of code abstraction based on subjective evaluations. Finally, based on the SOLO taxonomy proposed by Whalley et al. (2011), three researchers categorised the responses to investigate whether using SPEs made the mapping process more efficient.

The study produced a refined taxonomy by redefining the Multistructural level during analysis. A previous definition of Multistructural indicated that a response represented a translation that is close to a direct translation (Lister et al., 2010), whereas Whalley et al. (2011) suggested that the code may have been reordered to make

a 'valid' solution. However, during the analysis of the average calculation problem, some responses provided a direct translation that was a correct solution but was less integrated. While those responses were categorised as Multistructural, they tended to be over-categorised and should have been Unistructural. Therefore, Multistructural was redefined as 'a translation that is close to a direct translation. The code may have been reordered to make a more integrated and/or valid solution'.

Whalley et al. (2011) provided a rigorous methodology to analyse a large dataset to produce consistent SPEs for different code-writing questions. Their mapping process requires CS educators who are capable of identifying multiple alternative solutions or SPEs in which common features can be extracted. Student responses might be classified as Unistructural, which should indicate at least a single concept or SPE, whereas a Multistructural response should indicate a student's understanding of multiple concepts or SPEs, which may or may not provide an integrated solution.

Further, a Relational response should indicate that all concepts and SPEs have been integrated, manifesting a comprehension of the relationships between all elements and features. The CS educators should understand that classifying student responses is based on the level of translated specifications that are required to satisfy code implementations. In other words, the level of required specifications in a certain question affects student-response classifications but not necessarily that the classification could measure student knowledge.

A recent study by Izu et al. (2016) introduced an evaluation framework built on the SOLO taxonomy to investigate the complexity of questions by developing building blocks to identify programming constructs, such as assignment, iteration, and vectors. It was claimed that the mapping processes used in the previous research were not consistent in defining programming constructs at the Unistructural level (Sheard et al., 2008; Whalley et al., 2011; Jimoyiannis, 2013). Therefore, developing the building blocks may overcome the previous research limitations to identify programming constructs for the Unistructural level only. The building blocks should be derived from current course curricula while considering the knowledge that has been acquired by

students. In this study, iterative and vector questions were analysed, while applying the proposed building blocks, and the results showed that 44% of student performance achieved a Relational level and only 3% remained at a Unistructural level.

2.5 Reflection

Often, theories are complex and abstract, which leads to confusion when adapting and applying them in real-life contexts, especially in education. Gardner (2003) stated that educators misunderstood MI theory and its implications, which resulted in mixing other theories with MI theory. One way to make such a theory clear to implement is to conduct studies within a specific context and to replicate these studies, which might result in clear recommendations for educators. Although the three-ring conception provides a broadened definition of giftedness based on three clusters, those clusters could be manifested by individuals within a specific field of study. In addition, the conception provides more than the generic theory but introduces a blueprint for educators to go from theory into practice. Along with the introduction of the theory, the enrichment triad model was introduced, specifying different levels of enrichment followed by the introduction of the RDIM model. This model provides a clear methodology for how to identify large groups of high potential students, termed a 'talent pool', and how to involve this group in different levels of enrichment. In addition, this model has been implemented for a long period in many schools in the United States under the Schoolwide Enrichment Model project (Renzulli & Reis, 2000).

We believe that the three-ring conception can be initially adopted in programming, whereby gifted student programmers can be identified based on their academic performance. At this stage, we do not have a comprehensive understanding of the programming characteristics that might be exhibited by the students; therefore, there might be another identification method that could be used in the programming context. Thus, investigating programming characteristics through gifted students allows for developing a theory for defining giftedness in programming. Based on that, the enrichment triad model and RDIM can be incorporated, allowing CS educators to

identify the talent pool to include the top 20% of students using multiple identification methods derived from the developed theory. Then, CS educators can design the three levels of enrichment in the enrichment triad model. In programming, the first level of enrichment, general exploratory, might include advanced data structures and algorithms that could extend student interests. The second level of enrichment, group training activities, might include small group seminars to focus on extending student problem solving, think aloud, and technical communication abilities. The third level of enrichment, individual and small group investigation of real problems, might focus on solving real-life problems by developing a large-scale project. The RDIM outlines the methods and principles for the transition process between the levels of enrichment. Thus, the ultimate goal of providing breadth and depth enrichment for a wide range of students can be achieved without restricting the benefit of a gifted education to a small number of students.

The literature suggests multiple characteristics that a good programmer should possess in both educational and professional contexts. The characteristic that has been most discussed in the literature is mathematics ability. Previous research has investigated mathematical abilities, such as cognitive abilities, which could involve problem solving, reasoning, and abstraction. Mathematical abilities have been an indicator of programming aptitude. Most PATs include mathematical questions to measure problem solving, reasoning, and abstraction abilities. On the other hand, mathematics as a knowledge and pedagogy has been investigated to understand the role of mathematics in learning programming. Implications from previous research include designing mathematics modules that relate to programming concepts or introducing programming concepts and languages based on formal mathematics theory. Moreover, CS educators have established a great deal of research investigating those characteristics for novice programmers to help students who struggle when learning programming. However, there has not been enough investigation into those characteristics for gifted students, whether they manifest specific characteristics or not. The literature indicates that mathematical knowledge is important in learning programming, but does this mean

that gifted student programmers should be gifted mathematicians? In addition, previous studies have not provided significant statistical tests of the correlation between mathematics and programming. Thus, we aim to statistically investigate the correlation.

Some characteristics relate to cognitive ability, allowing the programmer to form a mental representation of a problem. Thus, mental representation can consist of problem solving, reasoning, and abstraction ability. A good programmer should possess problem-solving ability, which allows the programmer to analyse, decompose, compose, and solve a problem. Different strategies have been proposed in the literature (Mason et al., 2010; Polya, 2004; Vickers, 2008). In relation to knowledge-organisation strategies, it has been suggested that good programmers have different strategies for organising their knowledge in programming (Ormerod, 1990; McKeithen et al., 1981; Hoc, 2014). Knowledge organisation might be manifested by an expert programmer organising and recalling multiple and specific chunks of information that are related to solving a specific problem. In addition, implementing advanced data structures and algorithms, such as red black trees, lists, divide and conquer, and graphs is an important characteristic that should be manifested by a highly competent programmer. However, we do not know yet whether gifted student programmers use any of the suggested strategies for solving problems and organising their knowledge or whether they tend to use different strategies. Abstraction ability is an important aspect of cognitive ability, and it has a significant effect on learning programming concepts, such as recursion and iteration. Therefore, a good programmer should possess the ability to abstract from the implementation of recursion to understand the whole picture. However, we are not sure whether gifted student programmers are able to understand recursion or not. Although measuring student abstraction ability is a challenging task, previous CSE research has established a way to incorporate SOLO taxonomy to analyse student code-writing ability, which can help to investigate student abstraction ability.

Different coding strategies have been suggested, including coding standards, debugging, and optimisation. In the educational context, coding strategies have been

taught to achieve best coding practices. However, in the software development industry, bespoke coding rules can be used that might be different from the educational context. Following these rules can be an important characteristic of a good programmer.

In relation to attitudes and personal traits derived from educational and IT professional contexts, learning style and personality might have an effect on the student learning process. In addition, the IT professional literature highlights the importance of communication and cooperation skills, which are considered important criteria for a good software developer. In addition, a software developer should be eager to learn new technologies and should be curious and creative.

The literature suggested important characteristics, such as mathematical abilities, abstraction, problem-solving strategies, attitudes, and personal traits. These characteristics have been derived from different contexts, including educational and professional, and they have a great deal of overlaps. In addition, CSE researchers provides a great deal of effort in investigating the characteristics among novice students to provide them with support. However, there is a gap in the literature investigating the characteristics among gifted student programmers, as research in defining giftedness in programming and identifying gifted student programmers may not exist. These characteristics may or may not apply to CS educational contexts and gifted programming students may or may not exhibit specific characteristics. Therefore, our research questions have been developed to close that gap by investigating the characteristics of a gifted student within the context of programming.

CHAPTER 3. Methodology

The key element of excellent research is a reliable and valid research methodology that can produce similar results if the research is conducted again. Of course, in social science research, different variables that are mostly human-based can affect research results. Thus, choosing a research methodology that produces valid results depends on two aspects. First, researcher beliefs, which indicate how researchers understand, interact with, and view the surrounding world, can result in several methods to conduct research using different methodologies. Second, research paradigms that correspond with research methods that serve the purpose of the research and answer the research questions are another characteristic.

This chapter is divided into five sections. The first discusses basic concepts of research philosophy paradigms. The second discusses various research methodologies that would be appropriate for this study. The discussion points out advantages and limitations of alternative methodologies along with justifications for selecting particular methodologies in this study. The third section describes the chosen data collection and analysis methods, and the fourth addresses the validity and reliability of these methodologies. Finally, ethical considerations are addressed.

3.1 Research Paradigm

Weaver and Olson (2006, p. 460) defined paradigms as 'patterns of beliefs and practices that regulate inquiry within a discipline by providing lenses, frames and processes through which investigation is accomplished'. In addition, a variety of research paradigms derived from different philosophical views have been introduced to underpin

research projects.

Blaikie (2009, p. 69) discussed the importance of research paradigms that can help researchers plan and design research projects. In addition, Merriam emphasised that philosophically underpinning qualitative research is essential to help researchers understand 1) the nature of reality (ontology) that can exist or be constructed, 2) the researcher's knowledge about reality (epistemology), and 3) the methodology that outlines the procedures used to gain knowledge (Merriam & Tisdell, 2009, p. 8).

Research paradigms have been cumulatively derived from different research inquiries in social sciences in which each inquiry may have unique aspects. In this case, the research paradigm should be considered a model that can be constructed based on a variety of characteristics associated with every research project. Thus, descriptions of research paradigms have a certain degree of ambiguity and abstraction (Blaikie, 2009, p. 9). However, Cohen et al. (2011, p. 5) noted that social reality consists of two assumptions: ontological and epistemological. The ontological assumption determines whether social reality is an external factor that could affect an individual's behaviour or whether social reality is constructed based on an individual's beliefs. However, the epistemological assumption concerns the nature of knowledge in terms of asking what we know and how the knowledge can be acquired.

Based on researchers' ontological and epistemological beliefs, the research methodology determines the set of procedures that guide researchers regarding how knowledge can be obtained. For instance, a researcher who understands social reality as objective phenomena controlled by a set of surrounding variables might use a set of methods, such as surveys and experiments.

However, if the social reality is perceived as subjective phenomena constructed based on individual cognition and the surrounding world, procedures such as observations could be implemented (Cohen et al., 2011, p. 6). Thus, research paradigms are philosophical frameworks derived from the formulation of ontology, epistemology, and methodology.

Positivism is a philosophical paradigm introduced by Auguste Comte in the

nineteenth century, who contributed to establishing the discipline of sociology. The positivism philosophy considers social phenomena, which can be observed, perceived, and explained scientifically. It is objective and can be studied in a way similar to natural sciences, in which laws and theories might be generated (Cohen et al., 2011, p. 6).

A criticism of positivism is that laws and theories could be generated in social sciences, whereas the law of generalisation might not be applicable in certain cases in which social phenomena are controlled by a variety of surrounding external variables and individual cognition. For instance, in the educational context, different variables - most of them subjective and based on students, teachers, teaching methods, and curricula - can vary between classrooms or schools.

An additional concern regarding the positivism paradigm is dehumanisation, as positivism tends to detach human effects from social science (Cohen et al., 2011, p. 14). It has been argued that positivists strive to apply quantitative, computational, or statistical approaches to conclude with numerical results, in which a lack of explanation of understanding the social study and human behaviours could be apparent (Cohen et al., 2011, p. 6).

As a result of such a criticism, a post-positivist research paradigm emerged to consider social phenomena to be subjectively based on individuals, which cannot be researched in the same way as natural science inquiries. However, criticisms have again been raised regarding the possible lack of methodological rigour and biased reporting of results, as the researcher is the primary instrument conducting social research through the entire research process.

A mixed methods paradigm emerged to allow qualitative and quantitative methodologies to be combined, in which numerical methods can be enriched by more qualitative data. In addition, the mixed methods paradigm provides an approach that is less polar and allows a certain degree of trade-off between subjectivity and objectivity (Cohen et al., 2011, p. 21).

Rossman and Wilson (1985, p. 632) described various reasons for the need to

combine quantitative and qualitative research: 'First, combinations are used to enable confirmation or corroboration of each other through triangulation. Second, combinations are used to enable or to develop analysis in order to provide richer data. Third, combinations are used to initiate new modes of thinking by attending to paradoxes that emerge from the two data sources'.

The general aim of this study was to determine the characteristics that allow us to identify gifted student programmers. Those characteristics relate to multiple factors, including student ability, subject knowledge, and personal traits. In addition, the educational context of this study added another dimension that could affect certain characteristics (i.e. programming knowledge). This dimension gives our study a unique perspective of identifying gifted students' characteristics. Therefore, we believe that characteristics can be constructed based on student abilities and educational environment.

3.2 Overview of Research Methodologies

As mentioned, researchers' beliefs play a major role when choosing a research methodology and research purpose. Research questions must also be considered when determining a suitable methodology and data collection procedures (Cohen et al., 2003, p. 89).

Multiple methodologies are available to help researchers investigate how knowledge about certain problems can be obtained and to help researchers link methods and outcomes (Creswell et al., 2003, p. 5). Thus, this section provides a brief description of potential methodologies to be implemented in this research, with affordances and constraints explored for each so that the most appropriate methodology could be chosen for our research context.

3.2.1 Quantitative Methodology

The quantitative approach is used to quantify data that are collected to investigate a hypothesis and to analyse the data in numerical form. Bryman (2012, p. 24) defines detective research as the relation between theory and research in which a researcher has subscribed to a particular research field to allow for a hypothesis to be deduced based on the research knowledge. While a deductive approach starts from existing theory forming the hypothesis that needs to be evaluated, an inductive approach starts from the researcher's findings and observations, where theory could be generated at the end of the research.

Generalising results, which are produced by implementing a quantitative methodology, can be considered advantageous. However, in certain research contexts, where multiple social aspects can play a significant role on the nature of the research, results might not be generalised (Bryman, 2012, p. 176). Generalisation can be enormously affected by the sample size that should be representative of the population. However, the temptation of making claims resulting from representative samples should be carefully considered by researchers, as different elements of social aspects could be different in other contexts.

Another aspect of quantitative research is replication. If a study had been conducted and produced particular results, a replication of that study should yield similar results. That requires clear steps of how such research had been conducted to reduce subjectivity and to increase validity. Reliability and validity aspects of our research of will be discussed later in this chapter.

The quantitative methodology is an appropriate methodology for addressing the first research question, which is based on the hypothesis derived from the existing theory that mathematics correlates with programming. However, as mentioned in the literature chapter, previous research has shown mixed findings regarding the correlation between mathematical ability and learning programming, with concerns regarding sampling, sample size, and methodology. In addition, previous studies (Pacheco et al., 2008; Pioro, 2006; Tukiainen & Mönkkönen, 2002; Bennedsen & Caspersen, 2005;

Bergin & Reilly, 2005b) have indicated insignificant relationships that lack statistical weight in their results or do not have in-depth explanations derived from qualitative data to elaborate on their statistical findings. Therefore, investigating the correlation between mathematics and programming through quantitative methodology can prove or reject our hypotheses. Thus, we chose the quantitative methodology to test our hypotheses using descriptive statistics and statistical tests. To elaborate on our statistical findings, qualitative data will be gathered to understand the relationship between programming and mathematics based on participants' perceptions.

3.2.2 Ethnography

In ethnographic research, such as participant observations, the ethnographer deeply engages with a particular group of individuals, observing their behaviours and discussions for a long period of time in which rich information about individual perception and actions can be gathered through different procedures, namely observations, interviews, and documents (Bryman, 2012, p. 432).

A key feature of ethnography includes studying a social phenomenon to understand its unique elements through individuals. In addition, the ethnographer benefits from having been close to individuals, interacting with them for an extended period of time, where deep understanding of such a social phenomenon can be obtained.

Attaining granted access to participants within a social setting is the most challenging part of conducting ethnographic research, especially in private settings. Bryman (2012, p. 433) distinguished between open and closed settings. Closed public settings could be schools, hospitals, and firms that might require official procedures for access permission, whereas open public settings could be communities. Another disadvantage is that unstructured collected data could be produced from notes taken during observations that could be difficult to organise and analyse.

In the context of this research, the phenomena of giftedness and programming ability were explored within the context of education in which programming could be considered a sophisticated cognitive ability associated with multiple abilities and skills.

In addition, our research did not intend to investigate behavioural dimensions, which require close observation of participants over a long period of time. Moreover, time constraints and multiple access to participants can be obstacles that limit the possibility of conducting ethnographic research. Thus, we eliminated the option of using an ethnographic approach because our research purpose did not require investigation of behaviours, had a limited timeframe, and difficulties with frequent access.

3.2.3 Phenomenology

An alternative research methodology that might be suitable for this research was a phenomenological approach which 'focuses on the subjective experience of the individuals studied... at its heart is the attempt to understand a particular phenomenon' (Robson, 2004, p. 195). The role of the researcher is to accurately describe certain phenomenon, as Welman and Kruger (1999, p. 189) stated that 'the phenomenologists are concerned with understanding social and psychological phenomena from the perspectives of people involved'.

Gray (2013, p. 30) described phenomenology as an approach built on a theoretical view emphasising inductive research to allow researchers to collect data through unstructured methods that could produce a description in which unexpected elements regarding a particular phenomenon might emerge. Thus, the researcher tends to be open to what data can suggest and not limited to the researcher's view about certain phenomena.

Using unstructured methods in phenomenology could be considered disadvantageous in that it can affect generalisation rather than as methods that facilitate understanding specific phenomena within a specific context. As the study methods were structured, phenomenology was less suitable. In addition, phenomenology research is intended to focus on a specific phenomenon, examining surrounding elements to comprehend social, behavioural, and cognitive aspects that comprise such a phenomenon. This research encompassed two broad phenomena - giftedness and programming - with multiple aspects associated with each phenomenon and overlap of some aspects.

Based on the nature of this study, phenomenology therefore could not be used, as this research did not focus on an aspect of a specific phenomenon; rather, it sought to understand several aspects related to giftedness and programming within an educational context. The educational context could be considered a third dimension in our research in which several related aspects needed to be explored. Alternatively, the case-study methodology could be a more general approach that could be sufficiently flexible for qualitative and/or quantitative data collection (Gray, 2006, p. 124).

3.2.4 Case-Study Methodology

Case-study methodology can help researchers understand phenomena within a specific context, providing an in-depth explanation of such phenomena. Yin (1994, p. 8) defined a case study 'as an empirical inquiry that investigates a contemporary phenomenon within its real-life context; when the boundaries between phenomenon and context are not clearly evident; and in which multiple sources of evidence are used'. Creswell (2009, p. 13) described a case study as an approach that helps research exploring a programme, an activity, a process, and individuals with in-depth information.

However, different attempts to define what a case study is have resulted in confusion (Gerring, 2004, p. 342). Case studies provide unique examples of real people in real situations, enabling readers to understand ideas more clearly than simply presenting them with abstract theories or principles. Case studies can penetrate situations in ways that are not always subject to numerical analysis. In addition, Cohen et al. (2011, p. 289) argued that case studies can establish cause and effect. Yin (1994, p. 38) classified case studies based on number (single or multiple) and design (holistic or embedded) in which four categories result:

1. The single-case design focuses on an extreme unique case;

2. The embedded single-case design has multiple logical units that need to be analysed and are associated with a single case;

3. The multiple-case design provides comparative cases within the research;

4. The embedded multiple-case design has various units embedded within several cases.

The single-case design has been criticised for being too focused on the uniqueness of the single case, which can be questioned if the collected data do not indicate that the investigated case is not unique. Moreover, the single-case design should consider different factors that make the case unique, and these factors need to be critically investigated. However, the multiple-case-study design is advantageous in providing potential generalisation of research findings in which multiple cases provide different sources of data (Miles & Huberman, 1994; Patton, 1990) and allow triangulation of data from different sources, increasing the scope and validity of the investigation (Bonoma, 1985, p. 201). Campbell (1975, p. 180) argued that having two cases that help the researcher in the purpose of comparison is more valuable than having vast data from a single case. However, it depends on how unique the single case is.

In terms of case-study design, a holistic case study focuses on understanding the nature of a phenomenon within a context in which the case has a single unit of analysis. An obvious limitation of the holistic case-study design is the ability to provide deep understanding of the phenomenon limited by its single unit of analysis. An embedded case-study design allows multiple units of analysis to be included, providing better understanding of such a phenomenon.

Another classification of the case study is based on the types of case study: explanatory, exploratory, or descriptive Yin (1994, p. 8). The explanatory case study aims to understand interventions in a real-life context, which may be too complicated to investigate through a survey. The explanatory case study can be advantageous when testing a theory or the relationship between multiple aspects of a theory (Yin, 2009). The exploratory case study aims to explore a phenomenon to develop a theory rather than to test or establish causality. The descriptive case study can be useful in describing a phenomenon and its characteristics in a real-life context. Gerring (2004) argued that the descriptive case study can be used in developing a theory.

Another case-study classification, which is derived from research purposes, was

introduced by Stake (1995). The intrinsic case study emphasises providing a comprehensive understanding of the case rather than focusing on the uniqueness of the case itself. On the other hand, the instrumental case study switches the focus from the case itself to the study, where the aim is to understand the circumstances surrounding the case. Distinguishing between intrinsic and instrumental case studies depends on the different purposes of the case study. However, the collective case study implies that multiple cases can be investigated in a context where a theory can be generalised.

Therefore, case-study methodology was suitable for this research for several reasons. The research lent itself to explanatory and exploratory case studies, as we aimed to understand the phenomenon of giftedness through a real-life context of first-year university CS students. In addition, this study was a collective case study as a number of individual cases were investigated to identify characteristics that gifted students might possess. Thus, the embedded multiple-case study design was adopted in this study with each participant representing a single case and each single case embedded within multiple units of analysis that were the characteristics related to programming, as shown in Figure 3.1. The nature of the second research question and

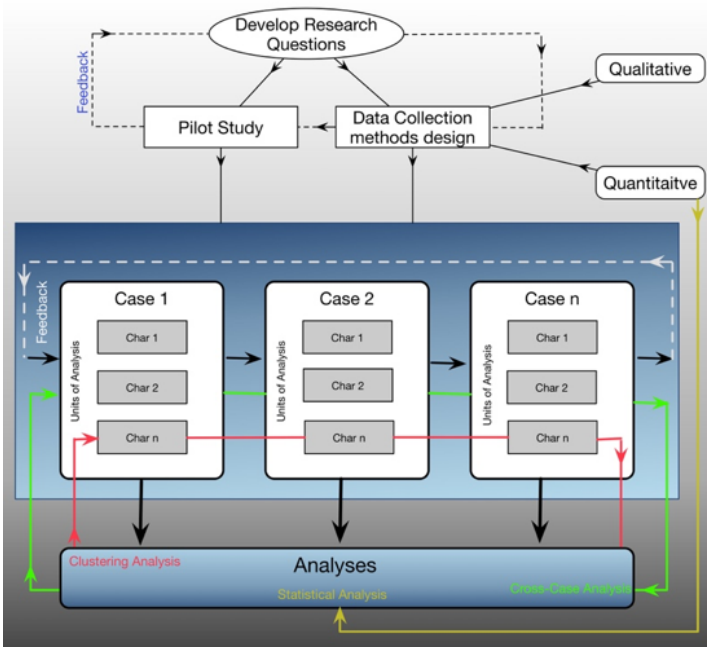


Figure 3.1: Embedded multiple-case design.

its sub-questions required a deep understanding of characteristics that could identify

gifted student programmers, which can be achieved through gifted CS students who have been identified based on academic performance. As discussed in the literature review, academic performance can be used as an identification method. However, there are multiple characteristics that a gifted student programmer might possess other than academic performance. Therefore, we used programming performance as an initial identification where students had been identified based on their performance in two programming modules.

In the educational context, variables such as the curriculum, teaching methods, and student abilities could contribute to their performance. Thus, the case-study methodology allows multiple sources of quantitative and qualitative data to be collected through different methods in which different variables can be investigated (Cohen et al., 2011, p. 289).

To summarise the methodologies adopted in this study, a quantitative approach was used to answer the first research question and its sub-questions to examine the correlation between mathematics and programming. A case-study methodology was then used to answer the second research question and its sub-question. A more complete discussion of validity and reliability of both methodologies is presented in Section 3.4.

3.3 Methods

Before describing our methods of collecting the data for this study, we shall remind the reader about the research questions and how did they emerge from the literature review.

RQ: 1 To what extent does mathematical ability correlate with programming ability in general?

RQ: 1.1 What is the correlation between student performance in discrete mathematics and programming modules?

RQ: 1.2 What is the correlation between student performance in calculus and programming modules?

RQ: 2 What are student perceptions about the relationship between mathematical ability and programming ability?

RQ: 3 What mental representation strategies do gifted students tend to use?

RQ: 4 What mathematics and programming knowledge do gifted students tend to have?

RQ: 5 What coding strategies do gifted students tend to use?

RQ: 6 What attitudes and personality traits do gifted students tend to possess?

The aim of this study is to investigate the characteristics related to programming through gifted students. The literature suggested multiple characteristics: mathematical ability, mental representation, knowledge organisation, coding strategies, and attitudes and personal traits. However, some characteristics had been investigated through students who were struggling in learning programming with limited evidence of what characteristics gifted students manifest. In addition, some characteristics have been derived from the professional programming context, which might be applicable in the educational context. In addition, the literature indicated a lack of significance regarding the statistical investigation of the correlation between mathematics and programming. Thus, the first research question (RQ1) and its sub-questions aim to statistically test the hypothesis suggested by the literature that mathematical ability correlates with programming ability. In addition, the investigation continues by addressing the specific correlation between discrete mathematics, calculus, and programming modules. Further investigation of the correlation was considered by investigating students' perceptions about the correlation in RQ2.

Knowledge organisation and mental representation, which could include problem-solving strategy and abstraction ability, have been important characteristics in learning programming (Blackwell, 2002; Teague & Lister, 2014; McKeithen et al., 1981; Joseph, 2015). It has been suggested that an expert programmer might have a particular

representation of a problem that will allow for deep understanding, reasoning, and implementing data structures and algorithms. In addition, the expert programmer tends to demonstrate a breadth and depth of knowledge in terms of how to implement data structures and algorithms. However, at this stage, we do not know whether a first-year gifted programmer possesses one or a combination of the above characteristics. Thus, we will investigate the above characteristics, addressing RQ3 and RQ4.

The literature suggested different coding strategies that a good programmer should adopt. These strategies might include naming, commenting, debugging, and organising code (Joseph, 2015; McConnell, 2004; Lammers, 1986). However, the literature indicates a gap in investigating whether gifted student programmers tend to use these strategies. In addition, some coding strategies have been derived from the IT profession, which can be different from the educational context. However, we are not sure which coding strategies are important to identify gifted student programmers. Thus, RQ5 will address the above concerns.

Personal traits, which include learning styles, curiosity, communication, and co-operative skills, are important characteristics. It has been argued that personal traits can affect how a student will engage with other students during the learning process. Working in pairs during lab sessions can be affected by communication and co-operative skills. In the IT professional context, these characteristics are highly required. However, there is a gap in the literature in terms of investigating whether specific personal traits are important in identifying gifted student programmers. Thus, RQ6 will investigate those characteristics among gifted students within the educational context.

It has been argued that mixed methods research defines research design based on philosophical assumptions and procedures that serve as a guide for the researcher of how data can be collected and analysed (Creswell & Clark, 2011). In addition, the guide provides a direction of how to combine qualitative and quantitative methodologies. However, Creswell (2009, p. 15) argued that each method will have limitations and certain degrees of bias, which a researcher may avoid by adopting other methods. Therefore, methodological triangulation allows qualitative and quantitative method-

ologies to be combined allowing for research findings to be more robust, where both methodologies are mutually corroborated (Creswell & Clark, 2011).

Cohen et al. (2011, p. 196) further distinguished triangulation types. The first type of triangulation, which was considered in our research, is a combined level of triangulation in which multiple levels of analyses are implemented, namely, analysis of a single case (participants) and interactive analysis between all cases. Using a combined level of triangulation during the analysis stage can provide a more sensible, complete view about social phenomena, in which each case could contribute by drawing from that view.

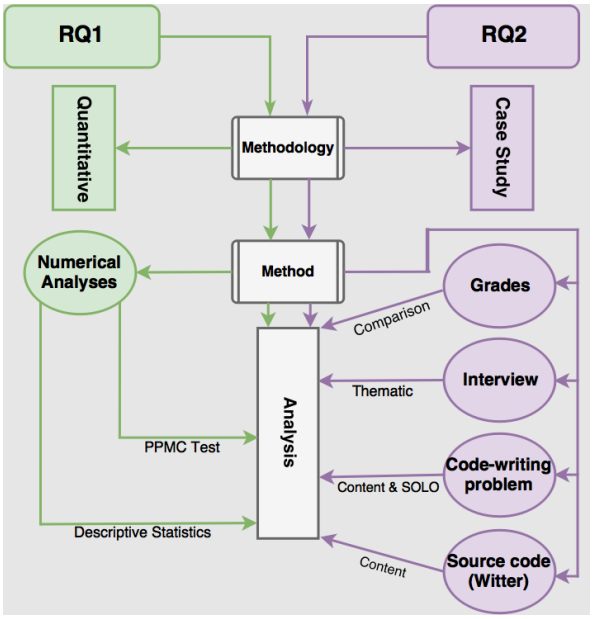


Figure 3.2: Study design in terms of methods and analyses.

The second type of triangulation that could be considered is methodological triangulation, where data are collected using more than one method. In this study, multiple sources of evidence (qualitative and quantitative data) were gathered, including the case-study methodology, which allows both qualitative and quantitative approaches to be used. Figure 3.2 illustrates the design with different methods, such as semi-structured interviews and supporting documents mainly used to gather qualitative data and numerical data regarding student performance, which was collected to increase the validity of the qualitative research.

As Cohen et al. (2011, p. 195) stated, triangulation involves using two or more

data sources to cross-check data to enhance the validity of the research. The specific data collection methods noted in the figure are described further in Sections 3.3.3 and 3.3.4.

3.3.1 Numerical Data

Student grades for programming modules and mathematics modules were collected to investigate the first research question and its sub-questions and to test our hypotheses below.

1. Mathematical ability correlates with programming ability in general;
2. Discrete mathematics grades correlate with programming module grades;
3. Calculus grades do not correlate with programming module grades.

Upon ethical approval, data were collected from the Department of Computer Science at the University of Warwick. The raw data in spreadsheet file format consisted of *n=2926* student marks during the period 1996 to 2014. The marks included two programming modules and two mathematics modules, which were core modules for first-year CS students.

In addition, student grades for the 2014/15 cohort (*n=134*) were collected to define the sample for the case studies. The data were analysed using descriptive statistics for comparative analyses between the performances of gifted students and the cohorts and between the performances of gifted students themselves. The data for this specific cohort included grades for two programming modules and mathematics modules, which consisted of coursework and final exams. Thus, we collected two sets of data for separate purposes. The first dataset was used to test our hypothesis for the first research question and sub-questions. The second set was used to address the second research question and sub-questions to identify gifted students for the academic cohort 2014/2015 and for descriptive statistics.

Statistical analyses (descriptive statistics and the Pearson product-moment correlation coefficient test) were used to determine the significance with which mathemat-

ics statistically correlated with programming. Both RSTudio and SPSS were used to analyse the data at different stages of the research, including:

1. Data cleaning;
2. Generating descriptive statistics;
3. Validating pre-correlation test assumptions;
4. Finding correlations;
5. Repeating the test for all cohorts;
6. Plotting diagrams.

3.3.2 Interviews

Conducting an interview is an essential method for gathering qualitative data that can help a researcher understand the meaning of certain research phenomena and understand how and why participants have particular perspectives (DiCicco-Bloom & Crabtree, 2006). Interviews allow us to capture individual beliefs and perceptions of how reality can be constructed (Punch, 2009). Interviews can be advantageous in gathering in-depth information about social inquiry that cannot be gathered through questionnaires, where closed questions limit responses in some cases (Hobson & Townsend, 2010; Wellington, 2000).

Using questionnaires, which could include open-ended questions, may have been suitable for our study. However, we chose not to, as the nature of our research was about identifying certain characteristics of gifted student programmers, which needed to be closely investigated, providing rich information from multiple aspects. Interviews were more suitable, as there was room for the researcher to follow up on some emerging points and themes in participants' responses. Interviews were suitable in this study, as the interviewer could simplify difficult questions and terms. For example, in CS,

multiple synonyms can be used that could confuse interviewees. In addition, the sample size was small, which could affect the statistical power of the questionnaire data.

Punch (2009) categorised interviews as unstructured, structured, or semi-structured. An unstructured interview allows the interviewer to use knowledge and memory to ask open questions to allow participants to freely answer. This type of interview could be considered a formal conversation, and an interview guide is not required (Bryman, 2012, p. 741). A structured interview requires an interview guide to be developed beforehand, where questions are asked in a certain order with little flexibility or responsiveness.

In contrast, a semi-structured interview still requires an interview guide to be developed, but the interviewer has more flexibility in altering the way that the questions are outlined as well as being responsive, asking questions on emerging themes. In addition, the interviewee has the advantage of freedom in expressing their views (Bryman, 2012, p. 741). Thus, in this study, semi-structured interviews were used to give participants the freedom to express their views through a mix of open-ended and closed questions (King & Horrocks, 2010, p. 75).

A variety of programmer characteristics were suggested in the IT industry and CS literature; these are categorised into broad concepts in Figure 3.3.

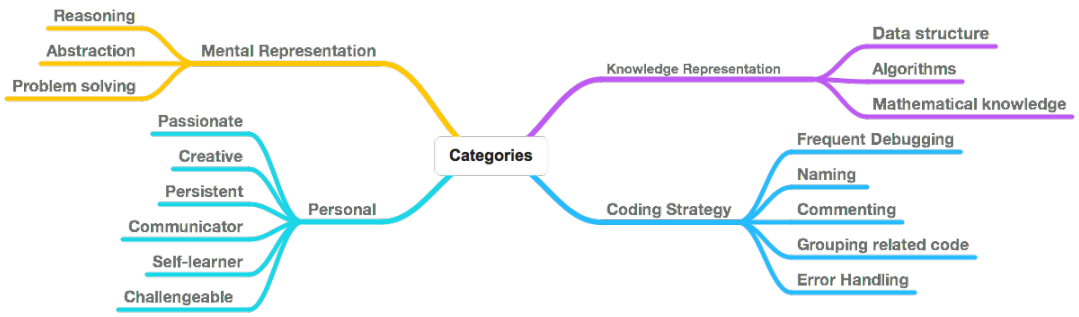


Figure 3.3: Categories of programming characteristics.

However, first-year CS students might possess different or specific characteristics, which can also develop during their study. In addition, the educational context includes multiple aspects that could affect characteristics related to giftedness in programming. The purpose of this research was to understand more about the characteristics of gifted

student programmers, which could help to provide a theoretical base for identifying gifted students at an early stage of their academic study, so they can be accelerated or enriched through special gifted education programmes to maximise their potential.

Semi-structured interviews were conducted to gather in-depth qualitative data from first-year gifted student programmers to answer the following research questions:

- What are student perceptions about the relationship between mathematical ability and programming ability?
- What mental representation strategies do gifted students tend to use?
- What mathematics and programming knowledge do gifted students tend to have?
- What coding strategies do gifted students tend to use?
- What attitudes and personality traits do gifted students tend to possess?

Broad categories derived from the literature were mental representation, knowledge organisation, personal traits, mathematics, and programming relationships. Participants were asked about certain characteristics related to each category, as shown in Figure 3.3. These characteristics could be programming and mathematics background, perception of mathematics and programming relationships, problem-solving strategies, data structure, algorithm implementation, and coding strategies. Consequently, responses regarding each question provided rich information to identify whether gifted programmers possess certain characteristics. A complete semi-structured interview schedule can be found in Appendix A.

3.3.2.1 Pilot Interview

A pilot study is useful in any research to indicate flaws, limitations, or weaknesses in data collection approaches. A pilot study also acts as an explorer study providing valuable input in the early stages of case-study design (Yin, 2009, p. 92). We decided to conduct a pilot study to serve the following aims:

- Validate the suggested categories of programming characteristics derived from the literature;
- Gain experience recruiting participants and conducting interviews;
- Refine the interview questions to maximise the chance of obtaining correct data.

Our intention was to interview five second-year students who had been identified based on their high performance in programming and mathematics modules. The students had been invited by email, but there were no responses. An alternative plan was then to ask PhD colleagues to participate and be interviewed, and five PhD students participated. During the interview, questions were about educational backgrounds, including programming and mathematics backgrounds, perceptions on the relationship between programming and mathematics, and problem-solving and coding strategies. The feedback from participants indicated the difficulty of some questions and the long duration of the interview. This feedback was considered, and some questions were refined or simplified. For example, a question that aimed to allow students to give an abstract explanation about software was modified to be clearer, instead asking students how they would analyse software based on its functionalities and data structures. Another unclear question was on investigating coding strategies for tracing code error and debugging. The initial question was 'How do you trace syntactical and semantical errors?' This was simplified to 'What are your debugging strategies?'.

3.3.2.2 Participants

Purposive sampling is a key feature for qualitative research in which the researcher selects participants based on their own perspective about certain groups of participants who possess certain characteristics that have been sought by the researcher (Cohen et al., 2011, p. 156). This type of sampling allows the researcher to have in-depth information that cannot be the case in random sampling, which provides greater breadth of participants but may not be as rich in meaning, as not all participants will possess the desired characteristics.

Based on the nature of this research, purposive sampling was used to select gifted students studying first-year CS at the University of Warwick. The selection was based on multiple criteria that were important in our study. The first criterion was that participants must have been at the end of their first year and must have studied two programming modules and two mathematics modules so that they could be selected based on their performance in these modules.

Thus, the second criterion was that participants should be in the top 10% of their cohort. However, as the cohort included 134 students, this gave a very small population size of 13 students and may not have guaranteed successful recruitment. We therefore included the possibility of increasing the target to the top 15-25% to increase the potential population size and, in turn, potential number of participants.

The third criterion was that the sample should include a negative case in which some participants might not confirm the initial theory suggested, that great mathematicians could be gifted programmers. Given the fact that our population size was small, based on the unsuccessful recruitment experience when conducting the pilot study, incentives were therefore included to maximise the probability of successful recruitment.

Recruitment involved two rounds of invitation via email. In the first attempt, four CS students who met the inclusion criteria agreed to participate; the second attempt secured five DM participants. A total of nine participants were interviewed. All participants were assigned pseudonyms and case IDs to protect their identities, with a list of pseudonyms and IDs stored confidentially by the researcher.

3.3.2.3 Coding and Themes

Analysing qualitative data includes organising, explaining, and reporting data through which a research problem can be investigated and understood by patterns, themes, and categories found in the data. Qualitative research produces large amounts of content, such as interview transcripts, memos, notes, and documents that help the researcher gain rich data. Thus, a researcher may feel uncertain as to how to analyse the data

(Bryman, 2012, p. 565).

Merriam and Tisdell (2009, p. 157) stated that the analysis stage could be the most daunting stage of a research study, as a mass of data could lead to ambiguity. Additionally, qualitative data could be interpreted differently by different researchers since there is no uniquely rigorous method of analysis. However, explaining and recording analysis procedures in detail can minimise subjective assumptions.

The nature of qualitative research implies that researchers are the prime instruments of the research. Researchers' perspectives that are derived from experience and knowledge might be valuable. However, researchers must introduce rigorous measures to minimise subjectivity. In addition, it is crucial that the researcher consider that the methods and analyses used must fit the purpose of the research. Although there are rules and guidelines for maximising validity in qualitative analysis, the degree of ambiguity is still obvious for the researcher as every set of qualitative data is unique and requires specific interpretation in which researchers' views cannot be avoided.

Choosing the type of analysis prior to beginning the research not only helps to increase research validity and reliability by ensuring fitness for purpose but can also aid the researcher in being clearer on how to organise, analyse, and report the qualitative research. Thus, the researcher must determine which type of analysis will be used: descriptive, narrative, thematic, discourse, or content analysis.

Thematic analysis, which is a widely used technique in qualitative analysis, aims to identify, analyse, and report themes derived from data (Braun & Clarke, 2006, p. 6). Thematic analysis is not considered a unique method of analysis, unlike other methods, such as grounded theory, which has been well-established and developed. In addition, finding themes is an activity that has been used in other analysis methods, which makes thematic analysis less distinctive (Bryman, 2012, p. 578).

As clear definitions of thematic analysis are lacking, it has often been adsorbed into other well-branded analysis methods, such as grounded theory or the narrative method (Bryman, 2012; Braun & Clarke, 2006). Although most qualitative analysis techniques share common steps of identifying themes that have emerged from data,

thematic analysis varies from other methods. For example, in grounded theory, themes derive from data, but a comprehensive theory may not have emerged.

In contrast, in thematic analysis, researchers might not subscribe to developing a theory, instead discovering themes based on participants' experiences and perspectives. Thus, 'thematic analysis can be a method which works both to reflect reality, and to unpick or unravel the surface of reality' (Braun & Clarke, 2006, p. 9).

Themes are built based on collected data, specifically from codes generated from transcripts in which themes must be related to the research questions. Opler, as cited in (Ryan & Bernard, 2003, p. 86), defined three concepts for themes: (1) themes are noticeable 'through the manifestation of expressions in data' and thus can be discovered; (2) some themes could be obvious and commonly used and others might be 'subtler, symbolic, and even idiosyncratic'; and (3) themes must relate to how frequently they appear and whether they appear in different ideas and practices.

Themes can be inductively generated from the data or deductively derived from literature or the view of the researcher. Patton, (as cited in Braun & Clarke, 2006, p. 12) stated that, in the inductive approach, the theme identification process is strongly related to the data. Thus, during the coding process, researchers do not code the data according to prior categories identified in literature or based on the researcher's view. Often, initial themes can be generated from interview questions, whereas themes derived from characteristics of such phenomenon that have been studied and found in literature reviews are partly empirical (Ryan & Bernard, 2003, p. 88).

Despite the fact that identifying themes is a common research activity in qualitative research, there are a wide range of terms used by social scientists for themes, including category, concepts, codes, and labels. However, Strauss, as cited in (Ryan & Bernard, 2003, p. 87), described the relation between expression and themes as 'conceptual labels placed on discrete happenings, events, and other instances of phenomena'. Thus, expressions could be grouped into themes, and themes could consist of sub-themes.

An alternative approach that helps researchers analyse multiple cases is cross-

case analysis, where comparisons between cases can be made to discover similarities and differences. It has been argued that cross-case analysis might force the data to be shaped in such a way as to make multiple comparable units. However, if the data have been collected based on prior themes, cross-case analysis can be advantageous (Miles & Huberman, 1994, p. 102).

One of the advantages of using the cross-case technique is to increase generalisation by comparing different cases against specific themes about such a phenomenon. Although generalisation in qualitative research has not been promising for social phenomena constructed based on unique individual participants and experiences, cross-case analysis could help us understand the circumstances that lead to similar findings about different individuals.

3.3.2.4 Analyses Procedures

Interview analysis procedures consist of transcription, organisation, coding, themes and sub-themes, narrative, and cross-case analyses. The interviews were audio recorded, and participants were informed about the audio recording before conducting the interviews. The analysis process for the data gathered through interviews started by transcribing the data, which could include detailed information and accurate perspectives of the participants. When transcribing, the researcher can either write the exact dialogue including repetition and hesitation in a verbatim transcript or choose to omit the original data by reporting direct sentences. A total of five hours of interview recordings were transcribed verbatim, noting issues with two interviews (non-English speaking and speech impediment), which meant transcription was more laborious.

All transcripts, audio file transcripts, and memos for all participants were uploaded to NVivo, which allows data to be electronically organised, managed, and coded. Using NVivo, a new project was created and named CGPS. In addition, nine cases were created, as each participant represents a case and can be identified based on case ID and pseudonyms. However, a list of participant names and university student numbers, along with their case ID and pseudonyms, was kept confidential for the purpose of

checking correct information. Demographic data were registered during the interview, such as gender and other data, and were extracted from the transcriptions. Table 3.1 shows demographic data including pseudonyms and case IDs.

ID	Name	Gender
91	Sara	Female
98	Bob	Male
36	David	Male
42	Joe	Male
55	Sam	Male
78	Steve	Male
14	Lee	Male
49	Robin	Male
79	Allen	Male

Table 3.1: Participant demographic data.

Under each case, the interview transcript was stored and coded by creating NVivo nodes and sub-nodes. Coding, which requires a careful review of interview transcripts to find significant data, was the next stage of qualitative data analysis, in which multiple levels of coding, such as open, axial, and selective coding could be applied (Cohen et al., 2011; Bryman, 2012). Open coding is a process of examination in which the researcher compares data to produce concepts, which refer to certain labels given to specific segments of data, which are then categorised. Selective coding focuses on determining core themes that can be more generic and sufficiently abstract to relate to such a phenomenon (Bryman, 2012, p. 569). The coding process should be consistent, moving back and forth to identify similarities and differences between themes from each data analysis. In this study, the coding process started by reading the transcript for the first participant and assigning meaningful text to NVivo nodes and sub-nodes. At the end of the coding process, numbers of nodes and sub-nodes were produced and checked by the supervisors. The coding process was repeated for all participant transcripts, and the researchers were open to any modifications, additions, and/or deletions of the previous nodes and sub-nodes. Themes emerged from grouping the related nodes. The resultant themes were early education, mathematics and programming, personal traits, coding strategies, mental representation strategies, and the programming project called

Witter. For example, the mental representation strategy theme consists of two sub-nodes, which are problem solving and abstract thinking. Figure 3.4 shows an example of the mental representation theme and its sub-nodes. The problem-solving sub-node

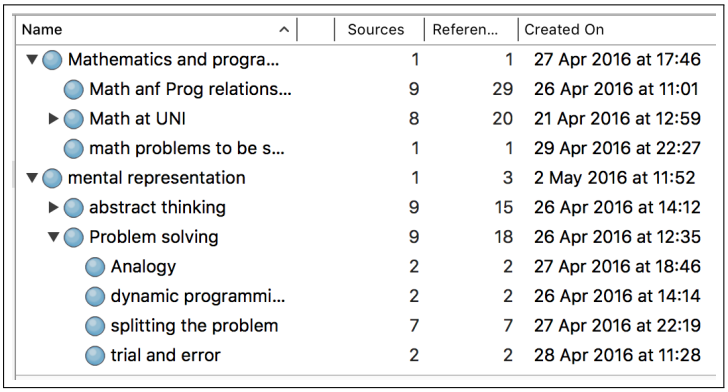


Figure 3.4: Example of NVivo nodes.

consists of sub-nodes as labels assigned to each participant's problem-solving strategy. If the participant's response was related to a specific strategy, a sub-node was created and assigned to the response. Figure 3.5 shows segments of each participant's transcript related to a specific problem-solving strategy, which was assigned to splitting the problem sub-node.

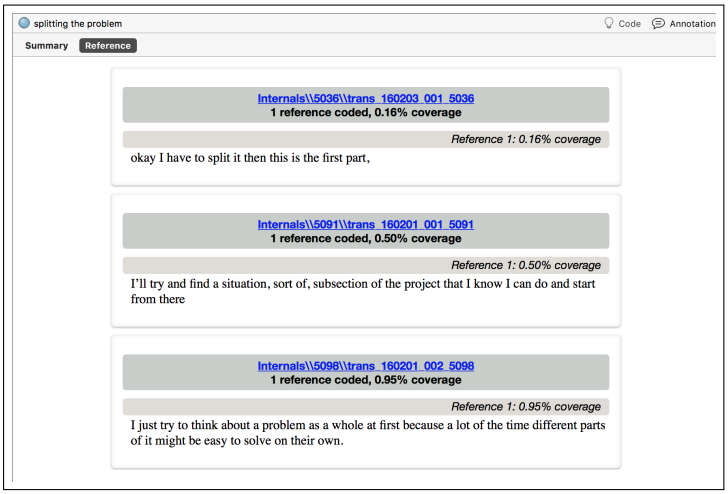


Figure 3.5: Segments of participant transcripts.

Narrative analysis, which provides interpretations of participants' experiences and stories, can provide a better understanding of such a phenomenon (Kim, 2015). In this study, the participants have been identified as gifted programmers; thus, conducting a narrative analysis can provide personal experience and stories that could

explain giftedness in programming. The narrative analysis was incorporated with the explanation of produced themes by interpreting participants' experience and stories. In addition, narrative analysis can highlight our participants for a wider spectrum of audience than CS educators. We include direct quotes from the participants when we need to emphasise our explanation of some characteristic or theme.

In addition to the thematic and narrative analysis techniques, we used a cross-case analysis to perform a comparison between cases. Cross-case analysis allows answers for specific themes to be grouped together to analyse central issues based on all views (Patton, 1990). We should emphasise that an early decision was made while designing our case study, which consisted of multiple cases, whereby each case included multiple units of analysis. To be more precise, each participant represented a case that consisted of multiple characteristics for gifted programmers. This design allows us to perform a comparison between participants based on identified characteristics and themes. Table 3.2 shows an example of a cross-case analysis related to the problem-solving strategy and the Witter theme. A comprehensive summarisation of the cross-case analyses will be discussed in Chapter 5.

ID	**Name**	**Problem-solving strategy**	**Data structure used in Witter**
91	Sara	Splitting a problem	Hash map and binary tree
98	Bob	Splitting a problem	Hash map and binary tree
49	Robin	Splitting a problem	Hash maps and array lists
55	Sam	Splitting a problem	AVL tree and adjacency list map
36	David	Splitting a problem	AVL tree and Hash map
		Dynamic programming	
78	Steve	Splitting a problem	Hash map, adjacency list map and array list
		Dynamic programming	
42	Joe	Splitting a problem	Binary tree, array and hash map
		Trial and error	
14	Lee	Analogy	hash maps
79	Allen	Trial and error	Hash map, linked lists, and adjacency list map
		Analogy	

Table 3.2: Example of cross-case analysis for problem-solving strategy and Witter.

3.3.3 Code-Writing Problems

Applying triangulation using other sources of data increases the level of validity (Bonoma, 1985, p. 201). As noted earlier, triangulation between methods allows multiple evidence bases to be collected using different methods. Collecting and analysing student code-writing problems triangulated evidence relating to their mental representations, indicating how gifted student programmers perform when writing code to solve a problem, and signs indicating that students possess a high level of abstraction when solving a complex problem, such as recursion.

Student exam scripts for a level 1 programming course (CS118), which covers programming fundamentals and OOP using the Java language, were collected through the Department of Computer Science administration following ethical approval. Three code-writing problems were selected, each of which included different programming constructs: array creation, linear search, and recursive method.

3.3.3.1 Analysis Procedures

Content analysis is a qualitative analysis technique in which the main objective is to describe and quantify a phenomenon. The content analysis approach allows researchers to investigate theories by analysing documents through categorising words that could have the same meaning in related classifications.

Bryman (2012) defined content analysis as an approach to quantify content based on predetermined categories in which analysis procedures should be systematic and replicable. Although content analysis allows a quantitative approach, for example, by counting the occurrence of specific words within the data, it has been claimed that providing a statistical analysis for social phenomena leaves researchers with less understanding of those phenomena rather than producing numerical facts. Hence, content analysis might be considered an approach that has a less qualitative nature (Elo & Kyngäs, 2008, p. 108). Another feature of content analysis is that it can be integrated with other approaches (Bryman, 2012).

Another technique of analysis was used and incorporated with content analysis

to maximise the validity of our analysis by utilising a well-developed educational taxonomy. As noted in Chapter 2, the SOLO taxonomy provides a framework to classify assessments based on what each exam question intends to measure, for example, at certain levels of student cognitive ability. As discussed in Chapter 2.4, CS educators established a framework utilising the SOLO taxonomy to classify student outcomes for code-writing problems (Lister et al., 2010; Whalley et al., 2011). We adapted the analysis approach of Whalley et al. (2011), as shown in Figure 3.6, to develop silent programming elements (SPEs) for each code-writing problem, to which the responses were openly coded. The SPE could be identified based on syntactical elements, program constructs, and features. Those features could be abstract and imply certain qualities related to code writing. Again, it is worth acknowledging the use of the term 'code' could refer to coding as a methodological approach for qualitative data content analysis and coding as writing a computer program. Additionally, the terms 'response', 'solution', and 'code' were used interchangeably. The final step of qualitative data analysis was to use the developed SPEs to categorise student responses according to the SOLO categories shown in Table 3.3.

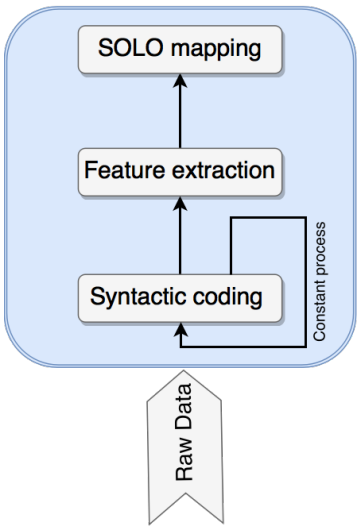


Figure 3.6: Bottom-up analysis approach.

As Bryman (2012) emphasised, analysis procedures should be rigorous, systematic, and replicable. Analysis procedures for this study are outlined in Appendix B

phase	SOLO category	Description
Qualitative	Extended Abstract - Extending [EA]	Uses constructs and concepts beyond those required in the exercise to provide an improved solution.
	Relational - Encompassing [R]	Provides a valid well-structured program that removes all redundancy and has a clear logical structure. The specifications have been integrated to form a logical whole.
Quantitative	Multistructural - Refinement [M]	Represents a translation that is close to a direct translation. The code may have been reordered to make a *more integrated and/or valid solution.*
	Unistructural - Direct Translation [U]	Represents a direct translation of the specifications. The code will be in the sequence of the specifications.
	Prestructural [P]	Substantially lacks knowledge of programming constructs or is unrelated to the question.

Table 3.3: Refined SOLO categories for code-writing solutions (Whalley et al., 2011).

and were designed to include all required analysis steps, which were replicated to reduce subjectivity. The researcher conducted the analysis for the three code-writing problems for nine students. In addition, two CS PhD students from Warwick University and Southampton University were also trained in the analysis procedures and replicated the analyses, as each researcher might extract different code features and give them different weights of importance, which could affect SOLO categorisation of student responses. Each researcher then consolidated their findings and agreed on final categorisation.

3.3.3.1.1 Array creation problem

The first problem was about writing a method that takes a single integer, n, as an argument to create an array of size n with random values between *0* and *100*. Students should write a valid code to demonstrate their knowledge of array declaration, initialisation, and iteration. In addition, the code could be implemented using Java built-in math or random objects and their functions to generate random values to be stored in the array. We assumed that those objects had been introduced to the students in the course. However, the question included a non-direct translation of the specifications, as the array must include values between *0* and *100* inclusive. In this

case, in math objects, there is a random method that returns a double value, greater than or equal to *0.0* and less than *1.0* that needs to be multiplied by *101* and converted to an integer. Thus, the array would include values from *0* to *100*. Comparing this with the SOLO taxonomy suggested that the question should be categorised as Multistructural. A sample student solution is shown in Figure 3.7. Coding student

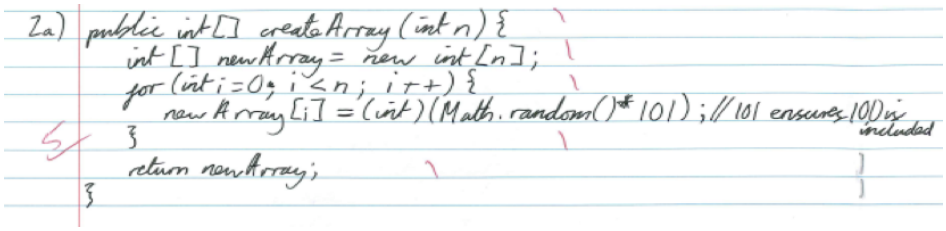


Figure 3.7: Sample student solution for the array creation problem.

answers was the next step to develop SPEs derived from the student program code to identify program constructs and syntaxes that could be used to implement a program to solve such a problem. As shown in Table 3.4, the main program constructs consisted of method declaration, array declaration, iteration, initialisation with random values, and a return statement. In most solutions, the methods were declared correctly to return the created array. However, some methods were declared to be static, which was not required in the question specifications. For the array declaration, all solutions declared the array with size n in a one-line statement, which is more efficient than using two-line statements.

All solutions implemented the array iterations using one finite loop, whereas there were two options to generate random values to be stored in the array. Both math and random Java objects were implemented; however, some solutions were not able to generate random values between *0* and *100*, including *100* as specified in the question.

Based on the resulting SPEs and features extracted from student codes, Table 3.5 shows the SOLO mapping. The third researcher categorised two answers (55 and 36) at a category higher than the highest possible category; therefore, we agreed to downgrade the answers to the Multistructural level.

Construct	Element	Feature
Method declaration	Public int[] array (int n)	Typical
	Public void array (int n)	Void method
Array declaration	int [] array = new int[n];	Efficient
Array iteration	1x for loop	Finite loop
Random value generation	Using Math object	Inclusive range
		Exclusive range
	Using Random object	Exclusive range
Return statement	Return array	Included
		Missing

Table 3.4: Program constructs and features for the array creation problem.

Construct	Feature	Solutions by Student Number								
		55	14	36	91	98	78	42	79	49
Method declaration	Typical	x	x	x	x	x	x	x	x	
	Void method									x
Array declaration	Efficient	x	x	x	x	x	x	x	x	x
Array iteration	Finite loop	x	x	x	x	x	x	x	x	x
Random value generation	Inclusive range	x		x	x					
	Exclusive range		x			x	x	x	x	
Return statement	Included	x	x	x	x	x	x	x		
	Missing								x	x
SOLO mapping (1st researcher)		M	U	M	M	U	U	U	U	U
SOLO mapping (2nd researcher)		M	U	M	M	U	U	U	U	U
SOLO mapping (3rd researcher)		R	U	R	M	U	U	U	U	U
Final and agreed SOLO mapping		M	U	M	M	U	U	U	U	U

Table 3.5: SOLO mapping for the array creation problem.
U=Unistructural; M=Multistructural; R=Relational.

3.3.3.1.2 Linear Search problem

The second problem was to write a method that takes an array and an argument *s* as arguments and performs a linear search on the array finding the index when *s* is found or returning *-1* if *s* is not found. The researchers agreed that the question specification can be translated directly and should be categorised as Multistructural.

All students demonstrated a clear understanding of the question and produced code that included the main constructs. One student's code tended to have a redundant declared variable to be returned; thus, it was considered a redundancy in the return statement. Different constructs extracted from the student code included method declaration, array iteration, selection, and the return statement. Student solutions were categorised based on derived features, as shown in Table 3.7.

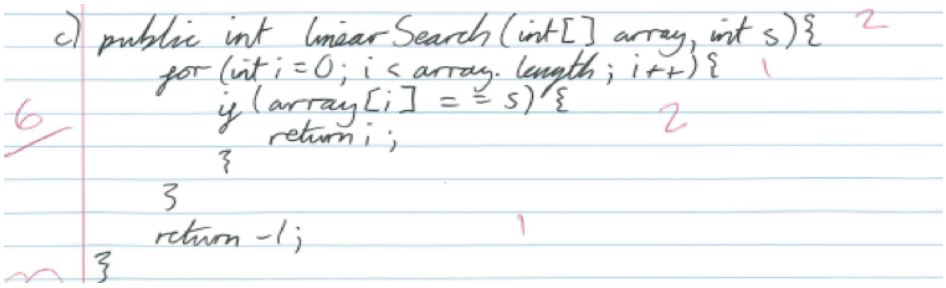


Figure 3.8: Sample student solution for the linear search problem.

Construct	Element	Feature
Method declaration	public int linearArray(int [] array, int s)	Typical
	public void linearArray(int [] array, int s)	Void method
Array iteration	for(int i=0;i<s;i++)	Finite loop
Selection	If statement	Valid condition
Return statement	int find = -1;	Redundant
	Return find	Non-redundant

Table 3.6: Constructs and features for the linear search problem.

3.3.3.1.3 Recursive problem

The third question was about writing a recursive method that calculates the sum of the differences between opposing pairs (i.e. the difference between *A[0]* and *A[n-1]*, *A[1]* and *A[n-2]*, etc.). The question aimed to measure a student's ability to implement a recursive method, which is considered a difficult concept for novice programmers to understand. Thus, this question was categorised as relational, as the question included additional complex constructs along with applying the recursion concept.

A typical solution passes an array with a variable that keeps track of the array index that traverses incrementally from left to right. Then, it is important to have a second variable that keeps track of the array index that traverses in the opposite way. In addition, the edges of the array must be checked to calculate differences between the edges. Figure 3.9 shows the sample student code, which meets the question specifications and is considered valid. Table 3.8 shows constructs, elements, and features extracted from the student code. The most important constructs that differentiated the solutions for the SOLO mapping were edges, difference calculation, and recursive method invocation.

Given the fact that the nature of recursion involves a degree of abstraction, novice students often encounter difficulties implementing recursive methods (Wirth,

Construct	Feature	Solutions by Student Number								
		78	98	91	79	14	49	42	36	55
Method declaration	Typical	x	x	x	x	x	x	x	x	
	Void method									x
Array iteration	Finite loop	x	x	x	x	x	x	x	x	x
Selection	Valid condition	x	x	x	x	x	x	x	x	x
Return statement	Redundant								x	x
	non-redundant	x	x	x	x	x	x	x		
SOLO mapping (1st researcher)		M	M	M	M	M	M	M	U	U
SOLO mapping (2nd researcher)		M	M	M	M	M	M	M	M	M
SOLO mapping (3rd researcher)		M	M	M	U	M	M	M	M	U
Final and agreed SOLO mapping		M	M	M	M	M	M	M	M	U

Table 3.7: SOLO mapping for the linear search problem.

U=Unistructural; M=Multistructural; R=Relational.

2014). Therefore, student solutions manifest different levels of SOLO categories ranging from the lowest to the highest (which is relational in this question). Two students were not able to understand the question requirements and provided solutions lacking constructs related to the question. Table 3.9 shows the SOLO categorisations.

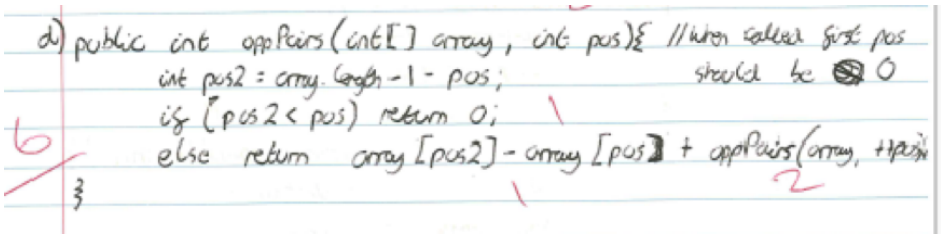


Figure 3.9: Sample student solution for the recursive problem.

Construct	Element	Feature
Method declaration	Public int oppPairs(int [] array, int pos)	Typical
	Public int oppPairs(int [] array)	Missing argument
	Public void oppPairs(int [] array, int pos)	Void method
Variable assignment	int pos2=array.length() -1-pos;	Efficient
Edges	If (pos2<pos)	Valid
Difference calculation	int diff = array[pos2]-array[pos] + array[pos2-j]-array[pos+i];	Invalid
	int diff = array[pos2]-array[pos] + oppPairs(array,++pos);	Efficient
Recursive invocation	oppPairs(array,++pos)	Valid argument
		Invalid argument
Return statement	Return array	Non-redundant

Table 3.8: Constructs and features for the recursive problem.

3.3.4 Project Source Code (Witter)

Witter is a social networking application that allows users to post information, called 'Weets'. The main aim of Witter is to allow a user to post a Weet and to display all user information including all Weets, followers, and other users who have been followed by

Construct	Feature	Solutions by Student number								
		49	14	79	91	98	78	55	36	42
Method declaration	Typical			x		x	x		x	x
	Void method	x			x			x		
	Missing argument		x							
Variable assignment	Efficient							x		x
Edges	Valid									x
	Invalid								x	
Difference calculation	Efficient								x	x
Recursive invocation	Valid argument				x	x	x	x	x	x
	Invalid argument			x						
Return statement	Non-redundant		x	x	x	x	x	x	x	x
SOLO mapping (1st researcher)		P	P	U	U	U	U	U	M	R
SOLO mapping (2nd researcher)		P	P	M	M	M	P	U	R	R
SOLO mapping (3rd researcher)		U	U	U	U	U	U	U	M	R
Final and agreed SOLO mapping		P	P	U	U	U	U	U	R	R

Table 3.9: SOLO mapping for the recursive problem.
P= Prestructural; U=Unistructural; M=Multistructural; R=Relational.

the user. Students in the Design of Information Structures CS126 module completed a project to write Witter codes to evaluate student programming ability using advanced data structures and algorithms. The project accounted for 40% of the final grade.

The project was considered to be a large-scale programming task to develop software that students can implement in different stages, including analysing requirements, designing the data structure and algorithms, implementing, and testing. Students had to implement three classes: User Store, Weet Store, and Follower Store.

The User Class was used to store all users who could be searched to reveal specific user information. The Weet Store class was used to store Weets that could be displayed on both user pages and the Weet page. Weets were searchable and trending Weets must be displayed on the main page. The Follower Store class was used to store user relationships, determining followers and who users follow.

All classes must have implemented methods that were included in pre-defined interfaces. Given that the application was a social networking platform, a large number of users were expected to use the application, generating huge numbers of Weets. Therefore, the data structure and algorithms needed to be designed carefully, considering time and space complexity.

Project marking was based on both automated tests and criteria of memory efficiency, time efficiency, design and understanding, and documentation. As multiple data structures and algorithm designs could be used, few designs could provide efficient application, considering fast computation and careful use of server resources.

Student Witter codes were collected through the module instructor based on prior ethical approval. The data were collected with the aim of accumulating evidence around particular programming characteristics.

First, the coding strategies that gifted student programmers used to write Witter code were investigated to explore what was used for the code conventions. Second, the mental representations used by the students were investigated to understand how they represented Witter problems and to understand their reasoning for choosing a particular data structure and algorithm designs. Third, student knowledge of data structure and algorithms was examined.

The data provided in-depth information about student abstraction abilities, as the assignment required concepts of object-oriented programming, such as classes, objects, inheritance, overriding, and polymorphism. Applying these concepts required a high ability of abstraction to understand the relations between classes, objects, and functions. In addition, the data provided evidence of how students applied certain types of data structures that could efficiently tackle specific problems within Witter.

3.3.4.1 Analysis Procedures

A content analysis approach was used to analyse student source code, allowing researchers to read the source code, conduct open qualitative coding (to look for certain evidence related to characteristics, such as student coding strategies, mental representation, and data structure knowledge).

The source codes contained an average of 2013 lines of Java code for each student, including Witter classes and functions. However, the analyses were narrowed to focus on identifying how students represented the Witter problem based on data structure and algorithms.

As part of the code file, students had to write descriptive comments, providing rationales for their approaches. The descriptive comments were hand-coded by the researchers to understand reasoning when choosing a particular data structure and algorithm. In addition, sections of the source code related to the main Witter classes,

and functions were coded to understand how students implemented their approach. As the source code was analysed, we investigated certain coding strategies related to code conventions.

Additional data related to Witter were collected through different methods. During the interviews, students were asked about their perception of Witter, in order for their responses to be used to triangulate findings on Witter performance. In addition, Witter grades were collected in the early stages to help us identify students who performed well in this project and were incorporated to provide a quantitative perspective on student performance.

3.4 Validity and Reliability

This section includes a discussion of validity and reliability of the two main methodologies - quantitative and qualitative case study - and explains how certain measures were applied to the data collection methods.

3.4.1 Validity and Reliability of Quantitative Methods

Validity in quantitative research, which concerns sampling accuracy, selecting the proper statistical test, and measurement validity, is a very important element for producing accurate research findings. However, achieving complete validity in research is impossible, as one of the characteristics of quantitative research has a certain degree of uncontrolled error that cannot be avoided; thus, validity can only be obtained up to a certain degree (Cohen et al., 2011, p. 179).

To attain a high level of sampling accuracy for our numerical sample, we aimed to carefully include correct data by conducting a data cleaning process to minimise sampling errors. Complete student records, which needed to include grades for two programming modules and two mathematics modules, were included in our sample. The raw data consisted of 3,060 student records from 1996 to 2015. In addition, we strived to achieve accurate results of testing the correlation by calculating the average

grades across the two programming modules and two mathematics modules.

In quantitative methodology, generalisation is considered advantageous over the qualitative methodology, as research findings could be extended to establish general laws derived from experimental research to determine causation. However, universal laws cannot be generalised from social research; instead, a degree of commonality can be established, especially in the educational context, where students, teachers, and curricula can have a certain degree of similarity (Cohen et al., 2011, p. 186). However, our numerical data, which were carefully processed, were relatively substantial compared to previous studies in which correlation results were statistically significant.

Reliability in quantitative methodology consists of three elements: stability, equivalence, and internal consistency. Improving reliability may be achieved by reducing the effects of uncontrolled or external factors, increasing standards for data collection and measurement, and excluding corrupted data during analyses (Cohen et al., 2011, p. 200-201). Dealing with null values is an important element to increase reliability. Therefore, data were cleaned to remove student records that did not include all grades for all modules and to check for outliers caused by typographical errors. A sanity check of the data was also conducted by the researcher and academic supervisors.

Stability concerns consistency of data measurement in which results should be consistent when an experiment or test is repeated with similar data. As mentioned, student grades were used spanning a long period to determine possible correlation between mathematical ability and programming ability; thus, the prime instruments to measure student performance were coursework assignments and final exams. Although our sample was large, student performance varied from one year to another and was affected by several external educational factors, including curricula, teaching methods, and assessments. These external factors could not be completely controlled, but we aimed to minimise their influence. A statistical comparison between all cohorts, determining the differences between means, was conducted to exclude cohorts that demonstrated comparatively abnormal student performance.

3.4.2 Validity and Reliability of Case-Study Methodology

Validity and reliability in a case study establish measures to increase qualitative research integrity (Christie et al., 2000, p. 16) and to reduce criticisms, such as biased views or lack of rigour (Yin, 2009, p. 14). Five measures were introduced by Miles and Huberman (1994) and Yin (2009, p. 40) to achieve reasonable research integrity and rigour validity: construct validity, internal validity, external validity, confirmability, and reliability.

Construct validity establishes an operational set of measures to reduce the researcher's subjective judgements derived from having full control of the case study and steering the results to a desired destination. In addition, a case study can be accused of lacking objectivity, as the researcher's views could prejudice the results and provide inadequate evidence. Yin (2009, p. 41) suggested that the case-study methodology can achieve construct validity by using multiple sources of evidence, establishing a chain of evidence, and having key external informants to review draft case-study reports. Various data sources can be used in a case study via triangulation to allow cumulative support, providing robust claims. We collected data from multiple sources of data, interviews, code-writing problems, project source code, and student grades, to build evidence around particular student characteristics. For example, participants provided qualitative information about how they approached the Witter project during the interview compared with the Witter source code data and grades.

Subjectivity in a case study can be addressed through careful review of the literature to consider different perspectives, including oppositional points of view. Additionally, constructing the interview guidelines, interview process, transcription process, and detailed analysis process can help the researcher provide rigorous analyses (Dick & Dick, 1990, p. 2) to increase objectivity. Careful planning, designing, and analysis of a case study can increase its research validity and reliability. Thus, in this research, the initial literature review provided several potential categories of characteristics of a gifted programmer to help develop an interview schedule to investigate these categories. A schedule was carefully designed to ensure that the researcher's views were not

imposed on participants by asking non-leading questions. In addition, the interview schedule was examined by two academic supervisors and was refined during the pilot study.

Internal validity in qualitative research ensures that the relationship between variables can be established and allows for identification of causes and effects. However, in descriptive and exploratory research, it can be impossible to determine relationships or/and causation that are not the main purpose of this type of research (Yin, 2009, p. 9). Approaches that were considered to achieve internal validity in these case studies included pattern matching, discussing findings compared with rival viewpoints, and linking findings with the literature review. Reporting research findings to a wider audience can also be beneficial to discuss, reflect, gather feedback, and refine. As an exercise for reporting our findings to a wider audience, we published two papers. The first reported findings on the correlation between mathematical ability and programming and was published at the Computer Science Education: Innovation and Technology Conference in 2015. The second paper, which reported findings on code-writing abilities for gifted student programmers, was published at the Psychology of Programming Interest Group Conference in 2017.

External validity concerns whether results can be generalised beyond the limitations of the case-study context. It could be argued that a single-case study produces insufficient generalisation of results, whereas a quantitative study tends to provide statistical results that can be generalised. Yin (1994, p.10) discussed the issue of scientific generalisation, as some research includes single cases and leads to limited results based on a specific context and small population. Single scientific experiments can also result in a biased conclusion, so replicating experiments can verify results and allow generalisation. This approach can also be taken in a case study by including multiple cases, where one case study can be conducted and constructive feedback injected into the next case study.

In addition, a case study relies on analytical generalisation in which the researcher investigates single and/or multiple cases within a context to obtain results

that can be generalised to a wider theory (Yin, 2009, p. 15). To overcome the lack of generalisation in case-study methodology in this research, an embedded multiple-case design was implemented across nine participants.

Reliability is a measurement to ensure that results can be replicated if the study is repeated (Merriam, 1988). In qualitative research, human behaviour is changeable and cannot be controlled. This is also true for external variables, such as the context of the study, which can affect the results. Additionally, the researcher's own beliefs and interpretations regarding the research problem and the methods for conducting the research can differ from one researcher to another, increasing complications for yielding the same results when repeating the study (Merriam, 1988). However, it could be argued that some research aspects related to specific contexts can be controlled. In this case, some educational aspects, such as curricula, teaching methods, and assessments related to the programming and mathematics modules, could be controlled to a certain degree.

Confirmability ensures that the research has been conducted in the same way that the researcher describes by keeping a record of collected data, including audio files, transcripts, and interview notes. In addition, the researcher must not be selective while reporting the case study and consider any information that contradicts the researcher's perspectives. In this study, all cases were reported in full.

3.5 Ethical Considerations

Ethics in research requires that a set of principles that could apply to a research context and human participants are not violated by the researcher, as participants' dignity must be respected. Additionally, the researcher must strive to report data honestly and avoid biased evaluations of the results as well as be open to rival perspectives.

Diener and Crandall (1978) proposed principles of ethical consideration, including informed consent and privacy. This study considered the ethical implications based on the proposed principles. The principle of harm to participants ensures that participants must be protected from any physical and emotional harm, which could affect

their well-being. For this study, measures were applied to ensure that participants were not harmed: participants' identities were protected and not exposed while reporting the research findings. Participant information, such as grades and exam scripts, was kept confidential and stored in the Department of Computer Science, and digital information was stored on the department's secure server.

Cohen et al. (2003, p. 77) outlined the importance of participant consent in social research, as participants could be exposed to an unpleasant experience during the research, such as stress or violation of privacy. Informed consent must provide a fair explanation of the research and possible consequences, allowing participants to have the choice to participate or not. In addition, participants have the right to withdraw at any stage of the study without any obligation or pressure.

In this research, ethical approval was obtained through the Humanities & Social Sciences Research Ethics Committee (HSSREC) before conducting data collection. In addition, consent was obtained through the Department of Computer Science for collecting emails, grades, exam scripts, and project Witter in which students already consented for the department to collect their information. Interview participants were invited by email to participate in the study and consent forms were explained and collected beforehand.

The informed consent form can be found in Appendix C. Participants were informed of the right to withdraw from the study at any time without any obligation. Participants were aware the interview was audio recorded. As noted earlier, in response to difficulties in recruiting for the pilot phase, participants were aware of and provided incentives of £10 Amazon vouchers and £100 Amazon vouchers for a prize draw. As interviews were an essential component of data collection for this study design, which requires in-depth data, there was a need to engage participants through incentives.

Another important ethical principle is privacy, as personal records and collected data must be confidential and used only by the researcher. The collected numerical grade data from 1996 to 2015 were anonymised beforehand by the department so student identities were not exposed to the researcher. However, interview participants

from the 2014/15 cohort were selected based on grades and their identities were revealed by the department so the researcher could contact them.

As participants were not anonymous, the researcher ensured that participants' identities and collected data were confidential, using techniques suggested by Cohen et al. (2011, p. 92). These techniques, which can be used to report research findings to the public, required deleting any sensitive information, such as names, addresses, and any possible traceable information. However, no confidential information was used in this research, as pseudonyms and case IDs were used for analysis and reporting. A list of participant names and student IDs was stored confidentially to track participants, for example, in case of withdrawal.

Summary

In summary, the methodologies selected to address the two main research questions were the quantitative methodology and case-study methodology. The quantitative methodology was a clear choice to address the correlation between programming and mathematics based on student performance in two programming modules and two mathematics modules. In addition, the correlations between these modules were investigated. A case-study methodology was utilised to tackle the second research question and its sub-questions, allowing for quantitative and qualitative data to be gathered to understand specific phenomena within that particular context. As noted, a case-study approach was chosen, rather than ethnographic or phenomenological approaches, as this study did not focus on behavioural aspects or single unique phenomena. Rather, we explored multiple characteristics regarding gifted programmers in a specific educational context.

CHAPTER 4. Correlation between mathematics and programming

Mathematical abilities have been an important foundation for CS. In fact, many computer scientists may obtain their first degree in mathematics. Moreover, many CS degrees require mathematics modules as a prerequisite for certain programming modules.

Although the literature indicates that mathematical ability affects student development in CS in general (Kurland et al., 1986), relationships between certain mathematical concepts and programming modules have not been obvious for CS students. For example, the role of calculus in studying introductory programming may not be clear to students, whereas mathematical concepts, such as logic and algebra, can be helpful in learning the basics of programming (Henderson & Stavely, 2014).

Investigating the relationship between mathematics and programming was the aim to answer RQ1 (To what extent does mathematical ability correlate with programming in general?), as the literature suggests that mathematical ability is an important characteristic in becoming a computer scientist. Programming, which involves aspects of analysis, algorithm design, coding, and testing, is one aspect of CS in which mathematical ability may be required. Thus, we aimed to test our hypotheses on the relationship between mathematics and programming.

Compared to previous research, which lacked statistical significance and has methodological limitations, a large raw dataset consisting of 3,060 student records that were analysed provided statistical weight to our results. Moreover, clear steps of numerical analysis were followed, providing rigorous methods that can be replicated to produce the same results.

To remind the reader about our hypotheses derived from the first research question and its sub-questions:

1. Mathematical ability correlates with programming ability in general;

2. Discrete mathematics grades correlate with programming module grades;

3. Calculus grades do not correlate with programming module grades.

We should address an important point related to measuring the mathematical and programming abilities of students. The measurement instruments were based on the academic performance of students, which involved sets of coursework and exam grades that comprise the final grade. We applied certain measures to maximise the validity of the measurement instruments, which are discussed in the analysis section 4.1.1.

This chapter includes three sections. The first discusses the results of investigating the relationship between mathematics and programming based on student academic performance, including discussion of numerical analysis procedures. The second section presents the results of investigating the relationship between DM and programming modules, and the third section discusses the relationship between calculus and programming modules.

4.1 Programming and Mathematics Correlation

This section presents detailed numerical analyses, procedures, and results to investigate the correlation between mathematics and programming based on student performance. The raw data consist of 3,060 student records of grades during the period from 1996 to 2015. The records consist of two programming modules and two mathematics modules, which were all core modules for first-year students seeking various degrees, provided by the Department of Computer Science at Warwick University.

The first module, Programming for Computer Scientists CS118, used first Pascal then Java since 2000 to introduce programming fundamentals. The second program-

ming module, Design of Information Structures CS126, involved data structures, algorithms, and advanced material, requiring CS118 as a prerequisite. For both modules, grades consist of examined and assessed components of a large programming assignment, contributing 40% of the total grade. In some cases, the split between examined and assessed components was not recorded, resulting in a combined final grade.

Mathematics modules, as conceived from 1996 to 2005, were Mathematics for Computer Scientists CS124 and Discrete Mathematics CS127. In 2006, as replacement modules, Mathematics for Computer Scientists 1 CS130 and Mathematics for Computer Scientists 2 CS131 were introduced. Moreover, CS130 involved the basics of mathematical terminology, formal definitions, logic, set theory, graph theory, and probability, while CS131 covered topics such as linear algebra and calculus, with calculus including limits, continuity, differentiable functions, differentiation of inverse functions, integration, and logarithms.

4.1.1 Analysis

The Pearson product-moment correlation coefficient (PPMCC) is a statistical test of the strength of a linear relationship between two variables. The PPMCC correlation value, r, ranges from -1.0 to 1.0, representing a negative or positive linear relationship. If $r=0$, it indicates no relationship between the variables. As r approaches +1, it indicates a stronger positive relationship, whereas as r approaches -1, it indicates a stronger negative relationship. Figure 4.1 shows the PPMCC formula. To verbally describe the strength of the correlation, a guide introduced by Evans (1996) suggested the following description:

- 0.00 to 0.19 ‘very weak’;
- 0.20 to 0.39 ‘weak’;
- 0.40 to 0.59 ‘moderate’;
- 0.60 to 0.79 ‘strong’;

- 0.80 to 1.0 'very strong'.

$$r = \frac{n(\sum xy) - (\sum x)(\sum y)}{\sqrt{[n\sum x^2 - (\sum x)^2][n\sum y^2 - (\sum y)^2]}}$$

Where:

- r denotes the Pearson correlation coefficient;
- n denotes the sample size;
- x denotes the programming averages;
- y denotes the mathematics averages;
- $\sum xy$ denotes the sum of the products of x and y;
- $\sum x$ denotes the sum of x;
- $\sum y$ denotes the sum of y;
- $\sum \sqrt{x}$ denotes the sum of the square of x;
- $\sum \sqrt{y}$ denotes the sum of the square of y.

Figure 4.1: Pearson product-moment correlation coefficient (PPMCC) formula.

Instead of investigating any correlation based on individual modules, we aimed to find the correlation between the average of the two programming modules, CS118 and CS126, and the average of the two mathematics modules, CS124 or CS130 and CS127 or CS131. Investigating the correlation based on average student performance could improve the validity of the correlation result. To ensure the data were valid, we tested that each student had recorded grades for all programming and mathematics modules using RStudio to run a command-based code to remove null values for any modules. Thus, the total observations are reduced to 1,912.

4.1.1.1 Pearson Product-Moment Correlation Coefficient (PPMCC) Assumptions

Before we ran the PPMCC statistical test, four assumptions were required to be fulfilled. The first assumed that variables are continuous and can be measured either at the interval or ratio level. Here, the two variables were the averages of student program-

ming grades and the averages of student mathematics grades, which were continuous and measurable.

The second assumption was that the two variables must possess a linear relationship that can be proved by creating a scatterplot and inspecting it visually. Figure 4.2 presents the scatterplot for the two variables and shows a positive moderate linear relationship.

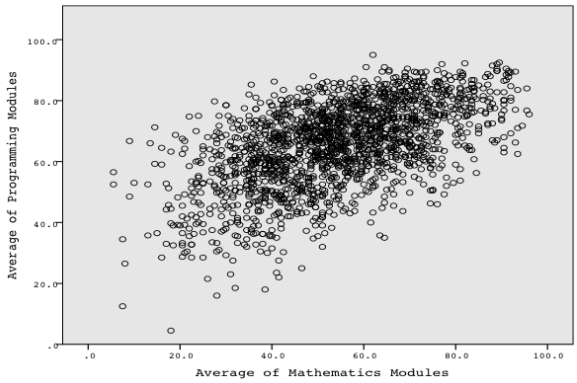


Figure 4.2: Scatterplot for mathematics and programming showing a positive linear relationship.

The third assumption was that there should be no significant outliers data that do not follow the pattern of other data points. To identify outliers, we used interquartile ranges to determine lower and upper bounds, as shown in Table 4.1. Data that were not within the boundaries were considered outliers. Visual inspection of a boxplot of the data in Figure 4.3 for the programming average suggested a number of student outliers, with an average below 31, as shown in Table 4.2. Figure 4.4 indicated that there were no outliers in mathematics averages.

	Lower bound	Upper bound
Programming	31	102
Mathematics	5	103

Table 4.1: Outliers using IQR.

In this study, there were two options for dealing with outliers. First, one option was to apply a trimming method to remove outliers caused by possible typographical and/or measurement errors. In this instance, outliers were checked against the raw

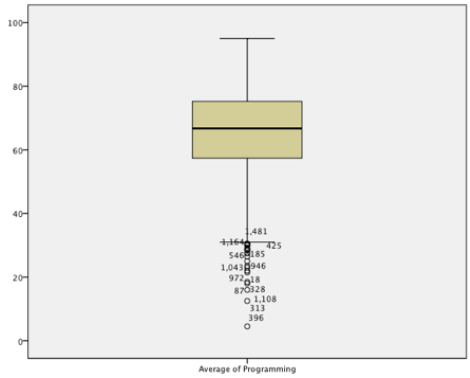


Figure 4.3: Outliers in programming.

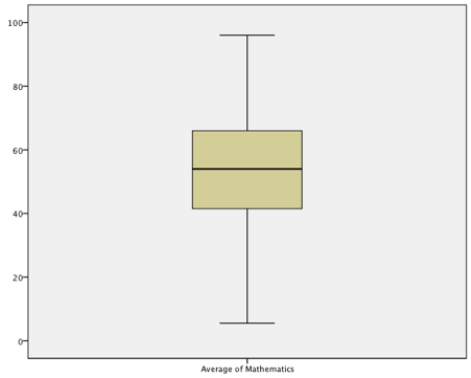


Figure 4.4: Boxplot for Mathematics.

	Anonymous ID	Programming	Mathematics
1	1021	22	26
2	1118	18	39
3	1147	31	22
4	1159	29	23
5	1256	25	47
6	1287	28	42
7	1289	27	8
8	1293	28	32
9	1420	13	8
10	1445	19	32
11	1529	5	18
12	1572	30	39
13	1803	29	30

Table 4.2: Outliers marks in the programming modules.

data, and no data entry errors were found. The second option was to modify (winsorise) the data located on both tails of the distribution to the next highest and/or lowest values within the distribution that were not believed to be outliers.

In our research context, it was realistic to have a number of students who were underachieving due to various factors; thus, we believe that those students' marks were not outliers, and they should therefore not be removed from the dataset. In addition, by removing the small sample of students, we believed that the correlation result may not have a significant effect.

The fourth assumption that needed to be addressed was to find whether variables were normally distributed, either using numerical tests or graphical methods. Numerical tests, such as the Shapiro-Wilk and the Kolmogorov-Smirnov tests, indicate

normality for small samples (Razali & Wah, 2011), whereas large sample sizes can be inspected visually for normal distribution using histogram and Q-Q plots. Figure 4.5 shows that the programming average had an imperfect normal distribution and unusual observations on the left tail, which could be outliers. In addition, the Q-Q plot in Figure 4.6 showed that most of the data points were located along a straight line, indicating that the data were normally distributed. In contrast, Figure 4.8 indicated that the average for mathematics modules had a much better normal bill-shaped distribution than the programming modules in Figure 4.7.

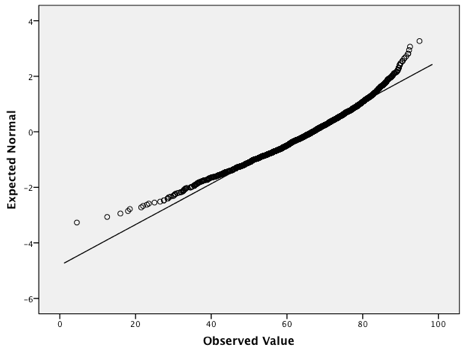


Figure 4.5: Normal Q-Q plot of average of programming modules.

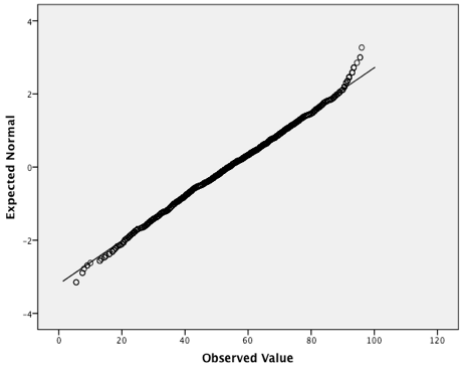


Figure 4.6: Normal Q-Q plot of average of mathematics modules.

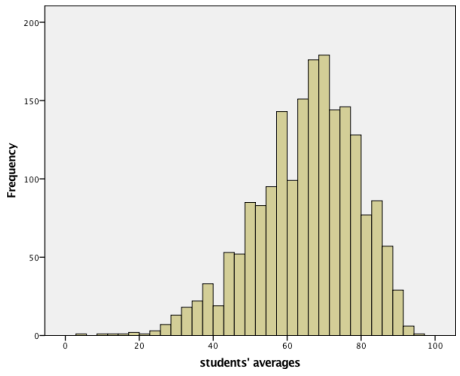


Figure 4.7: Distribution of student averages for programming modules.

.

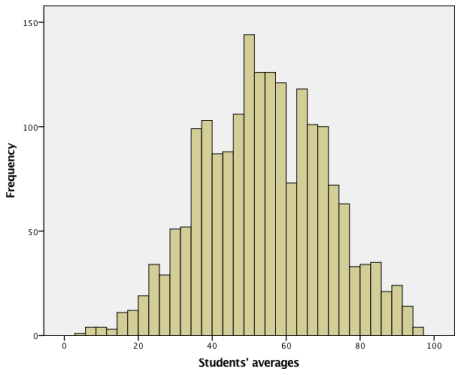


Figure 4.8: Distribution of student averages for mathematics modules.

4.1.2 Results

After investigating all the assumptions that need to be fulfilled, the PPMCC test indicated there was a moderately positive correlation between mathematics and programming, where *r=0.553*, as shown in Table 4.3. Table 4.4 shows descriptive statistics for the sample, whereas Table 4.5 presents student degrees included in the sample.

Programming	Mathematics	N	Correlation
		1912	0.553**

Table 4.3: Correlation between programming and mathematics.
** Correlation is significant to the level of 0.01.

	N	Min	Max	Mean	Std. Dev
Programming	1912	5	95	65	13.78
Mathematics	1912	5	96	54	16.84

Table 4.4: Descriptive statistics for student averages for programming and mathematics modules.

Statistical procedures for testing research hypotheses imply that the appropriate sta-

Degree		N
CS	Computer Science	1848
CSB	Computer and Business Studies	2
CSM	Computer and Management Sciences	57
CSE	Computer Systems Engineering	1
PwC	Physics with Computing	2
MwC	Mathematics with Computing	2
	Total	1912

Table 4.5: Student degrees.

tistical test should be run to either prove a researcher's alternative hypothesis *H1* or to reject the null hypothesis *H0*, which usually oppresses the researcher's hypothesis. As mentioned at the beginning of this chapter, hypotheses derived from the first research question and its sub-questions were tested. First, the alternative hypothesis *H1* was that mathematical ability correlates with programming ability in general, whereas the null hypothesis *H0* would indicate no correlation.

Determining the significance of the relationship can be investigated through hypothesis testing, based on the observed significance or p-value. The p-value determines the possibility of a null hypothesis to be true, whereas a low p-value indicates strong evidence to reject the null hypothesis (Isotalo, 2014). Therefore, if the p-value is below a significant level, a, then a decision can be made to reject the null hypothesis, but if the p-value is greater than the significance level, the null hypothesis is not rejected.

In this instance, based on the given data where $r=0.553$, $p=0.00001$, which is less than the significant level of $a=0.01$; therefore, we had strong evidence to reject the null hypothesis and to accept the alternative hypothesis. Thus, the correlation value is statistically significant at the 1% significance level.

As the correlation was based on student performance across multiple cohorts, it could be argued that a too low or too high performance could affect the correlation result. The performance variation between cohorts relates to multiple factors, including pedagogy and an individual's abilities. Therefore, a certain degree of performance variation could be acceptable in the context of education. However, we measured the correlation based on two important aspects: student averages for two programming and mathematics modules and a large sample that could mitigate abnormality in student performance. In addition, investigating each cohort's average and standard deviation for programming and mathematics grades indicated no significant variation between cohort performance. Therefore, the correlation value was not significantly affected by cohort performance variations.

4.2 Discrete Mathematics and Programming

Aiming to move from the general correlation between mathematics and programming, this section now presents results of the correlation investigation between different modules to determine whether one programming module correlated with a particular mathematics module. Two analyses were conducted. First, analysis aimed to investigate the correlation between DM module grades (CS127 and CS130) and introductory programming grades (CS118). The second analysis investigated the correlation between

DM module grades (CS127 and CS130) and data structure module grades (CS126).

4.2.1 Discrete Mathematics and Introductory Programming

4.2.1.1 Correlation between CS127 and CS118

The raw data included null values; thus, the total observations of 1,481 were reduced to 1,456 by removing missing values. A total of 888 students had separate marks for the exam and the assessment, as no combined final grades were registered. Therefore, data were processed to produce the final grades based on the mark weight of the modules: the CS118 final grade consisted of 40% coursework and 60% final exam, while the CS126 course consisted of 50% coursework and 50% final exam.

4.2.1.2 Correlation between CS130 and CS118

The raw data consisted of 682 observations that included missing values and duplicated entries. All null and duplicated values were removed and the observations were reduced to 668 records.

4.2.2 Discrete Mathematics and Data Structure

4.2.2.1 Correlation between CS127 and CS126

Initially, the data included 1,481 student records that had null values; thus, the cleaned data consisted of 1,433 observations, where 744 marks were combined and 689 students had marks for both the exam and the assessment.

4.2.2.2 Correlation between CS130 and CS126

The raw data included 682 records and some missing values. Thus, the cleaned data consisted of 643 students with no combined marks.

4.2.3 Results

The PPMCC test was run after validating the assumptions noted above, in which all assumptions were fulfilled. Results shown in Table 4.6 revealed that grades for both DM modules were moderately correlated positively with the introductory programming module grades with *r=0.52* for CS127 and *r=0.57* for CS130. Similarly, Table 4.7 shows that grades for both DM modules had a moderate relationship with the data structure programming module grades with *r=0.54* for CS127 and *r=0.55* for CS130. Given the correlation value was statistically significant at *a=0.01*, we had strong evidence to accept the second hypothesis, indicating that DM grades correlated with programming module grades.

Intro Programming	Discrete Math	N	Correlation
CS118	CS127	1456	0.52**
CS118	CS130	668	0.57**

Table 4.6: Correlation between introductory programming and discrete mathematics.
** Correlation is significant to the level of 0.01.

Data Structure	Discrete Math	N	Correlation
CS126	CS127	1433	0.54**
CS126	CS130	643	0.55**

Table 4.7: Correlation between data structure and discrete mathematics.
** Correlation is significant to the level of 0.01.

4.3 Calculus and Programming

This section presents the correlation analyses for investigating the relationship between calculus modules (CS124 and CS131) and programming modules (CS118 and CS126). First, we investigated the role of calculus in introductory programming. Second, the relationship between calculus and the data structure module was examined.

4.3.1 Calculus and Introductory Programming

4.3.1.1 Correlation between CS124 and CS118

The raw data had missing values, either for CS118 or for CS124. Records were removed and observations were reduced from 1,480 to 1,259. A total of 829 students had marks for the examined and assessed components, and 430 observations had only combined marks with no recorded split between the exam and the assessment.

We investigated the four assumptions mentioned above prior to assessing the correlation. First, the data were determined to be continuous and measurable variables. Second, the linear relationship between variables was investigated via a visual inspection of the scatterplot in Figure 4.9, which showed a weak relationship. As we assumed that the data had no outliers in our context, the assumptions that there were no significant outliers and that the data were normally distributed were fulfilled, as we were dealing with a subset of the total data.

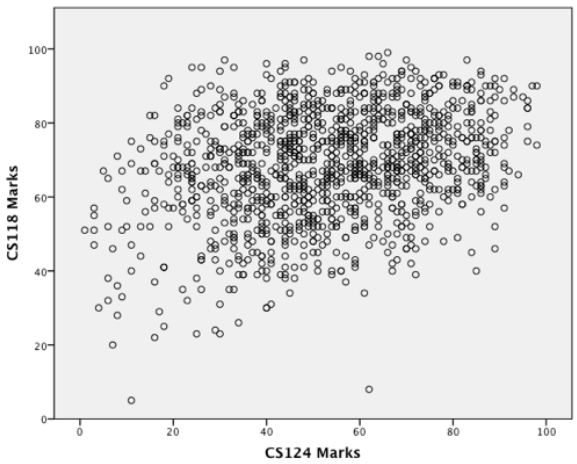


Figure 4.9: Linear relationship between CS118 and CS124.

4.3.1.2 Correlation between CS131 and CS118

This analysis aimed to find whether CS118 was correlated with CS131, which had been taught since 2006. The raw data had 682 records and, after cleaning for null values, a total of 642 observations were produced.

4.3.2 Calculus and Data Structure

4.3.2.1 Correlation between CS124 and CS126

The raw data had 1480 records that had missing values. The total number of observations after cleaning was 1237, consisting of 578 combined marks and 659 exam and assessment marks.

4.3.2.2 Correlation between CS126 and CS131

The initial data included 748 records that consisted only of exam and assessment marks and did not include combined marks. After removing null entries, the total number of observations was 642.

4.3.3 Results

The PPMCC was conducted to examine whether there was a relationship between programming module CS118 and calculus mathematics module CS124. The results revealed a weak and positive relationship, as shown in Table 4.8. In contrast, CS118 had a moderate relationship with CS124. Table 4.9 shows that calculus had a greater correlation with the data structure than with the introductory programming module. This could be explained by both modules having similar topics; for example, data structure modules included graph theory, which could be similar in mathematics modules. As our third hypothesis assumed that there is no relationship between calculus grades and programming grades, the results indicated that a positive weak-to-moderate relationship exists. Therefore, we had evidence to reject our hypothesis.

Intro Programming	Calculus	N	Correlation
CS118	CS124	1259	0.29**
CS118	CS131	642	0.36**

Table 4.8: Correlation between introductory programming and calculus.

** Correlation is significant to the level of 0.01.

Data Structure	Calculus	N	Correlation
CS126	CS124	1237	0.431**
CS126	CS131	624	0.433**

Table 4.9: Correlation between data structure and calculus.

** Correlation is significant to the level of 0.01.

Summary

The statistical analysis of student marks indicated that programming and mathematics had a positive moderate correlation that did not suggest any causation. The analysis was derived from a large sample of student grades and from a variety of CS and mathematics modules in which the results could be more accurate. In addition, these results suggested that DM grades have a moderate positive relationship with introductory programming and data structure grades. Our hypothesis assumed no correlation between calculus and programming, but the statistical analyses indicated the opposite; thus, the hypothesis was rejected and there is a positive correlation. In the educational context, in which multiple factors are involved, it is difficult to draw definitive conclusions from statistical analyses. Thus, the next chapter presents qualitative data to support these statistical analyses and to understand the phenomenon within an educational context. A discussion of quantitative and qualitative findings of the relationship between mathematics and programming will be presented later in the discussion chapter.

CHAPTER 5. Data Analyses and Findings

This chapter presents the findings of the participant data analyses. The data were collected for each participant through interview, Witter, code-writing problems, and grades and were analysed separately. The findings were then incorporated and presented together. As mentioned in the methodology chapter, numerous characteristics of gifted student programmers were investigated. Thus, all data on each participant are presented here to construct a complete picture of each gifted student to improve our understanding of multiple characteristics. As each participant was a unique case, we report each participant's perspective.

In this chapter, data analysis and findings for the participants are discussed in nine sections, with each section covering multiple themes: early education, mathematics and programming, attitudes and personal traits, coding strategy, mental representation strategies, and Witter. In the tenth section, the participants' cross-case analyses are presented.

5.1 Complete Data Analyses for David

5.1.1 Early Education

David was a second-year CS student who was identified as high-achieving among the cohort based on his first-year grades in programming modules. David attended state schools for his early education in one of the Eastern European countries. Although David wanted to study natural science, he could not because the limited high school class capacity made admission too competitive. He was placed in the mathematics and CS classes instead. David said, 'I didn't know what computer science meant'.

David achieved the highest grades among CS participants in both mathematics modules CS130 and CS131 as shown in Figure 5.1. In addition, Figure 5.2 shows David's performance in mathematics modules compared to the class average.

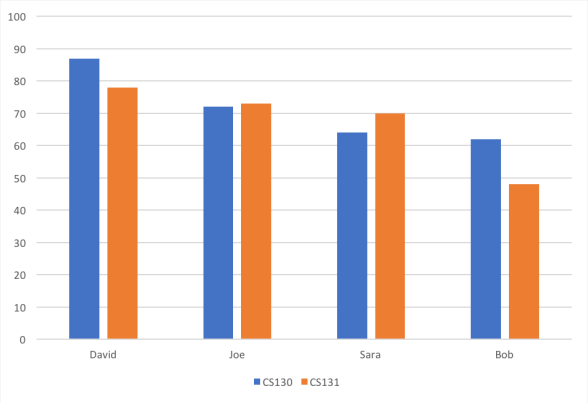


Figure 5.1: Participant mathematics module grades.

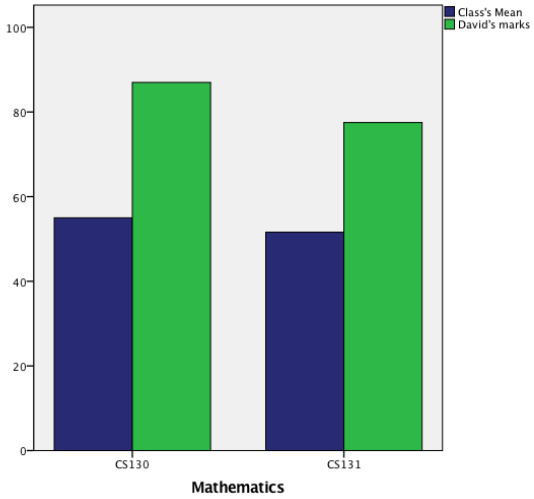


Figure 5.2: David's mathematics module grades compared to the class mean.

David's performance in both programming module assessments were above the class average, as shown in Figure 5.3 in which his overall grades for both modules were higher than the class average as shown in Figure 5.4. In addition, David's programming module average was the highest compared to other participants, as shown in Figure 5.5.

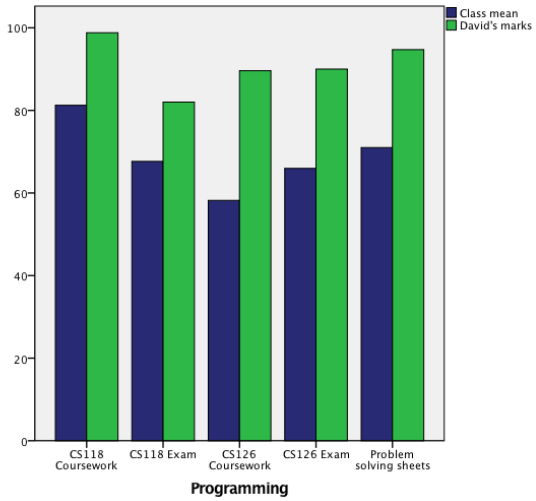


Figure 5.3: David's grades in programming assessments.

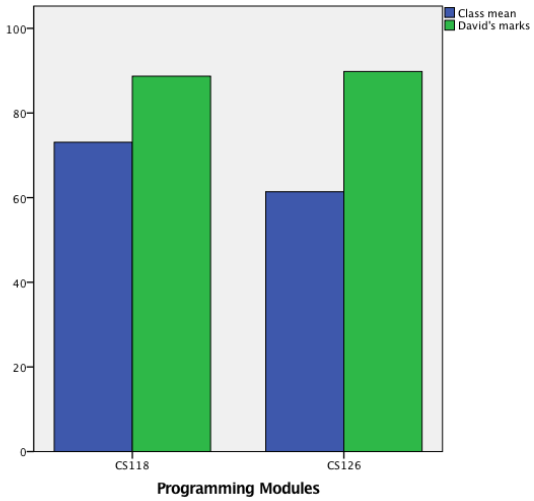


Figure 5.4: David's overall grades for programming modules.

David was exposed to a wide range of knowledge during his early education, especially

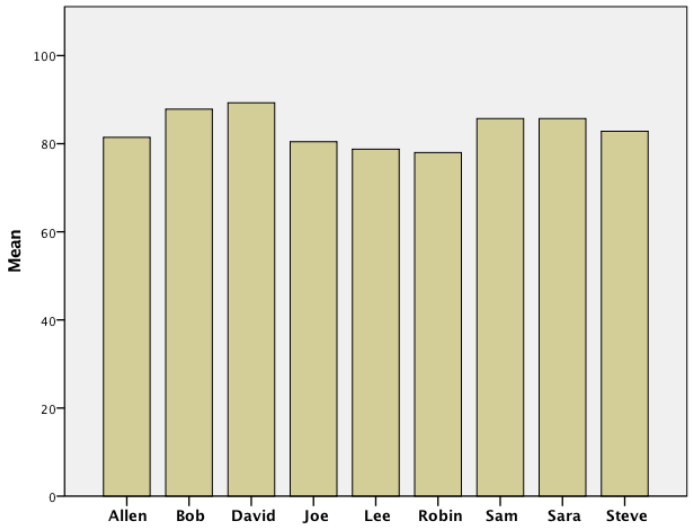


Figure 5.5: Participant programming module averages.

more in-depth knowledge in CS and mathematics, where the education system may have allowed him to maximise his potential.

Besides sports, David's hobbies included improving and comparing performance between different algorithms. He was enthusiastic about this, explaining differences between sorting algorithms. In addition, he spent his spare time solving different problems related to algorithms and mathematics, which helped him prepare for the Association for Computing Machinery (ACM) International Collegiate Programming Contest (ICPC).

5.1.2 Mathematics and Programming

David stated that having a good mathematical background promoted stronger abstract thinking, which helped him to understand and apply certain algorithms. He explained in detail how it was very important to him that a chosen algorithm for a problem was proven to perform well for unexpected cases to avoid costly time and space complexity solutions. In addition, he gave an example in which working in computer architecture required dealing with low-level coding that, in turn, required abstract thinking and a mathematical background.

David believed that different programming problems required certain mathe-

matical concepts in which the relationship between programming and mathematics depends on the context. He gave an example in which working on computer graphics required deep knowledge of calculus (i.e., matrices) that allowed a programmer to manipulate graphics pixels.

5.1.3 Attitudes and Personal Traits

5.1.3.1 Learning Style

David's learning style was fluid, depending on what he was trying to learn. If he decided to learn a new programming language, the best way for him was to follow some tutorials and to practise coding. For exam revision, he would study course materials and write down the most important part of the information. He believed both processes helped him to think, analyse, and remember important topics.

When David was asked if he preferred to work in pairs or alone during lab sessions, he cited two different experiences. He said, 'It was very nice working with others' but sometimes working in pairs might be disadvantageous for both students. In some cases, student abilities are different, and it could end up that the good student solves all the tasks without the contribution of other student, where both should equally work together. During the lab, if David struggled at one point, he would rather spend time trying to fix an error by himself than ask for help from a tutor.

5.1.3.2 Challenging Tasks

David preferred to solve challenging tasks rather than straightforward ones. David participated in many programming and mathematics contests, as most of the contest tasks were challenging. David would prefer to solve a range of tasks with different levels of difficulty rather than solving tasks that are all easy, 'more challenging tasks are usually nice'. Moreover, David would have liked to have extra tasks outside of the curriculum that he could do in his own time, including some research activities, which would have helped him gain research experience.

5.1.3.3 Communication Skills

When David was asked to assess his communication skills, he noted some presentations during his university studies and thought there were two major factors that could affect his presentation skills. As English was David's second language, he did not feel confident speaking in public, whereas he used to be the student vice president at his school in Romania for two years, where he gave frequent talks. David also felt he needed to be interested in a topic and be prepared to present. However, he preferred to communicate through speaking rather than writing, as he felt instant reactions and feedback could not be gained through written communication.

5.1.4 Coding Strategies

David said that when he wanted to write a small program, he usually started with the main function and included all codes inside it, but if the program was large and/or complicated, then he would split the codes into smaller functions. Reusable blocks of codes are common to build basic functions. When David was asked if he would use or optimise others' codes, he thought that it would depend on the task. For example, using a graphic user interface that automatically produced codes to build a website is convenient. If the task was complicated and he thought that he could understand and optimise someone else's code, then he would do so.

5.1.5 Mental Representation Strategies

When discussing problem solving, David said, 'I'm interested to find a more methodical approach'. David's problem-solving strategy involved dividing a problem into smaller, solvable parts where he tried to prove each part using either contradiction or induction proofs, saying: 'This sounds like dynamic programming'.

In addition, David demonstrated his abstract thinking while describing how he represented such a software application by first understanding the purpose of the application and how it works. Then, he thought of the details of the application

structure in terms of classes and functions to understand the connection between each part of the application.

When we analysed David's responses for code-writing problems (array creation, linear search, and recursion), he manifested the ability to provide a valid and integrated solution for all problems. Table 5.1 shows that David's responses could be classified at the highest possible SOLO category for each problem. Despite the fact that David's answers included an error, which was categorised as Multistructural, his response included all the required constructs for the problem so consensus was to categorise his response as Relational.

	Array Creation	**Linear Search**	**Recursion**
Allen	Unistructural	Multistructural	Unistructural
Bob	Unistructural	Multistructural	Unistructural
David	Multistructural	Multistructural	Relational
Joe	Unistructural	Multistructural	Relational
Lee	Unistructural	Multistructural	Prestructural
Robin	Unistructural	Multistructural	Prestructural
Sam	Multistructural	Unistructural	Unistructural
Sara	Multistructural	Multistructural	Unistructural
Steve	Unistructural	Multistructural	Unistructural

Table 5.1: Participant SOLO categories for the code-writing problems.

5.1.6 Witter

When David was asked about the Witter assignment he said, 'It was really interesting and challenging'. He spent considerable time researching, analysing, and thinking to minimise complexity before coding his solution. He said this type of problem does not have one right solution and that it takes a lot of time to decide which data structure can be used: 'I can use arrays but I can do it much better if I use some special data structures'.

David achieved a high score of 90 for the Witter assignment. When we analysed David's Witter project, it was clear that his approach was very articulate and creative. The implementation of the Adelson-Velskii and Landis (AVL binary search tree (Lafore, 2017) and a hash map for each class provided a unique approach considering time and

space complexity. The AVL tree has advantages of self-balancing and sorting data storage. In addition, David manifested the ability to articulate his approach where he made assumptions of the hash map size and provided implementation based on that. This suggested that David understood the limitations of each data structure type and the circumstances of such a problem. Moreover, David calculated time and space complexity for each function, explaining all possible case scenarios. However, he mentioned that his approach considered time efficiency over space complexity.

5.1.7 Summary

David's academic performance fell in the top 1% of programming and 5% of mathematics among the cohort. David was a hard-working individual who devoted considerable time learning programming and said that his hobby was to apply different algorithms to solve challenging problems.

In addition, David possessed different problem-solving strategies. Evidence gathered from the Witter project showed that David could deal with a large-scale project where he built all classes and functions, understanding the relationships and how to implement an appropriate data structure.

5.2 Complete Data Analyses for Joe

5.2.1 Early Education

Joe went to state school for his early education then joined a state college where he was introduced to computing for the first time. During his A-level studies, Joe took mathematics and further mathematics. Joe went to mathematics classes as an extra activity during the summer and said it was important for him to practice.

Joe studied computing, which he enjoyed a lot. The computing curriculum included programming with Visual Basic and a low-level programming language, then he taught himself Java and Android while he studied databases and basic algorithms, such as quick sort and binary search.

When Joe was asked why he chose to study CS, he was clearly enthusiastic, explaining his motivations, especially when he said, 'I enjoyed problem solving'. He said the combination of solving mathematical problems, which requires logical thinking and programming skills, was the reason he studied CS.

Figure 5.6 shows that Joe's performance in the mathematics modules at university was above the class mean, putting him in the top 10% of the cohort. Joe was the second highest mathematics student among CS participants, as shown in Figure 5.1.

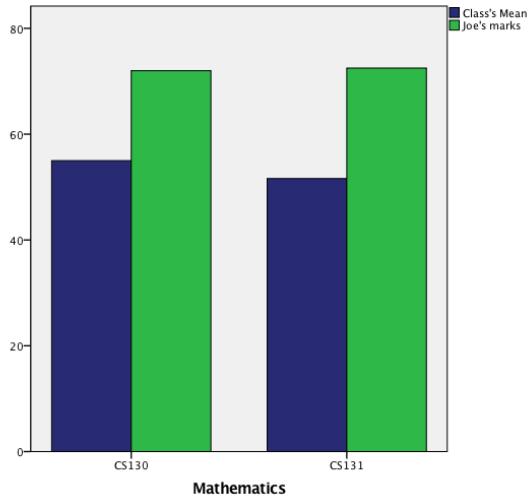


Figure 5.6: Joe's mathematics module grades compared to the class mean.

Joe's performance in the programming module assessments was above the class average except for his performance in the Witter project, where he performed below the class mean (Figure 5.7). Joe's overall grades for both programming modules were above the class average as shown in Figure 5.8. Figure 5.5 shows Joe's average compared to the participants of the programming modules.

5.2.2 Mathematics and Programming

Joe's perspective about the relationship between mathematics and programming was not about which causes the other, rather it was understanding the aspects that mathematics and programming have in common. Joe's view was that both fields shared

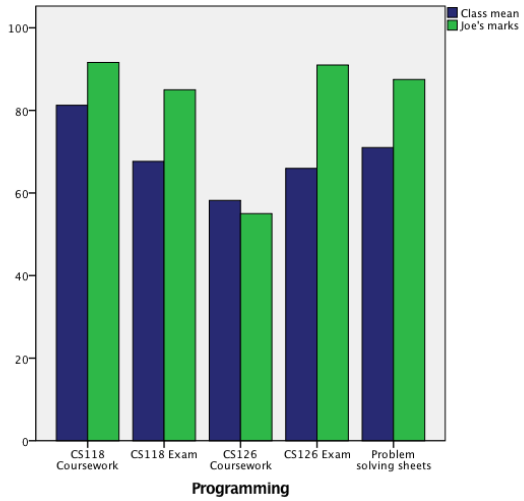


Figure 5.7: Joe's grades in programming assessments.

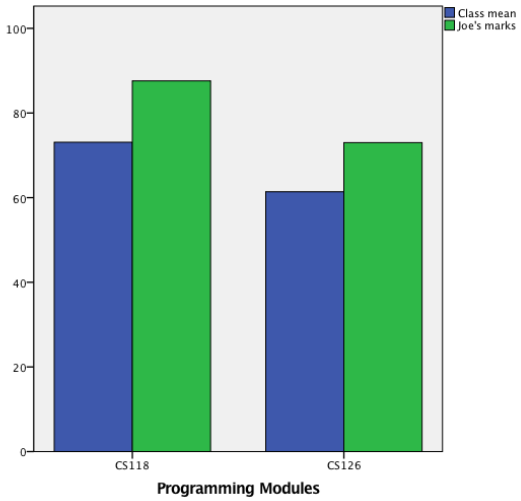


Figure 5.8: Joe's overall grades for programming modules.

similar cognitive aspects, such as logical thinking and reasoning. Thus, he felt that if someone was good at mathematics, it meant that he/she thought logically, which could then be useful in programming. Joe mentioned that when he did his first mathematics module (CS130), logic and algebra were covered. Then, when he progressed into his studies and took the algorithm module (CS126), which involved graphs, he was able to apply mathematics to programming. In addition, the relationship between relational algebra and database modules was obvious to him.

Although Joe could not understand how to apply calculus to programming, he thought that knowing and understanding calculus might make him a better programmer. He said the relationship between mathematics and programming became increasingly obvious once he progressed into his studies.

5.2.3 Attitudes and Personal Traits

5.2.3.1 Learning Style

Joe's learning style did not fit the typical protocol of attending lectures at university. Joe felt he learnt best through solving problems rather than reading a lot of material, as he found tasks that involve thinking, analysis, and coding engaged him more in learning and stimulated him to do more programming. Joe said, 'Going to seminars is more

useful', where seminars are usually a small class that involved analysing and solving problems in a more collaborative environment. During lab sessions, Joe preferred to work individually because he felt he learnt from his mistakes by fixing his code by himself and because working in pairs might make him miss something if the other student fixed his code.

5.2.3.2 Challenging Tasks

When Joe was asked whether he would like to solve challenging tasks or not, he emphasised that he loved programming specifically because of that aspect of problem solving; the harder the task was, the more engaged he would be. He said that it was not the coding process itself but the enjoyable feeling of satisfaction that came from solving a complicated problem. In addition, Joe suggested that having more tasks at the right degree of difficulty but not so complicated that they might take a lot of time to solve was something that he would enjoy.

5.2.3.3 Communication Skills

Joe spoke quickly when nervous so he could not pronounce his words very well during presentations. Despite this, he thought that his presentation skills were adequate. 'My presentation skills suffer as a result, but I find them okay'. Joe explained a situation in which he had to present his project during A-level studies. He said that it was not tedious, but rather a long process to get the audience to understand his project and that presenting was a rewarding experience. At the beginning of the interview Joe was nervous, but after some time he became more confident. Despite his speech difficulties, he preferred to communicate orally rather than in writing, as he could quickly rephrase his speech if someone did not understand him. He thought the processes of writing, editing, and rewriting took longer to communicate.

5.2.4 Coding Strategies

Joe's coding strategy was based in trial and error for assignments. He started by understanding the assignment requirements and the problem that needed to be solved. 'I try to get an initial idea as fast as possible', he said. Then, he would code his initial ideas and spend some time fixing the codes, as he felt he learnt from his mistakes. Joe liked to do code optimisation rather than rewriting a new code, which could simplify functions to print values, for example. In addition, Joe used comments on his code frequently, once he finished each section. However, evidence from the Witter project showed that Joe's code was not clear in some cases and could not always be understood. When error handling, Joe tried to find specific sections of code that he thought were problematic rather than scanning the entire code. He used print lines to track errors. In addition, Joe implemented Java exceptions to debug his Witter code. This strategy was advanced and the recommended method for identifying unexpected events when codes were compiled.

5.2.5 Mental Representation Strategies

When Joe was asked about his general problem-solving strategy, he said his approach to solving problems had changed since he studied the algorithm module (CS126). Instead of the trial-and-error approach that he used to do immediately without considering the data structure and algorithms, Joe instead spent considerable time investigating different algorithms before he solved a problem.

Joe responded that, for large programming assignments, he would break the assignment into subtasks and then prioritise the subtasks to implement the most important methods. Then, he would think about coding classes and functions and think about connections between them to keep progressing.

Joe's responses to the code-writing problem analyses showed that he understood the problem specifications and translated them into valid responses. For the first problem, array creation, Joe provided a code that was categorised at the second highest

possible SOLO category for the problem (Unistructural) as he missed some program constructs. In the second problem, linear search, Joe provided a valid response that was categorised at the highest possible SOLO category for the problem (Multistructural). For the last problem, recursion, Joe's response included all required constructs and was categorised as Relational.

5.2.6 Witter

When Joe did the Witter project, he initially spent time thinking about the appropriate algorithm. He thought that deciding the general algorithm for the assignment was important. Then, he spent some time analysing assignment functionalities and designing a graph to represent how data could be stored, retrieved, and deleted. He then considered which type of data structure he would use to build the project. Joe emphasised that time complexity was very important, and he would consider that in his project, which could be challenging in some cases.

However, Joe's performance in Witter was the lowest among the participants and was below the class average as shown in Figure 5.7. For the User Store class, Joe implemented a binary tree ordered by date, which could be efficient. However, this approach became inefficient if a new user was added as the tree would need to rebalance every 100 users, which would increase time complexity once the tree expanded. Joe's approach for the Follower class was an implementation of a two-dimensional array, which provided easy access but did not consider space complexity as the array size needed to be expanded. For the Weet class, Joe decided to implement a hash map that needed a second data structure to store and sort Weets, reducing time complexity. In this approach, finding a specific Weet required the user to search through all Weets, which made Joe's approach slow. Thus, Joe's overall approach was not appropriate for this type of problem, considering a large number of users and Weets, which could affect time and space complexity.

5.2.7 Summary

Joe performed better in mathematics modules, being in the top 10%, and his performance in programming was above the class average, achieving in the top 28%. Joe spent considerable time studying and learning programming, preferring to devote his time to solving complex problems and trying different algorithms. Based on the Witter project evidence, Joe's approach to programming lacked an understanding of how certain data structures could affect time and space complexity.

Joe's responses to code-writing problems were not exceptional except his response to the recursion problem. Joe could understand and implement programming concepts. However, he seemed to need to understand how different data structures and algorithms might affect time and space complexity based on certain problems and considering other factors. During the interview, he mentioned that he tried to solve any problem as quickly as he could, which could sometimes affect performance, as he had to modify his approach to solving problems on many occasions.

5.3 Complete Data Analyses for Bob

5.3.1 Early Education

Bob was studying CS and was a confident and well-spoken student who volunteered to deliver talks during departmental open days. Bob attended a state school for his early education and studied a range of A-level subjects including computing with Visual Basic, which he enjoyed, and before that he studied an information and communication technology (ICT) module for his general certificate of secondary education (GCSE). Bob studied two mathematics modules for his A-levels, mathematics and further mathematics, achieving A* and A, respectively.

At university, Bob studied two mathematics modules (CS130 and CS131), achieving 60% and 48%, respectively. He said, 'They are the most difficult modules, but I see how they're quite useful'. Bob decided to pursue his academic studies in CS, as

he thought that the field was rapidly changing and growing, so there would be job prospects. For Bob, the ideal job should be enjoyable and involve programming within a tech firm or the financial sector, attracting a reasonable income. Bob's hobbies included playing musical instruments, which he said took a lot of practice.

Bob's mathematics performance was the lowest among the participants in this study as shown in Figure 5.1. Figure 5.9 indicates that Bob performed slightly above the class average in CS130 and below the average in CS131. Bob started programming

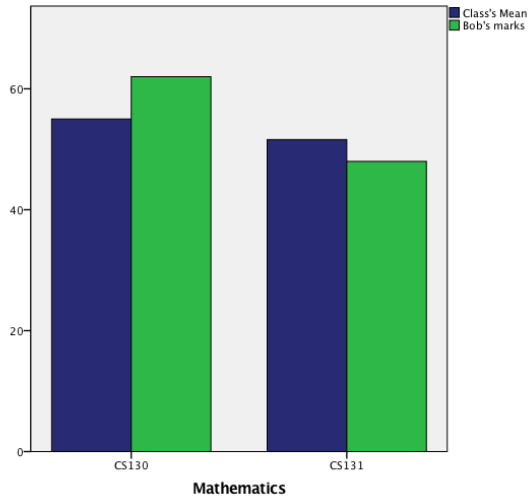


Figure 5.9: Bob's mathematics modules grades compared to the class mean.

when he was 13, experimenting with ActionScript with Flash. Then, he learnt how to build websites using HTML, PHP, and JavaScript. He learnt Visual Basic during his A-level study and Java at university when he studied CS118 and CS126. Bob achieved a high grade in CS118 in both the coursework and the final exam, whereas in CS126, he achieved a high grade in the coursework but not in the final exam. Bob enjoyed both modules and thought that his previous experience helped him with the learning, as some of the content was not new to him and some content built upon his previous learning. That might explain Bob's performance in CS126, which included advanced topics that Bob might not have been introduced to in his early education.

Apart from the CS126 exam, Bob managed to perform above the class average in all programming assessments, which makes Bob's overall grades for both programming modules above the class average, as shown in Figures 5.10 and 5.11. Moreover,

Bob's programming performance was the second highest average compared to the participants, as shown in Figure 5.5.

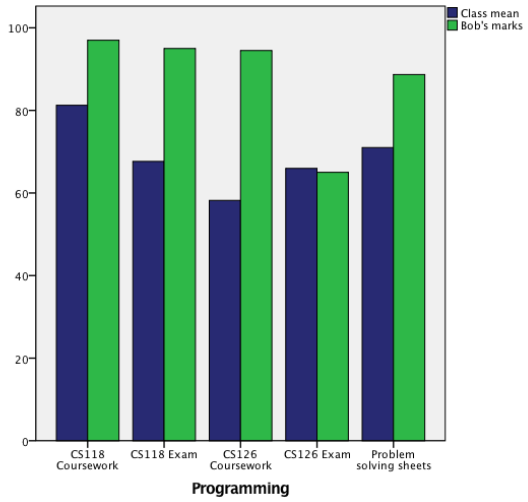


Figure 5.10: Bob's grades in programming assessments.

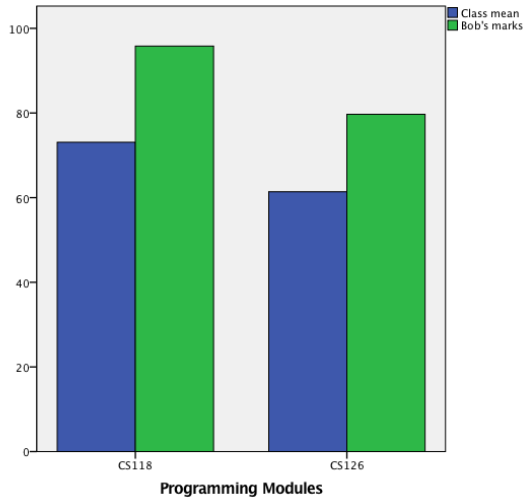


Figure 5.11: Bob's overall grades for programming modules.

5.3.2 Mathematics and Programming

When Bob was asked about the relationship between mathematics and programming, he instantly replied, 'Yeah, quite a lot' and emphasised that the more he studied mathematics, the more he understood how to apply mathematics to programming. He said, 'Logic as well comes up in maths which can be applied to computer science', that graphs can also be used in algorithms to solve such problems, and, obviously, mathematical problems in programming need mathematical knowledge to solve them. He also thought that calculus had some application in CS and 'It's not necessarily obvious what the connection is, but from my experience there is a good use for it'.

5.3.3 Attitudes and Personal Traits

5.3.3.1 Learning Style

Bob's learning style depended on trial-and-error strategies, as he usually started coding and learnt as he proceeded. He set some tasks to be achieved and tackled them one by

one, learning knowledge required to solve each task. During lab sessions, Bob thought a mixture of working alone and working in pairs was valuable, so that he could discuss a problem with his peers but would implement the solution by himself. He said, 'You get to learn specifically the things you don't know' but that having a discussion with others was not beneficial in all cases. When Bob struggled to solve a problem, he would usually try to solve it by himself, searching for a similar problem he had tackled in the past, but he would ask a colleague or a tutor if he could not solve it.

5.3.3.2 Challenging Tasks

Bob liked to be challenged by complex tasks but said the level of difficulty should be raised gradually in problem sheets, so he could learn key concepts and be confident in applying them. He said that challenging tasks needed to be included to help students build their knowledge base and boost their confidence, knowing that they can solve complicated tasks.

Bob thought that challenging tasks should be included in the curriculum and not as extra ungraded tasks that could distract students and be time-consuming. Bob liked to think in different ways when he was trying to solve a problem and 'To find solutions that aren't the obvious ones'.

5.3.3.3 Communication Skills

When Bob was asked about his communications skills, he said that he was considered a good presenter, having participated in many student activities at the university and having given many talks during open days. Thus, he felt confident speaking in public. He preferred to communicate verbally rather than in writing, as he sometimes found it hard to express his ideas through writing. In addition, he could explain and deliver complex technical ideas to his classmates through speaking.

5.3.4 Coding Strategies

Bob's strategy was to tackle a project by setting sub-goals and solving tasks accordingly. When Bob started coding a project, he would usually sort out all the elements and tasks and decide which he should solve first, solve it, then solve the second task, until he reached the end.

Bob organised his code into small functions. If he needed to use certain functions more frequently, his code was written in a linear style, where he divided the code with line breaks and comments. When dealing with programming errors, Bob thought that some programming languages were easier to debug, whereas he was constantly debugging his. Bob usually represented a certain application based on its functionality, where complex application included many functions. Bob considered complexity an important factor for large-scale software development. He gave an example of how avoiding declaring unnecessary variables and nesting too many loops could improve time and space complexity.

5.3.5 Mental Representation Strategies

Results from the code-writing problems indicated that Bob understood problem specifications and program constructs for the array creation problem. However, Bob's responses for this problem showed that direct translations of the problem specifications resulted in an invalid solution. The solution missed one program construct (i.e., generating exclusive random values) so his response was categorised as Unistructural. Bob provided a valid response for the linear search problem, and his response was categorised as Multistructural.

Bob's response for the recursion problem provided direct translations for the problem specification and resulted in invalid code. The code was invalid because important program constructs were missed, including recursive invocation and difference calculation. Thus, Bob's response for the recursive problem was categorised as Unistructural.

5.3.6 Witter

Bob's performance in Witter shows that he exhibited the ability to understand and implement data structure and algorithms considering time and space complexity. Bob's overall grade for Witter was 95. Bob's approach for the User Store and Weet classes consisted of two data structures providing good time over space complexity. The approach was a combination of a hash map and binary tree, which allowed data to be sorted, as most of the functions required data to be in order so that time complexity would be reduced. However, as the worst-case time complexity for searching in a tree depends on the tree height, if there are too many nodes in one side of the tree, time complexity is O(h). Thus, a self-balancing binary tree would make Bob's approach more efficient. Implementation for the Follower Store class was based on an adjacency matrix storing two directions of followers and follows, which provided good time complexity for accessing elements, whereas space complexity can be affected.

5.3.7 Summary

Bob's academic performance in programming was in the top 5% of his peers. However, Bob's mathematics average was in the 55^{th} percentile in which his performance in mathematics was only slightly above the class average. In comparison with the participants, Bob's performance in mathematics was the lowest; nevertheless, Bob achieved high grades in programming. Bob maintained the high academic performance exhibited through his early education and in higher education, which could not be achieved without devoting a lot of time and effort. In addition, Bob was motivated by external factors, in this case a successful career, which could lead him to dedicate more time and effort to achieve his goal.

As Bob said that he was always trying creative solutions, that was evidenced by his approach to the Witter assignment, which included a combination of different data structures and algorithms compared with results from the code-writing problems, which did not indicate any sign of originality. It seemed that Bob's performance in exams

was less successful than his performance in other assessments. This could suggest that including an open-ended project is a possible way of identifying creativity in programming.

5.4 Complete Data Analyses for Sara

5.4.1 Early Education

Sara attended grammar schools for her GCSE and sixth-form studies. At the age of 13, she started programming with Python but said she ended up being self-taught, as she thought that her teacher did not know much about Python, so the curriculum was a mix of unrelated topics, such as algebra and finite state automata. Sara chose to study CS to pursue her early interests in programming and game designing, and she expressed the view that the IT industry does not have enough women. Compared with other CS participants, Sara's performance in both mathematics modules were the second lowest, as shown in Figure 5.1, whereas she managed to perform above the class average in both CS130 and CS131, as shown in Figure 5.12.

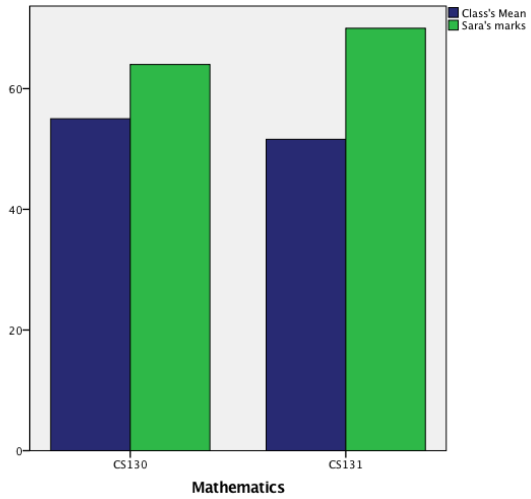


Figure 5.12: Sara's mathematics modules grades compared to the class mean.

Sara started teaching herself Java, which was difficult and slow, but she said she wanted to develop her Java skills before starting at university. It seemed that Sara had the

ability to determine her goals, and the knowledge to achieve them. She also had the ability and independence to be a self-taught learner.

Sara's performances in the programming module assessments were above the class average as shown in Figure 5.13, which resulted in achieving overall grades that were above the class average, as shown in Figure 5.14. In addition, Figure 5.5 indicates Sara's programming module average ranked fourth compared to other participants as shown in Figure 5.5.

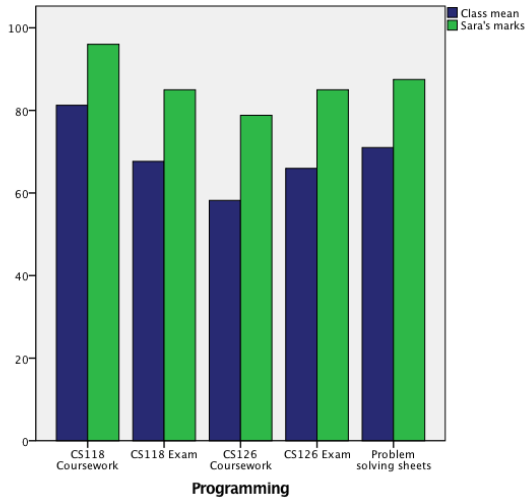


Figure 5.13: Sara's grades in programming assessments.

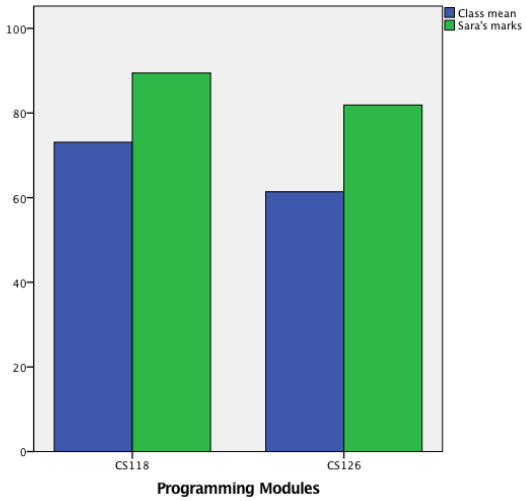


Figure 5.14: Sara's overall grades for programming modules.

When Sara expressed her feelings about programming, she emphasised that the most interesting part of programming was the feeling that she described as 'euphoric', once a complicated project was complete or an irritating bug had been eliminated. Sara said, 'It's what keeps me coming back and doing more and willing to learn more'.

5.4.2 Mathematics and Programming

When Sara was asked about the relationship between programming and mathematics, she thought that both were related to each other even though sometimes the connection was not clear. She said that, although mathematics and programming were in parallel, it was difficult sometimes while programming to distinguish whether she was thinking mathematically or logically, stating it was difficult to determine the causation.

She believed that it was not necessary to be a great mathematician to be a programmer, other than understanding mathematics basics, and in some cases a good programmer may not be good at mathematics. However, she did mention that the more she progressed in her studies, the more she could see how important mathematics could be. She gave an example of how mathematics was interlinked with programming by applying mathematics in code optimisation to split or combine loops.

5.4.3 Attitudes and Personal Traits

5.4.3.1 Learning Style

Sara was an independent self-learner and thought that learning style provided the most enriching strategy for her study. She said that despite the effort and the consumption of time, she learnt best in this way as long as she knew how the learning process worked and where the starting point was. In the lab, Sara preferred working in pairs, as social interactions, enthusiasm, and a slow-paced learning environment could help her self-taught process by discussing a problem with fellow students. 'I do believe group work is one of the best ways to learn', she said.

5.4.3.2 Challenging Tasks

Sara found solving challenging tasks more engaging and enjoyable, as she would not solve a problem that she already knew: 'I will just leave the problem'. However, in some coursework, she would invest more time to understand a challenging problem that kept her motivated to find the answer. She mentioned that solving the first programming module problem sheets was a problem for her, as she did not find the problems challenging or engaging at her level of ability, rather than having very slow-paced learning.

5.4.3.3 Communication Skills

When asked about her communication skills, Sara said that presenting was not something she has done much and that it made her nervous. However, she said that, during

the professional skills module when she had to do an essay and a presentation on a topic that she enjoyed and became passionate about, she found it easier to give a presentation on a topic she was confident in. Sara preferred to communicate in writing, as she could have other people to check her writing. She said, 'I think it's definitely that idea of being able to edit what you've said'.

5.4.4 Coding Strategies

When Sara started programming, she would first write and draw her ideas on a piece of paper, which would help her to understand the general structure of a program, including its functions and classes. In addition, this strategy helped her to understand the relationship between all classes, which she thought was very important in the early stages.

Sara thought that code comments were very important and avoided working with students who did not explain code, as it was time-consuming to look at messy code. She tried to comment on her code, especially for the coursework, when she thought that the written code was working correctly. For error handling, Sara usually put flags around sections that might have a problem and started investigating what could have gone wrong.

5.4.5 Mental Representation Strategies

Sara's method of solving such problems depends on their complexity. She would first assess the problem and see whether she understood what was required and whether she had the knowledge to solve it. If she did not feel confident, she would try to learn and find a way to understand the concepts behind the problem and find another similar problem that might have a subsection that could be a starting point.

Results from code-writing problem analyses indicated that Sara's responses for the first and second problems were categorised at the highest possible SOLO category, Relational and Multistructural, respectively, whereas, Sara's response for the recursion problem was categorised as Unistructural. Sara's response for the recursion method

did not include some constructs, such as checking the edge, as explained in Chapter 3.3.3.1.3, which was important for the method.

5.4.6 Witter

When Sara talked about Witter, she said that functionality was the main step, then choosing a suitable data structure to achieve a high level of efficiency. She said it was important to consider complexity: 'When I start nesting for loops, I immediately write is there another way I can do this?' as she always considered time complexity. Sara's performance in Witter showed that she understood the Witter requirements and implemented a solution that considered both time and space complexity.

Sara's approach for the User Store class implemented two data structures, the hash map and binary tree, where the approach achieved good space complexity using the hash map. Implementing the binary tree allows users to be ordered based on data and stored in a way in which retrieving specific users can be relatively fast, based on the number of users. However, a balanced tree would be more efficient in terms of achieving better time complexity for search functions, as large numbers of users might use the application. Sara explained in the preamble comments section that using a balanced tree would be more efficient but she did not implement it.

Sara implemented a similar approach to the Follower class using hash maps to store followers and follows, using a binary tree. For the Weet class, Sara also implemented a hash map and binary tree to store Weets, which were likely to be large numbers that could affect time complexity when searching for a specific Weet. Thus, time complexity for the search function could be improved by a balanced tree. Performance for some functions could be improved using a secondary data structure.

5.4.7 Summary

Sara's academic performance fell in the top 10% of programming and top 20% of mathematics among the cohort. Sara addressed the issue of the lack of women in the IT industry, and she was motivated to be successful and participate in this male-

dominated industry.

Sara implemented multiple data structures and algorithms to solve the Witter project. Her Witter approach was original in terms of implementing a combination of hash map and binary tree. Time efficiency for the search function could be improved by implementing a balanced tree, which she mentioned in her code comments, but her approach was relatively efficient.

Sara's responses for the array creation and linear search problems indicated that her codes were clear and elegant, reducing any redundancy in the code. However, Sara's response to the recursion problem showed that she understood the problem's requirements but could not provide a valid response.

5.5 Complete Data Analyses for Lee

5.5.1 Early Education

For his GCSE, Lee went to a state school that he believed was well-respected, but he discovered later that the school's Office for Standards in Education (OFSTED) report showed inadequate performance in many areas. Lee then attended a grammar school to study A-levels in mathematics, further mathematics, statistics, and music, achieving A*, A, B, and B, respectively. Lee did not study any CS subjects during his early education other than a basic GCSE ICT module. Lee said, 'I was quite interested in computer science but I just never learnt it'. Of all the participants, Lee had the least CS background, which made his case unique, as he performed above the class average in programming at university.

Lee's first exposure to programming was during his first year when he studied the CS118 module. He was worried at the beginning of the module as he was unable to understand the concept of arrays in programming; thus, he had to study hard through the material, which he said was well-organised and helpful. Then, he started understanding concepts and said he performed very well in both coursework and the final exam.

In the second programming module, CS126, Lee found most of the concepts again difficult and did not understand the data structure. Lee started understanding these concepts during lab sessions when he worked closely with other students in pairs to solve problem sheets. Lee said, 'Towards the end I felt like I was quite comfortable with this, the data structure and overall the course went fine'.

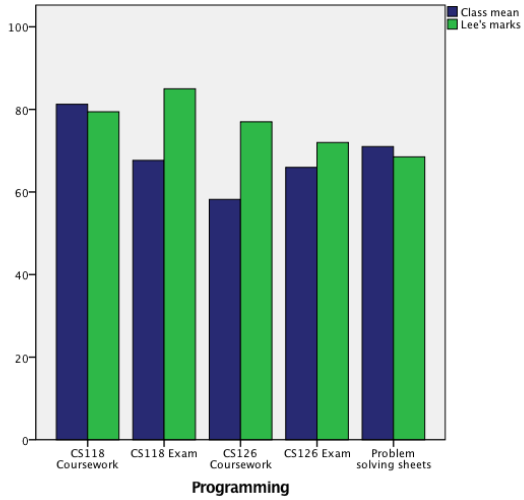


Figure 5.15: Lee's marks in programming assessments.

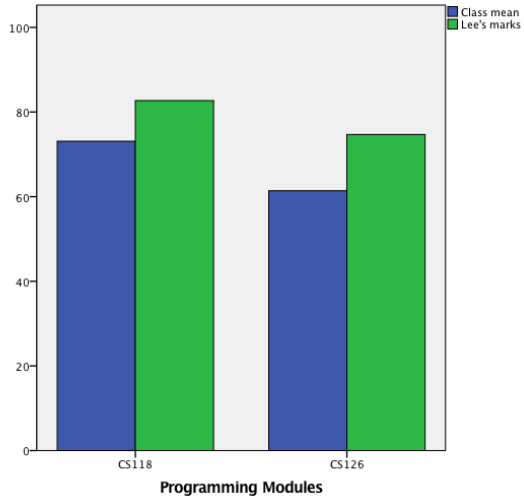


Figure 5.16: Lee's overall marks for programming modules.

Lee said that he did not do programming outside his study hours, and sometimes he felt programming was a tedious task because of error debugging, whereas other times it could be enjoyable. Lee had the ability to learn and to cope with subjects he had not been introduced to before. Not only that, he also performed well in both programming modules. Figure 5.16 indicates that Lee performed in both programming modules above the cohort average. It seemed that Lee was not passionate about programming or about being a hard-working student who would like to perform as well as he could in his studies.

5.5.2 Mathematics and Programming

When Lee was asked about the relationship between mathematics and programming, he said that during his first year he thought there was a relationship. He studied time complexity and convergence, which determines the value of functions based on certain

conditions. In addition, in CS126, he studied algorithms to prove time complexity, studying operations like *Big-O* notation.

He said that one does not need to be a great mathematician to be a good programmer. Lee said, 'I feel anyone can be good at programming, if they try hard' and that there is a relationship between both subjects, but it can be sometimes unclear. Lee was not sure about the relationship between calculus and any programming module.

5.5.3 Attitudes and Personal Traits

5.5.3.1 Learning Style

Lee's learning style relied on lecture attendance, note taking, and revision. During exam times, Lee tried to cover all topics by writing down a lot of material, but he thought this was not a good learning strategy. During lab sessions, Lee thought that working in pairs was very helpful, as he mentioned earlier that when he studied CS126, he benefited the most from his friend who was not afraid to ask questions. He thought that he learnt better in a group rather than studying alone, as discussion in pairs could bring about more ideas.

5.5.3.2 Challenging Tasks

When asked whether he would like to be challenged solving complicated problems or not, Lee said, 'I prefer challenging tasks if I know I have enough knowledge to do it'. Lee thought programming modules had enough challenging tasks and did not want to have more unless those tasks were not part of the assessment. Lee had a strategy for solving a new problem presented to him: find a similar problem that shared aspects with the new one, then look for patterns to help him solve the new problem. This was similar to how some students prefer to learn from solved examples in which some aspects relate to other problems.

5.5.3.3 Communication Skills

Lee was shy and his presentation skills were inadequate when he was younger. He started to build skills and confidence by playing instruments and performing on stage. 'That helped with my presentation skills now, although it's still lacking a bit.' In some situations, he said it was difficult for him to express a complicated topic and his thoughts on it to someone who was not familiar with that topic, as he used to draw his ideas to make them understandable to others. Lee preferred to communicate in writing more than verbally, as he thought that during the writing process he had time to explain his ideas and present them in a way that could convey his thoughts properly. He thought that because it took more time for him to deliver his thoughts, verbosity and confusion could result.

5.5.4 Coding Strategies

Lee's coding strategies depended on trial and error, where he started experimenting with some ideas until one part was done, then think about what methods and data structure could be used. Although he thought that planning, analysing, and designing algorithms and data structures on paper was considered the best practice, he planned what he wanted to code in his mind. 'I just like to come to a general plan in my head and then try to implement it.' In addition, he did not spend much time thinking about the connections between classes and methods, as he would keep implementing a part until he realised that a certain class was needed. In terms of code organisation, Lee tended not to comment on his code, which he admitted was a bad practice. Yet, he showed constant code commenting throughout the Witter assignment, perhaps as he realised his grade would be affected if he did not add comments.

Lee thought that reusing some segments of code was helpful, as he could customise sections of written code to suit certain purposes. He said that he had to use merge sort many times for the Witter assignment where he optimised merge sort for different purposes.

Lee used to ask someone else for help when he had code bugs, then he realised it

was hard for someone to understand a complicated code written by someone else. Lee then corrected his code errors by himself, using the print statement to track his code to find affected segments. Surprisingly, Lee implemented Java exceptions for handling code error in Witter.

5.5.5 Mental Representation Strategies

Lee's general strategy to solve a problem started by carefully reading the problem to understand what needed to be answered. Then, Lee tried to relate the problem to a similar problem that he had solved or to a similar problem where he could find a starting point. When Lee looked at a particular software, he was curious to know which algorithms and data structures had been used, as long as the software is not complicated.

However, when we analysed Lee's answers to code-writing problems, he provided direct translations of problem specifications for solving the array creation problem, with his answer classified as Unistructural. His answer missed one program construct that related to generalising an inclusive random value. Lee provided a valid answer for the second problem, which was categorised as Multistructural, as shown in Table 5.1. Lee's answer for the third problem, the recursive method, was categorised as Prestructural, as he showed a substantial lack of knowledge of the problem specifications. Table 5.1 shows that Lee's answer for the recursive method was categorised at the lowest SOLO category compared with other participants. In addition, Lee performed below the cohort average for the problem-solving assessment shown in Figure 5.15.

5.5.6 Witter

Despite Lee achieving a mark of 78 on the Witter assignment, he said that he could have performed better if he had managed his time better, as he started programming one week before the deadline. However, Lee said that it was not too hard to decide what type of data structure needed to be implemented as he chose the hash map. Lee understood the limitations of different data structures that could affect space and time

complexity, in which he said that using matrices would not be ideal, as more resources would be required. Lee said that applying matrices as a data structure takes much more space. He thought that trees would be more efficient, but he did not know how to implement balanced trees for Witter.

Lee provided very detailed comments at the beginning of each class, demonstrating his reasoning for making a data structure decision based on his assumptions. For example, Lee explained all possible sorting methods that could be used to sort the user ID, and he explained the time complexity for each sorting method.

Lee understood the mechanism of the hash map, how the function stored the user ID, and how to avoid hash map collision. He explained the possibilities of having full hash map buckets, where there was no empty bucket. Lee stated that if all buckets were full, increasing the size of the hash map by a larger prime number would solve the problem.

5.5.7 Summary

Lee was identified based on his academic programming performance in his first year at university. He performed above the class average in both programming modules. During the interview, it was clear that Lee was a very hard-working student who pushed himself to perform as well as possible. Lee's solution for the Witter assignment was not divergent from the norm, using a hash map as a data structure. During the interview, Lee mentioned that his solution would be better if he implemented a balanced tree, but he said that he did not know how implement an AVL tree. Communicating programming ideas is an important characteristic in programming, but Lee mentioned that was a difficult task for him.

5.6 Complete Data Analyses for Sam

5.6.1 Early Education

Sam attended a state school where he studied a wide range of AS-level and A-level subjects including mathematics, ICT, and science, in which he achieved A grades in mathematics and ICT. Sam mentioned that the only programming he did at that stage was HTML. Then, he questioned whether basic web development was considered programming or not.

Sam started programming when he was studying an ICT module, including the basics of web development languages as well as teaching himself Python. In addition, Sam did small projects that involved programming and HTML5. He said, 'Since I've got to university, it's been an explosion of programming languages'. Sam enjoyed programming, especially if a task had been described clearly to achieve tangible results.

Sam studied two programming modules. The first module was CS118 in which he performed well in the coursework, although he was not satisfied with his performance in the exam, as it was his first programming module. Although Sam thought that the Witter assignment covered many concepts and required a big effort, he performed very well in the second module CS126, in both the Witter assignment and the final exam. Figure 5.17 indicates that Sam's performance in all programming individual assessments were above the class average. Thus, Sam's overall marks in both programming modules were above the class average as shown in Figure 5.18. Sam's programming module average ranked third compared to other participants, as shown in Figure 5.5.

Sam had not been involved in any professional software development except small web development projects that he enjoyed doing in his spare time. Sam thought that Linux-based operating systems, such as Ubuntu, allowed for customising settings with more control for accessing the operating system functions, which could not be done with closed-source operating systems. Similarly, Sam liked to use an open-source IDE, as it is supposed to be a useful graphical environment that helps programmers.

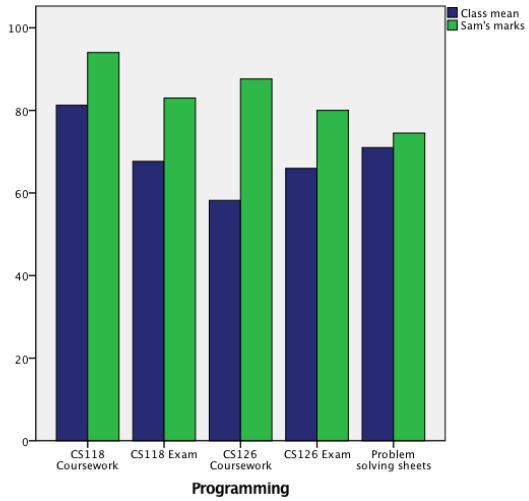


Figure 5.17: Sam's marks in programming assessments.

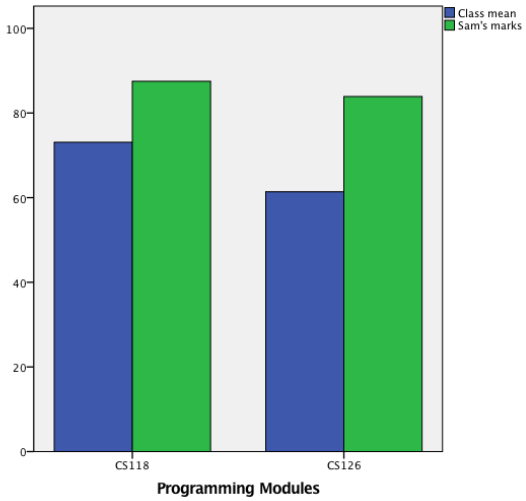


Figure 5.18: Sam's overall marks for programming modules.

5.6.2 Mathematics and Programming

When Sam was asked about his view on mathematics and programming, his answer was articulate in addressing such a complex issue. Sam thought there was a relationship between mathematics and programming and gave an example of how the discrete side of mathematics related to the practical side of CS. He thought there were shared theoretical underpinnings across both disciplines but he could not identify those aspects.

Sam explained how he implemented set theory to solve a programming task to generate prime numbers. His mathematical knowledge of set theory and prime numbers helped him solve the task by combining those concepts with a programming algorithm. Thus, Sam implemented a recursion algorithm to generate prime numbers based on his mathematical knowledge of related concepts. Sam thought that logical reasoning was the key aspect that both disciplines shared, 'It's more the link to mathematics and logical reasoning than mathematics itself'.

5.6.3 Attitudes and Personal Traits

5.6.3.1 Learning Style

Sam's learning style depended on finding examples that illustrated what Sam wanted to learn. He applied this style of learning for mathematical proofs, but it was sometimes difficult to understand why proofs worked in specific contexts. Sam thought the best way to learn programming was to practise solving many problems, as different problems had different specifications, limitations, and implementations. Sam said that learning from mistakes, especially when he was doing programming coursework, was very rewarding.

Sam found working in pairs during programming lab sessions was quite useful for discussing possible or alternative solutions, but he also thought that there might be risks working with a student who knew more than him. Sam wanted to learn rather than to have some tasks solved by his colleague. Thus, Sam preferred some time to work individually when solving such a task in the lab, and then at the end he would discuss his solution with another student.

If Sam encountered code error, he would try debugging the code and explored many possibilities to correct the error. When Sam eventually needed help, he would prefer to ask another student, as he thought that tutors sometimes did not read the problem sheet.

5.6.3.2 Challenging Tasks

Sam found that challenging programming tasks were both interesting and rewarding, especially if the purpose of the tasks was to build up knowledge and not to put students under pressure by setting a very difficult task. In addition, Sam mentioned that challenging tasks should be included in the lab sheets as a way of distinguishing students who were passionate from those might or might not be able to solve those tasks. Thus, including challenging tasks that were graded could be optional.

5.6.3.3 Communication Skills

Generally, Sam could deliver presentations, especially if he was passionate about a topic, without any difficulties, but sometimes did find presentations daunting, which made him nervous. Sam said that he could explain and deliver a complex problem to other students. His friends had told him that he had a good way of explaining, as Sam was visual and liked to draw diagrams, which helped others to understand. Sam preferred to communicate in writing rather than speaking. He found that sometimes he needed to prepare before speaking to others, as he could not get straight to the point; thus, his speech was not always succinct.

5.6.4 Coding Strategies

Sam's approach to solving a programming problem started by analysing the problem requirements and identifying any specific programming language that was required to be learnt. Then, Sam divided the task into small parts, creating a skeleton for each part and solving them one by one.

In terms of code organisation, Sam mentioned that following code standards was the best programming practice, as he usually commented his code at the final stages of the project. Evidence from the Witter project indicated that Sam's code had very detailed comments, and the code was easy to follow.

Sam had a different way of handling code errors, using a web search to find the cause. Sam thought that reusing and optimising others' code could be beneficial for a large-scale project if he knew how the code worked, and it did not affect his skills. He thought that code optimisation was a very important skill that he was learning.

5.6.5 Mental Representation Strategies

Sam's approach to solving a problem started by conducting research of similar problems that had been solved. Then, he started analysing previous approaches, finding advantages and limitations that could be improved in his own approach before imple-

menting. Recently, after studying data structure and database modules, Sam thought that it was important to think about the structure behind any software application rather than considering the interface design.

Results from analysing Sam's responses for code-writing problems indicated he was able to understand problem specifications and to provide a translation that might be a valid response. For the array creation problem, Sam's response provided clear translations for the problem specifications, resulting in a valid solution that was categorised as Multistructural. Sam's response for the linear search problem, however, was unexpectedly invalid, and the response included redundancy in declaring unwanted variables; thus, the response was categorised as Unistructural. For the recursion problem, Sam's response missed important program constructs that affected his solution, which was categorised as Unistructural.

5.6.6 Witter

Sam mentioned that choosing appropriate data structures and algorithms for Witter was critical if he would like to be more creative in his approach. He said that most students implemented only a hash map, but he was intrigued by implementing trees that allowed Sam to visualise his approach. In addition, he said that implementing trees provided a good time complexity. Sam's approach for the User Store class implemented two self-balancing binary trees (AVL), storing users ordered by user ID and user join date. This approach provided good time complexity for conducting insertion, searching, and deleting operations with complexity of *O (log n)*. However, the time complexity of a search operation by user ID could be improved by considering a secondary data structure, such as a hash map, allowing for a complexity of *O (1)*. In addition, using AVL trees provided a good space complexity.

Sam's approach for the Follower Store class implemented an adjacency list map consisting of a hash map containing user IDs with followers and another hash map containing the user ID with follows. The decision to use an adjacency list map led to very efficient performance, in terms of time and space complexity. For the Weet

class, Sam's approach implemented a combination of different data structures that were suitable for the specific problem.

5.6.7 Summary

Sam's academic performance in programming suggested that he had above-average ability compared to his peers, achieving in the top 10%. Sam's high academic performance could not exist without having devoted substantial time and effort. Interview evidence indicated that Sam had taught himself different programming languages, which required hard work and commitment.

Sam's performance in the Witter assignment showed that his approach of combining multiple data structures was creative and original and stood out from the rest of the participants. Further evidence of Sam's creativity in programming was that his problem-solving strategy depended on finding an analogy that might trigger programming creativity.

5.7 Complete Data Analyses for Robin

5.7.1 Early Education

Robin attended grammar school studying a wide range of subjects for GCSE including ICT, which involved web development scripting languages. Robin said that his passion and interest in mathematics was the reason that he applied for a DM degree. He said that getting a degree with combined subjects made a person more well-rounded and more attractive to potential employers. Robin's ideal job would involve heavy programming for financial models.

Robin's programming experience had not been developed until he started his university study where he learnt programming in CS118. Then, he studied CS126 using Java. Robin said that he enjoyed both modules mainly because of the coursework, robot maze and Witter, which were designed based on real-life contexts, which Robin thought was the reason that both modules were interesting to learn.

Robin did not have any programming experience before studying these modules but he performed well in the programming coursework, as shown in Figure 5.19. However, Robin's performance on the exams was not as high as his performance in the coursework. Figure 5.19 indicates that Robin's performance on the CS118 exam was slightly below the class average, whereas his performance in the coursework, robot maze, Witter, and problem sheets was higher than the class average. Robin's performance in Witter was the second highest grade compared to the other participants' performance, where he achieved 92. However, Robin's programming average for both modules was the lowest among the participants, as shown in Figure 5.5 because of the low performance on the CS118 exam.

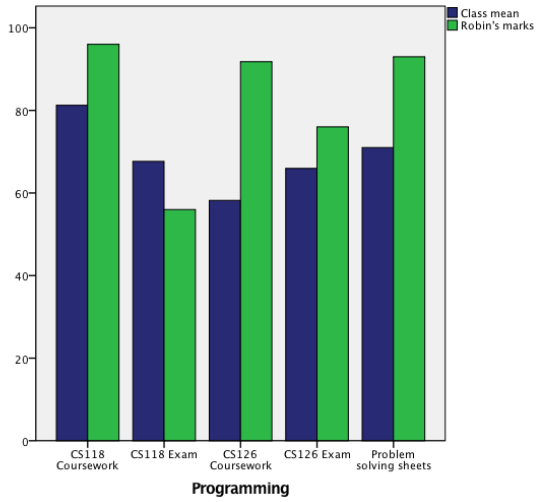


Figure 5.19: Robin's marks in programming assessments.

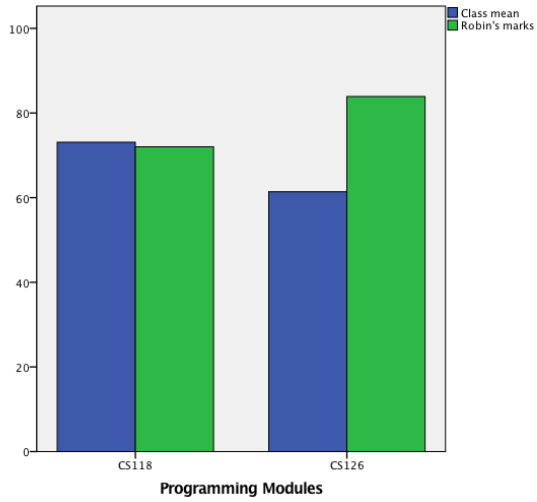


Figure 5.20: Robin's overall marks for programming modules.

5.7.2 Mathematics and Programming

Robin thought that mathematics and programming could be related and raised an interesting point about the relationship direction, believing that he could apply some programming methods to solve mathematical problems. For example, finding prime numbers in large datasets could be easier by developing a computer program algorithm that could compute prime numbers. Robin said he struggled to realise how applied mathematics related to CS but thought the relationship definitely existed.

Another interesting point raised by Robin was that both subjects consist of

two aspects that could help individuals to learn either mathematics or programming. Cognitive aspects that require logical and methodological thinking and problem-solving strategies and skills that are related to both subjects could be acquired and developed by the learner.

5.7.3 Attitudes and Personal Traits

5.7.3.1 Learning Style

Robin's general learning strategy depended on reading lecture materials, books, and extra resources. He preferred to learn individually and be an independent learner. In programming, Robin said that, when particular programming topics were delivered during the lectures, he would read more about the topic and would try to solve problems, as he thought he learnt better by experimenting with what he had been taught. During the lab sessions, Robin preferred to work in pairs where both students could learn from each other. When solving a problem, one student might use a different algorithm that may or may not be better; however, discussing and analysing potential solutions could be rewarding.

5.7.3.2 Challenging Tasks

Robin found that challenging tasks engaged him, as he liked to solve complicated tasks and be pushed to overcome difficulties, which was the best way for him to learn. He thought that easy tasks were not interesting or engaging. Robin said, 'I find [it] a lack of a challenge', but some CS126 tasks were challenging and enjoyable, such as solving a problem that required implementing a hash map. Robin believed that having additional unassessed challenging tasks would be a good idea if they did not affect the workload.

5.7.3.3 Communication Skills

Robin had a good perception about his communication skills: 'I'm a good presenter'. He was involved in fundraising activities and gave presentations to pupils in schools

about his adventure climbing Mt. Kilimanjaro. Thus, Robin liked to communicate in speaking rather than in writing, as he could interact with people and observe their reactions during presentations. He thought that writing did not give him the chance to instantly refine or explain his ideas. In addition, Robin could explain complicated technical problems to another person, which meant he had the ability to communicate his ideas effectively.

5.7.4 Coding Strategies

Robin's coding strategy started by considering the problem specifications and analysing what was required. Then, Robin drew a flow chart explaining his plan to reach the final goal. After that Robin worked backwards by splitting the problem into small approachable tasks, and he started tackling the basic task and its related tasks. In this case, he might write the main class and its functions. He made sure to test his approach, considering time and space complexity.

Robin's coding strategies included organising his code in terms of writing small functions to be called from the main function, which he tried to keep short. In addition, Robin tried to comment on his code in case someone read the code, but if he was implementing a small code or a private code, then he did not comment. Moreover, if Robin wanted to optimise other code, then comments would make the process much easier. Robin's debugging strategy started by locating affected code to then 'talk through out loud what it's doing'. He found this strategy effective, as it helped him to resolve the code errors.

5.7.5 Mental Representation Strategies

Robin followed problem-solving strategies to accomplish certain tasks. Robin identified the final goal of solving a problem and worked backwards to determine what should be implemented first. Then, Robin built up his solution until he achieved the final goal. Robin looked at a certain software application based on its functionality, and he thought of the application core structure related to classes, functions, and data

structure. Robin said that it could be helpful to gather some ideas from other software applications regarding design and implementation.

Robin manifested different SOLO categorisations in his responses for code-writing problems. Robin's response for the array creation problem was categorised as Unistructural, as his code missed two program constructs (inclusive random values and return statement). However, Robin produced a valid response for the linear search problem and the response was categorised as Multistructural. Robin's response for the recursive problem indicated that he did not understand the problem specifications, which resulted in invalid code. His code lacked basic recursion constructs including valid arguments and method invocation, which resulted in the code being categorised as Prestructural. As code-writing problem responses had been written during the CS118 exam, it seemed that Robin's performance during the examination settings was not as good as his performance in programming assessments. As mentioned earlier, Robin's performances in coursework were higher than for exams.

5.7.6 Witter

Although Robin thought that the Witter assignment was challenging at some points, it was enjoyable and rewarding and he said, 'I learnt a lot through it'. Robin mentioned that time and space complexity were important factors to consider, especially if Witter was to become publicly available on the Internet. Then, the difference between *Big-O(n)* and *Big-O(n2)* matters. Thus, he devoted time and effort to implementing an efficient approach. Robin said, 'It was a nice bit of independent study I could do in my own time and work through'.

Robin's approach for the User Store included a hash map and array lists in which the approach considered time complexity over space complexity. Robin justified his decision for the chosen data structures, as the approach had good time complexity for the searching operation with a considerable memory space trade off. Robin's approach for the Follower class included the implementation of two hash maps. Although Robin thought the approach increased space complexity, time complexity for iteration was

also an advantage of the approach. Robin's implementation for the Weet class included hash maps and array list data structures, which allowed for efficient time and space performance. In addition, Robin implemented extra data structures to store trending Weets, which is efficient when displaying them. Robin's overall performance in the Witter assignment indicated his abilities to solve large-scale problems and his ability to understand different data structure advantages and limitations.

5.7.7 Summary

Although Robin's performance in programming was the lowest compared to other participants, for both programming modules, he achieved an average of 78, which fell in the 50^{th} percentile.

However, Robin's performance in exams was not as good as in assignments. If we consider that coursework for both programming modules was weighted at 40% of the overall score, Robin achieved 96 in CS118 coursework and 92 in Witter.

In addition, Robin's first exposure to programming was at university, whereas some other participants had programming experience during their early education. That may have affected Robin's performance in the first programming module, CS118, whereas his performance in the second module CS126 that included advanced topics was better. Thus, Robin manifested above-average ability in a specific area of assessment based on his performance but not in overall performance in programming.

Based on Witter evidence, Robin's approach was to implement different data structures, which indicated Robin's abilities in understanding advantages and limitations. In addition, Robin implemented certain data structures for certain problems, which made the approach efficient and creative.

5.8 Complete Data Analyses for Allen

5.8.1 Early Education

Allen was shy and did not engage in detailed discussion; instead, he gave short answers. However, as the interview progressed he was more forthcoming in explaining his answers. Allen attended state primary and secondary schools for GCSEs. Then, he attended state secondary school for his sixth form, studying AS-levels and A-levels in mathematics, further mathematics, and physics. Allen studied ICT modules, which involved the basic skills of using computers, including Microsoft Office applications, but the modules did not include any programming. Allen joined the Scratch Club as an extracurricular activity and found it interesting; it also helped him to grasp the concepts of programming.

Allen started programming when he was 13 through extracurricular activity aimed at developing students' programming skills using Scratch. In addition, Allen taught himself programming with Java during the summer before he started his study at the university, saying that he wanted to get a head start. Allen said, 'I didn't really have any experience'. He found that programming can sometimes be frustrating, especially when debugging complicated code that might result in an inefficient solution but that he felt quite satisfied in overcoming such difficulties.

Allen studied two programming modules in the first year where he achieved grades higher than the class average for both modules, as shown in Figure 5.22. Allen thought that CS126 was more theoretical, which he found difficult at the beginning, but he said, 'By the exam, I kind of got my head around it so I did better'. Allen said that he was not happy with his performance in the Witter assignment, and he thought he could have obtained better grades. He said that Witter was a huge task and 'quite daunting'. However, Allen's performance in each programming assessment was higher than the class average as shown in Figure 5.21.

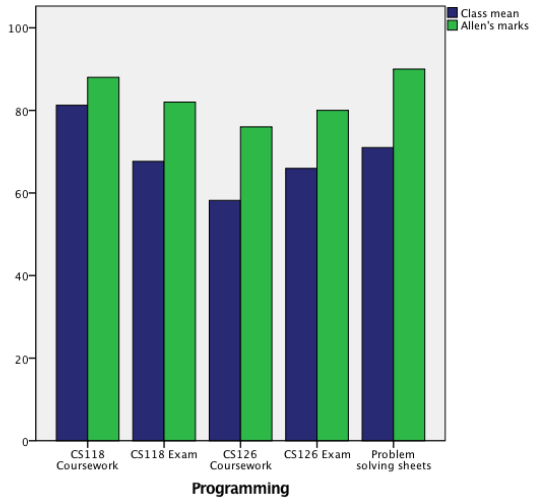


Figure 5.21: Allen's marks in programming assessments.

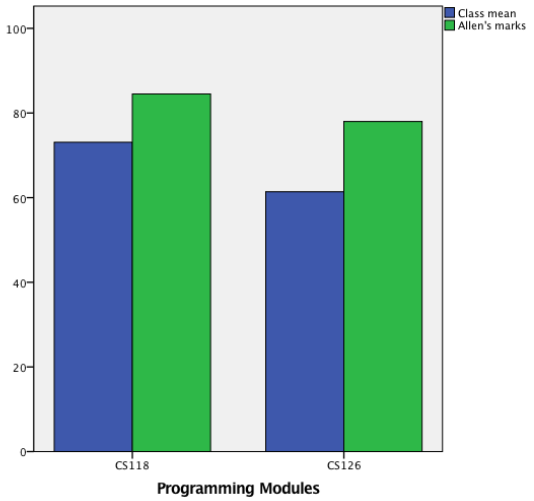


Figure 5.22: Allen's overall marks for programming modules.

5.8.2 Mathematics and Programming

When Allen was asked about his view on the relationship between mathematics and programming, he said he thought that it depended on the nature of the mathematics and programming modules. He said, 'In some modules, they are quite strongly related', especially in the theoretical aspects of both subjects. Allen found that studying combinatorial optimisation, which involved graphs, trees, and networks, related to CS modules, such as Design Information Structure and Algorithmic Graph Theory. However, Allen thought that calculus modules were separate in the mathematics department, and he said, 'I've never seen it as being related to computers'. Allen thought that it was not necessary to be a good mathematician to be a good computer scientist, but some people are good at both subjects, saying, 'I wouldn't say one necessarily means the other'.

5.8.3 Attitudes and Personal Traits

5.8.3.1 Learning Style

Allen thought that attending lectures and taking notes helped him to gain a basic understanding. Moreover, programming problem sheets allowed Allen to 'get [a] deeper understanding of the topic'. Thus, the best learning strategies for Allen were attending

lectures, taking notes, and solving problem sheets. During the lab, Allen preferred to study on his own, allowing him to progress at his own pace. He thought that working individually kept his focus on one task at a time and that working in pairs might slow his progress if the other student needed help with some aspects of the problem. If Allen had a problem during the lab, he spent some time, but not too much, debugging his code before asking other students who might have encountered the same problem.

5.8.3.2 Challenging Tasks

Allen liked to solve challenging tasks that had certain levels of difficulty in which solving the challenging task was not time-consuming. Otherwise, if the task was manageable, then it was quite rewarding. Allen said that including assessed challenging tasks in the problem sheets would encourage him to spend time on them, knowing that he might gain extra marks. Otherwise, he thought that if tasks were not assessed, then there was a chance that he would be learning less about that topic because he may spend all of the time solving difficult tasks. Allen gave an example of a challenging task that he had found rewarding, which was when he solved the robot maze task for CS118. He said, 'Once I had got the solution, not necessarily an elegant solution, but one that worked, that was quite satisfying'.

5.8.3.3 Communication Skills

Allen found presentations quite a daunting experience, especially without preparation. He said that he felt better if he presented a topic that he knew and that interested him. Allen said, 'I can communicate well, I've been able to plan ahead of time'. It seems that speaking was not Allen's favourite way to communicate with others as he preferred writing that allowed him to edit his ideas. Allen found that he could express and explain complicated technical problems to other students, especially if they were 'on same wave length', whereas it was difficult for him to explain such topics to other students who had different thoughts on a particular topic.

5.8.4 Coding Strategies

Allen possessed a unique coding strategy in the cohort. He started solving a problem by drawing a picture of what he wanted to achieve, like drawing a graph or flow chart. Then, he wrote down the basic steps of implementation and pseudo code. After that, he wrote a guideline of his implementation within Java files as comments, explaining each part of the implementation and its purpose. Allen said, 'I always try at least a basic plan of what I'm going to do'. Allen tried to organise his code based on an initial code plan, and he included detailed comments to help him understand and remember complex code. However, if Allen wrote basic code, he would not usually comment on the code. Allen had a strategy for tracking code errors by printing comments to determine the location of corrupted code to fix the errors.

5.8.5 Mental Representation Strategies

Allen had different problem-solving strategies for general versus programming problems. In general, Allen tried to solve a problem by following his intuitive thinking about the problem. For programming problems, he analysed the problem context and understood the requirements before thinking of a method of tackling the problem. Allen depended on what he learnt during lectures, labs, and tutorials to help him solve the problem. When Allen represented a problem or software, he considered the details 'behind the scenes', thinking about how the software had been developed or what data structure had been implemented.

Results from the code-writing analysis of Allen's responses fell across different SOLO categories, showing Allen's abilities to understand problem specifications and to translate those specifications into valid solutions. Allen's response for the array creation problem was categorised as Unistructural, as he could have improved his response by generating inclusive random values. However, Allen's response for the linear search problem was categorised as Multistructural, which was the highest possible SOLO category for this problem. Allen's response for the recursion problem missed important

program constructs that affected his solution, although he used interesting and different methods to approach the problem. Allen tried to use overriding methods, but the code was not clear and was categorised as Unistructural.

5.8.6 Witter

Allen thought that Witter was a challenging assignment because of the way the module was organised. He commented that the module started with many theoretical topics and that suddenly all these topics needed to be practised and implemented in one large assignment. Allen thought the most important aspect of Witter was to choose the most efficient data structure with appropriate consideration of time and space complexity. Allen said that he chose a hash map and linked list as his general approach.

Allen's approach for the User Store consisted of a hash map and linked lists storing users ordered by date, which gave good time complexity in accessing users by ID, but the approach could have been improved by considering a binary search. The Follower Store approach consisted of an adjacency list map, which gave good time and space complexity. However, if more edges had been used, accessing time could be affected. Adjacency list maps provide high space complexity requiring needed memory for the exact size of edges, whereas an adjacency matrix could waste memory if expanded. Allen stored Weets in linked lists that were sorted as hash maps. Time complexity was reasonably good but could be improved for accessing Weet IDs and dates by storing them in extra data structures.

5.8.7 Summary

Allen's performances in his early education and higher education show that he had above-average ability. Allen achieved high grades in his A-level modules and achieved in the top 20% in programming modules at university. Although Allen performed well in mathematics for A-level modules, his performances in first-year mathematics modules were not consistent.

Allen's Witter approach indicated that he implemented creative approaches for

particular tasks. For example, his approach to the Follower class was creative and original, implementing an adjacency matrix. However, the approaches for other tasks were simply effective rather than creative and did not stand out from the rest of participants' approaches. Allen understood programming concepts, and he implemented advanced programming concepts in Witter.

5.9 Complete Data Analyses for Steve

5.9.1 Early Education

Steve was a second-year DM student. He attended a state school, studying a variety of GCSE subjects including mathematics, further mathematics, statistics, and creative art. Steve's plan was to apply for mathematics for his university degree, which required achieving an A* in both A-level mathematics modules. As Steve achieved an A in one module, the DM degree provided by the Department of Computer Science was offered to him instead.

Steve started learning programming at school when he was 14 years old, exploring computer commands and experimenting with operating system batch files. Eventually, he started programming batch files to solve his mathematics homework. At university, Steve studied two programming modules in the first year: CS118 and CS126. During the second year, Steve studied functional programming and found it very interesting, as he had never done any declarative programming language study.

He thought that the CS118 module was a good introduction to programming basics and performed well in both the coursework and final exam, as shown in Figure 5.23. In addition, Steve performed above the class average in all programming assessments, as shown in Figure 5.23.

Steve found that the CS126 module was very helpful in understanding different types of data structures and algorithms. He performed better in the Witter assignment than in the final exam, yet his overall score was above the class average, as shown in Figure 5.24. He suggested that the CS126 lab sheet problems might become more

interesting if the problems related to a real-life context instead of building general stack or queue implementations. Steve did not have specific preferences for operating systems or IDE, but he thought that Linux provided customisable features, such as an open-source operating system. He thought that including some graphical user interface IDEs might reduce a programmer's control of the code.

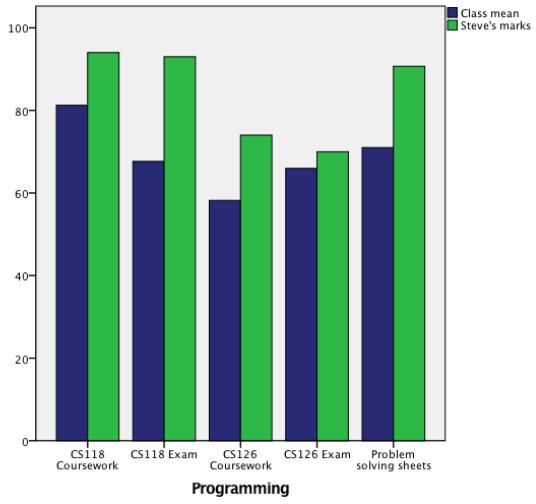


Figure 5.23: Steve's marks in programming assessments.

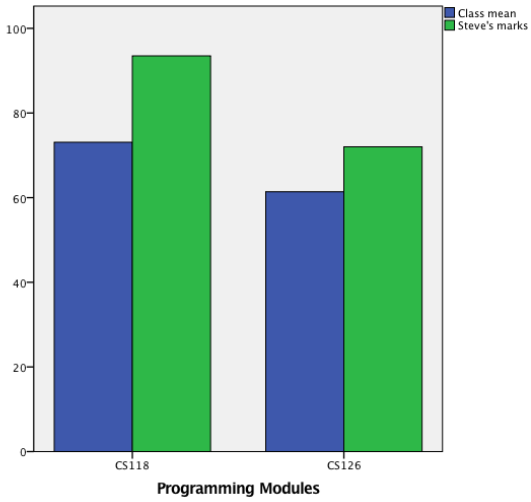


Figure 5.24: Steve's overall marks for programming modules.

5.9.2 Mathematics and Programming

Steve had interesting thoughts about the relationship between mathematics and programming. He thought that mathematics was independent from programming and that mathematical modules were different from programming modules. He said that the analysis module had rigorous proofs, and he did not think that programming modules implemented analysis concepts. Although CS126 included graphs that needed to be proved, mathematical analysis concepts were not implemented rigorously.

Steve gave an example, explaining how mathematical concepts had been superficially implemented in programming. When the runtime for specific algorithms needed to be proved, he said that programmers often used the mathematical approach of 'hand-waving' by suggesting that specific operations run in N times because 'we know it does' without conducting rigorous proofs. Steve said, 'A lot of the programming modules that I do don't really intercept with calculus quite distinctly'. However,

he thought that the DM module, which was based on CS concepts, had obvious connections and implementations in programming modules. Steve thought that it was not necessary to be good at mathematics to be a good programmer. However, having mathematical analysis ability might help a programmer to write a more efficient computer program. Steve said that the functional programming module was mathematically oriented, which required a great deal of mathematical ability.

5.9.3 Attitudes and Personal Traits

5.9.3.1 Learning Style

Steve's general learning style depended on reading and revising material related to specific subjects, whereas Steve's learning strategy for learning programming and algorithms depended on problem analysis and algorithm implementation. He found that he learnt algorithms better when he experimented with code. During the lab, Steve preferred to work on his own, thinking that working in pairs might not guarantee that both students were at the same level. He thought that other students might hold him back from learning more by experimenting with the code and he said, 'I like to change values, tweak and see how it works'. When Steve faced code bugs, he liked to spend considerable time solving the bugs, and he did not mind doing so until he ran out of options. Then, Steve would seek help.

5.9.3.2 Challenging Tasks

Steve would like to solve challenging tasks, as he found them more intriguing to solve and could learn more from them. Steve mentioned that the CS118 tasks were quite easy, and he had done them before, whereas CS126 included challenging tasks, which he found interesting. He mentioned that he would like to have graded challenging tasks, which are weighted as a small percentage of the lab marks. He thought that the marks were not important, but it gave an indication to measure his performance in a specific area.

5.9.3.3 Communication Skills

Steve thought that his communications skills were good, as he said that one of his hobbies was role playing games, which involved a lot of verbal communication. However, Steve preferred to communicate in writing rather than in speaking, as he thought that communicating in writing allowed him to develop his arguments better than communicating verbally.

5.9.4 Coding Strategies

Steve described his coding strategies as 'usually quite bad', where if he has been given a problem, he liked to think it through for some time before getting a rough idea of what the code could look like. Once he had the idea in his mind, he started implementing it without a clear plan until he reached a point where he could not progress further. Then, he would start over, analysing the problem specifications, designing charts, planning his solution, and implementing his code.

Steve had a constant order when organising his code, starting with constructer, mutator methods, generic methods, and inherited methods at the end. In addition, Steve usually commented on his code, especially if the code was huge, to help remind him of the code explanation, but if he tested some ideas then he omitted comments. Steve would not mind reusing and optimising other code that he already wrote and could be reusable, such as a printing function. Steve usually handled code bugs by fixing them in a linear way, one at time, then he recompiled the code and moved on to fixing the next bug. Based on Witter evidence, Steve implemented Java exceptions, which is a sophisticated and recommended strategy for handling code error.

5.9.5 Mental Representation Strategies

Steve's general problem-solving strategy depended on breaking down problems into small approachable parts. In addition, he used dynamic programming to solve parts of a problem that contributed to an optimal solution for the whole problem. When

Steve looked into a software application, he usually thought about the application as an entity. Sometimes, he was intrigued by the application features including data structures and algorithms.

Results from analysing Steve's responses for the code-writing problems had different SOLO categorisations for each problem. Although Steve's response for first problem indicated that he understood the problem specification and translation and could implement his solution, the code had a small bug regarding generating random values. The response did not generate random values including 100, which made the response Unistructural. For the second problem, Steve's response manifested his ability to understand what was required and to provide a valid code. The response was categorised as Multistructural. Steve's response to the third problem indicated that he did not understand how to implement recursion, but he managed to include most of the program constructs; thus, his response was categorised as Unistructural.

5.9.6 Witter

Steve described Witter as a 'tedious' task, as he encountered technical problems with the provided files that were supposed to create the webpage for Witter. He was not able to test his code to display the Witter information on the webpage instead of using too many print functions, which was the only option for him to test his code.

Despite the technical problem, Steve managed to find a time-consuming alternative solution for testing his code. He provided a valid approach for all Witter classes achieving an overall grade of 74. Steve's approach for the User Store included a hash map of an array list. The approach provided a good time complexity, accessing users with *Big-O (1)* but for the insertion, it depended on the length of the array. Steve mentioned that using some sort of graph for storing followers and follows in the graph edges would be better than using data structures that use nodes. Thus, Steve's approach consisted of an adjacency matrix that provided fast accessing time, whereas using an adjacency matrix can affect space complexity caused by a sparse matrix. The approach can be improved by implementing, for example, two hash maps instead of an

adjacency matrix. Steve's approach for the Weet Store class included a hash map of an array list, which provided good time complexity in general, whereas time complexity could be improved for some functions, like obtaining a trending topic by including additional structure.

5.9.7 Summary

Steve's academic performance throughout his early and higher education specifically in programming showed that he had above-average ability among the cohort. He achieved in the top 15%. Throughout Steve's early and higher education, his mathematical abilities were manifested through his high performance. In addition, Steve's high academic performance indicated his dedication and hard work. Steve was certainly passionate about programming, as he spent his spare time at home programming batch files, solving mathematical problems, and writing source code for games. Witter evidence indicated that Steve's approach was appropriate according to each specific problem, achieving good time and space complexity. However, his approach was not the optimal solution, and in some cases, the time complexity could have been improved significantly.

5.10 Cross-case Analysis

5.10.1 Early Education

Table 5.2 indicates that all participants shared two A-level mathematics modules achieving high grades. However, some participants studied different third A-level modules, including statistics, physics, and computing. Moreover, three participants attended selective schools for which students must meet certain academic requirements to be admitted, whereas the rest of the participants attended regular non-selective schools.

	Math	Further Math	Statistics	Physics	Computing	Prog. Experience	School
Steve	A*	A	NA	NA	A	Yes	Non-selective
Sara	A*	A	NA	NA	NA	Yes	Selective
Lee	A*	A	B	NA	NA	No	Both
Bob	A*	A	NA	NA	B	Yes	Non-selective
Robin	A*	A	NA	NA	NA	No	Selective
Allen	A*	A	NA	NA	NA	No	Non-selective
David	A*	A	NA	NA	NA	Yes	Non-selective
Joe	A*	A	NA	NA	NA	Yes	Non-selective
Sam	A*	A*	NA	A	NA	Yes	Non-selective

Table 5.2: Spread of the early education theme across the participants.

5.10.2 Mathematics and Programming

Table 5.3 illustrates the ranking among the cohort and the participants themselves for programming and mathematics modules. As participants studied different degrees, four participants shared two mathematics modules, whereas five participants shared other mathematics modules. For participants who studied DM degrees, their cohort grades for mathematics modules were not collected. Thus, some rankings for the cohort are not applicable.

	Programming			Mathematics			Relationship
	Class rank out of 73	Participants rank out of 73	Contests	Class rank	Participants rank	Contests	
Steve	19	5	0	NA	3/5	0	Yes/No
Sara	11	4	1	13/73	3/4	0	Yes
Lee	32	8	0	NA	2/5	1	Yes
Bob	3	2	0	34/73	4/4	0	Yes
Robin	38	9	0	NA	5/5	0	Yes
Allen	25	6	0	NA	4/5	0	Yes/No
David	1	1	1	2/73	1/4	2	Yes
Joe	36	7	0	6/73	2/4	0	Yes
Sam	10	3	0	NA	1/5	0	Yes

Table 5.3: Spread of mathematics and programming themes across the participants.

5.10.3 Attitudes and Personal Traits

Table 5.4 shows that seven participants preferred to learn programming by solving a problem based on coding experimentation and practising, whereas two participants had a different learning strategy that included reading, note taking, and self-teaching. Regarding the preferences in terms of paired working, three participants were in favour of working with another student during lab sessions. In relation to communication skills, two participants preferred to communicate verbally, and three participants had no preference between spoken and written communication, whereas four participants preferred to communicate only through writing. In relation to the ability to explain complex technical problems, four participants who prefer to communicate verbally can explain complex problems to other students. Concerning challenging tasks, every participant would like to be challenged to solve problems; however, two participants were concerned about including assessed challenging tasks.

	learning style				Communication skills			Challenging Tasks
	Experimental	Reading and note taking	Self-taught	Working in pairs	Speaking	Writing	Explaining a complex problem	
Steve	✓	✓			✓		✓	✓
Sara	✓		✓	✓		✓		✓
Lee		✓		✓		✓		
Bob			✓		✓	✓		✓
Robin	✓	✓		✓	✓	✓	✓	✓
Allen	✓	✓			✓	✓	✓	
David	✓	✓				✓		✓
Joe	✓							✓
Sam	✓	✓			✓		✓	✓

Table 5.4: Spread of attitudes and personal trait themes across the participants.

5.10.4 Coding Strategies

In relation to coding strategies, Table 5.5 indicates that seven participants split a problem into small parts to be solved, whereas three participants preferred to start the coding process by analysing the problem, planning their solutions, drawing graphs, and implementing the code. Two participants preferred the trial-and-error strategy. All participants adhered to coding conventions, including comments and optimisation. The most-used code debugging strategy is to print lines to indicate the location of

the code error. Another strategy to detect the code error is to go through the whole code, investigating what has possibly gone wrong. However, the strategy of exceptions, which is more advanced, was applied by three participants.

	Coding strategies							
	Strategy			Comments	Optimisation	Debugging		
	Splitting a problem	Trial and error	Planning and drawing flow chart			Printing lines	Exceptions	Sanity check
Steve	✓			✓	✓		✓	
Sara			✓	✓	✓	✓		
Lee		✓		✓	✓	✓	✓	
Bob	✓			✓	✓			✓
Robin	✓			✓	✓			✓
Allen			✓	✓	✓	✓		
David	✓			✓	✓	✓		
Joe		✓		✓	✓	✓	✓	
Sam	✓		✓	✓	✓			✓

Table 5.5: Spread of coding strategy theme across the participants.

5.10.5 Mental Representation Strategies

In relation to mental representation strategies, Table 5.6 indicates that seven participants tend to solve problems by breaking them down into small sub-problems, whereas two participants tend to adopt a trial-and-error strategy.

	Problem solving				Abstraction	
	Analogy	Dynamic prog.	Splitting a problem	Trial and error	Data structure and algorithm representation	Recursion problem
Steve		✓	✓			
Sara			✓		✓	
Lee	✓					
Bob			✓			
Robin			✓		✓	
Allen	✓			✓	✓	
David		✓	✓		✓	✓
Joe			✓	✓	✓	✓
Sam			✓		✓	

Table 5.6: Spread of the mental representation strategy theme across the participants.

In addition, two participants tend to find a solved problem that shares some aspects with the current problem. The analogy strategy can be useful sometimes to solve simple or generic problems but not for large-scale problems. Another strategy that had been used by two participants is dynamic programming, which is similar to splitting a problem strategy. However, in dynamic programming, the sub-problems share the same attributes, and the result of one sub-problem can be used to solve other sub-problems. Four participants tend to use a combination of problem-solving

strategies. In relation to abstraction, six participants were able to provide a representation of a software based on its data structure and algorithms rather than providing a superficial representation related to the software interface and design. However, only two participants were able to solve the recursion problem and were categorised in the higher SOLO categorisation.

5.10.6 Witter

Table 5.7 indicates the data structure types that have been used to solve the Witter project. Three participants used a combination of data structures, including hash maps and binary trees, whereas two participants used a more efficient combination of data structures, including hash maps and an AVL tree. In addition, two students implemented hash maps and array and linked lists, whereas three participants implemented hash maps and an adjacency matrix.

	Witter						
	Data structure						Grades
	Hash map	AVL tree	Binary tree	Adjacency matrix	Linked lists	Array list	
Steve	✓			✓		✓	74
Sara	✓		✓				88
Lee	✓				✓		78
Bob	✓		✓				95
Robin	✓					✓	92
Allen	✓			✓	✓		77
David	✓	✓					90
Joe	✓		✓				55
Sam	✓	✓		✓			88

Table 5.7: Spread of the Witter theme across the participants.

Summary

In this chapter, we presented our findings for four case-study CS participants. The findings from the different data collection methods were incorporated for each participant and presented as multiple themes and sub-themes. These themes will be discussed in the discussion chapter.

CHAPTER 6. Discussion

This chapter is divided into two sections. The first addresses the research questions outlined in the methodology chapter, which are addressed to provide a description of programming characteristics derived from our findings. The second section provides an overview of the proposed model of giftedness.

6.1 Evaluation of the Research Questions

6.1.0.1 Mathematics and Programming (RQ1 and RQ2)

The first research question 'To what extent does mathematical ability correlate with programming ability in general?' addressed the statistical correlation between mathematical ability and programming ability in general based on student performances. The first and second sub-questions asked, 'What is the correlation between student performance in discrete mathematics and programming modules?' 'What is the correlation between student performance in calculus and programming modules?' These sub-questions addressed the specific correlation between DM and calculus and programming. Quantitative approaches including descriptive statistics and PPMCC test were used. The second research question asked, 'What are student perceptions about the relationship between mathematical ability and programming ability?' This question addressed the student perceptions of the relationship to clearly support our findings of the first research question.

The results of our study indicated a positive relationship between mathematical ability and programming ability. This result supports previous studies that concluded the existence of the relationship (Pacheco et al., 2008; Bennedsen & Caspersen, 2005;

Bergin & Reilly, 2005b; Pioro, 2006). However, the study results contradict the findings of Tukiainen and Mönkkönen (2002), which were not statistically significant but suggested a negative relationship between student pre-university mathematics grades and introductory programming grades. Their study sample size was 33 students, and there was no justification of such a controversial result. Our study design addressed issues with previous research on methodological approaches and with inadequate sample size.

Although the statistical results indicated a positive relationship between mathematics and programming, causation cannot be determined. We do not have evidence to support the hypothesis that a good programmer is a good mathematician. The relationship involves other factors that make causation unclear. The context of programming can be either educational or professional. In the educational context, mathematics modules have been designed in relation with specific programming modules where relationships between certain mathematics and programming concepts are obvious. In the IT profession, there are different types of programming paradigms used where a bespoke programming paradigm is designed for a specific context. For example, in the financial sector, mathematical models are important in predicting stock prices.

There is no doubt that learning programming involves some sort of mathematical knowledge. As the literature suggested, concepts such as logic and algebra play important roles in learning programming (Henderson & Stavely, 2014). The degree of involvement varies across institutions, depending on the CS curricula. Our findings from the interviews indicate that seven out of nine participants realised the existence of the relationship, whereas two participants said that the relationship depends on other factors, such as the type of mathematics and programming. In addition, participants identified the relationship between linear algebra and graph theory in the CS126 programming module.

Computer science graduates exposed to several mathematics modules may not be able to apply mathematical concepts in the context of workplace tasks (Baldwin et al., 2013). Therefore, there is a need to understand flexibility of programming

and mathematics curricula to adapt to the rapid changes in the CS field. Thus, the main focus for CSE is to consider which specific CS discipline - computer engineering, software engineering, or artificial intelligence - requires certain mathematics topics to be taught.

Discrete mathematics plays a role in CSE. Most CS degree providers urge first-year students to enrol in DM, and this discipline has been included in the Association for Computing Machinery (ACM) curriculum. However, the significance of DM should be based on multiple factors, such as teaching methods, that help students to grasp, apply, and evaluate mathematical concepts within programming modules. Another element that needs to be considered by curriculum designers is how certain categories of mathematics could be appropriate to specific programming paradigms; for instance, teaching in a functional-driven language paradigm could be linked to function concepts in DM (Power et al., 2011; VanDrunen, 2017).

In contrast, in an object-oriented paradigm, the use of predefined set classes might not require understanding of mathematical set theory. Results from previous research suggested a correlation between DM performance and introductory programming course performance (Sutner, 2005). Our statistical result was similar, showing positive correlations between the DM module and data structure module.

However, integrating DM with CS curricula has several challenges that need to be addressed. Some higher education institutions enforce certain mathematics modules as core subjects to be delivered for all science degree students. This could cause confusion regarding how to link the provided mathematics modules with certain sciences, such as CS. Another challenge is that many mathematics modules are taught by mathematics faculty and not by computer scientists (VanDrunen, 2017). Students may benefit by studying DM along with data structure modules taught by CS faculty (Decker & Ventura, 2004).

Furthermore, teaching calculus to undergraduate CS students is required by many institutions, but relevant questions include ‘How much calculus do students need?’ and ‘How do they apply calculus in programming?’ Requiring calculus as

a prerequisite for intermediate or upper-level mathematics courses that CS students might take (e.g. combinatorics, graph theory, or logic) may be nonsensical as knowledge of calculus plays essentially no role in such courses. Instead of calculus, DM would develop essential skills for CS students (Ralston, 2005).

Calculus could be less correlated with introductory programming than DM, as prior research suggests that students enrolled in calculus perform significantly less well in programming courses than those enrolled in DM (Pioro, 2006; Pacheco et al., 2008). This suggests learning programming basics requires basic algebra and DM. First-year CS students may benefit from taking DM as an introductory module more than from taking calculus (Leblanc & Leibowitz, 2006). Our results, however, indicate that calculus has a more positive effect on student performance in data structures compared with the role of calculus in introductory programming. Thus, calculus is a general mathematics concept that could play a role in learning programming by teaching specific problems that require the implementation of calculus. Mathematical concepts, such as differential and integral calculus, may not have obvious implications or connections to introductory programming modules, yet those concepts can be helpful for specific programming modules, such as functional programming. Further investigations are needed into the role of calculus in different programming paradigms, such as functional programming.

6.1.0.2 Mental Representation (RQ3)

Mental representation considers how such a programming problem can be understood and solved based on how the programmer recalls and applies specific knowledge. The mental representation ability allows the programmer to construct a problem's perception that might influence the solution. It depends on the type of programming problem that needs to be solved. We are considering problems that require reasoning, problem-solving strategy, and abstraction ability. Thus, mental representation ability consists of reasoning, problem-solving strategy, and abstraction ability.

Based on the evidence from the Witter assignment in this study, reasoning could

be manifested when students decide to implement a certain data structure or algorithm to solve a specific problem. Participants showed their reasoning during the interviews and in the Witter assignment preamble comments. In addition, participants had a wide knowledge of data structures and algorithms, and they could justify their reasons for choosing an appropriate data structure and algorithm in the Witter project. Participants understood the advantages and limitations of different types of data structures.

Anderson (cited in (Robertson, 2017)) defined problem solving as 'any goal-directed sequence of cognitive process' in which the cognitive process comprises different phases. This begins by (1) defining the problem by analysing the specifications that need to be met; (2) defining all possible algorithms that specify data types, data structures, and logical sequences; (3) defining the advantages and limitations of each algorithm to select the best algorithm; (4) implementing the algorithm; and (5) evaluating the solutions against the possible solutions that might be applied by different algorithms (Sprankle & Hubbard, 2012). Regarding problem-solving strategies, gifted student programmers might implement one or multiple problem-solving strategies that suit the specific problem. Most participants adopted the strategy of splitting a problem into small tasks, which was suggested by Polya (2004). However, Hoc (2014) distinguished between the problem-solving strategies of expert and novice programmers, stating that experts tend to break down the problem, whereas novices tend to solve the problem by coding line by line.

In relation to abstraction, the literature suggested that abstraction can be manifested in different ways, including recursion and deep representation of software application (Kramer, 2007; Hazzan & Kramer, 2016; Koppelman & van Dijk, 2010). In this study, participants struggled to solve the recursion problem as only two participants provided a valid solution. It has been suggested that recursion is a difficult concept to grasp by programmers, as it requires the programmer to abstract from the implementation to construct a deep perception beyond the boundaries of coding. However, the participants manifested another method of abstraction that requires a deep representation of a software application. (Hazzan & Kramer, 2016) suggested that providing a

higher representation of a software application can indicate the ability to abstract from the superficial representation, which includes the graphic interface design, to construct representations based on the data structure and algorithm. In this study, six participants provided perceptions of a certain software application, indicating that they would consider the application based on what and how the data structures and algorithms had been implemented. However, some participants represented the application based on its general purpose and its graphical design. Interestingly, a few participants mentioned that they used to have superficial representations of some problems, but after studying the data structure modules, they became interested in more abstract perceptions. It would be very interesting to investigate whether or not educators could enhance a student's cognitive ability.

In addition, participants' perceptions of the Witter assignment were well articulated in terms of explaining how they found efficient solutions, meaning that time and space complexity were important factors to consider in the Witter problem rather than knowing what Witter does. As participants implemented Witter based on OOP in which one of its features is abstraction and given the fact that Witter was a large-scale problem, participants manifested the ability to understand the connections between different classes, objects, and interfaces.

6.1.0.3 Knowledge of Mathematics and Programming (RQ4)

The acquisitions of mathematical concepts, data structures, and algorithms are important characteristics in identifying gifted student programmers. The participants were introduced to multiple A-level mathematics modules and achieved high grades, and as they progressed in their academic studies, advanced mathematics modules had been introduced. In relation to programming, one participant had been introduced to a wide range of advanced modules of data structure and algorithms at a high school in a European country. The rest of the participants had been introduced to ICT modules. A few participants had no programming knowledge before attending university, whereas other participants had taught themselves Java and Python. At university, participants

had been introduced to two first-year programming modules including introductory and data structure modules. McKeithen et al. (1981) stated that expert programmers tend to recall multiple meaningful areas of knowledge to provide an efficient solution rather than recalling single items of information. Participants were able to implement different data structures and algorithms to solve Witter, and all participants implemented multiple advanced data structures and algorithms. That also confirms the suggestion by Joseph (2015) that being highly competent in programming requires the programmer to implement advanced data structures, such as AVL trees. Two participants implemented AVL trees for Witter, whereas other advanced data structures have also been implemented by the participants. The study findings confirm the suggestion in the literature that the acquisition of mathematics and CS knowledge allows a good programmer to understand a problem and to apply data structures and algorithms not only to solve the problem but to provide an efficient solution. The knowledge can be acquired either during early education or higher education and can be self-taught, which might be a sign of the individual's determination and motivation.

6.1.0.4 Coding Strategies (RQ5)

In relation to the research question that asked, 'What coding strategies do gifted students tend to use?', participants implemented certain coding strategies to start understanding the problem and writing the code. The strategy that was most used was to split the problem into small tasks. Five participants adopted this strategy. However, three participants adopted the strategy of planning and drawing the flow charts, whereas two participants adopted the strategy of trial and error. In addition, participants implemented additional coding strategies, such as commenting, code optimisation, and code error handling. Some participants also mentioned that adding explanatory comments allowed their code to be readable by both themselves and others. Some participants thought that commenting on their code was helpful in some cases when the code was complex, but not for simple code. Moreover, some participants thought that the importance of code comments was derived purely from being

an assessment criterion rather than being valuable for code readability. However, evidence from the Witter project indicated that participants provided clear and extensive comments to make the code easy to understand.

Code optimisation is another strategy to improve, modify, and use an existent piece of code. In some cases, some codes can be used multiple times within one project; thus, tweaking a few lines can be effective in terms of time. Our findings suggest that all participants implemented code optimisation, such as reusing quick sort algorithms in Witter for different purposes. Some participants tweaked some codes from lab session exercises to be used in Witter. In some cases, code optimisation cannot be helpful if the programmer does not understand the code or its purpose or how optimisation can be done for the specific context (McConnell, 2004).

Code error handling is a basic coding strategy that programmers should know; yet, implementing certain code error handling techniques, such as exceptions, returning, and printing values, could indicate a gifted programmer. In addition, a gifted programmer should be able to master different code error handling techniques to overcome both syntactical and logical errors, which are more difficult to handle. Most participants in this study tended to handle the code error using a printing technique to help them trace the error and locate a small segment of the code that might need to be fixed. However, some participants were able to implement different techniques including exceptions, constant debugging, and recompiling along with a sanity check and printing techniques. In addition, one participant mentioned that searching the Internet to research an error that had been output by a programming language complier can also help to detect syntactical errors. Often, the programming language compiler produces ambiguous error messages that do not help, especially for logical errors.

6.1.0.5 Attitudes and Personality Traits (RQ6)

In relation to the attitudes and personality characteristics that gifted student programmers tend to possess, learning style, communication skills, and the ability to solve challenging tasks were investigated. It is important to mention that the aim of the in-

vestigation was not to classify participants' learning styles but rather to describe how gifted student programmers learn programming and how they interacted with their partners during the learning process. Thus, the two main areas of investigation were programming learning style and participants' tendency to learn programming in pairs during lab sessions. Our findings indicated that most participants learnt programming by experimenting, as some participants mentioned that learning from mistakes can be rewarding. However, some participants adopted the traditional style of learning programming by attending lectures, reading material, and taking notes. Tutorial sessions are another learning activity whereby a small number of students can analyse and discuss how to solve specific problems in theory. These sessions allow students to brainstorm before attending lab sessions to implement their ideas. Some participants highlighted the importance of tutorial sessions. Pair programming is a collaborative learning activity whereby two students share one computer to solve a specific problem. It has been effective in boosting student confidence when learning programming. However, student learning styles might differ within pairs, which could have either a positive or negative effect on performance. In this study, six participants preferred not to work in pairs, as it could prevent their own ideas and solutions from being implemented as the other student's knowledge or opinions may not be the same. Another reason could be that some gifted students might lack certain personal traits that would allow them to communicate and collaborate with other students. In this case, pair programming may not be an ideal method of learning for gifted students.

We believe that communication skills play an important part in the learning process, as interaction might be affected by communication skills. Participants in this study self-evaluated their oral communication skills during interviews, based on giving presentations in the professional skills module; the skills varied. Four participants, two of whom were non-native English speakers, found presenting a daunting experience and preferred to communicate in writing rather than speaking. In addition, the same four participants said they would find it difficult to express a complex programming-related issue to their partner and would try to illustrate the ideas through drawing. Some

participants evaluated their oral communication skills to be average, depending on the topics they chose to discuss. Two participants were comfortable presenting as they were involved in social activities that required superior oral communication skills (e.g. giving talks at events, such as departmental open days). Other participants preferred to communicate in writing rather than speaking to allow more time to articulate ideas and arguments. One participant mentioned that communicating verbally required him to premeditate his thoughts to be understood, which made speaking not the best option. It seems that gifted student programmers might have inadequate communication skills. As communication skills are important characteristics in both educational and IT professional contexts, more pedagogical activities might need to be included (Chinn & Vandegrift, 2008; Havill & Ludwig, 2007).

Accepting challenging programming tasks could be a characteristic indicating that a gifted programmer has the confidence, domain knowledge, and motivation to deal with complex problems. In addition, intrinsic motivation can be boosted by solving challenging tasks, as self-satisfaction can result in learning and solving a difficult task. Often a programmer experiences joy and self-satisfaction when unpacking completed code and designing and implementing an efficient algorithm. In this study, participants had a positive view of accepting challenging tasks to solve. Some participants preferred to be unassessed for these parts of assignments, whereas others preferred optional assessment. In addition, three students were involved in multiple programming, mathematics, and game development competitions, which often consist of challenging tasks to be solved within a limited time.

6.2 Model of Giftedness in Programming

As discussed earlier in the literature review chapter, the three-conception theory of giftedness consists of three clusters: above-average ability, creativity, and task commitment. We decided to adhere to this theory to provide us with initial steps of identifying gifted programming students. However, our findings suggested that the three-conception theory does not fully apply to the context of programming for which

academic performance cannot be the only indication of giftedness in programming, as numerous characteristics could be another indication. Although the initial identification method of our participants was derived from the three-conception theory (i.e. based on participants' academic performance as an indicator of intelligence), the context of programming consists of multiple factors that could be as important as intelligence. We argue that gifted students might manifest some characteristics other than intelligence or some gifted students might manifest combinations of different characteristics. Thus, we suggest a model that can be used to identify gifted students in programming.

The model consists of three profiles as shown in Figure 6.1, mathematical ability, creativity, and personal traits, where each profile consists of multiple characteristics that will be discussed in detail. Given the nature of this research, which derived from investigating different characteristics of programming and considering educational factors, overlap between profiles will also be presented to explain the model. The importance of this model is that characteristics have be presented based on inquiry where gifted students who manifest specific profiles or combination of profiles, have been the cornerstone of this research.

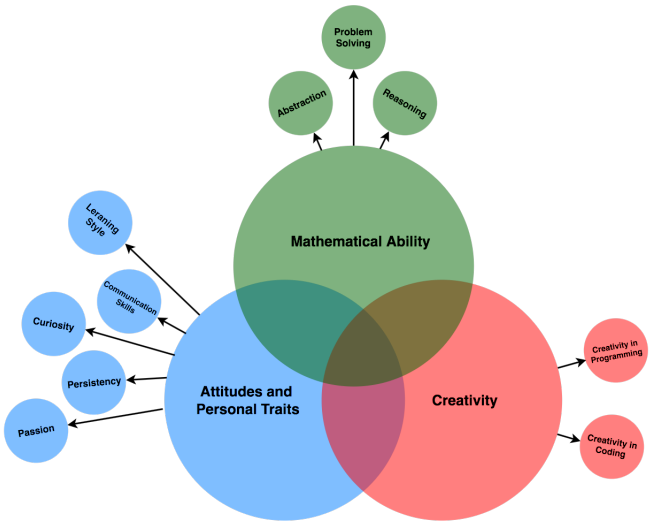


Figure 6.1: Model of giftedness in programming.

6.2.1 Mathematical Ability

The role of mathematics in CS has been discussed and investigated in a growing body of research that aims to unpack the relations, effects, and causalities. Statistical-based evidence found that there was a correlation between student mathematical performance and programming performance (Owolabi et al., 2014). Another body of literature focused on investigating the success factors of learning programming in which mathematical ability was an important factor when learning programming and predicting students who might excel in CS (Pacheco et al., 2008; Watson & Li, 2014; Simon et al., 2006). In addition, investigating mathematical ability has been a major part of designing programming aptitude tests, including questions to measure student reasoning, pattern recognition, and logic abilities. However, the reliability of the PAT is questionable because different results were found when replicating the test in different institutions (Dehnadi, 2009).

Despite these considerable studies, determining the causality between mathematics and programming is still questionable when research findings indicate a lack of evidence that, for example, studying programming has positive effects on mathematical development (Kurland et al., 1986). Thus, the focus of investigating the role of mathematics in CS should change from proving the correlation to understanding these relations and their effects on the student learning process.

We need to step back and understand other factors around programming and mathematics. There is a need to define programming and the context from which our definition of programming is derived. We need to investigate the implications of teaching mathematics to CS students. There is a need to understand which type of mathematical concepts can be taught to CS students for specific programming paradigms.

After discussing the relationship between mathematics and programming, both disciplines share a similar cognitive process that requires abilities such as reasoning, logic, problem solving, and abstraction. Although measuring these abilities is still questionable in terms of reliability and validity, we noticed that students manifest single or multiple abilities through different methods of assessments.

The acquisition and application of mathematical concepts relies on cognitive abilities, such as reasoning, problem solving, and abstraction. Therefore, these cognitive abilities, which could also be referred to as mathematical abilities, underlie cognitive processes that help students grasp mathematical concepts, which can be applied to solve problems within different domains. Cognitive abilities may have positive effects on student mathematical achievements (Floyd et al., 2003). Thus, we claim that students who exhibit academic achievements in mathematics modules possess single or multiple cognitive abilities that might accelerate their ability to learn programming.

The findings in this thesis show that participants have been exposed to a wide range of mathematical modules and have been high achievers, with some participants in the top 10% of the cohort. In addition, participants exhibited mathematical ability in their early education, achieving high grades in high school mathematics subjects. Mathematical abilities allow students to create a mental representation of programming problems.

We believe that participants could apply their mathematical knowledge to understand computational complexity. For instance, participants' knowledge of graph theory and how to compute time and space complexities, implementing *Big-O* notations, allowed them to be able not only to implement a data structure but also to analyse the effects of the implementation of the data structure and algorithms.

6.2.2 Creativity

Creativity is a buzzword concept that has different interpretations related to fields of study, such as psychology, art, or science. The ambiguity of the meaning of creativity in CS in general and particularly in programming needs to be clarified. In this section, we briefly explore different definitions of creativity before a discussion of the literature on creativity in CS.

Sternberg and Lubart (1999) implied that creativity is the ability that allows original and unique work to be produced. Another definition, derived from a large review of previous studies, suggested that creativity is an interaction in an individual,

or a team of individuals, between ability and process, resulting in producing an outcome that is novel and useful in a specific social context (Plucker & Beghetto, 2004).

A study of individual motivation behaviour (Deci & Ryan, 1985) suggested that individuals strive to be self-determined and expert. Being a self-determined individual is derived from behaving according to an individual's own desire to achieve a goal and/or competence, instead of acting based on someone else's desire. Consequently, self-determined people are motivated to engage with new challenging tasks, allowing them to expand their ability and knowledge. Renzulli (1984a) stated that the creative-productive trait describes human activity based on aspects such as self-determination, motivation, and competency, when individuals provide their best ability to be in the development of unique thoughts, solutions, or products.

In addition, Renzulli (1984a) emphasised the role of creativity in education to promote student creativity by focusing on the implications of knowledge to be integrated with inductive reasoning of real-world problems. In this case, students are expected to be self-determined and motivated, solving new challenging problems to transform them from passive to active learners. Thus, a creative-productive trait means that student abilities are enabled and empowered by both the educational system and teachers, allowing them to learn specific fields of knowledge by adapting inductive reasoning and solving challenging real-world problems.

One shared aspect that can be derived from different creativity definitions is the ability to achieve something novel. Boden (2004) described two aspects of creativity related to achievements as historical creativity, achieving something that nobody else has achieved before, or psychological creativity, achieving something that is novel and original to the person. Psychological creativity is more important in education for understanding how to improve student creativity based on pedagogy.

6.2.2.1 Creativity in Programming

As we discuss the general aspects of creativity, it is important to address the literature on creativity epistemology within the CSE context to understand what creativity means

and to allow us to identify creativity aspects that can be exhibited by students. Often, computing has been perceived as a non-creative discipline among students (McCormack & d'Inverno, 2012) where CS curricula has had inadequate emphasis on creative thinking and creative problem-solving skills (Romeike, 2007a).

There are numerous aspects of creativity in CS in general, and in programming in particular, and some of these aspects related to our research will be discussed. Romeike (2007b) emphasised the importance of creativity in CSE and identified three aspects: individuals with motivation and passion, environment, and software design. These aspects can boost creativity in CSE, where creativity can be exhibited by a student who is passionate about software design and motivated by solving a challenging problem that requires certain CS knowledge. Thus, the interrelationship between the three aspects can result in creative ideas and solutions.

Motivation is what drives an individual to achieve desired goals. In the context of programming, a study conducted by Bergin and Reilly (2005a) investigated the correlation between intrinsic motivation and programming performance using the Motivated Strategies for Learning Questionnaire (MSLQ), which is an instrument to measure student motivation, and using programming module grades to measure the programming performance of 110 students. They found a significant correlation to prove the study hypothesis that highly motivated students perform better in programming.

Programming and learning programming are time-consuming tasks in some cases, where open-source programmers may spend long hours programming, yet they are motivated and determined to accomplish challenging tasks or to learn something new, pushing them out of their comfort zone. Thus, motivation can be a strong force for students to perform better in programming and motivation might trigger creativity in a specific area of interest.

The findings from participant interviews in this study indicated that participants' interests in studying CS were based on both intrinsic and extrinsic motivation. Some participants manifested abilities, such as mathematical ability, above-average ability in programming, and creativity, allowing them to strive to accomplish their

study with high performance. In contrast, other participants were driven by extrinsic motivation factors, such as exceptional career opportunity or rewards, which could be getting high grades to be competitive with others.

Previous research suggested that the CS field shared some aspects of creativity with other disciplines, such as mathematics and engineering. Saunders and Thagard (2005) stated that computer scientists can share similar processes with engineers to solve problems by implementing techniques to build a solution where creativity can be manifested by developing an original solution. In addition, creativity in CS can be triggered by by frustration from engaging with a complicated problem.

Another aspect of creativity in CS is abstract thinking that could be also manifested in mathematics. Brooks (1995) quoted that programmers 'build castles in the air, from air, created by exertion of the imagination'. Solving a problem in CS may require multiple cognitive abilities, such as reasoning, pattern recognition, and mathematical ability to produce a creative solution. However, Saunders and Thagard (2005) introduced another aspect that may trigger creativity for solving such a problem by analogy, which he described as 'analogy is far from being the only source of creative solutions in CS, but its importance is illustrated by many historical examples'. Two types of analogy have been identified: local and distant. The local analogy refers to analogies derived from the same domain, whereas distant analogies are derived from different fields. For example, neural networks in CS derives its conceptual idea from biology. Thus, creativity in solving a problem can be derived from considering problems in the field or from related fields.

Creativity can be manifested by designing and programming a game in an original way, as developing a game allows a programmer to creatively think about game characters, levels, and behaviours (Bennett et al., 2013). Creativity as divergence can be applied to other types of software that can be designed and programmed in original ways that are different from the normal solution. Li et al. (2015) interviewed 59 experienced software engineers from Microsoft, identifying 53 factors that might make a great programmer. One of the factors that emerged from the study was creativity,

which involved novelty in producing a solution for a specific problem. Thus, a student, who provides a solution that is divergent from the norm by considering problem limitations and the original algorithm may possess creativity in programming.

The Witter project in this study required participants to solve a large-scale problem to design and implement a software that simulated Twitter functionalities, which required designing algorithms and data structure while considering time and space complexity. Multiple options of algorithms and data structures could have been used for each function, where each algorithm and data structure had limitations that needed to be considered. Participants had to make decisions around implementing their own solutions for the problem.

As mentioned earlier, any programming problem has multiple options of algorithms that could be implemented, which could be simple, obvious, or novel. Participants' Witter solutions varied based on different algorithms and data structures, with hash maps being the most common data structure implemented. However, some participants were able to provide a novel solution that was divergent from the common solutions, as a combination of two or more different types of data structure. One original solution consisted of implanting an AVL binary search tree and hash maps and another was a combination of an adjacency list and AVL tree.

Often, writing a computer program requires a programmer to be imaginative an can think beyond the laws of reality to produce creative solution (Bronish, 2012). Thus, programming tasks are considered to be a creative activity that allows a degree of freedom to solve a problem in many ways, which may result in producing a novel solution. Yet, the advantage of freedom could produce a solution with mistakes and costly errors. Programming involves phases, such as requirement analysis, design, implementation (coding) and testing, where creativity could be manifested in different ways according to each phase. In this case, multiple forms of creativity could be defined in CS in general and in programming in particular.

When a programmer solves a problem, one important process is the design phase, where multiple aspects should be considered, such as problem requirements, constraints

of the problem domain, limitations of the programming language, and constraints of knowledge as algorithms and data structures. During the design phase, a programmer considers multiple possible algorithms and data structures where experiences may or may not be helpful. In this case, the combination of previous aspects may result in original designs that might lead to innovative products or software, yet the following phases of implementation and testing may also affect the final product.

A mixed quantitative and qualitative study conducted by Salgian et al. (2013) used different instruments to measure student creativity in a course on conducting robots through a pre-developed creativity test (Torrance test), self-report of creativity rating, assessment to rate student projects, and a focus group. Although mixed methods had been used to measure student creativity, there were no correlations between certain instruments, such as the creativity test and the self-report. An explanation of the lack of correlation might be that creativity has multiple definitions related to specific fields where the creativity test considers different aspects of creativity from specific aspects of creativity only in CS. The study results suggested three aspects of creativity could be manifested by students:

- design: a student can produce creative designs that meet requirements within certain constraints;
- problem solving: a problem solution should be original;
- knowledge acquisition: a student should possess various concepts of data structure, algorithms, and problem-solving strategies, allowing for a novel solution to be produced.

6.2.2.2 Creativity in Coding

Writing a source code is the phase where a programmer implements designed algorithms and data structures using programming languages that a computer can understand. In this phase, a gifted programmer writes source code that is to be readable by a computer, the programmer, and others. It sounds peculiar to think that writing computer code

can be creative, like writing a poem, where abstraction can be manifested through a sequence of statements written in a symbolic way to be understood by a machine and human at the same time. John Warnock, known for developing the PostScript language, was interviewed by Lammers (1986, p. 52) argues that writing a program writing a program is a challenging task similar to authoring a book, combining ideas and concepts in a way that makes a reader think. Thus, he stated that ideas manifested in the code become the programmer's commodity.

If a programmer does not adhere to programming language rules, then the computer cannot understand the written code. In addition, if a programmer writes a code that cannot be understood when he or she revisits the code after a while when code modification is needed, then the code is also not readable. Similarly, often programmers who develop large-scale software strive to write simple, clear, and consistent code. Thus, code readability is an important aspect during the implementation phase.

The art of code consists of numerous aspects that affect code readability and quality. Writing a small code where redundant statements are removed is considered an important aspect of producing elegant code. Verbose coding can be hard to trace, causing confusion for the reader. For example, a programmer might declare extra unwanted variables or functions. However, in some cases, we cannot write small code, but the code can still be split into functions to reduce complexity. Often, programming a statement that represents a step of algorithms can be written in one statement or can be split. For example, if we need to declare and initiate an array a of size $n=3$ to store integer values of *10, 20,* and *30* using the Java language, then there are multiple statement options, as shown below.

```
1 // declares an array of integers
2  int[] a;
3 // allocates memory for 3 integers
4 a = new int[3];
5 // initialise first element where the first index of the array is 0
6 a[0] = 10 ;
7 // initialise first element
```

```
a[1] = 20 ;
// initialise third element
a[2] = 30;
```

Listing 6.1: First option of declaring and initialising an array.

```
// create and initiate an array of integers
int[] a = new int[]{10,20,30};
```

Listing 6.2: Second option of declaring and initialising an array.

It is obvious that the second option is shorter, but what if we asked the code to store *1000* integers? If we use the first option, then *1000* lines of initiation would be written. In this case, even the second option would not be efficient. Thus, an iteration should be used to store *1000* values.

```
// create and initiate an array of integers
int[] a = new int[1000];
// using iterative for loop to store integers in the array
  for (int i=0; i<999; i++)
       {
           a[i]= i;
       }
```

Listing 6.3: Initialising an array using iterative loop.

Another aspect of writing readable code is naming variables, classes, objects, and functions. Often a programmer tends to use names that are shortened, are abbreviations, or are meaningless, which would not be clear to a reader. Names should be specific and descriptive, avoiding general names, such as 'tmp' (temporary) or 'get', where using 'readFromFile' is more concrete. Consistency in the name format allows a programmer to specify names with personal style or with style that has been agreed on with other programmers working on a team.

Code optimisation allows a programmer to improve code that is already functioning to be more efficient by modifying classes and functions where small changes can improve code readability and performance. For example, tweaking a few lines can

reduce runtime. However, code optimisation may not be helpful to improve code performance. In some cases, when an applied algorithm is not efficient, a programmer should pause and redesign the algorithms and data structure. Performance can be enhanced by refining iteration, which can affect the runtime and computer memory if the iteration is infinite or if nested iteration is unnecessary in some cases.

Code comments allow a programmer to add explanations of the code by adding a prefix (double slashes in Java) to notify the programming language complier to ignore the comments during execution. Therefore, syntactical errors are not generated when comments are embedded in the code. Comments should be added at the beginning of the code to explain the purpose of the code functionality. In addition, comments can be added before declaring a function to explain the function aims and parameters that need to be passed on and the values that will be returned. Thus, comments should be descriptive and precise.

Code organisation can increase readability, especially for large-scale software development, as numerous classes, objects, and functions are likely to be created. Grouping variables and functions in blocks makes the code easy on the eye and on the brain, as it helps to navigate and locate certain code within large volumes of code. It is important to keep a consistent layout throughout the whole code.

6.2.3 Attitudes and Personal Traits

6.2.3.1 Learning Style

Edsger Dijkstra (cited in (McConnell, 2004)) postulated that, as programming is a human activity, personal traits have been overshadowed for a while, where other characteristics, such as mathematical ability and intelligence, have dominated employers' attention. It could be argued that it is inappropriate to consider personal traits to be required and desired criteria for a profession like programming. However, the nature of programming as a profession might be different in some respects. For example, a programmer could spend long hours working remotely without being supervised to solve challenging tasks. In the context of the work environment, a programmer should

possess certain traits to work alongside colleagues and clients with integrity, honesty, and collaboration.

Certain personal traits not only derive their importance from an industrial context but also from an educational point of view, where students should manifest motivation and curiosity along with traits related to the learning environment and learning style. Learning style theories suggest numerous ways regarding how students learn, perform, and collaborate within a learning environment.

It is important to understand how students can learn in different ways according to learning style and personal traits, as we think that higher education institutions must cater, in terms of programming pedagogy, for all students who might be struggling, coping, or excelling. Thus, educators should understand learning styles and understand the psychology, epistemology, and methodology of acquiring programming to provide multiple patterns of learning that can cater to all students.

6.2.3.2 Challenging Tasks

In the educational context, programming assignments could have a constant level of difficulty, assuming that students share a similar level of knowledge, ability, and acceptance of challenging tasks. Of course, designing assessments is a separate issue, where multiple factors are involved, such as marking criteria and pedagogy. However, including challenging tasks could boost gifted student motivation, rather than leaving students bored while solving tasks that were introduced or solved before. Those advanced tasks that may be designed to build student knowledge or to introduce an advanced topic can be unassessed if student grades might be affected, or these types of assignments could be considered 'bonus marks'.

Another implication could be to introduce a research project or open-ended assignment where a gifted student could be allowed to expand boundaries as far as possible. In addition, involving gifted students in programming and mathematics competitions could provide enriched experiences derived from a competitive environment. A further pedagogical implication is to introduce changeable programming tasks de-

rived from another field of science, such as biology. Interdisciplinary tasks could bring new elements of motivation and new challenges.

We believe that any pedagogical implication that can boost student motivation, learning, and experience should be inclusive to allow any student to take part in any activity. My view is that it is possible that we have a mixture of students within any class studying one curriculum, where students who are struggling, coping, or excelling might be provided with the same learning and experience. As we should cater for all students, CS educators devote great effort to address issues around struggling students; yet, gifted students might struggle to excel more or to discover great potential.

6.2.3.3 Communication Skills

There is a great emphasis from an industrial IT perspective on communication skills that allow a programmer to communicate ideas, design, and implementation to colleagues and clients (Li et al., 2015; Joseph, 2015; McConnell, 2004). Thus, a programmer can be integrated and engaged within a working team. It would be interesting to see how communication skills can be developed through pedagogy to meet industry requirements for programmers. Communicating technical concepts to a general audience can be challenging. Moreover, explaining complex programming problems to colleagues requires good communication skills. Another form of communicating ideas, design, and implementation is writing code and technical reports.

A large body of literature derived from both educational and industrial perspectives emphasises communication skills as desired skills for hiring a programmer. Moreover, CS students understand the need and significance of oral and writing skills for employment, as they rate communication skills to be the second most important employment criteria after technical skills (Chinn & Vandegrift, 2008). Nonetheless, the nature of studying CS involves spending a considerable amount of time solving technical issues where student communications skills might be affected because of the lack of social interactions and oral communication. It has been recommended by the National Association of Colleges and Employers (NACE) that oral communications skills should

be considered for CS curricula (Havill & Ludwig, 2007).

In addition, writing in CS is an important skill to communicate ideas, designs, and technical issues to CS professionals 'writing to discipline' and to a public audience. Moreover, writing has become a mandatory employment skill and curriculum standard, as suggested in computing curricula (Hoffman et al., 2006). In CSE, writing has a different purpose, including critical thinking enhancement, communicating algorithms through coding, and learning activities. Thus, coding can be considered a mode of expression in which a programmer communicates ideas and 'algorithms' to both machine and humans.

6.2.4 Curiosity

In his Turing award lecture 'The Humble Programmer', Edsger Dijkstra stated: 'The competent programmer is fully aware of the strictly limited size of his own skull' and that programming can be considered a human activity for compensating our skull limitations (Dijkstra, 1972). In addition, a humble programmer should approach programming activity accepting that he or she might be humiliated if the programming problem required ability, knowledge, and/or skills beyond their capacity. Being a humble programmer means that programmers should be curious and be tough on themselves to learn and acquire knowledge and skills. Thus, curiosity can trigger a programmer to learn and keep up with rapidly evolving technology, languages, and tools.

Brooks (1995) emphasised learning as an important characteristic when listing the joys of the craft in his book *The Mythical Man-Month*, as learning could contribute to self-satisfaction and increase programmer motivation. Therefore, a programmer should be humble to know that programming is a difficult task (Dijkstra, 1972) that requires curiosity and being a constant learner through different forms of learning, such as reading, practising, and being involved in new software development projects.

6.2.5 Persistency

Programming could be a dull task requiring time to unpack complicated problems or to fix complex code bugs. Even during the process of learning theoretical concepts of programming, students may lose interest in acquiring often abstracted and sophisticated concepts that are required to solve programming problems. What makes a difference in being a gifted programmer is being persistent, as quitting what one is studying, programming, or debugging is not an option. If more domain knowledge, skills, and/or technology are required to overcome obstacles, then the key is to keep learning and practising.

One aspect of mastering musical instruments is constant practice, and a similar aspect could be applied to programming, especially in coding and debugging. To draw an analogy between programming and music, there is a suggestion that a programmer might be a good musician. Although we have not come across rigorous research supporting this suggestion, composing music requires abstraction and considerable practice, which indicates a musician must be persistent. Inquiries of the relationship between mathematics and music have been conducted. A study conducted by Schmithorst and Holland (2004) investigating the neural correlations between musical training and mathematics performance, which concluded that musical training activated the left fusiform gyrus and prefrontal cortex. The study hypothesised that musical training and mathematical performance correlated with improved working memory and improved abstract representation of numbers. As the relationship between programming and mathematics has becomes obvious, programming might inherit this relationship with music in some way, which could be an interesting field of inquiry. Three of the nine gifted participants in this study mentioned playing musical instruments as a hobby, but this could not be generalised for all participants, as video gaming was a popular hobby among other participants.

6.2.6 Passion

The normal daily task of programming might become tedious and boring, yet a gifted programmer keeps tackling the task, solving problems with enthusiasm and passion. That could be applied to general human activities, including learning programming. If programming is approached as a hobby, that would generate motivation to keep programming and to be good at it. Being passionate about programming would be a desired personal trait for IT industry employment, where a programmer may spend long hours programming. In the context of education, where the study load may be enormous, passionate students may turn spare time into time spent developing their own programming projects.

The findings from our research indicate that some participants do study programming during summer vacations, based on their own desire to learn a new programming language. In addition, some participants spent their spare time comparing the performance of different algorithms or developing their own projects. One participant created his own computer program to help him solve mathematical tasks.

Weisfeld (2013) stated that a great programmer should be passionate, persistent, and creative. In addition, the author discussed how programmers may share traits of being casual, informal, and friendly, which may contribute to building a successful small IT enterprise that could start from 'the garage'.

Summary

In this chapter, we discussed our findings and introduced a model consisting of three profiles that could indicate gifted students in programming. These profiles, which are mathematical ability, creativity, and personal traits, consist of characteristics that can be manifested as a combination of profiles or as a single profile. Mathematical ability in turn consists of three characteristics, reasoning, problem solving, and abstraction, which play an important role in programming and can be exhibited by gifted students in different forms related to programming. The second profile of creativity can be man-

ifested in different forms in relation to programming. Gifted students can be creative in designing algorithms that can be efficient and original as well as in coding by writing readable and elegant code. The third profile is personal traits, which include important characteristics related to learning style communication, curiosity, persistency, and passion.

CHAPTER 7. Conclusion

This final chapter provides a summary of the study aim and the main findings from addressing the research questions. This is followed by a summary of the study significance and contributions to the wider CSE literature and practitioners. The limitations of this study are outlined, along with the ways in which these may be considered in future studies. Future work and considerations for key stakeholders are presented.

7.1 Summary of the Study and Main Findings

The aim of this study was to investigate characteristics related to giftedness in programming through gifted student programmers in the Department of Computer Science at Warwick University. The study aim derived from an attempt to define giftedness in programming to establish a theoretical foundation in CSE. Giftedness in a specific context is the cornerstone of identifying gifted students for enrichment and acceleration programmes. The literature of computer science education revealed little support for gifted student programmers through differentiated curricula or competitions. A lack of theories on giftedness in programming could cause misunderstandings that might affect gifted student identification and lead to inappropriate interventions.

This thesis began by introducing the study background in terms of motivation, research questions, context, and significance. Relevant literature on gifted education and the psychology of programming was examined to understand the ontological and epistemological views of both disciplines. Moreover, characteristics related to programming were highlighted, including mathematical abilities, abstraction, problem solving, personal traits, learning style, IT industry-related characteristics, and coding strate-

gies.

Although mathematical ability was the most obvious and relevant characteristic suggested by the literature, there was a lack of statistically significant proof of the relationship between programming and mathematics, which leads to the first research question and sub-questions. A quantitative approach was used to test the hypothesis for the relationship between programming and mathematics, and the PPMCC test was used along with descriptive statistics to analyse student grades. A statistically significant positive correlation was found between mathematics and programming abilities, and our dataset was the largest among relevant recent research.

The study evolved to investigate the sub-questions of the first research question. Correlations between grades for different mathematics and programming modules were examined, showing a positive correlation between DM and programming modules and a positive correlation between calculus and programming modules.

A case-study methodology was used to answer the rest of the research questions. The second research question addressed students' perceptions of mathematics and programming relationship. Most participants recognised the relationship and understood how certain mathematics are implemented when learning programming. Mathematical graph theory was implemented in CS133 programming module whereas some mathematical theories were not related to the introductory programming module. Two participants mentioned that the relationship might depend on the type of programming paradigm.

The investigation of a set of characteristics suggested by CSE and IT professional literature were addressed in RQ 3, 4, 5 and 6. Those characteristics, which have not been investigated in the context of giftedness and education, included mental representation, knowledge of mathematics and programming, coding strategies, and attitudes and personal traits. We collected quantitative and qualitative data to investigate whether gifted programmers possess certain characteristics. Several data collection methods were used, including student grades, interviews, code-writing problems, and a Witter project analysis.

In relation to the question: 'What mental representation strategies do gifted students possess?' multiple data sources found that participants possessed either one or a combination of abilities including problem solving and abstraction. Participants constructed a mental representation of Witter by understanding the requirements and by explaining their approach and their decision to use a particular data structure or algorithm. Some participants shared similar problem-solving strategies by breaking down a problem into smaller, achievable tasks, while others tried to find an analogy to another previously solved problem. Two participants exhibited abstraction ability based on solving the difficult recursion code-writing problem, but most participants failed to produce valid answers.

In terms of the research question related to mathematics and programming knowledge, participants varied in their mathematics experience, with all studying A-level and first-year mathematics modules. All participants achieved high grades in the A-levels, but their performances at university varied. In addition, participant perceptions of the relationship between mathematics and programming were different.

The participant knowledge in CS and in programming during their early education was limited to basic ICT topics with the exception of a few who pursued their own interests in learning programming and expanding their knowledge. One participant with A-level-equivalent high school education was introduced to advanced algorithms and data structures in his home country. All participants were introduced to two programming modules during their first year of university. They acquired basic programming concepts and advanced data structure and algorithms.

Evidence on coding strategies gathered from interviews and the Witter project indicated that participants possessed some coding strategies, including code comments, organisation, optimisation, and debugging. Most participants adhered to code commenting standards by placing a header description on each code file that explained the purpose of the code and by including in-text comments that explained specific lines of code. Some participants were able to write an organised code that was easy to read and to trace. Participants demonstrated their ability to optimise other code and to refine

their code. In terms of debugging strategies, some participants searched the Internet to find an explanation for the complier error message, while others used printing lines to trace an error or more sophisticated approaches, such as Java exceptions.

As far as attitudes and personal traits are concerned, participants possessed single or combination of several attitudes and traits. In relation to learning styles, the study found that most participants adopted a trial-and-error strategy and problem-solving sheets to learn programming. In addition, some participants learned by attending lectures, with provided material, and problem-solving exercises were also beneficial. The findings suggest that a few participants did not prefer to work in pairs (paired programming) during lab sessions because of a perceived mismatch in learning style, pace, or fear of losing control of implementing different solutions. The study also indicated that including challenging tasks might increase participant engagement. Most participants thought that solving challenging tasks could be rewarding and motivating, but perspectives were mixed about whether these challenging tasks should be formally assessed.

Their communication skills varied, with some who expressed their ideas well through verbal communication and others who preferred to communicate in writing. It was difficult for some participants to explain complex technical situations to their peers, and some felt they would benefit from drawing.

Most participants were passionate about programming from an early age, and some had taught themselves programming. One participant was not passionate about programming (not internally motivated), yet he achieved good grades in programming modules, perhaps due to external motivation of achieving high academic performance or a lifelong goal of obtaining a well-paying job.

In the early stages of this study, the theory of the three-ring conception was adopted to provide a starting point for a giftedness theoretical framework to identify participants based on their academic performance. However, this identification method measured only one aspect of the theory: above-average ability but not creativity. In addition, the theory did not provide a definition for creativity that could be applied in

the context of programming. Based on the findings from this study, some participants manifested a single characteristic, which made the three-ring conception an unsuitable theoretical base for giftedness in programming.

Therefore, we introduced a model consisting of three general profiles with sub-characteristics: mathematical ability, creativity, and personal traits, in which gifted student programmers might manifest combinations of profiles and sub-characteristics. The profile of mathematical abilities includes reasoning, problem solving, and abstraction, whereas creativity refers to creativity in programming and in coding. Personal traits include learning styles, persistency, curiosity, and passion. The profiles and sub-characteristics relate to single or multiple gifted student programmer abilities that can be exhibited through single or multiple programming activities. Specific activities can be used as an identification method, including academic performance, class activities, problem-solving assignments, mathematical assessments, teacher observations, self-nomination, or software development projects.

7.2 Strengths of the Study

This study contributes to the CSE literature, filling a gap in previous research on gifted education within the context of programming. This study provides a unique investigation of gifted student programmers with a focus on specific characteristics. The literature suggested several general characteristics related to programming. However, this study examined selected characteristics based on specific contexts of education and giftedness, which give this study strength. The study added to the literature on gifted student programmers, which could enrich CS teachers' understanding and practice on how to identify gifted students to include them in special education programmes. Knowing these characteristics allows teachers to choose appropriate identification methods for specific characteristics; thus, all potential students can be supported.

Another contribution of this study is the statistical investigation of the relationship between programming and mathematics in a large dataset, compared with previous studies. The investigation evolved to examine a specific relationship between

different DM, calculus, and programming modules. To understand how programming and mathematics can be related and to unpack the statistical results, student perceptions of the relationship were collected via interviews. These findings could provide CS educators with an overview of how students perceive studying mathematics modules and how certain mathematical concepts can be implemented for specific programming concepts. As a study, it has several strengths:

- Provides a breadth of literature on gifted education and programming education;
- Adopts multiple methodologies, allowing for quantitative and qualitative data to be collected through different methods;
- Implements methodological triangulation, thus validity and trustworthiness were increased;
- Provides a detailed explanation of data analysis procedures for each method that can be replicated by other researchers.

7.3 Limitations of the Study

Despite the contribution and significance of this study, several challenges and limitations were encountered during different stages. As the case study was limited to a small population and specific context, the interview data collection stage encountered issues related to time constraints and a lack of participants. Possible participants were identified based on first-year programming academic performance. Thus, the identification process was delayed until the end of the academic year 2014/15 for both programming module results. However, during that period, we conducted our investigation to address the first research question using numerical grade data gathered for all previous cohorts from 1996 to 2014.

Later, the 2015 cohort data were also included, and the investigation of the general correlation between programming and mathematics was repeated. As the case study focused on a small population of gifted students, which is the nature of a qual-

itative approach, investigating nine participants with in-depth data collected through different methods was far better than using a shallow quantitative approach.

As this study was limited to unique cases, generalisation must be done cautiously. However, including multiple-case studies could increase generalisation, and analytical generalisation of multiple cases could contribute to wider theory (Yin, 2009, p. 15).

The study encountered challenges during the qualitative analyses. Lack of rigorous analysis procedures in previous CSE research analysing student programming code highlighted a challenge to incorporate different analysis procedures in this study to increase validity and reliability. To overcome this concern, we adopted a previous procedure proposed by Whalley et al. (2011), which appeared to be the most rigorous analysis procedure derived from the SOLO theory (the framework used in this study). In addition, the analysis was repeated by two other independent researchers to increase the reliability of the data analysis. The limitation of the code-writing problems was that the problems were extracted from exam questions and limited to measure certain programming ability.

The exam setting may have affected students' answers. In addition, it would have been beneficial if the problem were tailored for this study. However, time constraints and difficulties in approaching some participants were the obstacles to designing the problems.

7.4 Future Work

Future work includes evaluating, refining, and implementing the model. Evaluating the model in another context would expand or modify the model profiles and its sub-characteristics; for example, a different context might provide different insight derived from different programming pedagogy and student abilities. Refining the model would have positive effects in general for developing the theory of defining giftedness in programming.

We plan to implement the model in developing an online platform for identifying and supporting gifted student programmers. Implementing the online platform allows

educators to identify gifted student programmers in the early stages, and a clear and prompt intervention can be introduced.

The platform will incorporate multiple programming characteristics along with different identification procedures. Each procedure is suitable for identifying specific characteristics. The identification process will maximise the inclusion of gifted students by categorising them into different levels. Each category will be provided with enriched and tailored material. The platform will include programming tasks that have different stages of difficulty to allow students to evolve to the next stage of tasks.

7.5 Key Considerations

This section provides key considerations and questions that might need to be addressed by educational policy makers, instructional designers, and practitioners.

1. Educational policy makers

    - Would different paths of study and acceleration procedures maximise student academic progress?
    - Would institution and private sector collaboration help develop the experience of gifted students?

2. Instructional designers

    - Would teaching mathematics modules by CS faculty ensure that students see the connection between mathematics and programming through real-life problems?
    - A possible method of ensuring the connection is obvious is to show how mathematical problems can be solved using specific algorithms or how mathematical concepts can be used to analyse or prove data structure efficacy;
    - Lab sheets and tutorial problem sheets should be designed to build student knowledge and should include challenging tasks to be solved as homework, which may be awarded with bonus grades;

- Would student motivation be increased by introducing interdisciplinary problems related to different science disciplines?

3. Practitioners

- The process of identifying gifted student programmers should be based on a variety of identification methods to capture single or combined characteristics and to increase the number of potential gifted students;
- Academic performance can be used to identify giftedness; yet, gifted programmers can be identified based on teacher observations, programming tasks, personal traits, self-nomination, software development, and mathematical ability;
- Enrichment strategies could include extracurricular activities related to programming and mathematics, competitions, and professional placement programmes;

References

Aaby, A. A. (2004). *Theory introduction to programming languages.* Creative Commons Attribution 2.0 Generic. Retrieved 23-02-2017, from `http://www.freetechbooks.com/introduction-to-programming-languages-t84.html`

Albon, R., & Jewels, T. (2008). Gifted university students: last chance to 'come out of the closet'. In *Proceedings of the 10th Asia-Pacific Conference on Gifted Education, Singapore.*

Almstrum, V. L., Hazzan, O., Guzdial, M., & Petre, M. (2005). Challenges to computer science education research. *SIGCSE Bulletin*, *37*(1), p. 191-192.

Baldwin, D., Walker, H. M., & Henderson, P. B. (2013). The roles of mathematics in computer science. *ACM Inroads*, *4*(4), p. 74-80.

Beck, L. L., & Chizhik, A. W. (2008). An experimental study of cooperative learning in CS1. *SIGCSE Bulletin*, *40*(1), p. 205-209.

Bennedsen, J., & Caspersen, M. E. (2005). An investigation of potential success factors for an introductory model-driven programming course. In *Proceedings of the 1st International Workshop on Computing Education Research, Seattle, US* (p. 155-163).

Bennedsen, J., & Caspersen, M. E. (2006). Abstraction ability as an indicator of success for learning Object-Oriented Programming? *SIGCSE Bulletin*, *38*(2), p. 39-43.

Bennedsen, J., & Caspersen, M. E. (2007). Failure rates in introductory programming. *SIGCSE Bulletin*, *39*, p. 32-36.

Bennett, V. E., Koh, K., & Repenning, A. (2013). Computing creativity: divergence in computational thinking. In *Proceeding of the 44th ACM Technical Symposium*

on Computer Science Education, Denver, US (p. 359-364).

Bergin, S., & Reilly, R. (2005a). The influence of motivation and comfort-level on learning to program. In *Proceedings of the 17th Workshop of the Psychology of Programming Interest Group, University of Sussex, UK* (p. 293-304).

Bergin, S., & Reilly, R. (2005b). Programming: Factors that influence success. *SIGCSE Bulletin*, *37*(1), p. 411-415.

Biggs, J. B. (1999). *Teaching for Quality Learning at University.* Buckingham: Open University Press, UK.

Biggs, J. B., & Collis, K. F. (1982). *Evaluating the Quality of Learning: The SOLO Taxonomy Structure of the Observed Learning Outcome.* Academic Press, NY, US.

Binstock, A. (2012). *What makes great programmers different?* Retrieved 01-04-2014, from `http://www.drdobbs.com/architecture-and-design/what-makes-great-programmers-different/240001472`

Blackwell, A. (2002). What is programming. In *Proceedings of the 14th Workshop of the Psychology of Programming Interest Group, Brunel University, UK* (p. 204-218).

Blaikie, N. (2009). *Designing Social Research.* Polity Publisher, Cambridge, UK.

Bloom, B. (1982). The role of gifts and markers in the development of talent. *Exceptional Children*, *48*(6), p. 510-522.

Boden, M. (2004). *The Creative Mind: Myths and Mechanisms.* Routledge.

Boesch, C., & Steppe, K. (2011). Case study on using a programming practice tool for evaluating university applicants. In *Proceedings of the of 2nd International Conference on Computer Science Education: Innovation and Technology, Singapore.*

Bonoma, T. V. (1985). Case research in marketing: opportunities, problems, and a process. *Journal of marketing research*, *22*(2), p. 199-208.

Bornat, R. (2014). *Camels and humps: a retraction.* Retrieved 21-11-2017, from `http://www.eis.mdx.ac.uk/staffpages/r\_bornat/papers/`

Brabrand, C., & Dahl, B. (2009). Analyzing CS competencies using the SOLO taxonomy. In *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation*

and Technology in Computer Science Education, Paris, France.

Bracken, B. A., & McCallum, R. S. (1998). *Universal Nonverbal Intelligence Test.* Riverside Publishing, IL, US.

Braun, V., & Clarke, V. (2006). Using thematic analysis in psychology. *Qualitative Research in Psychology*, *3*(2), p. 77-101.

Bronish, D. (2012). *Abstraction as the Key to Programming, with Issues for Software Verification in Functional Languages.* The Ohio State University, US.

Brooks, F. (1995). *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition.* Pearson, London, UK.

Bryman, A. (2012). *Social Research Methods.* Oxford University Press, UK.

Campbell, D. T. (1975). III. "degrees of freedom" and the case study. *Comparative Political Studies*, *8*(2), p. 178-193.

Carter, J., Efford, N., Jamieson, S., Jenkins, T., & White, S. (2007). The TOPS project - teaching our over performing students. In *Proceedings of the 8th Annual Conference of the Higher Education Academy for Information and Computer Science, UK.*

Carter, J., White, S., Fraser, K., Kurkovsky, S., McCreesh, C., & Wieck, M. (2010). ITiCSE 2010 working group report motivating our top students. In *Proceedings of ITiCSE, Ankara, Turkey* (p. 29-47).

Chinn, D., & Vandegrift, T. (2008). Uncovering student values for hiring in the software industry. *Educational Resources in Computing*, *7*(4), p. 1-25.

Christie, M., Rowe, P., Perry, C., & Chamard, J. (2000). Implementation of realism in case study research methodology. In *International Council for Small Business, Annual Conference, Brisbane, Australia* (p. 1-21).

Cohen, L., Manion, L., & Morrison, K. (2003). *Research Methods in Education.* Routledge, UK.

Cohen, L., Manion, L., & Morrison, K. (2011). *Research Methods in Education.* Routledge, UK.

Colangelo, N., Assouline, S. G., & Gross, M. U. (Eds.). (2004). *A nation deceived:*

How schools hold back america's brightest students. the templeton national report on acceleration (Vol. 2). ERIC.

Colangelo, N., & Davis, G. (2002). *Handbook on Gifted Education.* Allyn & Bacon, MA, US.

Coleman, M. R. (2003). The identification of students who are gifted. *The ERIC Clearinghouse on Disabilities and Gifted Education. The Council for Exceptional Children, US*, p. 2-6.

Creswell, J. W. (2009). *Research Design Qualitative, Quantitative, and Mixed Methods Approaches.* SAGE.

Creswell, J. W., & Clark, V. L. P. (2011). *Designing and Conducting Mixed Methods Research.* SAGE.

Creswell, J. W., Clark, V. L. P., Gutmann, M. L., & Hanson, W. E. (2003). Advanced mixed methods research designs. In A. Tashakkori & C. Teddue (Eds.), *Handbook of Mixed Methods in Social and Behavioral Research* (p. 209-240). SAGE.

Davis, H., Carr, L., Cooke, E., & White, S. (2001). Managing diversity: Experiences teaching programming principles. In *Proceedings of the 2nd LTSN-ICS Annual Conference, UK.*

Deci, E., & Ryan, R. (1985). *Intrinsic Motivation and Self-Determination in Human Behavior.* Springer, US.

Decker, A., & Ventura, P. (2004). We claim this class for computer science: A non-mathematician's discrete structures course. *SIGCSE Bulletin*, *36*(1), p. 442-446.

Dehnadi, S. (2009). *A cognitive study of learning to program in introductory programming courses* (Unpublished doctoral dissertation). Middlesex University.

Dehnadi, S., & Bornat, R. (2006). The camel has two humps. In *Proceedings of the 18th Workshop of the Psychology of Programming Interest Group, University of Sussex, UK.*

DFE. (2008). *Identifying gifted and talented learners - getting started.* Department for Education. Retrieved 06-09-2016, from `http://www.education.gov.uk`

DiCicco-Bloom, B., & Crabtree, B. F. (2006). The qualitative research interview.

Medical Education, *40*(4), p. 314-321.

Dick, B., & Dick, R. (1990). *Convergent Interviewing*. Interchange, Australia.

Diener, E., & Crandall, R. (1978). *Ethics in Social and Behavioral Research*. University of Chicago Press, US.

Dijkstra, E. W. (1972). The humble programmer. *Communications of the ACM*, *15*(10), p. 859-866.

Elo, S., & Kyngäs, H. (2008). The qualitative content analysis process. *Journal of Advanced Nursing*, *62*(1), p. 107-115.

Evans, J. D. (1996). *Straightforward Statistics for the Behavioral Sciences*. Brooks/ Cole, CA, US.

Eyre, D. (1997). Teaching able pupils. *Support for Learning*, *12*(2), p. 60-65.

Eyre, D. (2011). *Room at the Top: Inclusive Education for High Performance*. Policy Exchange, London, UK.

Fasko, D. (2001). An analysis of multiple intelligences theory and its use with the gifted and talented. *Roeper Review*, *23*(3), p. 126-130.

Fincher, S., & Petre, M. (2004). *Computer Science Education Research*. Taylor & Francis, UK.

Floyd, R. G., Evans, J. J., & McGrew, K. S. (2003). Relations between measures of Cattell-Horn-Carroll (CHC) cognitive abilities and mathematics achievement across the school-age years. *Psychology in the Schools*, *40*(2), p. 155-171.

Fuller, U., Johnson, C. G., Ahoniemi, T., Cukierman, D., Hernán-Losada, I., Jackova, J., ... Thompson, E. (2007). Developing a computer science-specific learning taxonomy. *SIGCSE Bulletin*, *39*(4), p. 152-170.

Gagné, F. (1985). Giftedness and talent: Reexamining a reexamination of the definitions. *Gifted Child Quarterly*, *29*(3), p. 103-112.

Gagné, F. (2000). *A differentiated model of giftedness and talent (DMGT).* Retrieved 12-08-2014, from `http://campbellms.typepad.com/files/gagne-a-differentiated-model-of-giftedness-and-talent-dmgt.pdf`

Galpin, V. C., Sanders, I. D., & Chen, P.-y. (2007). Learning styles and personal-

ity types of computer science students at a South African University. *SIGCSE Bulletin*, *39*(3), p. 201-205.

Gardner, H. (1987). The theory of multiple intelligences. *Annals of Dyslexia*, *37*(1), p. 19-35.

Gardner, H. (2003). Multiple intelligences after twenty years. In *The American Educational Research Association.*

Gardner, H. (2006). *Multiple Intelligences: New Horizons in Theory and Practice.* Basic Books, NY, US.

Gerring, J. (2004). What is a case study and what is it good for? *The American Political Science Review*, *98*(2), p. 341-354.

Ginat, D. (2004). Do senior CS students capitalize on recursion? *SIGCSE Bulletin*, *36*(3), p. 82-86.

Gomes, A., & Mendes, A. J. (2008). A study on student's characteristics and programming learning. In *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications, Vienna, Austria* (p. 2895-2904).

Gray, D. E. (2006). *Doing Research in the Real World.* SAGE.

Gray, D. E. (2013). *Doing Research in the Real World.* SAGE.

Green, T. R. (1989). Cognitive dimensions of notations. In Sutcliffe & L. Macaulay (Eds.), *People and Computers V* (p. 443-460).

Han, J., & Beheshti, M. (2010). Enhancement of computer science introductory courses with mentored pair programming. *Computing Sciences in Colleges*, *25*, p. 149-155.

Harris, J. (2014). Testing programming aptitude in introductory programming courses. *Computing Sciences in Colleges*, *30*(2), p. 149-156.

Hartmanis, J. (1993). Some observations about the nature of computer science. In R. K. Shyamasundar (Ed.), *Foundations of Software Technology and Theoretical Computer Science: 13th Conference Bombay, India* (p. 1-12). Springer Berlin Heidelberg.

Hattie, J., & Purdie, N. (1998). The SOLO model: Addressing fundamental measure-

ment issues. In G. Dart Barry; Boulton-Lewis (Ed.), *Teaching and Learning in Higher Education* (p. 270). Australian Council for Educational Research.

Havill, J. T., & Ludwig, L. D. (2007). Technically speaking: Fostering the communication skills of computer science and mathematics students. *SIGCSE Bulletin*, *39*(1), p. 185-189.

Hazzan, O., & Kramer, J. (2016). Assessing abstraction skills. *Communications of the ACM*, *59*(12), p. 43-45.

Henderson, P. B., & Stavely, A. M. (2014). Programming and mathematical thinking. *ACM Inroads*, *5*(1), p. 35-36.

Hobson, A. J., & Townsend, A. (2010). Interviewing as educational research method(s). In D. Hartas (Ed.), *Educational Research and Inquiry: Qualitative and Quantitative Approaches* (p. 223-238). London: Continuum.

Hoc, J. (2014). *Psychology of Programming*. Elsevier Science.

Hoc, J., & Nguyen-Xuan, A. (1990). Language semantics, mental models and analogy. In J. Hoc (Ed.), *Psychology of Programming* (p. 139-152). Academic Press.

Hoffman, M. E., Dansdill, T., & Herscovici, D. S. (2006). Bridging writing to learn and writing in the discipline in computer science education. *SIGCSE Bulletin*, *38*(1), p. 117-121.

Isotalo, J. (2014). *Basics of Statistics*. CreateSpace Independent Publishing Platform.

Izu, C., Weerasinghe, A., & Pope, C. (2016). A study of code design skills in novice programmers using the SOLO taxonomy. In *Proceedings of the ACM Conference on International Computing Education Research, Melbourne, Australia* (p. 251-259).

Jenkins, T., & Davy, J. (2000). Dealing with diversity in introductory programming. In *Proceedings of the 1st Annual LTSN-ICS Conference, UK* (p. 81-87).

Jensen, A. R. (2002). Galton's legacy to research on intelligence. *Journal of Biosocial Science*, *34*(2), p. 145-172.

Jimoyiannis, A. (2013). Using SOLO taxonomy to explore students' mental models of the programming variable and the assignment statement. *Themes in Science*

and Technology Education, *4*(2), p. 53-74.

Johnson, D., on Disabilities, E. C., & Education, G. (2000). *Teaching Mathematics to Gifted Students in a Mixed-ability Classroom*. ERIC Clearinghouse on Disabilities and Gifted Education, The Council for Exceptional Children, US.

Joseph, S. (2015). *Programmer competency matrix*. Retrieved 09-02-2015, from `http://sijinjoseph.com/programmer-competency-matrix/`

Kanij, T., Merkel, R., & Grundy, J. (2013). An empirical study of the effects of personality on software testing. In *Proceedings of the 26th International Conference on Software Engineering Education and Training, San Francisco, US* (p. 239-248).

Kim, J. (2015). *Understanding narrative inquiry: The crafting and analysis of stories as research*. SAGE.

King, N., & Horrocks, C. (2010). *Interviews in Qualitative Research*. SAGE.

Knuth, D. (1992). *Personal communication*. Letter.

Koppelman, H., & van Dijk, B. (2010). Teaching abstraction in introductory courses. In *Proceedings of the 15th Annual Conference on Innovation and Technology in Computer Science Education, Ankara, Turkey* (p. 174-178).

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

Kurland, D. M., Pea, R. D., Clement, C., & Mawby, R. (1986). A study of the development of programming ability and thinking skills in high school students. *Journal of Educational Computing Research*, *2*(4), p. 429-458.

Lafore, R. (2017). *Data Structures and Algorithms in Java*. Pearson, London, UK.

Lahtinen, E., Ala-Mutka, K., & Järvinen, H.-M. (2005). A study of the difficulties of novice programmers. *SIGCSE Bulletin*, *37*(3), p. 14-18.

Lammers, S. (1986). *Programmers at Work*. Harper & Row Publishers, UK.

Leblanc, M. D., & Leibowitz, R. (2006). Discrete partnership: A case for a full year of discrete math. *SIGCSE Bulletin*, *38*(1), p. 313-317.

Li, P. L., Ko, A. J., & Zhu, J. (2015). What makes a great software engineer? In *Proceedings of the 37th International Conference on Software Engineering,*

Florence, Italy (p. 700-710).

Lister, R. (2011). Concrete and other neo-Piagetian forms of reasoning in the novice programmer. In *Proceedings of the 13th Australasian Computing Education Conference, Perth, Australia* (p. 9-18).

Lister, R. (2016). Toward a developmental epistemology of computer programming. In *Proceedings of the 11th Workshop in Primary and Secondary Computing Education, Münster, Germany* (p. 5-16).

Lister, R., Clear, T., Simon, Bouvier, D. J., Carter, P., Eckerdal, A., ... Thompson, E. (2010). Naturally occurring data as research instrument: Analyzing examination responses to study the novice programmer. *SIGCSE Bulletin*, *41*(4), p. 156-173.

Lister, R., Simon, B., Thompson, E., Whalley, J. L., & Prasad, C. (2006). Not seeing the forest for the trees: Novice programmers and the SOLO taxonomy. *SIGCSE Bulletin*, *38*(3), p. 118-122.

Marland, S. (1971). Education of the gifted and talented - volume 1: Report to the congress of the united states by the u. s. commissioner of education. *ERIC*, *1*, p. 126.

Mason, J., Burton, L., & Stacey, K. (2010). *Thinking Mathematically*. Pearson, London, UK.

McConnell, S. (2004). *Code Complete*. Microsoft Press, US.

McCormack, J., & d'Inverno, M. (2012). Computers and creativity: The road ahead. In J. McCormack & M. d'Inverno (Eds.), *Computers and Creativity* (p. 421-424). Berlin, Heidelberg: Springer Berlin Heidelberg.

McKeithen, K. B., Reitman, J. S., Rueter, H. H., & Hirtle, S. C. (1981). Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, *13*(3), p. 307-325.

Merriam, S. B. (1988). *Case Study Research in Education: A Qualitative Approach*. Jossey Bass, CA, US.

Merriam, S. B., & Tisdell, E. J. (2009). *Qualitative Research: A Guide to Design and Implementation*. Jossey Bass, CA, US.

Miles, M. B., & Huberman, A. M. (1994). *Qualitative Data Analysis: An Expanded Sourcebook*. SAGE.

Ormerod, T. (1990). Human cognition and programming. In J. Hoc (Ed.), *Psychology of Programming* (p. 63-82). Academic Press.

Owolabi, J., Olanipekun, P., & Iwerima, J. (2014). Mathematics ability and anxiety, computer and programming anxieties, age and gender as determinants of achievement in basic programming. *GSTF Journal on Computing*, *3*(4), p. 47.

Pacheco, A., Gomes, A., Henriques, J., de Almeida, A. M., & Mendes, A. J. (2008). Mathematics and programming: some studies. In *Proceedings of the 9th International Conference on Computer Systems and Technologies, Gabrovo, Bulgaria* (p. 1-15).

Passow, A. H., & Frasier, M. M. (1996). Toward improving identification of talent potential among minority and disadvantaged students. *Roeper Review*, *18*(3), p. 198-202.

Patton, M. Q. (1990). *Qualitative Evaluation and Research Methods*. SAGE.

Petersen, A., Craig, M., Campbell, J., & Tafliovich, A. (2016). Revisiting why students drop CS1. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research, Koli, Finland* (p. 71-80).

Petre, M. (1990). Expert programmers and programming languages. In *Psychology of Programming* (p. 103-113). Academic Press.

Pioro, B. T. (2006). Introductory computer programming: Gender, major, discrete mathematics, and calculus. *Computing Sciences in Colleges*, *21*(5), p. 123-129.

Plucker, J. A., & Beghetto, R. A. (2004). Why creativity is domain general, why it looks domain specific, and why the distinction does not matter. In J. L. S. Robert J. Sternberg Elena Grigorenko (Ed.), *Creativity: From Potential to Realization* (p. 153-168). American Psychological Association.

Polya, G. (2004). *How to Solve It: A New Aspect of Mathematical Method*. Princeton University Press, NJ, US.

Power, J. F., Whelan, T., & Bergin, S. (2011). Teaching discrete structures: A

systematic review of the literature. In *Proceedings of the 42th ACM Technical Symposium on Computer Science Education, Dallas, US* (p. 275-280).

Punch, K. (2009). *Introduction to Research Methods in Education.* SAGE.

Ralston, A. (2005). Do we need any mathematics in computer science curricula? *SIGCSE Bulletin*, *37*(2), p. 6-9.

Ralston, A., & Shaw, M. (1980). Curriculum '78 - is computer science really that unmathematical? *Communications of the ACM*, *23*(2), p. 67-70.

Razali, N. M., & Wah, Y. B. (2011). Power comparisons of Shapiro-wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling tests. *Journal of Statistical Modeling and Analytics*, *2*(1), p. 21-33.

Reis, S. M., & Renzulli, J. S. (2010). Is there still a need for gifted education? an examination of current research. *Learning and Individual Differences*, *20*(4), p. 308-317.

Renzulli, J. S. (1976). The enrichment triad model: A guide for developing defensible programs for the gifted and talented. *Gifted Child Quarterly*, *20*(3), p. 303-326.

Renzulli, J. S. (1978). What makes giftedness? reexamining a definition. *The Phi Delta Kappan*, *60*(3), p. 180-261.

Renzulli, J. S. (1984a). The three ring conception of giftedness: A developmental model for creative productivity. In *The Annual Meeting of the American Educational Research Association, New Orleans, LA.* ERIC.

Renzulli, J. S. (1984b). The triad/revolving door system: A research-based approach to identification and programming for the gifted and talented. *Gifted Child Quarterly*, *28*(4), p. 163-171.

Renzulli, J. S. (2002). Expanding the conception of giftedness to include co-cognitive traits and to promote social capital. *Phi Delta Kappan*, *84*(1), p. 33-58.

Renzulli, J. S., & Reis, S. M. (2000). The schoolwide enrichment model. In K. Heller, F. Mönks, R. Subotnik, & R. Sternberg (Eds.), *International Handbook of Giftedness and Talent* (p. 367-382). Elsevier Science.

Robertson, S. I. (2017). *Problem Solving: Perspectives from Cognition and Neuro-*

science. Routledge, UK.

Robins, A. (2015). The ongoing challenges of computer science education research. *Computer Science Education*, *25*(2), p. 115-119.

Robinson, N. M. (1997). The role of universities and colleges in educating gifted undergraduates. *Peabody Journal of Education*, *72*(3/4), p. 217-236.

Robson, C. (2004). *Real World Research*. Wiley, US.

Romeike, R. (2007a). Applying creativity in CS high school education: Criteria, teaching example and evaluation. In *Proceedings of the 7th Baltic Sea Conference on Computing Education Research, Koli, Finland* (p. 87-96).

Romeike, R. (2007b). Three drivers for creativity in computer science education. In *Proceedings of the IFIP-Conference on Informatics, Mathematics and ICT: A Golden Triangle, Potsdam, Germany.*

Rossman, G. B., & Wilson, B. L. (1985). Numbers and words combining quantitative and qualitative methods in a single large-scale evaluation study. *Evaluation Review*, *9*(5), p. 627-643.

Ryan, G. W., & Bernard, H. R. (2003). Techniques to identify themes. *Field Methods*, *15*(1), p. 85-109.

Salgian, A., Nakra, T. M., Ault, C., & Wang, Y. (2013). Teaching creativity in computer science. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education, Denver, US* (p. 123-128).

Saunders, D., & Thagard, P. (2005). Creativity in computer science. *Creativity Across Domains: Faces of the Muse*, p. 153-167.

Schmithorst, V. J., & Holland, S. K. (2004). The effect of musical training on the neural correlates of math processing: a functional magnetic resonance imaging study in humans. *Neuroscience Letters*, *354*(3), p. 193-196.

Seyal, A. H., Mey, Y. S., Matusin, M. H., Siau, N. H., & Rahman, A. A. (2015). Understanding students learning style and their performance in computer programming course: Evidence from bruneian technical institution of higher learning. *International Journal of Computer Theory and Engineering*, *7*(3), p. 241.

Sheard, J., Carbone, A., Lister, R., Simon, B., Thompson, E., & Whalley, J. L. (2008). Going SOLO to assess novice programmers. *SIGCSE Bulletin*, *40*(3), p. 209-213.

Simon, Fincher, S., Robins, A., Baker, B., Box, I., Cutts, Q., ... Tutty, J. (2006). Predictors of success in a first programming course. In *Proceedings of the 8th Australasian Conference on Computing Education, Hobart, Australia* (p. 189-196).

Sooriamurthi, R. (2001). Problems in comprehending recursion and suggested solutions. *SIGCSE Bulletin*, *33*(3), p. 25-28.

Sprankle, M., & Hubbard, J. (2012). *Problem Solving and Programming Concepts*. Pearson, London, UK.

Stake, R. E. (1995). *The Art of Case Study Research*. SAGE.

Sternberg, R. J., & Lubart, T. I. (1999). The concept of creativity: Prospects and paradigms. In R. J. Sternberg (Ed.), *Handbook of Creativity* (p. 3-15). Cambridge University Press.

Sutner, K. (2005). CDM: Teaching discrete mathematics to computer science majors. *Educational Resources in Computing*, *5*(2).

Teague, D., & Lister, R. (2014). Longitudinal think aloud study of a novice programmer. In *Proceedings of the 16th Australasian Computing Education Conference, Auckland, New Zealand* (p. 41-50).

Tukiainen, M., & Mönkkönen, M. (2002). Programming aptitude testing as a prediction of learning to program. In *Proceedings of the 14th Workshop of the Psychology of Programming Interest Group, Brunel University, UK* (p. 45-57).

VanDrunen, T. (2017). Functional programming as a discrete mathematics topic. *ACM Inroads*, *8*(2), p. 51-58.

Venkatesan, V., & Sankar, A. (2014). Investigation of student's personality on pair programming to enhance the learning activity in the academia. *Journal of Computer Science*, *10*(10), p. 2020-2028.

Vickers, P. (2008). *How to Think Like a Programmer: Problem Solving for the Bewildered*. Cengage Learning.

Warne, H. (2013). *7 ways more methods can improve your program.* Retrieved 12-08-2014, from `http://henrikwarne.com/2013/08/31/7-ways-more-methods-can-improve-your-program/`

Warne, H. (2014). *What makes a good programmer?* Retrieved 12-08-2014, from `http://java.dzone.com/articles/what-makes-good-programmer`

Watson, C., & Li, F. W. (2014). Failure rates in introductory programming revisited. In *Proceedings of Conference on Innovation and Technology in Computer Science Education, Uppsala, Sweden* (p. 39-44).

Weaver, K., & Olson, J. K. (2006). Understanding paradigms used for nursing research. *Journal of Advanced Nursing*, *53*(4), p. 459-469.

Weisfeld, M. (2013). *Becoming a programming rock star: 5 traits that make a great programmer.* Retrieved 12-10-2014, from `http://www.informit.com/articles/article.aspx?p=2135212`

Wellington, J. (2000). *Educational Research: Contemporary Issues and Practical Approaches.* Biddles, UK.

Welman, J. C., & Kruger, F. (1999). *Research Methodology for the Business and Administrative Sciences.* Oxford University Press, UK.

Whalley, J., Clear, T., Robbins, P., & Thompson, E. (2011). Salient elements in novice solutions to code writing problems. In *Proceedings of the 13th Australasian Computing Education Conference, Perth, Australia* (p. 37-46).

Wilson, B. C., & Shrock, S. (2001). Contributing to success in an introductory computer science course: A study of twelve factors. *SIGCSE Bulletin*, *33*(1), p. 184-188.

Wirth, M. (2014). The canny skipper - a puzzle for demonstrating data structures and recursion. In *Proceedings of the Western Canadian Conference on Computing Education, Richmond, Canada.*

Yin, R. (1994). *Case Study Research: Design and Methods.* SAGE.

Yin, R. (2009). *Case Study Research: Design and Methods.* SAGE.

APPENDIX A. Interview Schedule

Introduction

I want to thank you for taking the time to meet with me today. My name is Ayman and I would like to talk to you about programming. Specifically, my research aims to investigate certain characteristics that programmers might possess. The interview should take less than an hour and fifteen minutes.

I will be recording the interview because I don't want to miss any of your comments (is that OK?). All responses will be kept confidential and the information will be only shared with my supervisors. We will ensure that any information we include in our report does not identify you as the respondent.

Remember, you don't have to talk about anything you don't want to and you may end the interview at any time.

Are there any questions about what I have just explained?

ID: **Date:** **Audio_NO:**

Interview Schedule

T	Main Topic	Main Question	Follow up QA	Probing QA
5 Minutes	About Student	Can you please tell me about yourself?	What schools did you attend?	(e.g. state school, private school)
			Did the schools teach computer science?	Can you give me an example of what did you study?
			Why did you choose to study CS?	(e.g self-interest, family-interest)
			Please describe the ideal job for you following graduation?	Why?
			What are your interests and hobbies?	Playing any musical instruments?

ID: **Date:** **Audio_NO:**

Interview Schedule

10 Minutes	**Academic Background**	**Mathematics**	**Can you tell me about your mathematics background?**	**Have you done (GCSE, A level, IP)?**	**How did you do?**
				What mathematics modules have you done at UNI?	**Can you give me an example of what did you study?**
					What did you think of studying these modules?
					Tell me why did you hate….?
					Tell me why did you enjoy….?

ID: **Date:** **Audio_NO:**

Interview Schedule

10 Minutes	**Academic Background**	**Programming**	**Tell me about your programming background?**	**When did you start programming?** **When you are programming, what does it feel like?** **What programming modules have you done at UNI?** **What kind of programming have you done at home?** **Have you done any professional software development? When?** **What programming languages have you used?** **Do you prefer any particular operating system?** **Do you prefer any particular integrated development environment (IDE)? Open source?**	**Can you explain why you ...?** **What did you think of studying these modules?** **Tell me why did you hate....?** **Tell me why did you enjoy....?** **How many hours do you spend on programming?** **Can you tell me what did you develop?** **Tell me please, why do you prefer ……?** **Why?** **Could you say something more about that** **Tell me why?**

ID: **Date:** **Audio_NO:**

Interview Schedule

10 Minutes	**Mathematics and programming**	**As you study both programming and mathematics, are they related to each other or independent from each other?**	**Could you expand on that point?** **Can you tell me about your thoughts on the relationship between mathematics and programming?**	**If yes, can you think of any example where you found mathematics is helpful in solving a programming problem?** **Which math topics do you usually apply during programming?** **(e.g. algebra, calculus and discrete)** **Does being good at maths help you with programming?** **Can you give an example ?**

ID: **Date:** **Audio_NO:**

Interview Schedule

10 Minutes	**Characteristics**	**Personal**	**Can you tell about your learning style?**	**How do you learn?**	**e.g. self-learner**
				In the lab, how do you prefer to work while programming?	**(e.g. individual, with peer partner) Why?**
				In the lab, what do you do if you become stuck while programming a specific task?	**(e.g. keep trying, ask for help (from who) or quit)**
				Would you prefer to solve challenging tasks or easy tasks?	**why?**
				Would you like to have more challenging tasks in the lab?	
				What sort of programing tasks that make you excited to solve?	**How do you solve it?**
			Creativity	**Tell me about a problem where you felt that the normal solution would not be suitable?**	**An example in programming?**
				Have you ever tried a new way of doing things in general?	

ID: **Date:** **Audio_NO:**

Interview Schedule

5 Minutes	**Characteristics**	**communication**	**Tell me about your communication skills?**	**Tell me about your presentation skills?** **Tell me about a situation when you had to explain complex problem to a friend or tutor.** **Are you a better communicator when writing or speaking?**	**Give an example in programming?** **Tell what make you prefer ….?**

ID: **Date:** **Audio_NO:**

Interview Schedule

5 Minutes	**Characteristics**	**Coding strategies**	**Can you tell about your coding strategies?**	**Tell me about the whole process when writing a program, what do you first?** **Do you organise your code in typical way?** **How frequently do you use comments when you are coding?** **How do you handle programming errors?**	**(e.g. experiment first, doodle)** **Why?**

ID: **Date:** **Audio_NO:**

Interview Schedule

5 Minutes	**Characteristics**	**Mental representation**	**Can you tell what do you do when you asked to solve a problem?**	**What problem-solving strategy do you use?** **Tell me how would you represent a particular software system?**	**Do you break down a project into smaller problems?** **Can you give an example in programming?** **How would you analyze a particular system?** **Would you look at it as an application, set of classes, or algorithm and data structure?**

ID: **Date:** **Audio_NO:**

Interview Schedule

10 Minutes	**Characteristics**	**Knowledge representation**	**Can you tell me how do you decide to choose data structure and algorithms?**	**Do you consider the complexity in terms of time and machine resources? Why?**	**Can you expand on that point?**
5 Minutes	**Conclusion**	**Would you like to add anything else?**			

APPENDIX B. Analysis procedures for the Code-Writing Problems

Instructions:

1. In part one, you need to carefully read Table 1;
2. In part two, you need to provide all valid possible solutions that you can come up with to each question (three questions);
    - You need to extract program constructs, syntax elements, feature for each question;
3. In part three, according to Table 1, you need to categorise student's answers.

Part 1: Introduction to SOLO taxonomy.

SOLO taxonomy is a categorization used to classify student's cognitive ability. Table 1 is the categories to classify student answers for typical programming writing-code questions.

SOLO category	Description
Extended Abstract – Extending [EA]	Uses constructs and concepts beyond those required in the exercise to provide an improved solution
Relational – Encompassing [R]	Provides a valid well structured program that removes all redundancy and has a clear logical structure. The specifications have been integrated to form a logical whole.
Multistructural – Refinement [M]	Represents a translation that is close to a direct translation. The code may have been reordered to make *a more integrated and/or valid solution.*
Unistructural – Direct Translation [U]	Represents a direct translation of the specifications. The code will be in the sequence of the specifications.
Prestructural [P]	Substantially lacks knowledge of programming constructs or is unrelated to the question.

Table 1 SOLO taxonomy for the code-writing problems.

Part 2: Provide all possible solutions using Java

Question 1: Write a method that, when called with a single integer argument, n, creates an array of n integers with random values between 0 and 100 inclusive.

Solution 1 (efficient)	Alternative Solution 2	Alternative Solution 3

`public int[] randomArray(int n){`
`int[] array = new int[n];`
`Random randomGenerator = new Random();`
`for(int i=0;i<n;i++){`
`array[i] =`
`(int)(Math.random() * 101);`
`}`
`return array;`
`}`
	`public int[] randomArray(int n){`
	`int[] array = new int[n];`
	`Random randomGenerator = new Random();`
	`for(int i=0;i<n;i++){`
	`array[i] =`
	`randomGenerator.nextInt(101);`
	`}`
	`return array;`
	`}`	

You need to extract program constructs, syntax elements, features (code quality) for question 1.

Construct	Element	Feature
e.g. array iteration	for loop	Finite Redundant
Method declaration	Public int[] array (int n)	Normal
Array declaration	int [] array = new int[n];	Efficient
Array iteration	1x for loop	Finite loop
Random value generation	Using Math object	Inclusive range
	Using Random object	Exclusive range
Return statement	Return array;	Included

Question 2: Write a method that, when called with an array and an integer argument, s, performs a linear search on the array reporting the array index of the first instance of s in the list, or returning -1 if s is not found in the array.

Solution 1 (efficient)	Alternative Solution 2	Alternative Solution 3

`public int search(int[] array, int s){`
`int index = -1;`
`for(int i=0;i<array.length();i++){`
`if(array[i] == s){`
`index = i;`
`return index;`
`}`
`}`
`return index;`
`}`
	`public int search(int[] array, int s){`
	`for(int i=0;i<array.length();i++)`
	`if(array[i] == s)`
	`return i;`
	`return -1;`
	`}`	

You need to extract program constructs, syntax elements, features (code quality) for question 2.

Construct	Element	Feature
e.g. array iteration	for loop	Finite Redundant
`Method declaration`	`public int linearArray(int [] array, int s)`	`Normal`
`Array iteration`	`for(int i=0;i<s;i++)`	`Finite loop`
`Selection`	`If statement`	`Valid condition`
`Return statement`	`int find = -1;` `Return find;`	`Redundant`

Question 3: write a recursive method that calculates the sum of the differences between opposing pairs (i.e. the difference between A[0] and A[n-1], A[1] and A[n-2], and so forth). For example, the array { 3, 6, 34, 65 } results in the calculation: (65 3) + (34 6) = 90. You may assume the list will always be even in length.

Solution 1 (efficient)	Alternative Solution 2	Alternative Solution 3

`public int arrayPairs(int[] array, int first){`
`int last = array.length() - first - 1;`
`if(last < first)`
`return 0;`
`else`
`return array[last] - array[first] + arrayPairs(array, ++first);`
`}`
	`public int arrayPairs(int[] array, int first){`
	`int last = array.length() - first - 1;`
	`if(last <= first)`
	`return 0;`
	`else{`
	`int difference = array[array.length-index-1] - array[first];`
	`return difference + arrayPairs(array, ++first);`
	`}`
	`}`	

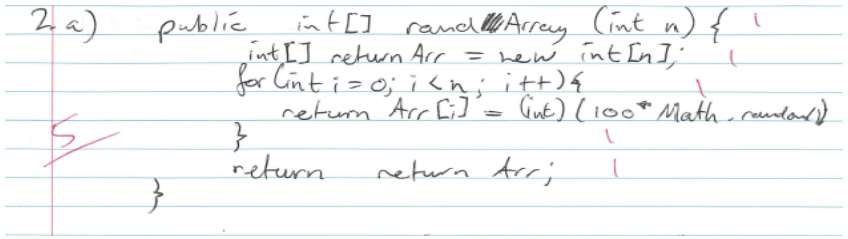
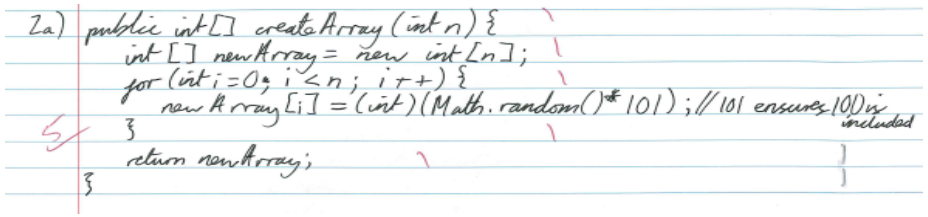
You need to extract program constructs, syntax elements, features (code quality) for question 3.

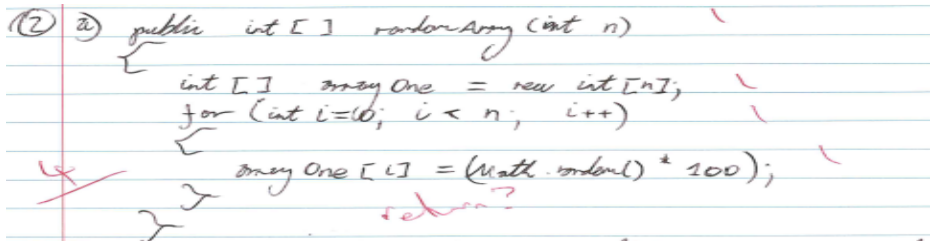
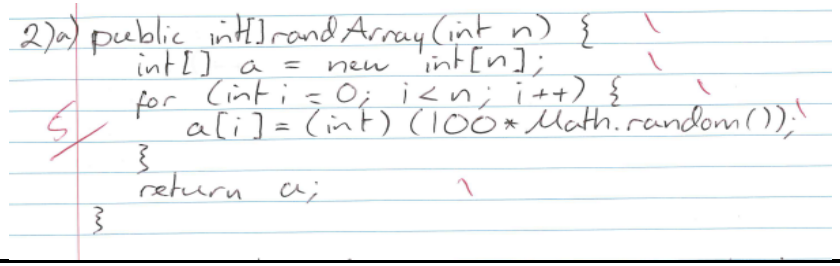
Construct	Element	Feature
e.g. array iteration	for loop	Finite Redundant
Method declaration	`Public int oppPairs(int [] array, int pos)`	Normal
Variable assignment	`int pos2=array.length() -1-pos;`	Efficient
edges	`If (pos2<pos)`	Valid
Difference calculation	`int diff = array[pos2]-array[pos] + oppPairs(array,++pos);`	Efficient
recursive invocation	`oppPairs(array,++pos)`	Valid argument
Return statement	`Return array;`	non-redundant

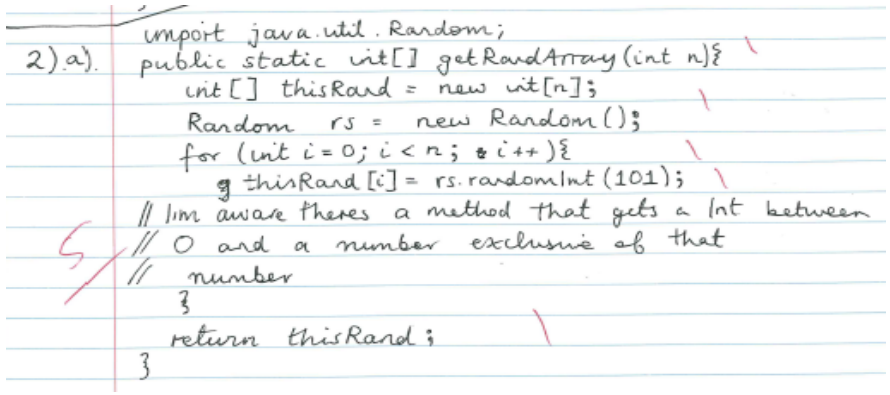
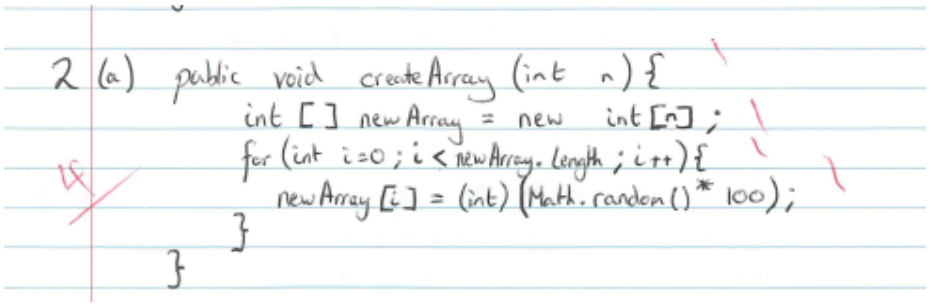
Part 3: For each previous question, you will have 9 students answers that need to be categorised based on Table 1.

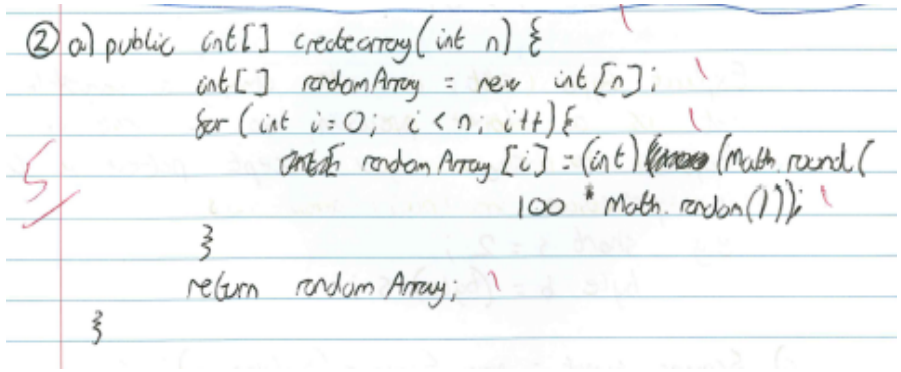
Instructions:

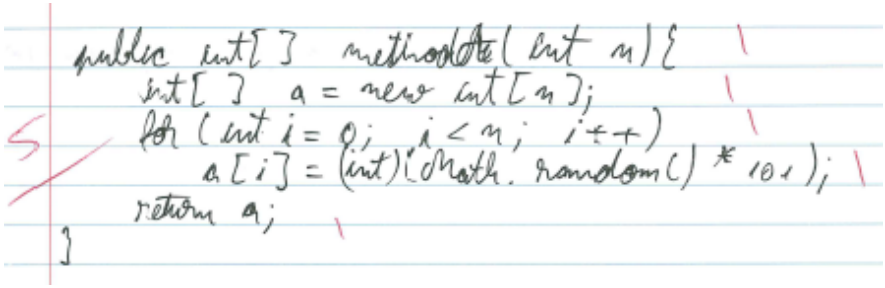
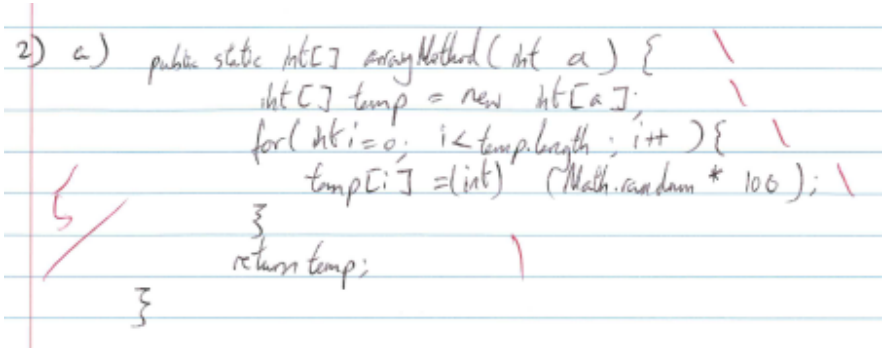
1. Read the students answers carefully.
2. Rate the students answer based on Table 1 categories (P, U, R, M, R and EA)
3. Give your justifications for your rating.

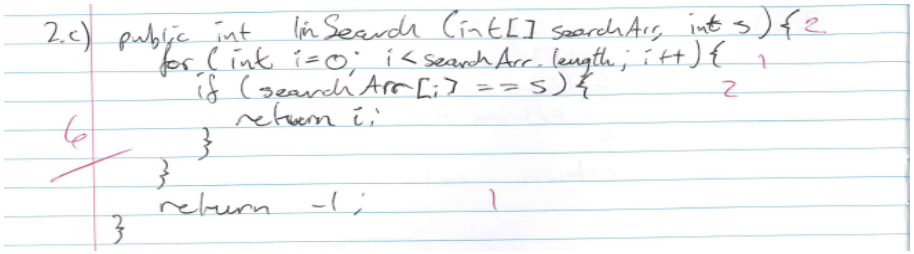
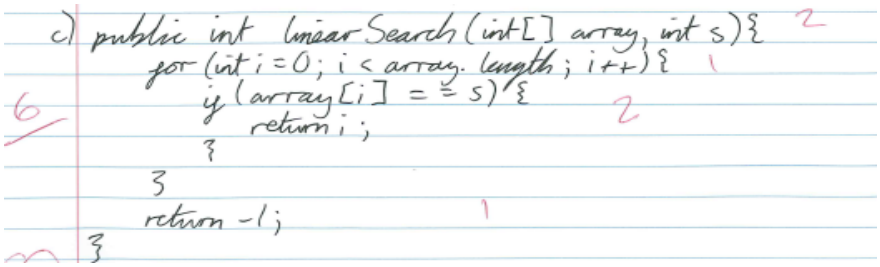
Case No	Answers for question 1	Rating	Justification
Question1 : Write a method that, when called with a single integer argument, n, creates an array of n integers with random values between 0 and 100 inclusive.			
5098		U	The student has understood what was asked in the question, tried to use constructs he has learned in class and followed the logical structure of the question.
5091		M	The solution is well structured, based on the structure of the question and the student has implemented the solution with no mistakes.

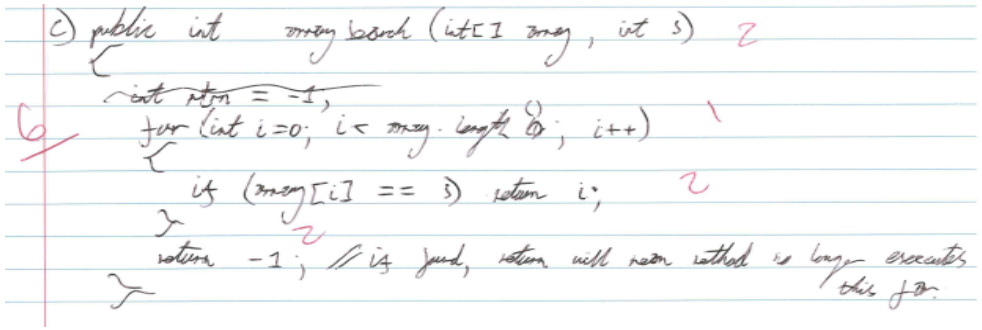
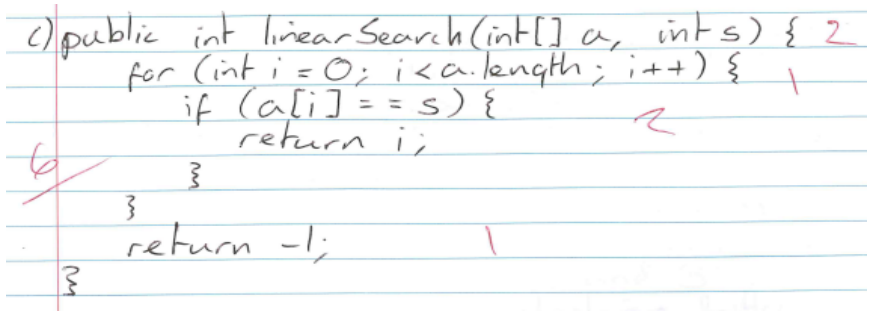
5079		U	The solution is well structured, however the student made a serious mistake (no return clause).
5078		U	The solution is well structured, based on the structure of the question and the student has implemented the solution with a little mistake (Math.Random() not incremented to include 100).

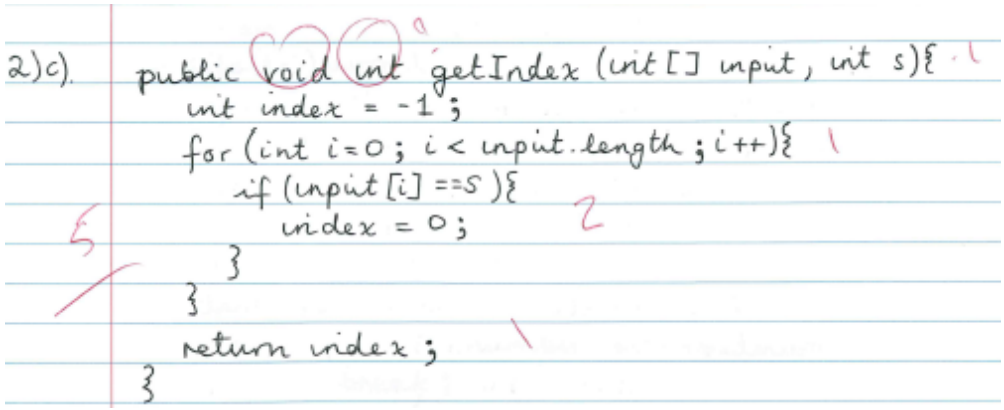
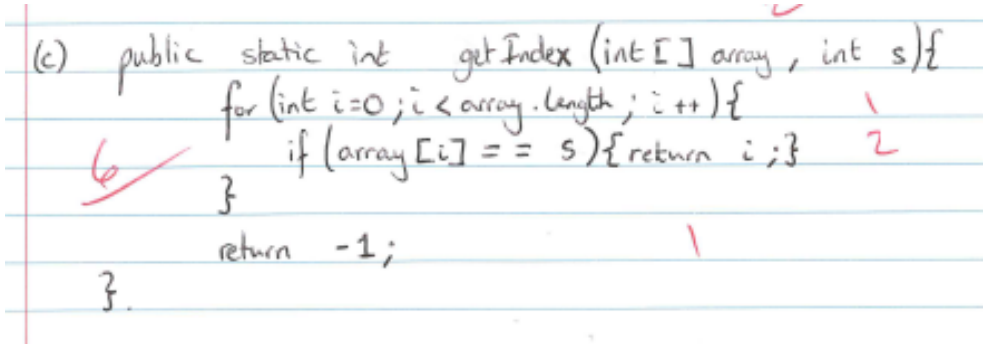
Case No	Answers for question 1	Rating	Justification
Question1 : Write a method that, when called with a single integer argument, n, creates an array of n integers with random values between 0 and 100 inclusive.			
5055		M	The student provided a more delicate solution than the usual, included comments in his code, as well as the import statement, although it was not asked. He has made a little mistake (nextInt and not randomInt), which would not be easy to make if he was writing in an IDE.
5049		U	The solution is well structured, the student uses an unusual feature (array length), however the student made two significant mistakes (100 is not included, no return clause).

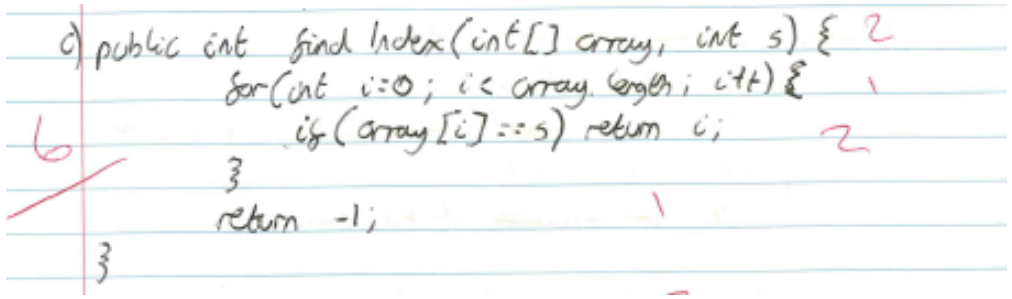
5042		U	Not a very clean, although unusual solution, which also does not lead to the correct output.

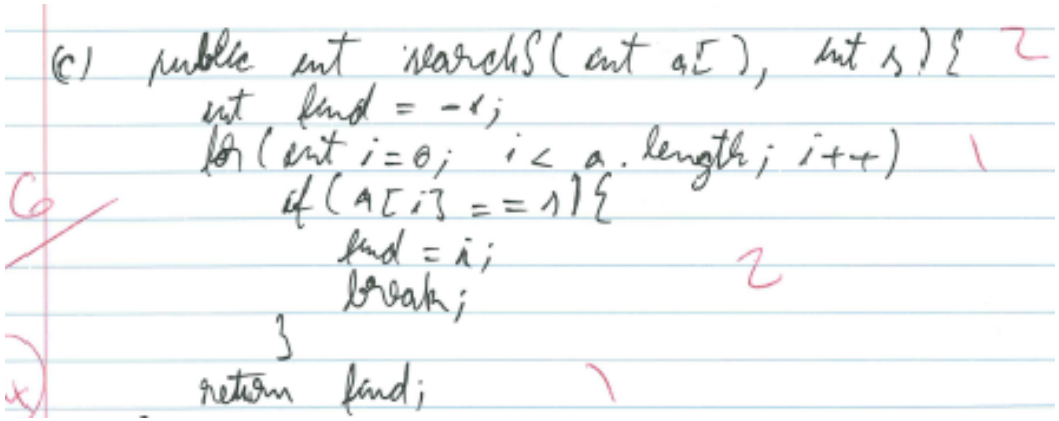
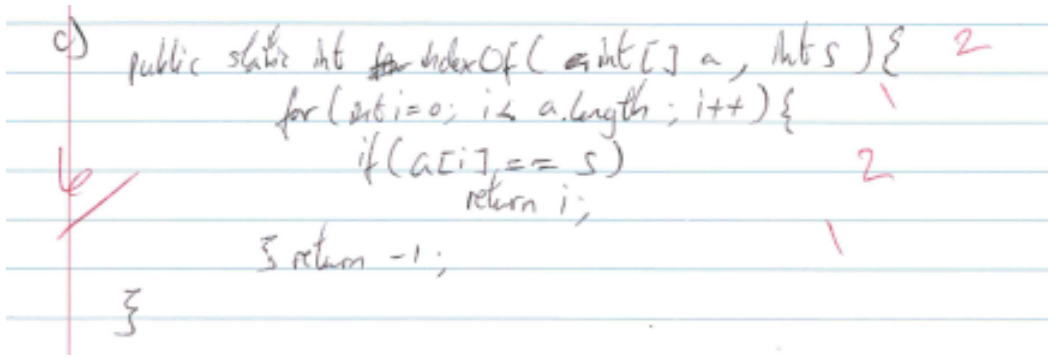
Case No	Answers for question 1	Rating	justification
Question1 : Write a method that, when called with a single integer argument, n, creates an array of n integers with random values between 0 and 100 inclusive.			
5036		M	The solution is well structured, based on the structure of the question and the student has implemented the solution with no mistakes.
5014		U	The solution is well structured, the student uses an unusual feature (array length), however the student made one little (100 is not included).

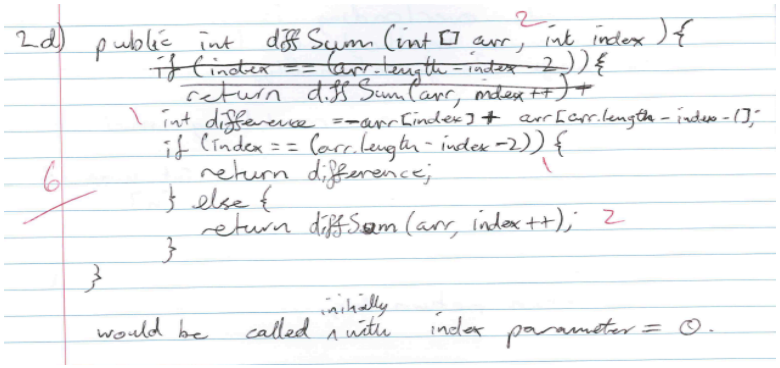
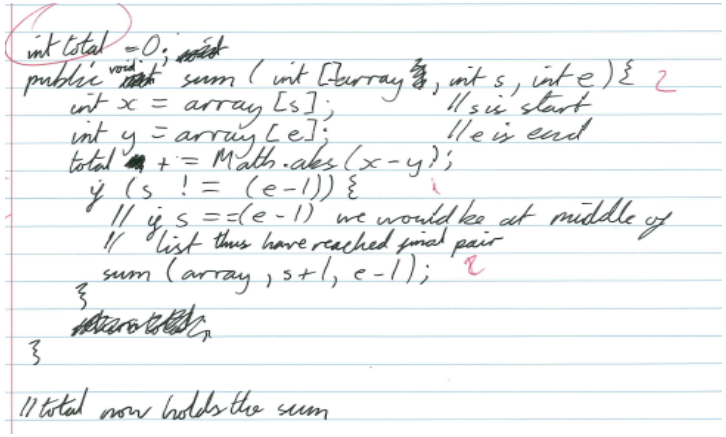
Case No	Answers for question 2	Rating	justification
Question 2: Write a method that, when called with an array and an integer argument, s, performs a linear search on the array reporting the array index of the first instance of s in the list, or returning -1 if s is not found in the array.			
5098		M	The student has removed any redundant statements and has provided a clean and correct solution.
5091		M	The student has removed any redundant statements and has provided a clean and correct solution.

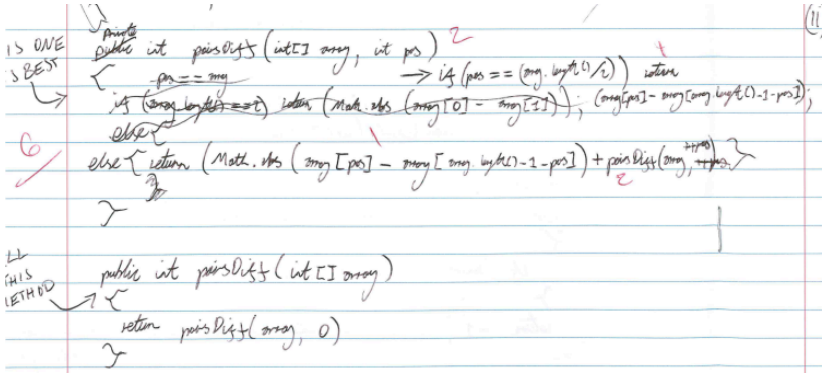
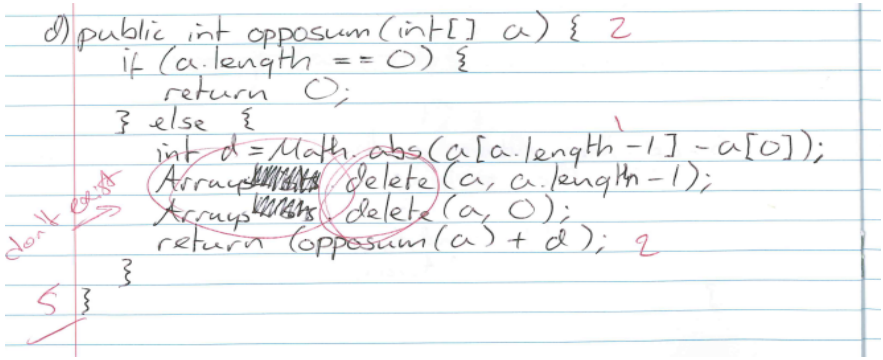
5079		M	The student has removed any redundant statements and has provided a clean and correct solution.
5078		M	The student has removed any redundant statements and has provided a clean and correct solution.

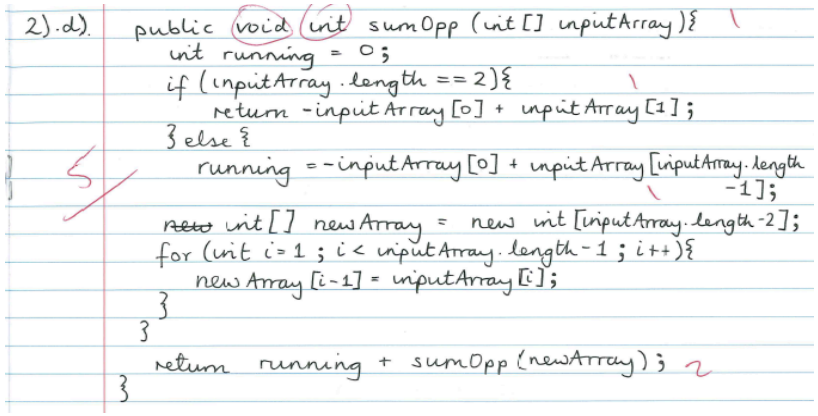
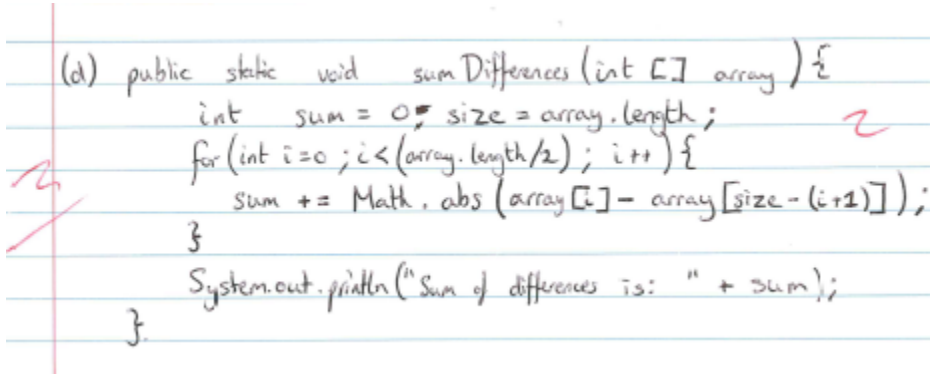
Case No	Answers for question 2	Rating	Justification
Question 2: Write a method that, when called with an array and an integer argument, s, performs a linear search on the array reporting the array index of the first instance of s in the list, or returning -1 if s is not found in the array.			
5055		U	The student has included a redundant statement (int index = -1) and has made several mistakes throughout his code (index=0;, void int).
5049		M	The student has removed any redundant statements and has provided a clean and correct solution. However, he uses the static construct in the method signature, which is neither needed nor asked.

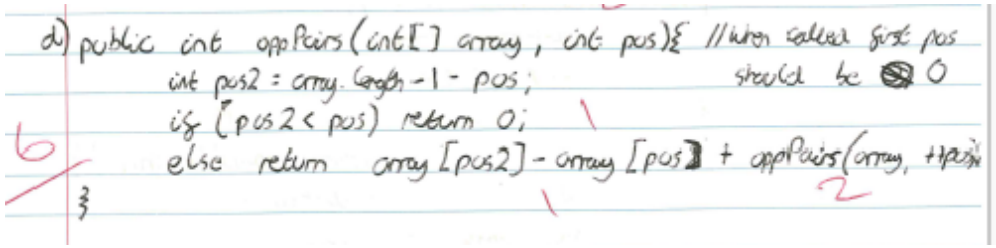
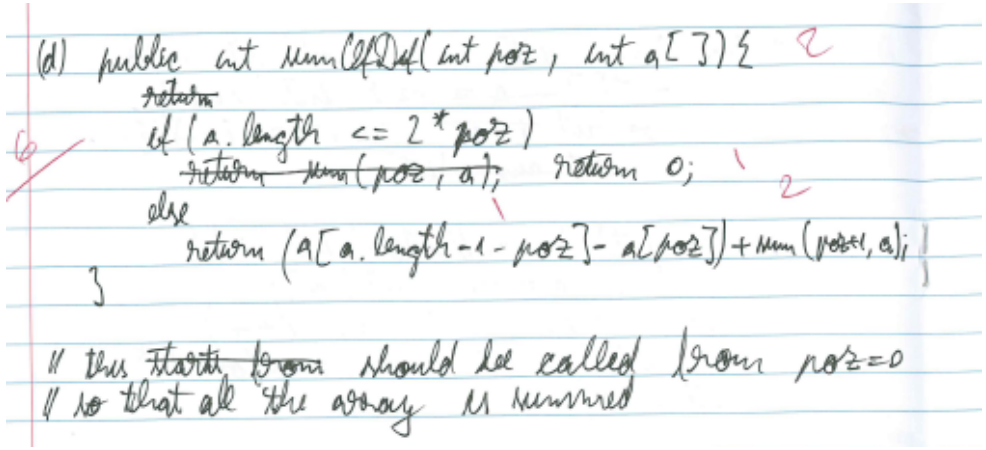
5042		M	The student has removed any redundant statements and has provided a clean and correct solution.

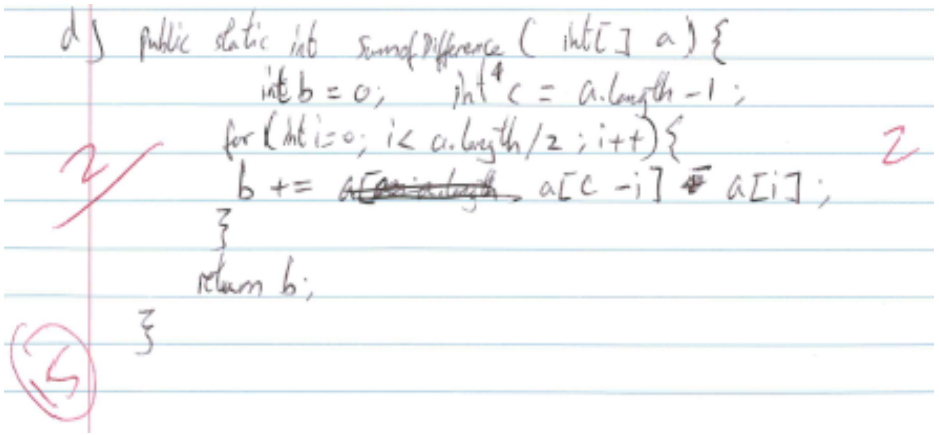
Case No	Answers for question 2	Rating	Justification
Question 2: Write a method that, when called with an array and an integer argument, s, performs a linear search on the array reporting the array index of the first instance of s in the list, or returning -1 if s is not found in the array.			
5036		U	The student provides a less clean solution, with redundant statements and variable assignments. However, his solution leads to a correct output.
5014		M	The student has removed any redundant statements and has provided a clean and correct solution. However, he uses the static construct in the method signature, which is neither needed nor asked.

Case No	Answers for question 3	Rating	Justification
Question 3: write a recursive method that calculates the sum of the differences between opposing pairs (i.e. the difference between A[0] and A[n-1], A[1] and A[n-2], and so forth). For example, the array { 3, 6, 34, 65 } results in the calculation: (65 3) + (34 6) = 90. You may assume the list will always be even in length.			
5098		U	The solution is well structured; however, it leads to a wrong output. It doesn't sum up the output of the recursive executions of the method and it doesn't take into account the second number being less than the first, in which case the difference should not be calculated and added.
5091		U	The student has removed some redundancies (e.g. in the calculation of indices) and used an unusual construct (abs), although in a wrong way. However, his solution doesn't take some edge cases into account (e.g. the second number being less than the first). Moreover, the solution does not meet all the requirements (variable outside the function)

5079		U	The student tries to use some more advanced features (e.g. method overriding). However, the program is not clean and the edge cases are not considered correctly.
5078		U	Overall good algorithmic way of thinking. However, the implementation is wrong in many ways. Some irrelevant and wrong actions are included in this program (deletion of array elements, although not used in the correct manner). Edge cases are not considered correctly.

5055		U	Uses a recursion, but it modifies the input data, which is not included in the requirements. Tried to bypass a problem instead of trying to solve it in a cleaner way.
5049		P	This implementation does not follow the guidelines of the question (recursion).

Case No	Answers for question 3	Rating	Justification
Question 3: write a recursive method that calculates the sum of the differences between opposing pairs (i.e. the difference between A[0] and A[n-1], A[1] and A[n-2], and so forth). For example, the array { 3, 6, 34, 65 } results in the calculation: (65 3) + (34 6) = 90. You may assume the list will always be even in length.			
5042		R	The optimal solution, uses just enough code to get things done
5036		M	The optimal solution, uses just enough code to get things done. However, a not so clean way is preferred to calculate the edge case.

5014		P	This implementation does not follow the guidelines of the question (recursion).

This informed consent form is for students participating in the interview. This form has two parts:

- Information sheet (to share information about the study with you);
- Certificate of consent (for signature if you agree)

Lead Researchers:

- Dr Mike Joy (Academic supervisor, Department of Computer Science);
- Dr Adam Boddison (Academic supervisor);
- Ayman Qahmash (PhD Student).

PART I: Information

Introduction

In this research study, we would like to find out more about programming experiences of Computer Science students. In particular, we would like to investigate particular characteristics that programmer might possess in which those characteristics could be identified. In general, much of this research is likely to inform the ongoing research on computer science education to ensure that curricle can meet students' expectation by including enriched content and challenging tasks.

Should there be anything in this form that you do not understand or that would like further information about, then please do not hesitate to contact academic supervisors, Dr Mike Joy (M.S.Joy@warwick.ac.uk) and/or Dr Adam Boddison (adam@adamboddison.com). Please be aware that you can contact me (a.qahmash@ warwick.ac.uk) at any time throughout the study using these details should you need to.

Voluntary Participation and Right to Withdraw

Participation in this study is optional. Also, you have the right to withdraw your consent to participate in this study at any time and you do not need to give a reason.

Research Process and Duration of Study

This study will focus on data collected through of interview with you. The interviews will be conducted through face to face meeting and will approximately last no more than forty minutes. Interviews will generally be recorded and copies are available to participants on request. The questions asked will be focused on your programming experience. Should you not wish to answer a particular question, then this is fine; they must just let the interviewer know at the time.

In addition to the interviews, students may be asked to give their permission to collect data from another source of information such as exam script and sample of programming codes.

Benefits and Risks

As a gesture of good will for giving up their time to participate in this study, you will enter a prize draw to win £120 Amazon gift card and you will receive £10 Warwick eating voucher (or equal amount in cash). Please note these benefits will be given regardless of any positive or negative views expressed during interviews.

We have judged that there are no significant risks to you from taking part in this study, but should you become concerned, you can contact the researchers at any time.

Anonymity and Confidentiality

Participants in this study are not anonymous, since they have been selected by the researchers. However, the identity of you will remain confidential for the purposes of the publication of any research findings. Pseudonyms will be used to protect the your

identity from other students and the wider research audience. Any personal information that might be used to identify you will not be made available in the public domain.

PART II: Consent Form

I have read the information provided and have been given the option to ask any questions that I might have. I consent voluntarily to participate in this study.

- Printed name of participant .....
- Signature of participant .....
- Date .....