An Exploration of Traditional and Data Driven Predictors of Programming Performance

WATSON, CHRISTOPHER

An Exploration of Traditional and Data Driven Predictors of Programming Performance

Christopher Watson

BSc(Hons), FHEA, PGCAP, PhD

School of Engineering and Computing Sciences
Durham University

A thesis submitted for the degree of
Doctor of Philosophy

April, 2015.

ABSTRACT

This thesis investigates factors that can be used to predict the success or failure of students taking an introductory programming course. Four studies were performed to explore how aspects of the teaching context, static factors based upon traditional learning theories, and data-driven metrics derived from aspects of programming behaviour were related to programming performance.

In the first study, a systematic review into the worldwide outcomes of programming courses revealed an average pass rate of 67.7%. This was found to have not significantly changed over time, or to have differed based upon aspects of the teaching context, such as the programming language taught to students.

The second study showed that many of the factors based upon traditional learning theories, such as learning styles, are context dependent, and fail to consistently predict programming performance when they are applied across different teaching contexts.

The third study explored data-driven metrics derived from the programming behaviour of students. Analysing data logged from students using the BlueJ IDE, 10 new data-driven metrics were identified and validated on three independently gathered datasets. Weaker students were found to make a greater percentage of successive errors, and spend a greater percentage of their lab time resolving errors than stronger students. The Robust Relative algorithm was developed to hybridize four of the strongest data-driven metrics into a performance predictor. The novel relative scoring of students based upon how their resolve times for different types of errors compared to the resolve times of their peers, resulted in a predictor which could explain a large proportion of the variance in the performance of three independent cohorts, $R^2 = 42.19\%$, 43.65% and 44.17% - almost double the variance which could be explained by Jadud's Error Quotient metric.

The fourth study situated the findings of this thesis within the wider literature, by applying meta-analysis techniques to statistically synthesise fifty years of conflicting research, such that the most important factors for learning programming could be identified. 482 results describing the effects of 116 factors on programming performance were synthesised and consolidated to form a six class theoretical framework. The results showed that the strongest predictors identified over the past fifty years are data-driven metrics based upon programming behaviour. Several of the traditional predictors were also found to be influential, suggesting that both a certain level of scientific maturity and self-concept are necessary for programming. Two thirds of the weakest predictors were based upon demographic and psychological factors, suggesting that age, gender, self-perceived abilities, learning styles, and personality traits have no relevance for programming performance.

This thesis argues that factors based upon traditional learning theories struggle to consistently predict programming performance across different teaching contexts because they were not intended to be applied for this purpose. In contrast, the main advantage of using data-driven approaches to derive metrics based upon students' programming processes, is that these metrics are directly based upon the programming behaviours of students, and therefore can encapsulate such changes in their programming knowledge over time. Researchers should continue to explore data-driven predictors in the future.

Keywords: programming, predictors, learning strategies, programming behaviour, compilation behaviour, error quotient, watwin, robust relative, meta-analysis.

Declaration

No part of this thesis has previously been submitted for any degree at any institution. Three of the studies presented in this thesis have appeared in the papers [197,198,200], all of which have been subject to peer review. At the beginning of each chapter we mention where the results presented in that chapter have been published.

Acknowledgements

I'd like to thank the EPSRC for the enhanced scholarship which funded this research. Without the funding they provided, this research would not have been possible. It is because of their support, that I had the opportunity to present my work at so many conferences, and to make some international collaborators in the process.

Next, I thank my supervisor Dr. Frederick Li for helping me to secure funding, encouraging me to publish, and for allowing me the independence to pursue the research topics which interested me the most. I also thank you for the comments on the thesis, along with your support during my BSc and PhD. Also, thanks to my second supervisor Dr Ioannis Ivrissimtzis for his timely and very useful comments on my research. You helped me to understand the wider implications of my thesis, and your guidance and support during the writeup phase were invaluable.

Additionally thanks to my examiners Prof. Quintin Cutts MBE and Dr. Stefan Dantchev for their useful comments on the thesis, and for making the viva one of the most enjoyable, and memorable experiences of my time in academia.

I would like to thank Prof. Nick Holliman and Dr. Steven Bradley for allowing me to recruit their programming students for the experiments conducted in this thesis. I'd also like to thank Steven for your guidance, the demonstrating, lecturing, research, and outreach opportunities which you afforded me over the past four years.

Durham University Outreach, most notably Dr. Lorraine Coghill, Dr. Pete Edwards, Christine Durrand, Chris Saunders, and Chris Callow, for allowing me to bring computer science into secondary school classrooms through the *Digital Physics* project.

During my time in Durham I have been fortunate enough to meet many great people. I'd like to thank three future Doctors in particular for their support - Yan Birch, Dave Roberts, and Chris Wastell. Yan, thanks for all of the memorable advise and marathon pool sessions that we enjoyed. Dave, thanks for the comments on the thesis and for joining me on a fantastic trip to New York. Chris, for always having the time for a chat, and a soothing pint of real ale when things weren't going well. I wish you all the very best in the future, and look forward to your graduations.

I'd like to thank Jamie Godwin for the time which we spent together over the past seven years in Durham. I'm glad that you decided to come back from Japan to join me on this journey, and I consider myself fortunate to have had your support while we both endured the PhD process together. Thanks for keeping me on the square and level, and I wish you every success in whatever you choose to pursue in the future.

I'd also like to thank my friends from school who helped to keep me sane by dragging me away from the office. Gregg, thanks for all of the late nights in the Grey Horse and for being a constant source of support over the past twelve years of my life. You'll probably never know how much I needed our long pool or snooker sessions to take my mind off things, and you were always at the other end of the phone whenever I needed anything. Anthony, the same, but also thanks for joining me on my adventures to *creaky* Sweden and Finland. Those trips would have not been the same without you there, and I hope we can do more travelling in the future. Neil, thanks for the many hours of GTA and CoD, and your hospitality in Sunderland. I'm looking forward to spending many more nights over there in the future. You all made a massive difference to me completing this PhD, possibly more than any of you will ever realise, and thanks to you all for your support and friendship.

Finally, and most importantly, thanks to my parents Gloria and Denis for their constant patience, loving care, encouragement, and support throughout my life. To them, I owe everything, and to them this thesis is dedicated.

To my parents.

Contents

1 Introduction 1

1.1 Background . . . 2

1.2 Research Objectives . . . 5

1.3 Research Questions . . . 9

1.4 Contributions . . . 10

1.5 Thesis Organisation . . . 12

1.6 Publications . . . 13

2 Background 15

2.1 Definitions and Terminology . . . 16

2.2 The Difficulties of Learning to Program . . . 18

2.2.1 Program Comprehension . . . 18

2.2.2 Program Generation . . . 19

2.2.3 Choice of Programming Language . . . 20

2.2.4 Threshold Concepts . . . 22

2.2.5 Mental Models and Misconceptions . . . 23

2.2.6 Multiple Skills . . . 24

2.2.7 Teaching Methodology . . . 25

2.2.8 Differences in Student Abilities . . . 25

2.3 Predictors based upon Traditional Learning Theories . . . 26

2.3.1 Cognitive Predictors . . . 27

2.3.2 Psychological Predictors . . . 29

2.3.3 Academic Predictors . . . 31

2.3.4 Context Independent Predictors . . . 32

2.3.5 Hybrid Predictive Models . . . 32

2.4 Predictors based upon Programming Behaviour . . . 39

2.4.1 Stoppers, Movers, Tinkerers . . . 39

2.4.2 The Errors that Students Make . . . 40

2.4.3 Compilation Behaviour . . . 41
2.4.4 Supplementing Compilations with Testing Data . . . 43
2.4.5 Massive Datasets and Data Mining . . . 44
2.4.6 CVS Repositories . . . 46
2.4.7 Online Submission Systems . . . 47
2.5 Summary . . . 52

3 Research Method 53
3.1 Research Objectives . . . 54
3.2 Research Questions . . . 55
3.3 Teaching Context . . . 56
3.3.1 Learning Content and Materials . . . 57
3.3.2 Assessment Components . . . 57
3.4 Research Methods . . . 61
3.4.1 RQ1: Failure Rates in Introductory Programming . . . 61
3.4.2 RQ2: Traditional Predictors based upon Learning Theories . . . 63
3.4.3 RQ3: Data-Driven Predictors based upon Programming Behaviour 65
3.4.4 RQ4: Meta-Analysis of Fifty Years of Research on Factors that can Predict Programming Performance . . . 67
3.5 Statistical Techniques . . . 69
3.6 Summary . . . 70

4 Failure Rates in Introductory Programming 71
4.1 Introduction . . . 72
4.2 Related Work . . . 74
4.3 Research Design . . . 76
4.3.1 Research Questions . . . 76
4.3.2 Data Collection Method . . . 76
4.3.3 Description of the Sample . . . 79
4.4 Results . . . 80
4.4.1 Worldwide Pass Rates of Programming Courses . . . 80
4.4.2 Pass Rates of Programming Courses over Time . . . 81
4.4.3 Moderating Aspects of the Teaching Context . . . 82
4.5 Discussion . . . 86
4.5.1 Threats to Validity . . . 87
4.6 Conclusion . . . 89

5 Traditional Predictors based upon Learning Theories 91

5.1 Introduction 92

5.2 Related Work 94

5.3 Research Design 95

5.3.1 Research Questions 95

5.3.2 Participants 95

5.3.3 Instruments 96

5.3.4 Criterion Variable 96

5.3.5 Predictor Variables 97

5.4 Results 98

5.4.1 Performance of the Cognitive Predictors 98

5.4.2 Performance of the Psychological Predictors 100

5.4.3 Performance of the Academic Predictors 101

5.4.4 Overall Performance of the Predictors 103

5.4.5 Regression Analysis 104

5.5 Discussion 105

5.5.1 Threats to Validity 106

5.6 Conclusion 107

6 Data-Driven Predictors based upon Programming Behaviour 109

6.1 Introduction 110

6.2 Related Work 112

6.3 Research Design 114

6.3.1 Research Questions 114

6.3.2 Criterion Variable 115

6.3.3 Data Collection Method 115

6.3.4 Data Preparation and Cleaning 116

6.3.5 Datasets and Participants 121

6.4 Results 124

6.4.1 Descriptive Statistics of the Datasets 124

6.4.2 Relations between Different Error Types and Performance 127

6.4.3 Resolve Times as a Predictor 132

6.4.4 Developing a Predictive Algorithm 137

6.4.5 Comparative Analysis 145

6.5 Discussion 150

6.5.1 Threats to validity 151

6.6 Conclusion 152

7 Meta-Analysis of Fifty Years of Research into Predictors of Programming Performance 153

7.1 Introduction . . . 154

7.2 Related Work . . . 155

7.3 Research Design . . . 157

7.3.1 Research Questions . . . 157

7.3.2 Systematic Review and Data Extraction . . . 157

7.3.3 Meta-Analysis Procedure . . . 169

7.4 Theoretical Framework . . . 172

7.4.1 Aptitude Predictors . . . 173

7.4.2 Academic Predictors . . . 173

7.4.3 Cognitive Predictors . . . 174

7.4.4 Psychological Predictors . . . 175

7.4.5 Programming Behaviour Predictors . . . 176

7.4.6 Demographic Predictors . . . 176

7.4.7 Identifying Interacting Factors . . . 177

7.5 Results . . . 178

7.5.1 Aptitude Predictors . . . 178

7.5.2 Academic Predictors . . . 181

7.5.3 Cognitive Predictors . . . 184

7.5.4 Psychological Predictors . . . 188

7.5.5 Programming Behaviour Predictors . . . 192

7.5.6 Demographic Predictors . . . 195

7.5.7 Moderator Analysis . . . 196

7.6 Discussion . . . 197

7.6.1 Practical Implications for Teaching . . . 199

7.6.2 Implications and Applications of the Framework . . . 200

7.6.3 Threats to Validity . . . 201

7.7 Conclusion . . . 202

8 Conclusion and Future Works 203

8.1 Contributions . . . 204

8.2 Research Questions . . . 207

8.3 Implications for the Theoretical Framework . . . 211

8.4 Limitations . . . 213

8.5 Future Directions . . . 213

8.6 Summary . . . 216

Appendices **217**

A Statistical Tests **217**

A.1 Assumption Testing 218

A.2 Association 219

A.3 Prediction 219

A.4 Differences Between Groups 221

A.5 Meta Analysis 223

A.5.1 Effect Size Calculations 223

A.5.2 Effect Size Corrections 224

A.5.3 Meta Analytic Model Fitting 226

B Example Assessment Materials **229**

B.1 Example Term One Bench Test 230

B.2 Example Term One Project 234

B.3 Example Term Two Bench Test 236

B.4 Example Lab Worksheet 238

C Paper Awards **241**

C.1 ACM ITiCSE 2014 242

C.2 IEEE ICALT 2013 243

D Publications **245**

List of Figures

1.1 Theoretical framework of factors which will be explored by the studies performed in this thesis. The dotted lines indicate interacting factors between different classes of predictors and were identified in Chapter 7. 7

2.1 Predictors which researchers have examined over the previous fifty years. Interacting factors were determined by related studies in the literature, and are shown on the diagram by using dashed arrows (further detail is presented in Section 7.4) 26

3.1 Overview of the research conducted within this thesis, showing how the three research objectives related to the overall research theme, and how each of the four research questions were related to the research objectives. 56

4.1 Bar chart showing the number of CS1 outcomes coded for each year. Different decades are indicated by different colors. 79

4.2 Geographical distribution of the CS1 outcomes used by both studies. . . 80

4.3 Pass rates of this study compared to Bennedsen and Caspersen [13] . . . 81

4.4 Bar chart showing the mean percentage of non-passing students for each year that was covered by this study. Different decades are indicated by different colors. 82

4.5 Bar chart showing the mean percentage of non-passing students by the different countries that were covered by this study. 83

4.6 Bar chart showing the mean percentage of non-passing students by the different programming languages taught by courses covered by this study. 85

6.1 Screenshot of the Code Browser Tool which we developed to assist us in visualizing students' programming activities logged through our BlueJ extension. The classes the student interacted with during a session are shown on the left hand side, and squares represent events being executed. Red indicates a compilation error, Green indicates a compilation success, and Purple indicates an invocation. The researcher could obtain more detailed information on the events by hovering over them, and could zoom into an area of interest by selecting an area of the graph. 117

6.2 Example cases where errors would be inaccurately marked as resolved by previous approaches of building compilation pairings. 118

6.3 Distribution of the performance of all participating students $n = 141$. . 123

6.4 Most common errors found in the three datasets used by this study. . . 125

6.5 Scatter graphs showing the combined correlations between percentage of different types of pairings with the performance of all three cohorts, $n = 141$. 128

6.6 Correlations between the percentage of different types of error pairings and performance over the duration of all three courses. 131

6.7 Scatter graphs showing the combined correlations between the percentage of lab time students spent working on different types of pairings with the performance of all three cohorts, $n = 141$. 133

6.8 Correlations between the percentage of time spent working on different types of error pairings by students and performance, over the duration of all three courses. 136

6.9 The scoring algorithm used by Robust Relative algorithm, showing the penalties which were identified through applying the genetic training algorithm. 141

6.10 Variance explained by the Robust Relative and Error Quotient algorithms over the duration of the course. 144

6.11 Figures showing the variance explained by both Robust Relative and the Error Quotient algorithms compared to the frequency based metrics previously explored in this chapter. 147

6.12 Figures showing the variance explained by both Robust Relative and the Error Quotient algorithms compared to the time based metrics previously explored in this chapter. 148

6.13 Comparing the correlation strengths of the predictors based upon programming behaviour (PB) against the predictors based upon traditional learning theories (T) which were explored in Chapter 5. 149

7.1 Overview of the Systematic Review and Data Extraction Procedure . . 158
7.2 Overview of the Meta-Analysis Procedure Applied by this Study 169
7.3 Framework of internal factors that are predictive of programming performance. Interacting factors were determined by related studies in the literature, and are shown on the diagram by using dashed arrows. 172
7.4 Chart Showing Meta-Analysis Results for the Aptitude Predictors . . . 179
7.5 Chart Showing Meta-Analysis Results for the Academic Predictors . . . 182
7.6 Chart Showing Meta-Analysis Results for the Cognitive Predictors . . . 186
7.7 Chart Showing Meta-Analysis Results for the Psychological Predictors . 190
7.8 Chart Showing Meta-Analysis Results for the Programming Behaviour Predictors 193
7.9 Chart Showing Meta-Analysis Results for the Demographic Predictors . 195
7.10 Updated theoretical framework, displaying the strongest fifteen predictors found by this meta-analysis. Interacting factors were determined by related studies in the literature, and are shown on the diagram by using dashed arrows. 200

8.1 Updated theoretical framework, displaying the strongest fifteen internal factors from the meta-analysis, and the two influential external aspects of the teaching context identified by this thesis. 212

List of Tables

2.1 Regression models designed by previous researchers to predict programming performance and their explanatory power. 33

2.2 Most common errors identified by previous researchers. 41

2.3 Overview of programming behaviours identified by recent studies (2005-2014). 49

3.1 Table showing the weightings of the various assessment components for all three cohorts. All of the assessments were summative unless stated otherwise, a dash (-) indicates that an assessment was not used in the course during a given year. 58

3.2 Table showing the assignments/concepts that were covered in the weekly lab sessions for all three cohorts. Lab assignments that were not reused are shown in italics. Labs reserved for course assessments are shown in bold. 58

5.1 Table showing the correlations (r) of the cognitive predictors explored by this study with programming performance, and compared to prior research if applicable. (* $p < .10$, ** $p < .05$, *** $p < .01$). 99

5.2 Table showing the correlations (r) of the psychological predictors explored by this study with programming performance, and compared to prior research if applicable. (* $p < .10$, ** $p < .05$, *** $p < .01$). 101

5.3 Table showing the correlations (r) of the academic predictors explored by this study with programming performance, and compared to prior research if applicable. (* $p < .10$, ** $p < .05$, *** $p < .01$). 102

5.4 Table showing the strongest five context independent predictors identified by this study, compared to the average correlation strength reported by previous research. 104

6.1 Example Logs of Two Files to Illustrate Pair Construction 120

6.2 Descriptives of the participating students from all cohorts 122
6.3 Most common errors found in the three datasets used by this study, compared to most common errors found by previous researchers. 125
6.4 Median (Md) and Interquartile Range (IQR) of the resolve times in seconds.126
6.5 Table showing the correlations (r) of performance with the percentage of different types of pairings logged for students (* $p < .10$, ** $p < .05$, *** $p < .01$) 129
6.6 Table showing the results of the regressions based upon the percentage of different types of event pairings logged for students. 130
6.7 Table showing the correlations (r) of performance with the percentage of lab time students spent working on different types of event pairings. (* $p < .10$, ** $p < .05$, *** $p < .01$) 134
6.8 Table showing the results of the regressions based upon the percentage lab time students spent working on different types of pairings. 134
6.9 Low (L) and High (H) penalty bounds (seconds) of the most common errors. Medium penalties are applied for times between the low and high bounds. 139
6.10 Table showing the results of the regressions based upon the Robust Relative and Error Quotient algorithms. 142
6.11 Results of running the Error Quotient using the Robust Relative Penalties.144

7.1 Table showing examples of inconsistent findings of research into predictors of programming performance, that were not previously explored in this thesis. 156
7.2 Summary Table of the Sample of Studies based upon the Coding Criteria 162
7.3 Overview of the data coded from each study included in this meta-analysis, and the predictors that each study examined in relation to the proposed theoretical framework of interacting internal factors (Figure 7.3).163
7.4 Example interacting factors within the theoretical framework classes. Many of these are hypothesised based upon the systematic review we performed. 177
7.5 Meta-Analysis Results for the Aptitude Predictors 180
7.6 Meta-Analysis Results for the Academic Predictors 183
7.7 Meta-Analysis Results for the Cognitive Predictors 187
7.8 Meta-Analysis Results for the Psychological Predictors 191
7.9 Meta-Analysis Results for the Programming Behaviour Predictors . . . 194
7.10 Meta-Analysis Results for the Aptitude Predictors 195

7.11 Results of the Moderator Analysis 196
7.12 The Overall Strongest 15 Predictors found by this Meta-Analysis 198
7.13 The Overall Weakest 15 Predictors found by this Meta-Analysis 198
7.14 Summary of effect sizes by class sorted by mean strength ($\overline{ES_{\bar{r}}}$) 199

List of Symbols

Symbol	**Description**
CI_L	95% confidence interval (lower)
CI_U	95% confidence interval (upper)
df	Degrees of freedom
$ES_{\bar{r}}$	Weighted Effect Size
$\overline{ES_{\bar{r}}}$	Mean of all effect sizes for a class
F	F-ratio (used in ANOVA)
H	Kruskal-Wallis test value
HSD	Tukey's Honestly Significant Difference
IQR	Interquartile Range
I^2	Index magnitude of heterogeneity
k	Number of subsamples or groups
M	Sample mean
Md	Median
μ	Population mean
n	Number of participants in subsample
p	Probability
Q_t	Total heterogeneity
r	Pearson's correlation
R^2	Coefficient of determination
r_s	Spearman's Rho
SD	Sample standard deviation
SM	Mean of All Sample Sizes
t	t-test value
T	Wilcoxon test value
U	Mann-Whitney test value
χ^2	Chi-square test value
z	Test of Effect Size

Chapter 1

Introduction

In this chapter we set the context for the research which will be presented in this thesis. Background information is first presented and motivations for examining predictors of programming performance are discussed. The three research objectives of this thesis are discussed, and an overview of the four research questions is presented. Finally, this chapter briefly discusses the contributions of this thesis and lists the publications which arose from this work.

1.1 Background

The demand for skilled programmers is constantly increasing on a global scale. Recent projections from the United States Bureau of Labour Statistics [186] suggests that the growth of computing careers is set to continue through 2020, and that various computing skills will be in strong demand for the foreseeable future. For example, the job market for systems software developers is projected to increase by 32% by 2020, application developers by 28%, and database administrators by 31%.

A fundamental entry requirement for many of these computing careers are strong programming and software engineering skills. However, the global education system is still not producing enough graduates with the skills required to satisfy these future labour demands. Within the US, the number of available high school computing courses have decreased by 17%. Although enrolment in computing courses has seen a slight increase over recent years, retention is still a problem, and the number of majors awarded per US computer science department is still half that it was in the year 2000 [217]. Because of this, it is estimated that of the future 1.4 million jobs that are to be be created in computing fields, only 30% will be filled by 2020 [90].

To address this labour shortage, governments throughout the world are in the process of bringing programming into the classroom environment, so that students can be better prepared to meet the future demands of a digital economy. In the US, Barack Obama has called for high schools to "create classes that focus on science, technology, engineering and math - the skills today's employers are looking for, to fill jobs right now and in the future" [90].

Within the United Kingdom, a major drive is currently underway to introduce programming as part of core curriculum of subjects. Prior to 2014, the UK government are aiming to have after school "Coding Clubs" introduced into at least 25% of Primary Schools [35]. From September 2014, UK schools will replace the Information and Communication Technology (ICT) course with Computing. Pupils aged 5-7 will be expected to understand "what algorithms are" and how to "create and debug simple programs". By the age of 11, pupils will have to "design, use and evaluate computational abstractions that can model the state and behaviour of real-world problems, and physical systems" [43].

Classroom instructors are naturally concerned about the proposed changes. A recent poll of UK secondary school teachers showed that 74% of current ICT teachers do not believe they have the right skills needed to deliver the new computing curriculum, and fear that they have neither the time, or ability, to learn the new skills that they require, to teach programming to students [44].

Given the introduction of compulsory programming courses into ordinary classroom environments throughout the world, there is a greater need than ever to develop an understanding of precisely which aspects of the external teaching context, and which characteristics internal to students are influential on their programming ability.

Identifying struggling programming students can be challenging. Introductory programming courses generally have a high student-to-lecturer ratio, an average of 50:1 or greater in the case of our University, and often lecturers do not know how well students are performing until after they have completed the first formal assessment. This may not take place until several weeks after the course has started, and given potentially high enrolment numbers, it can take an instructor a considerable amount of time and effort to process these assessments. Even if an assessment was indicative of overall performance, by the time it was processed, it may be too late for struggling students to withdraw, or for instructors to intervene to prevent students from failing [16].

This is a cause of great concern for computer science educators, and unsurprisingly over the past fifty years these concerns have led to an abundance of research focusing on identifying predictors of programming performance. As well as identifying characteristics that can be used to predict struggling students without the need to use formal assessments, other motivations for exploring such predictors include [32]:

1. Exploring relationships between programming and other cognitive abilities.
2. Providing automated interventions for students based upon their characteristics, such as different compilation feedbacks based upon prior knowledge.
3. Advising students on major selection.
4. Improving programming classes for non-computing majors.
5. Determining the importance of different characteristics that influence the ability of students to acquire programming skills.

Numerous predictors have been explored over the past fifty years. The majority of early work conducted during the 1960's and 1970's focussed upon using standardized programming aptitude tests as predictors of performance. Popular instruments included the IBM Programmer Aptitude Test (IBM PAT), and the Computer Programmer Aptitude Battery (CPAB). Predictors researched during these eras included: arithmetic reasoning, letter series reasoning, and figure classification [3, 8, 30]. However, despite measuring students' levels of *programming aptitude*, no single instrument or characteristic emerged as a conclusive predictor of performance.

This led to an expansion of the types of characteristics explored, and during the 1980's and 1990's, researchers expanded the search for predictors to include academic, cognitive, demographic, personality, and psychological traits. These included: performance in academic courses (with an emphasis on math [161] and science [159]), spatial ability [91], intellectual development, gender [113], personality style (Myers-Briggs [202]) and learning styles (Kolb's LSI [42]). As with the previous two decades, no single characteristic emerged as a conclusive predictor of programming performance.

During the 2000's, researchers mainly repeated efforts of the previous two decades and continued to explore various academic, cognitive, and psychological predictors of performance. These included: comfort level [210], self-efficacy [189] and learning strategies [16]. Again, the majority of predictors explored were found to be incapable of consistently predicting performance across different teaching contexts.

During the 2010's, researchers began to apply data-mining and statistical techniques to *big data* which was directly logged from an IDE and described students' programming behaviours [82, 200]. These predictors have shown more promise than the previously explored predictors, possibly due to their data-driven nature.

Whilst an abundance of predictors have been explored over the past fifty years, the main shortcoming of research to date is that the identified predictors cannot perform consistently when they are applied in a range of different teaching contexts. Researchers examining predictors of programming performance have a tendency to judge the value of a predictor, based upon the statistical significance of how the predictor performed on a single sample of students, working within a single teaching context. However, this approach is problematic, as verification studies of the same predictor in a different teaching context have a tendency to yield inconsistent results [200].

Consider the often cited predictor of Math performance. Using the SAT Math instrument to measure this ability, [203] reported a strong correlation of $r(88) = .51$ between the math and programming performance of college students. [91] reported a similar moderate correlation of $r(32) = .48$. On the other hand [3] reported a weak correlation of $r(50) = .13$, and [80] reported a weak correlation of $r(45) = .13$.

Varying findings of this nature can be found for almost all of the predictors examined to date. In other words, despite fifty years of research, there is still no general consensus among researchers on which factors can support students to learn programming. This is of greater concern to computer science educators nowadays given the introduction of programming lessons into early classroom environments. Without an understanding of such factors, it becomes extremely difficult to design and implement any effective pedagogical tools or strategies, which can automatically be applied to identify and better support the weakest students.

1.2 Research Objectives

The wider motivation of this thesis is to help students succeed in programming. This thesis investigates factors that can be used to predict the success or failure of students taking an introductory programming course. Both factors that can be measured prior to, and during a course are examined. These factors are classified into three broad groups: external aspects of the teaching context, factors based upon traditional learning theories, and data-driven metrics derived from aspects of programming behaviour.

The factors that are examined for use prior to course commencement include those which are based upon aspects of the teaching context, and those which are derived from traditional learning theories. These factors are generally measured by using surveys and are mainly static in nature. In other words, these factors mostly remain unchanged in response to changes in students' programming knowledge (e.g. math background). Such factors may be influential on course performance, and could be applied to advise students on their likelihood of success prior to their enrolment.

In contrast, the metrics that are examined as predictors for use during a course are data-driven in nature. Instead of being based upon data gathered from surveys, these metrics are based upon analysing data that is directly gathered from an IDE which captures the programming process of students. The data gathering involves collecting snapshots of source code and error messages, usually when students' perform actions such as saving the project they are currently working on. This data then enables the utilization of data-driven approaches for illuminating the *symptoms* of struggling students based upon aspects of their programming behaviour (e.g. repeating errors).

The broad research question which will be explored in this thesis is:

> *Which factors can be used to predict the success or failure of students taking an introductory programming course?*

Consequently the three research objectives (RO) of this thesis, are defined as follows:

> *RO1: To explore which aspects of the external teaching context can influence the performance of students taking an introductory programming course.*

> *RO2: To explore which internal factors derived from traditional learning theories can be predictive of the performance of students taking an introductory programming course.*

> *RO3: To explore which data-driven metrics based upon analysing the programming behaviour of students within an IDE, can be predictive of the performance of students taking an introductory programming course.*

These three research objectives were selected so that both external (teaching context) and internal (student characteristics) influences on programming performance would be explored. Derived from the work conducted later in this thesis, a framework showing the interacting internal and external factors which are under investigation is shown in Figure 1.1. The design of this framework is based upon our results from later in the thesis, and justification of its design can be found in Section 7.4. The framework is only presented at this stage to illustrate to the reader which factors will be under investigation in the remainder of this thesis. In the remainder of this section, motivations for including each of the three research objectives are discussed.

RO1: Exploring Aspects of the External Teaching Context

> *RO1: To explore which aspects of the external teaching context can influence the performance of students taking an introductory programming course.*

The first group of factors that will be explored are taken from the external teaching context. These aspects can include the country in which the course was taught, grade level of the institution, cohort size, and the programming language taught. It is well known that educational practices and assessment criteria can vary across different continents. We therefore hypothesized that the these factors would have a moderating effect on students' successes (pass and failure rates) of programming courses.

RO2: Exploring Factors Derived from Traditional Learning Theories

> *RO2: To explore which internal factors derived from traditional learning theories can be predictive of the performance of students taking an introductory programming course.*

The second group of factors that will be explored are traditional predictors of programming performance, which describe the internal characteristics of the students. These include predictors derived from learning theories and predictors that are based upon aspects of academic background and performance in different subjects. These internal characteristics are always measured by requiring students to complete a series of test instruments or surveys which are then utilized to form predictions.

Although such predictors have been the focus of research over the past fifty years, there are two major shortcomings of previous research concerning the generalisability of such predictors to a range of different teaching contexts. The first problem is that there is a distinct lack of verification of predictors across different contexts. Researchers examining predictors have a tendency to judge the value of a predictor based upon

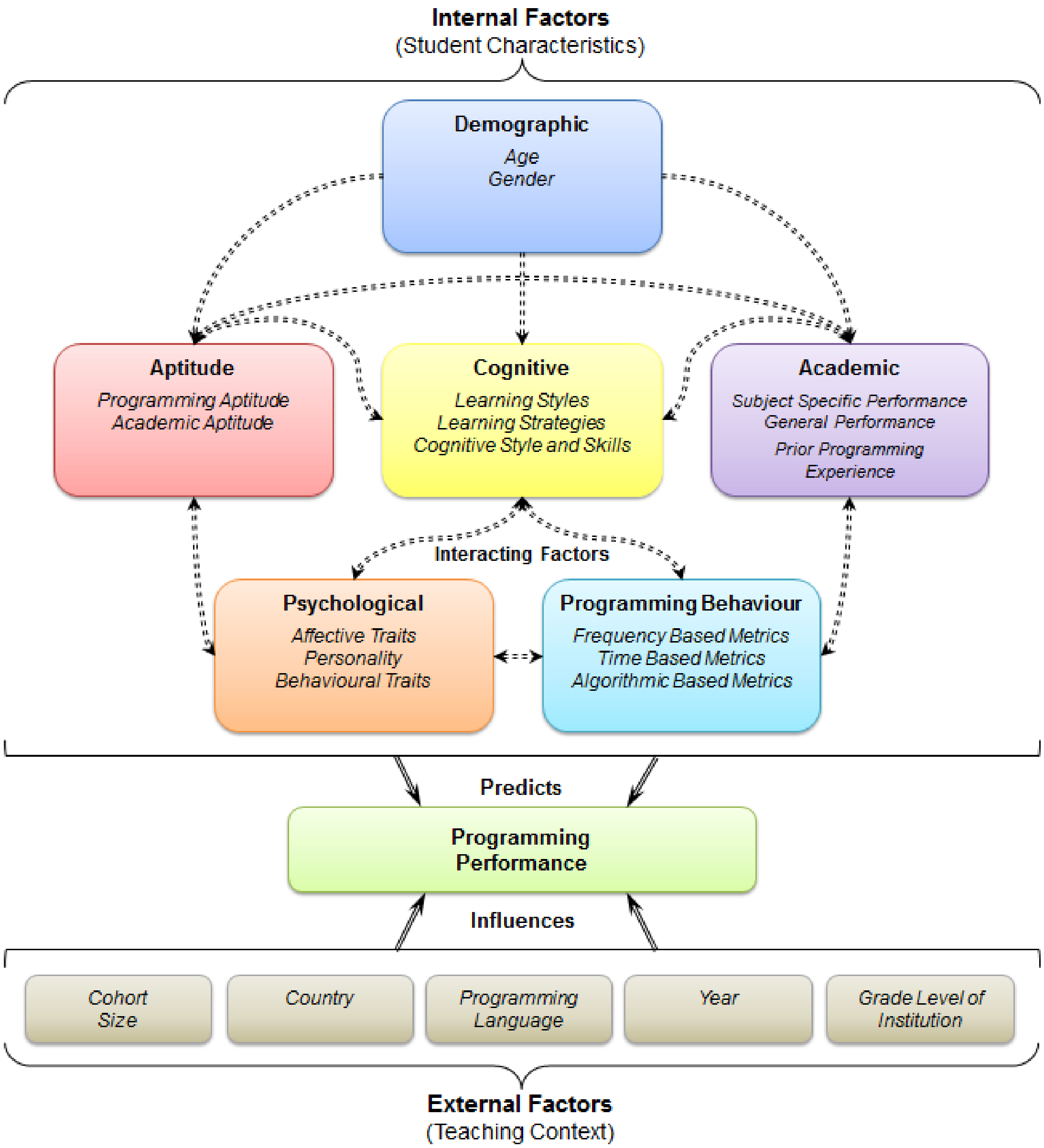


Figure 1.1: Theoretical framework of factors which will be explored by the studies performed in this thesis. The dotted lines indicate interacting factors between different classes of predictors and were identified in Chapter 7.

the statistical significance of the results on a single sample of students. This links to the second problem, which is when predictors are applied within different teaching contexts, they have a tendency to yield inconsistent results. In this thesis, we argue that this is because predictors considered to date are fundamentally limited by their static nature. That is, the traditionally measured attributes do not change in response to an increase in programming knowledge, and therefore cannot be consistently predictive of programming performance across a range of different contexts over time.

RO3: Exploring Data-Driven Metrics Derived from Aspects of Programming Behaviour

> *RO3: To explore which data-driven metrics based upon analysing the programming behaviour of students within an IDE, can be predictive of the performance of students taking an introductory programming course.*

The third group of factors are data-driven in nature, and are designed to be applied during a course for autonomously monitoring students' programming processes. Instead of being based upon data gathered from surveys, these metrics are based upon analysing data that is directly gathered from an IDE which captures the programming process as students' write programs. The data gathering involves collecting snapshots of source code and error messages, usually when students' perform actions such as saving the project they are currently working on. This data then enables the utilization of data-driven approaches for illuminating the *symptoms* of struggling students based upon aspects of their programming behaviour, e.g., the number of errors that students have made, or the time which they take to resolve errors when compared to their peers.

The main benefit of data-driven metrics based upon how students' solve programming errors, or whether they pay attention to code quality, is that, they are directly based on their regular programming activities of a student, and therefore can reflect changes in their learning progress over time. This is not the case for the traditional predictors explored over the past fifty years, such as age, gender, and high school performance, which remain static within the context of a course.

These metrics could be also applied to drive an expert system, so that students exhibiting *struggling symptoms* could be provided with appropriate pedagogical interventions when required. Such interventions would be difficult to provide using traditional predictors based upon learning theories, and would be incapable of automatically adapting to changes in a student's learning progress over time. Researchers have only recently begun to explore such data-driven predictors, and to date there has been no comparison of the performance of such predictors against the previous body of research.

1.3 Research Questions

To satisfy the three research objectives, this thesis conducted four successive quantitative studies which formed the basis of the four research questions. These research questions are:

RQ1 To what extent are students' programming performances influenced by aspects of the teaching context, including: year, country, grade level of the institution, cohort size, and the programming language taught in the course? *(RO1)*

RQ2 Which traditional learning theories describing the psychological and cognitive aspects of learning, and which aspects of students' academic backgrounds are predictive of their programming performances? *(RO2)*

RQ3 Which data-driven metrics derived from data describing students' programming behaviours are predictive of their programming performances? *(RO3)*

RQ4 How do factors based upon traditional learning theories, academic background, and programming behaviours, compare when they are used to predict students' programming performances across different teaching contexts? *(RO2, RO3)*

To present an overview to the reader RQ1 was answered by performing a systematic review of the literature on programming education. From the available literature, data describing the worldwide pass and failure rates of programming courses was extracted, and analysed by grouping the data based upon aspects of the teaching context.

RQ2 was based upon examining the relations between 34 predictors based upon traditional learning theories with the programming performance of one sample ($n = 39$) of students studying the 2012/13 introductory programming course at our University. These predictors were selected either as previous research had yielded inconsistent results, or, because no researcher had attempted to verify previous findings.

RQ3 was based upon exploring the data gathered from the BlueJ IDE which described the programming activities of three samples of students taking the programming course at our university. Samples were taken from 2011/12 cohort ($n = 37$), 2012/13 cohort ($n = 45$), and 2013/14 cohort ($n = 59$). Based upon our experiments, several aspects of programming behaviour which predicted performance were identified.

RQ4 was based upon performing a thorough meta-analysis to integrate the results from previous research with the results of this thesis. Based upon this study, 482 individual results describing 116 predictors were statistically synthesised, illustrating to researchers precisely which factors are the most critical for programming success.

1.4 Contributions

This thesis makes six major contributions to our understanding on the relevance of different factors for supporting students to learn programming. These are:

- Systematic review into the worldwide failure rates of programming courses and an exploration into whether aspects of the teaching context moderate the failure rates. Previously, only a single study [13] had attempted to provide any quantitative evidence to support the often cited claim of high worldwide programming failure rates.

- Evaluation of 34 traditional predictors based upon learning theories. These predictors had previously only been explored in a limited number of teaching contexts, or previous results were inconsistent. Re-evaluating these predictors in our context contributes to the knowledge on their context dependency, and provides evidence on their wider applicability as enablers of programming success.

- Identification of 10 new data-driven predictors based on aspects of programming behaviour. Five of these predictors were based upon the percentage of different types of compilation pairings that were logged from students. A further five predictors were based upon the percentage of lab time students spent working on different types of pairings. Nine of these predictors were statistically significant, but more importantly, and unlike the traditional predictors, were found to yield consistent results on three independently gathered datasets.

- Development of a data-driven predictive algorithm (*Robust Relative*) which predicts the programming performance of students based upon quantifying four aspects of their programming behaviour. The algorithm incorporates a novel scoring technique where students are relatively penalized based upon how their resolve times for different types of error compares to the resolve times of their peers. This allows both the difficulty of resolving different types of errors, and the students own programming abilities to be taken into account. A regression analysis showed that the scores produced by the Robust Relative algorithm could consistently explain a large amount of the variance in the performance of three samples of students, R^2 = 42.19%, 43.65% and 44.17% - almost double of the variance explained by Jadud's Error Quotient and the predictors based upon traditional learning theories. This suggests that programming behaviour, in particular error resolve times are important predictors of performance.

- Present the findings of a thorough meta-analysis, which aimed to synthesise over fifty years of conflicting quantitative research into predictors of programming performance. The comprehensiveness of this study can be highlighted from the number of predictors which were identified from the analysis. From the systematic review phase, 482 previous results describing the relations between 116 predictors and programming performance were extracted from relevant articles. Meta-analysis techniques were then applied to statistically synthesise the findings of multiple studies which had examined the same predictor of programming performance across different teaching contexts, revealing the most important factors for programming success. By utilizing the Random Effects model [78], the results of the meta-analysis can be generalised to different teaching contexts, regardless as to differences in setups (e.g. language taught) or aspects of the students themselves (e.g. prior experience). In other words, the meta-analysis answers the overall question of this thesis by identifying the factors which enables certain students to develop programming skills whilst other students endlessly struggle.

- Applied knowledge transformation by classifying the 116 predictors which were identified from the meta-analysis into a six class theoretical framework. This framework was applied to consolidate the meta-analysis findings by highlighting which types of factors are more relevant to programming than others, and could be applied by future researchers to derive practical applications.

This thesis also makes several minor contributions to knowledge, which are presented alongside the studies documented in Chapters 4-7. Examples include:

- Quantitative evidence suggesting that the failure rates of programming courses have not substantially changed over time.

- A predictive model built using the context independent predictors that were based upon traditional learning theories which we identified in this thesis.

- Validation of the previous research into the most common types of errors which novice programmers generate.

- Comparison of the data-driven predictors and the predictors based upon traditional learning theories which were explored in this thesis.

- To provide a synthesized benchmark on the effects of different predictors on programming, which future researchers can compare their results with.

1.5 Thesis Organisation

This thesis is structured using 8 chapters, including the current chapter, and comprises research undertaken at the University of Durham over the period from October 2010 to March 2014.

- **Chapter 2** provides background information relating to the three research objectives of this thesis. The difficulties of learning to program from a theoretical perspective are first discussed. Previous research into predictors based on traditional learning theories and programming behaviours are presented.

- **Chapter 3** presents the research methodology employed by this thesis. The methods used to answer the four research questions are discussed, and the statistical tests used throughout this thesis are outlined.

- **Chapter 4** explores whether aspects of the external teaching context have a moderating effect on the success rates of programming courses. Data describing the outcomes of 161 worldwide courses was gathered through a systematic review process and analysed.

- **Chapter 5** evaluates 34 predictors based upon traditional learning theories and their relation to the programming performance of a sample of students from our context. Based upon these results, a context-tuned regression model is presented.

- **Chapter 6** explores predictors based upon programming behaviour. Using three datasets describing the logged programming activities of three cohorts from our context, 10 new predictors based upon programming behaviour are identified. Based upon these findings, we propose a predictive algorithm, designed to quantify several aspects of programming behaviour which describes how desirable a student's programming behaviour has been over the duration of a session.

- **Chapter 7** presents a comprehensive systematic meta-analysis of the quantitative prior research into predictors of programming performance, and attempts to resolve the conflicting findings which have plagued the research over the past fifty years. A total of 482 results describing the relations of 116 distinct factors with programming performance are synthesised. Based upon our results, a six class theoretical framework of factors predictive of programming is proposed, which highlights to researchers the most important factors for programming success.

- **Chapter 8** concludes and summarises the contributions of this thesis and discusses possible directions for future work.

1.6 Publications

The work presented in this thesis appears in the following peer reviewed publications:

- C. Watson and F.W.B. Li. Failure Rates in Introductory Programming Revisited. In *Proceedings of the 19th ACM Conference on Innovation and Technology in Computer Science Education* (ITiCSE '14), pages 39-44, 2014, ACM.
  Best Paper Award.

- C. Watson, F.W.B. Li, and J.L. Godwin. No Tests Required: Comparing Traditional and Dynamic Predictors of Programming Success. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education* (SIGCSE '14), pages 469-474, 2014, ACM.

- C. Watson, F.W.B. Li, and J.L. Godwin. Predicting Performance in an Introductory Programming Course by Logging and Analyzing Student Programming Behavior. In *Proceedings of the 13th IEEE International Conference on Advanced Learning Technologies* (ICALT '13), pages 319-323, 2013, IEEE.
  Outstanding Paper Award.

The following publications are in preparation for submission, or currently under review:

- C. Watson et al., Meta-Analysis of Fifty Years of Research into Predictors of Programming Performance. *ACM Transactions on Computing Education.*

- C. Watson et al., Watwin Revisited: An Exploration of Students Programming Behaviour in BlueJ. *IEEE Transactions on Education.*

- C. Watson and A. Vihavainen. The CS1 Failure Rate Phenomenon. *ACM Inroads.*

Parts of this thesis are referenced or extended by the following publications:

- A. Vihavainen, J. Airaksinen, and C. Watson. A Systematic Review of Approaches for Teaching Introductory Programming and their Influence on Success. In *Proceedings of the 10th ACM Conference on International Computing Education Research* (ICER '14), pages 19-26, 2014, ACM.

- C. Watson, F.W.B. Li, and J.L. Godwin. BlueFix: using Crowd-Sourced Feedback to Support Programming Students in Error Diagnosis and Repair. In *Proceedings of the 11th International Conference on Advances in Web Based Learning* (ICWL '12), pages 228-239, 2012, Springer.

Chapter 2

Background

In 2001 McCracken et al., [117] published a multi-national, multi-institutional study of the programming abilities of students following their completion of either a first or second programming course. 216 students from four different universities completed an identical assessment of programming skills. This assessment was composed of three problem solving exercises based upon arithmetic expression evaluation (e.g. prefix notation). The researchers felt that these problems would be solvable by students taking any computer science program, but the results were surprising. Students performed terribly on the assessment, and only scored an average of 22.89/110 points. The main question which followed from this study was *why did the students perform so badly?*

Lister et al., [100] hypothesised that one of the reasons why students performed badly was because the assessment asked too much of them. A new instrument consisting of 12 multiple choice questions was designed to assess programming ability based upon two dimensions. In the first part, students were tested on their ability to predict the outcomes of executing a small fragment of code. In the second part, students were tested on their ability to identify the correct fragments to complete a block of code to achieve a specific outcome. 556 students from 7 different countries participated. The results showed that students fared better than those under the McCracken study, but an average score of 60% was still disappointing. The question still remains, *why did the students perform so badly?*

In this section we provide background material for the studies which were performed in this thesis. Firstly, general terminology on programming concepts are defined. Secondly, the difficulties of learning to program from a theoretical perspective are discussed. Following this, predictors based upon traditional learning theories which will be applied in our teaching context are discussed. Finally, a review of data-driven predictors which are based upon programming behaviours is presented.

2.1 Definitions and Terminology

We first briefly define concepts which are referred to throughout the remainder of this chapter. These definitions are based upon the detailed descriptions provided in [146].

Types of Programming Languages

There are many different types of programming languages. We briefly define three types which are relevant to this thesis.

- *Object-Oriented (OO) Programming* is based upon the fact that in our daily lives we interact with thousands of "real world" objects, and each of these objects have their own set of capabilities which we utilize. Example languages include Java and C++. Generally OO languages consist of the following features:
  - *Classes and Objects.* A class consists of both the variables needed to define the computer representation of an object, and the methods which are needed to interact with it. Informally, the class can be thought of as a "master template" and an object a copy of the template with specific settings. For example, a Ball class could contain a *variable* to store the size of a Ball. By creating *instances* of the Ball class, it is possible to create many Ball objects from the same class, but by changing the value of the size variable, it is possible to create an infinite amount of different Ball objects.
  - *Inheritance* is an OO feature which allows classes to reuse the variables and methods provided in another class. For example, a specialised BeachBall class can inherit the methods and variables to store BeachBall size by inheriting these aspects from the Ball class.
  - *Polymorphism and Overloading.* Different kinds of objects often have similar methods. Similarly, the same operator in a language can have different meanings depending on what data types it is being applied to. For example, the "+" operator in Java is overloaded and can be applied for both String concatenation, and numerical addition.
  - *Encapsulation* refers to the ability to keep the detailed workings of a class private, which in turn promotes program reliability. Many OO languages make use of *libraries* of classes which provide programmers with a small core of essential features (such as math routines). The underlying implementations of these classes are hidden from the programmer, but once they understand the capabilities of different classes then it becomes straightforward to understand the overall operation of the program.

- *Procedural Languages* are based upon the execution of set of statements which are executed in sequence, although branching (if statement) and repetition of statements (loops) can be used to control the program until a required condition is satisfied. An example of a procedural language is C.
- *Functional Languages* differ to procedural languages in that the values of functions are not assigned to variables. Instead the functions are manipulated directly together with the data items which are arranged in lists. These languages have been suggested as advantageous to programmers who prefer to state problems in a more purely mathematical way. An example of a functional language is LISP.

Programming Process

The process of creating a program generally involves several steps, including:

- *Program Design* refers to the process of developing a mapping between the expected functionality of the program and the syntax used to construct it.
- *Program Generation* refers to the process of writing syntax to build a program.
- *Compilation* refers to the converting of a program written in a source language, and generating an equivalent program which is capable of running on a computer.
- *Execution* refers to the running of the program following a successful compilation.
- *Debugging* refers to the process of identifying errors in programs. Syntax errors can prevent the successful compilation of the program. Runtime errors can terminate program execution following a successful compilation. Semantic errors refer to differences between the expected and actual behaviour of the program.

Tools for Program Implementation

Generally, two types of IDE's (Integrated Development Environment) can be used:

- *Professional Development Tools*, such as Netbeans, are industry standard tools which can be used to construct complex programs. These tools usually feature a text-based editor for entering syntax, and provide numerous support mechanisms to the programmer such as debugging assistance and code completion.
- *Novice Friendly Environments*, such as BlueJ, are tools which are targeted specifically at novice programmers. They are generally less complex than the professional development tools and offer less advanced support features.

2.2 The Difficulties of Learning to Program

It has been stated that it can take up to 10 years to transform a novice programmer into an expert [153, 212]. In this section we briefly discuss some of the theories which have been put forward to explain why programming is widely perceived to be such a difficult skill to learn. These include difficulties relating to program comprehension and generation, the programming language used in the course, threshold concepts, mental models, multiple skills, teaching methodology and differences in students' abilities.

2.2.1 Program Comprehension

Program comprehension describes the cognitive processes which are applied by students in order to understand programs. It is a particularly important skill for novice programming students to master, as a large proportion of their learning is reliant upon their ability to comprehend example programs [206].

Several strategies which programmers employ to comprehend and build a mental representation of programs have been identified. These include top-down strategies, bottom-up strategies, opportunistic strategies, and integrated metamodels [163].

Top-down strategies describe program comprehension as the process of reconstructing knowledge about the domain of the program, and then mapping this knowledge to the source code. Rather than studying programs line by line, programmers form a hypothesis about the general nature of the program, and then verify and refine their hypothesis by examining the source code for the presence of specific functions and structures [23]. In contrast, bottom-up strategies describe program comprehension as the process in which programmers start with individual code statements, and continually group these statements into higher level abstractions until a complete mental representation of the program is formed [137, 179].

Other researchers reject the notion of either a dominant top-down or bottom-up strategy. Opportunistic strategies describe programmers who adopt a top-down or bottom-up strategy depending upon the nature of the task they are trying to perform [104]. Integrated metamodels further refine this notion by describing program comprehension in terms of a domain, program, situation, and knowledge base, which programmers utilize to comprehend source code through applying a hypothesis driven set of *how*, *what*, and *why* conjectures [98, 179].

The main difficulty of programming comprehension for novice students is that all four of the comprehension strategies we have discussed require skills and knowledge which they may not necessarily have. Knowledge of the problem domain is one of the advantages that experts possess over novices, even if they know no more about the programming language than novices, they also know about the problem domain and can utilize that knowledge to help them comprehend programs [153]. In contrast novices are limited to surface level and superficially organised knowledge as they lack detailed mental models, fail to apply relevant knowledge, and approach problems line by line, rather than applying meaningful program chunks and structures [212].

The expertise required for program comprehension appears to differ based upon the type of programming language. Wiedenbeck et al., [206] compared the program comprehension skills of novice students studying Pascal to novice students studying C++. In the case of short programs, no overall differences in the comprehension skills of students was found, but the C++ subjects were superior to Pascal students in answering questions about program function. In the case of large programs, the Pascal students outperformed the C++ students both in terms of overall score, and on each of the individual task scales (operations, control flow, data flow, functions). They suggested that the problem was that novices had a tendency to focus upon understanding the program domain, rather than the problem domain. This causes great difficulties when trying to use C++ where the entire premise of object oriented languages is to enable students to focus upon the problem domain.

2.2.2 Program Generation

Even if students are able to comprehend example programs, previous research has suggested that there is little correspondence between the ability of students to comprehend a program, and their ability to actually write one [212]. These difficulties are possibly exemplified by the fact that even after passing an introductory programming course, students can still struggle to design and implement basic software systems [55].

In early work Rist [151] identified several differences in the approaches to program design adopted by intermediate and novice students. Intermediate students were found to design programs by retrieving almost complete code blocks from memory, and expanding and integrating these block to form a program in a top-down manner - working from the beginning of the program to the end. However students who are working on a problem which they have not encountered before do not have any blocks available in memory, and are forced to develop a program by linking knowledge fragments together, block by block, in a bottom-up approach.

Venables et al., [188] suggested that the programming skills of novice students are developed in a hierarchical manner. As students gain proficiency in code tracing, their ability to comprehend and explain code develops. When students are reasonably competent at both code tracing and explaining, the ability to systematically write code then develops.

2.2.3 Choice of Programming Language

The programming language which is taught in the course is widely believed to have a direct effect on the amount of difficulties which novice students face. There has been a great deal of debate over the most appropriate choice of paradigm and language for introducing programming to novices [136].

We first consider the choice of paradigm. Historically many programming courses were taught using procedural languages such as C [99], but these languages also came with many criticisms. Mody [127] observed that students encountered significant difficulties when transitioning from C to a higher level language. He also raised concerns over the impact C had on the approaches students took to learning in other courses. In particular, data structures was described as a course which had degenerated into a set of pointer exercises, rather than focussing upon implementing discrete structures. Tang [45] also criticised teaching C as a first programming language, stating that students with little to no programming knowledge are easily overwhelmed by the complex pointer and address operations the language requires.

Unsurprisingly in modern times procedural languages have mostly been replaced by object-oriented (OO) languages such as Java and C++ [48]. The original motivation behind this paradigm shift was to lower the learning curve for novice programming students. By creating programs using virtual objects, in theory students would only have to understand the problem domain in order to design the computer implementation. However, this was quickly found not to be the case, and researchers such as Robins [153] suggested that programming using an OO language can in fact be more difficult than using procedural languages, due to the conceptual difficulties that novices face when performing this task. Novices can easily become confused when identifying objects and are hindered by identifying objects that are not useful to solving the specific problem.

Wiedenbeck et al., [206] suggested that the distributed nature of OO programs (i.e. spreading functionality across a set of interacting objects) leads to a harder learning curve than for procedural languages. They found that students studying OO languages were superior to students studying procedural languages when answering questions about program functions. However, procedural students were superior to OO students when answering questions about program comprehension. They suggested that the OO paradigm may need to be taught after novices have gained some experience with basic programming concepts, such as data types, assignment, branching or looping.

Clark, MacNish and Royal [38] compared Java to procedural languages such as Pascal. They noted that one of the difficulties of teaching OO programming is that students are usually required to work on relatively small scale projects, and may struggle to appreciate the problems that OO programming tries to address, e.g. polymorphism.

Biddle and Tempero [18] examined the potential issues of using Java as a language to teach novice programmers. Although not commenting on whether the OO paradigm was itself reasonable for teaching novices, they concluded that the high level of Java may be detrimental for novices who first need to understand the fundamentals of data and control in order to understand how computing works. A number of pitfalls that educators must be aware of when teaching Java to novices were also presented, such as the differences between primitive and object types, and the separation of interface and implementation is undermined.

Other researchers have studied the appropriateness of different OO languages for teaching programming to novices. Generally, the main issues which arise across all studies is that general purpose OO languages such as Java or C++ have not been designed specifically for educational purposes, and therefore may be less effective at teaching novices than using those languages which have specifically been designed for this purpose (e.g. LOGO, Scratch, Alice, or Eiffel) [136].

Hadjerroult [69] compared the teaching of Java to C++ as a first programming language. He concluded that although file handling was more difficult in Java than in C++, the majority of shortcomings of both languages concerned the general difficulties associated with learning the OO paradigm [153].

Kölling [89] presented 11 requirements of a well designed programming language for teaching purposes. Applying these principles to evaluate C++, Smalltalk, Eiffel and Java, Kölling found that each language had shortcomings. C++ failed to meet almost all of the requirements, with the most serious criticism being the lack of type safety. Smalltalk was found to have a lack of static typing. Eiffel was criticised for the complexity of the language. Java was criticised for the lack of clear distinction between primitive and object types for novice students. However when considering experience reports from instructors in addition to the language evaluations, he concluded that OO languages were seen unequivocally as a powerful and valuable teaching tool. The main difficulty reported was switching from the procedural languages to the OO paradigm, such as decomposing a problem into appropriate objects.

Manilla and de Raadt [109] performed a similar study to Kölling [89]. Analysing the work of previous creators of languages that are intended specifically for teaching (LOGO, Pascal, Python, Eiffel) a total of 17 requirements of a well designed programming language were defined. These requirements were then applied to evaluate the appropriateness of 10 different languages used for teaching novice programmers. Eiffel and Python were found to satisfy the most requirements (15), and were closely followed by Java (14) and C++ (11). Although designed specifically for teaching novices, the LOGO (9) and Pascal (7) languages were among the lowest ranked by the study.

2.2.4 Threshold Concepts

Another explanation as to why programming is difficult for novice students to learn is because there are a number of threshold concepts that are associated with programming. Meyer and Land [122] proposed that within academic disciplines, there are a number of troublesome concepts (thresholds) which can hinder the students' learning progress until these concepts are mastered. The threshold concepts usually correspond to the core concepts within a discipline, which provide a potentially transformative point in students' understandings of the entire discipline. Generally, threshold concepts are:

- *transformative*: they significantly transform how a student perceives a subject, or part thereof, and perhaps even causes a shift in personal identity.
- *integrative*: they connect concepts in ways that were previously unknown to a student by exposing their interrelatedness.
- *irreversible*: they are difficult for a student to unlearn, making the transformation of knowledge unlikely to be forgotten or undone.
- *boundary markers*: they mark boundaries in the conceptual space between disciplines or schools of thought.
- *troublesome*: potentially very troublesome for a student to overcome, for any of a variety of reasons including conceptual complexity and counter-intuitiveness.

Understanding the OO paradigm has been identified as a threshold concept [22], and authors such as Luker have argued that that learning OO programming, "requires nothing less than a complete change of the world view" [106]. There are numerous other concepts which are transformative of the way which students' perceive computing, and programming in particular [215]. Examples include:

- Pointers, in particular when used as parameters [22].
- Distinction between classes and objects [56].
- Recursion [158].
- Polymorphism [129].
- Distinction between the existence of a program as code, and its existence as a dynamic execution time entity within a computer [172].
- Object interaction [172].
- Abstraction, in particular modularity, decomposition and information hiding [128].

2.2.5 Mental Models and Misconceptions

Other researchers have explored how novices mental models relate to their learning of programming concepts. In general, the bulk of research into mental models and programming performance has suggested that novice programmers can struggle to understand programming concepts as they lack a clear mental model of how their programs relate to the underlying system [153].

Mayer [111] suggested that the use of concrete models to present the computer system and encouraging novices to describe technical information in their own words can help them to solve problems which are not explicitly taught within the course. Pea [135] hypothesised that students' misconceptions across a range of different programming languages may stem from a mental "superbug". He found that three different types of misconceptions could be attributed to students incorrectly believing that the computer has intelligent interpretive powers, and is capable of automatically going beyond the code which students supply to help them achieve their goals.

Boulay [21] suggested that many of the misconceptions novice students make about variables could be caused by the misapplication of analogies in materials. As an example, the analogy of using a box as a place to store variables could lead students to build the wrong mental model that a variable can hold more than a single value at a time. He also suggested that novices can find the syntax and underlying semantics of a programming language difficult to understand as they lack knowledge about the capabilities of a computer. In other words, their mental models about the machine they are trying to program is inadequate for their learning requirements.

Perkins and Simmons [139] noted that novices can also foster misconceptions about the names of variables, describing a case where students studying Pascal incorrectly had the notion that by naming a variable "largest" that the computer would automatically know to store the largest of a series of numbers it reads into that variable because it understands the semantic unit "largest".

Bayman and Mayer [11] asked a sample of novice programming students to explain the steps that a computer would take to execute a range of statements written in BASIC. They found that students held several misconceptions relating to variable assignment, initialisation, storage, and printing. For example, students believed the equals symbol in a statement was an equality. They suggested programming practice is often insufficient to teach novices programming, as long as their underlying conceptions are still limited by an inaccurate mental model of the computer.

Fleury [62] examined the correctness of mental models of parameter passing which were held by Pascal students. Several misconceptions were identified e.g. when the value of a local variable is changed the value is then available throughout the program.

Ma et al., [107] examined mental models of assignment (of values and references) held by students who had completed a Java programming course. On course assessments, the students who held correct mental models were found to significantly outperform students who held incorrect models. The concerning finding of this study was that 33% of students who completed the course still held incorrect mental models on value assignment, and only 17% of students held correct models of value assignment.

Dehnadi and Bornat proposed a performance predictor based upon mental model consistency [49], although later studies suggested the predictor struggled to generalise to other teaching contexts [19, 33].

More recently Kaczmarczyk et al., [86] identified four types of misconceptions that were common to novice programming students. These included misunderstandings on the relationships between language elements and underlying memory usage, the process of while loop operation, the object concept, and debugging.

2.2.6 Multiple Skills

Another source of difficulty concerns the fact that programming is not simply a single skill which can easily be acquired. Rather, it is a complex cognitive activity where competence can only be developed through mastery of the entire programming work flow [153]. Boulay [21] suggested that the five potential sources of difficulty facing novice programming students include:

- *Orientation*: developing a general understanding of what kinds of problems can be solved by using programs, and the advantages of learning the skill.
- *The Notional Machine*: developing a mental model about the machine that they are learning how to control with the programming language.
- *Notation*: mastering the syntax and semantics of the formal language they are trying to understand.
- *Structures*: after mastering notation, students attempt to master structures or plans, which can be used to achieve small-scale goals (e.g. computing a sum).
- *Pragmatics*: learning the skill of how to specify, develop, test, and debug a program using whatever tools are available.

The difficulty is that none of these skills are entirely separable from the others. Students will be unable to satisfy the pragmatics until they have an understanding of the notation. It is thought that some of the difficulties that novice programmers faced can be attributed to the students ineffective attempts to deal with all difficulties at once.

2.2.7 Teaching Methodology

Other difficulties that novices face can stem from an inappropriate teaching methodology. Gomes and Mendes [64] raised several criticisms over the approaches which are generally used to teach programming. Firstly, there is the classic criticism that the instruction is not personalised, and that students are not provided with useful immediate feedback during problem solving. Secondly, it is difficult for students to understand program dynamics when instructors rely upon using static learning materials. Thirdly, the teaching of programming tends to be focussed upon the syntactic correctness of programs, rather than promoting problem solving using a programming language.

Studies have also shown a disparity between which concepts instructors believe students find difficult, and the concepts which students themselves believe are difficult. For instance, Lahtinen et al., [94] conducted an international survey of over 500 students and instructors to get opinions about the difficulties of programming. They found students felt confident to study alone and that instructors perceived course contents as more difficult than the students did themselves. They suggest that one of the main difficulties facing novices is the lack of effective problem solving materials.

These results were comparable to a similar study by Milne and Rowe [126] who found that instructors and students agreed on the most difficult concepts, but instructors scored them as more difficult than students.

2.2.8 Differences in Student Abilities

Finally novices face the challenge of learning in an environment where the abilities of their classmates can vary considerably. Jenkins [83] suggests that the wide range of students who take programming courses can make it difficult for instructors to teach programming at a rate which keeps all students sufficiently engaged. Even for novice students, Robins et al., [153] suggest that there is a clear distinction between what you may class as an "effective novice" (i.e. does not need much support), versus an "ineffective novice" (i.e. cannot learn without an excessive amount of support which is detrimental to their classmates).

Gomes and Mendes [64] also suggest that different students will have different problem solving abilities, and motivations. Unlike the other subjects which students may be studying, programming is not a single skill which can be learned from a textbook. It even takes a considerable amount of effort for a novice to write and understand a basic "hello world" program. Essentially, the effort required to produce a compilable program, combined with the lack of instant gratification could have a negative effect on students motivation and cause them to suffer from a lack of persistence.

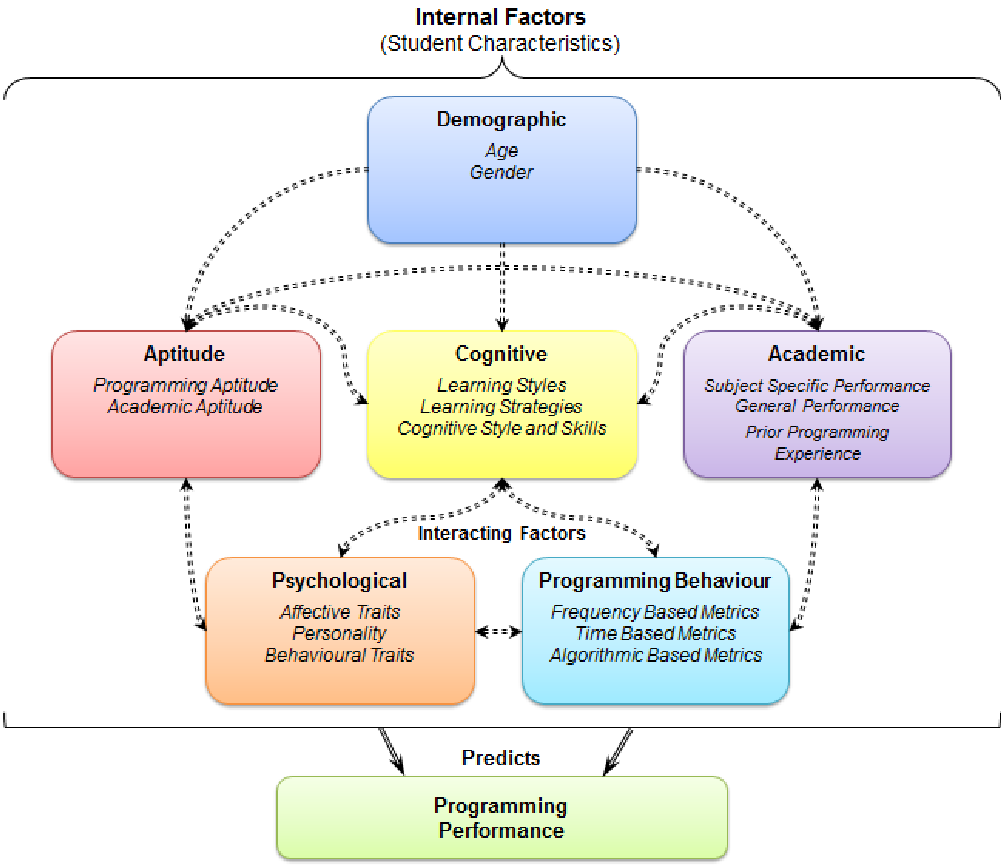


Figure 2.1: Predictors which researchers have examined over the previous fifty years. Interacting factors were determined by related studies in the literature, and are shown on the diagram by using dashed arrows (further detail is presented in Section 7.4)

.

2.3 Predictors based upon Traditional Learning Theories

Exploring the relations between traditional learning theories and programming performance has been the focus of research for over fifty years. Many researchers have explored how various academic, psychological, demographic, behavioural, and cognitive factors relate to the programming performance of students. Most recently, researchers have begun to explore data-driven predictors, which describe how the programming behaviour of students relates to their performance. An example of some of the predictors which have been under investigation is shown in Figure 2.1.

Whilst numerous predictors have been examined over the past fifty years, previous research has been limited by both a lack of verification of predictors in different teaching contexts, and the conflicting results which are commonly reported when verifications take place. Due to these issues there is no general consensus among researchers on which factors support students in developing programming skills. This makes it extremely difficult to design and implement any effective pedagogical tools or strategies which can be applied to identify and support the weakest students.

In this subsection, we present some evidence of the inconsistent findings surrounding the predictors which will be evaluated in our teaching context. This subsection is not intended to serve as a complete guide on predictors of programming performance. The remaining predictors (such as those based upon programming aptitude) are presented in Chapter 7, where we performed a substantial meta-analysis to synthesize the findings of multiple studies that have examined the same predictors across different teaching contexts. The comprehensiveness of the work can be highlighted from the number of predictors which are included in the analysis. From the initial review, 482 previous research results describing the relations between 116 predictors and programming performance were synthesised and classified into a theoretical framework of factors predictive of programming performance.

For now, we ask that the reader accepts our conjecture that inconsistent findings are common across almost all of the traditionally explored predictors, and that further evidence supporting this claim will be presented later in this thesis (Chapter 7).

2.3.1 Cognitive Predictors

Several researchers have considered the impact of various cognitive characteristics on the learning of programming. These characteristics are based upon the various mental processes that influence the ways in which learners think and learn, and the cognitive learning styles and strategies that learners employ to make knowledge and skill acquisition possible. The influence of different cognitive factors have been well explored over the past fifty years, however they are also one of the classes of predictor where researchers have reported a high number of inconsistent results. The relations between two forms of cognitive characteristics and programming performance will be explored in our teaching context: learning styles and learning strategies.

Learning Styles

A learning style describes how learners perceive, interact with, and respond to different learning situations. The implication of learning styles is that different learners will respond uniquely to different teaching approaches, and in order for instruction to be most effective, the teaching approach employed should be as closely aligned with an individual's learning style as possible.

Kolb's Learning Styles Inventory (LSI) consists of four distinct styles, which are based on a four-stage cycle of learning. Kolb's model describes learning as a continual cycle of involvement, where learning commences with concrete experiences and is followed by a period of reflection, observation, and application of those experiences in different situations to solve problems. Based on the four quadrants of the cycle, there are four corresponding learning styles: converger, diverger, assimilator, and accomodator. Research on the relations between Kolb's model and programming performance has yielded inconsistent results. Considering the assimilator dimension, Corman [42] found no relation between scores obtained on the assimilator dimension and the programming performance of a sample of 83 university students, r = .06. On the other hand, Chammilard and Karolick [36] reported a weak negative correlation, r = -.13, on a sample of 877 students, and Campbell and Johnstone [29] reported a moderate correlation, r = -.36, on a sample of 74 students. The exact relation between Kolb's LSI and programming performance is therefore unclear and warrants further exploration.

The Gregorc Style Delineator also consists of four distinct learning styles, which are based upon perceptual quality and ordering ability. Perceptual quality is used to define learners on two dimensions: concrete, which describes learners who prefer to learn from authentic experiences, and abstract, which describes learners who prefer to use their own intuition for learning. Ordering ability is further used to classify concrete/abstract learners on two further dimensions: sequential, which describes learners who prefer to learn from information presented in a logical sequence, and random, which describes learners who prefer to receive information in their own way. Two studies have previously explored the relations between Gregorc's Style Delineator and programming performance, and as with studies which explored Kolb's LSI, inconsistent results were reported. Considering the scores obtained on the abstract/sequential dimension, Lau and Yuen [95] reported a weak correlation, r = .13, on a sample of 217 high school students. Two years later, the experiment was repeated, and Lau and Yuen [96] reported a moderate correlation, r = .30, on a sample of 131 high school students. The exact relation between Gregorc's Style Delineator and programming performance is therefore unclear and warrants further exploration.

Learning Strategies

A learning strategy describes the techniques and methods that learners employ to satisfy the learning objectives of a course. They differ from learning styles, as they describe techniques that learners employ in specific learning situations, rather than characteristics describing the general approach to learning that they adopt. The learning strategies that learners adopt can vary depending on the nature of the academic tasks that they are required to perform. It follows from recent research that learners who possess a high level of motivation, and use efficient learning strategies, are more likely to satisfy learning objectives and be more successful than their counterparts who do not [34].

Self-regulated learning is one form of learning strategy. Self-regulated learning is an active constructive process whereby learners set goals for their learning and monitor, regulate, and control their cognition, motivation, and behaviour, guided and constrained by their goals and the contextual features of the environment [143]. The Motivated Strategies for Learning Questionnaire (MSLQ) is an instrument designed to measure a learner's motivations and self-regulated learning in classroom contexts. It is based on a cognitive view of motivation, and is composed of two sections: a motivation section and a learning strategies section [14]. To date, only Bergin and Reilly [14] have explored the relations between the scores obtained on the MSLQ scales and programming performance. Using a sample of 34 University students, several strong correlations were reported, including: critical thinking, $r = .58$, self efficacy, $r = .57$, task value, $r = .54$, and resource strategy (effort), $r = .62$. Whilst these results are encouraging, they are only derived from a single teaching context, and we currently do not know whether these findings are generalisable to different teaching contexts.

2.3.2 Psychological Predictors

Other researchers have considered the impact of various psychological characteristics on the learning of programming. These characteristics are based upon the various non-cognitive factors that influence the ways in which learners think and learn. They describe the personal characteristics of the student, describing how they differ from, or are similar to others, in a non-cognitive sense, such as in terms of their personality, self esteem, or self efficacy. The influence of different psychological factors have only been recently explored over the past ten years, therefore a lack of verification is the limiting factor of research in this area. The relations between two types of psychological predictors and programming performance will be explored in our teaching context: affective characteristics and behavioural characteristics.

Affective Characteristics

Affective characteristics generally describe the emotional state of a learner, and include factors that can reflect their feelings and attitudes during the learning process. The relations between two different affective characteristics and programming performance will be explored in this thesis. Several researchers have suggested the importance of self-esteem as a critical component for learning. As a result, teachers have focused efforts on boosting the self-esteem of learners, working on the assumption that if their self-esteem can be improved, then it will in turn foster improved learning outcomes. The benefits of self-esteem can be divided into two categories: enhanced initiative and pleasant feelings. Enhanced initiative comes from the observation that people with high levels of self-esteem tend to be more confident and willing to speak up in groups, and to be critical of the group's approaches. Pleasant feelings comes from the observation that people with high levels of self-esteem claim to be more likable and attractive, have better relationships, and to be less stressed than people with low self-esteem. However, whether these benefits can lead to an enhanced learning performance is still the subject of much debate [9, 10].

The Rosenberg Self-Esteem Scale is one of the most widely used instruments used to evaluate an individuals self-esteem. The instrument consists of a 10-item scale that assesses an individual's feelings of self-worth when the individual compares themselves to others. Only one previous study to date has attempted to identify the relation between programming performance and self-esteem. Bergin and Reilly [14] reported a moderate correlation, $r = .36$, between the scores obtained on Rosenberg's Scale, and the programming performance of a sample of 54 University students. However, we do not know whether this finding holds in a different teaching context.

The second affective characteristic which will be explored in this thesis is attributional style. An attribution describes the reason that people perceive to be the cause of events that are related to themselves or others. An attributional style defines the consistent way in which people tend to account for the outcomes of these events. Several studies suggest that the types of attributions and both the expectancy beliefs and emotions that learners experience as a result of the attributional process can determine their future behaviours [5, 211]. One of the classic attributional style models was proposed by Weiner [201], who defines the processes through which learners form attributions in terms of two dimensions: locus of causality, and stability. Locus of causality refers to whether the cause of an event was perceived to be internal or external to the learner. For instance, if a learner believed that they failed an exam because they lacked ability, then this would be an internal cause. The stability dimension refers to whether the cause of event can change over time. Stable causes are those that tend to influence events and behaviours consistently over time and situations. Unstable factors are comparatively easy to change over time, such as the amount of effort exerted to complete a task. Based upon the four combinations of stability and locus of control, four attributions are derived: luck, ability, effort, and task difficulty.

To date, only three studies have explored the relations between attributional style and programming performance. Whereas Henry et al., [74] reported a weak positive correlation, r = .25, between attributions to task difficulty and the programming performance of 45 University students, Wilson and Shrock [210] reported a weak negative correlation, r = -.20, on a sample of 105. Other inconsistencies are common. Wilson and Shrock also reported a weak positive correlation, r = .19, between attributions to effort and performance, whereas, Ventura [190] reported no relation, r = .06, for a sample of 249 students. This suggests that the relations between attributional style and programming performance are unclear and warrants further investigation.

Behavioural Characteristics

Behavioural characteristics generally describe aspects of a learners day to day behaviour that can influence their learning progress. In our teaching context two behavioural characteristics are explored: hours worked in a part time job, and lecture attendance.

Only a single study to date has reported any correlations between lecture attendance and programming performance. Allert [1] reported a weak correlation, r = .12, between these two variables on a sample of 139 University students. The relation between hours spent working in a part time job and programming performance has also only been explored by a single study. Bergin [16] reported no correlation, r = .05, between these two variables on a sample of 30 University students.

2.3.3 Academic Predictors

There has been considerable interest in developing an understanding of the wider influence that programming skills can have on a student's performance in related academic subjects, with an emphasis on Math and Science. As with the previously discussed predictors, there are also a high number of inconsistent findings.

Previous Programming Experience

Although the intuitive notion would perhaps be that previous programming experience would be advantageous to students' taking a programming course, previous research has yielded inconsistent results on the benefits of prior experience. Jones and Burnett [84] reported a moderate correlation, r = .44, between the presence of prior programming experience and the programming performance of 49 University students, and similar correlations were reported by Wiedenbeck [205], $r(120)$ = .27, and Sheard [164], $r(51)$ = .28. However, other researchers have reported no-to-weak correlations, including: Byrne and Lyons [28], $r(110)$ = .10, and Ventura and Ramamurthy [189] $r(256)$ = .06. A possible explanation is that all students reach a similar level of programming ability after completing a course.

Academic Background

As with the previous predictor groups, inconsistent results can be found across a range of academic predictors. Capstick [30] reported a strong correlation, $r = .58$, between calculus and programming performance of 24 University students, whereas Stein [178] only reported a moderate correlation $r(160) = .20$. An initial study by Koubek [91] found a strong correlation, $r(134) = .62$, between chemistry and programming performance, whereas a later study only revealed moderate correlations, $r(606) = .31$. Whipkey [203] identified a strong correlation, $r(88) = .73$, between high school GPA and programming performance, whereas Werth [202] found no relation, $r(58) = .07$.

2.3.4 Context Independent Predictors

In terms of identifying context independent predictors of programming performance, a multi-national, multi-institutional study was conducted in 2004 using students from 11 different institutions (based in Australia, New Zealand, and Scotland). Four diagnostic tests were used to measure different traits of the students. These included: paper folding test (spatial visualization and reasoning ability), map sketching (ability to articulate decisions), phone book searching (ability to articulate a search strategy), study process questionnaire (deep or surface approaches to learning). The findings of this study were that spatial visualization was associated with success, along with adopting a deep learning approach, being able to articulate a search strategy in detail, and map drawing style (landmark, route, or survey) [168,169,184]. Whilst these studies were interesting, they also take considerable effort to perform. As such the majority of experiments into predictors of CS1 performance usually take place across considerably different contexts, often yielding considerably different results [198].

2.3.5 Hybrid Predictive Models

In this section, we have shown that the majority of predictors which will be examined in our teaching context have yielded considerably varying results when they are applied in different teaching contexts. However, it has long been recognised that constructing predictive models based upon different combinations of predictors can yield stronger results than when the predictors are considered singularly. Examples of such models are shown in Table 2.1. As with the performance of the predictors when considered singularly, it can be seen that there is a great deal of variation in the models which have been explored by previous research. Different predictors are almost always featured in different regression models, possibly suggesting that different factors are more relevant for programming in different teaching contexts than others.

Table 2.1: Regression models designed by previous researchers to predict programming performance and their explanatory power.

Researcher	Year	Level	Language	n	Predicting	Predictors	R^2
Biamonte [17]	1964	University	Not Stated	106	Overall letter grade.	Programmer aptitude test score, and logical analysis score.	34.8
Bauer [8]	1968	University	Fortran	68	Overall letter grade.	College GPA, IBM PAT Part 2 (Figure Series), and Strong Vocational Interest Bank (SVIB): Computer Programmer scores.	65.6
Willoughby [209]	1971	University	Not Stated	49	Overall letter grade.	CPAB reasoning, number ability, and diagramming scores, and ATGSB verbal an quantitative scores.	34
Willoughby [209]	1971	University	Not Stated	30	Overall letter grade.	CPAB reasoning, letter ability, number ability, and diagramming scores, and ACT mathematics and social studies score.	72
Mussio [131]	1971	College	Not Stated	93	Overall letter grade.	CPAB Diagramming score, Russell's Scale of Motivation for School Achievement score, Strong Vocational Interest Bank: programmer key and age related interests.	52
Alspaugh [3]	1972	University	Assembly	50	Total score on three programming exams.	IBM PAT Total, highest math course completed, TTS (impulsive, sociable, reflective, dominant, emotionally stable, active, vigorous) scores.	33.1
Alspaugh [3]	1972	University	Fortran	50	Total score on two programming exams.	IBM PAT Total, highest math course completed, TTS (impulsive, sociable, reflective, dominant, emotionally stable, active, vigorous), and Watson-Glaser critical thinking appraisal.	37.7
Jacobs [80]	1973	College	Fortran	45	Total score on the final exam.	SAT verbal score, Watson-Glaser inferential ability, and Watson-Glaser deductive ability.	18.1

Continued on next page

Table 2.1 – *Continued from previous page*

Researcher	Year	Level	Language	n	Predicting	Predictors	R^2
Capstick [30]	1975	University	Cobol	24	Overall numerical score.	Second semester calculus grade, IBM PAT letter reasoning score, and IBM PAT figure series score.	49.9
Newstead [132]	1975	University	Fortran	131	Overall letter grade.	College GPA, time spent on the course, and perceived ability.	41
Kurtz [93]	1980	University	Fortran	23	Overall letter grade.	Level of intellectual development.	63.3
Leeper [97]	1982	University	Unknown	92	Overall letter grade.	Units of high school english, math, science and foreign languages completed, SAT verbal and math score, and high school rank (position in graduating class).	26
Hostetler [77]	1983	University	Fortran	79	Overall numerical score.	CPAB reasoning and diagramming score, college GPA, most advanced math course completed, and whether a student perceived themselves as ’sober’ or ’happy go lucky’.	42.7
Whipkey [203]	1984	College	Pascal	88	Average score obtained on three separate exams of programming skills.	High school GPA and SAT Math score.	57
Nowaczyk [134]	1984	University	Fortran	160	Overall numerical score.	Age, gender, previous computer experience, previous performance in courses (math, english, foreign languages), expected grade in the course, attitude towards computer science, perceived locus of control, and number of word problems correctly solved.	45

Continued on next page

Table 2.1 – *Continued from previous page*

Researcher	Year	Level	Language	n	Predicting	Predictors	R^2
Nowaczyk [134]	1984	University	Cobol	60	Overall numerical score.	Age, gender, previous computer experience, previous performance in courses (math, english, foreign languages), expected grade in the course, attitude towards computer science, perceived locus of control, and number of word problems correctly solved.	81
Koubek [91]	1985	University	Fortran	1235	Overall letter grade.	SAT Math and Verbal scores, High School english, math, and science grades, and College chemistry, math, physics, and english grades.	29
Koubek [91]	1985	University	Fortran	497	Overall letter grade.	SAT Math and Verbal scores, High School english, math, and science grades, and College chemistry, math, physics, and english grades.	50
Koubek [91]	1985	University	Pascal Fortran	485	Overall numerical score.	SAT Math and Verbal scores, High School english, math, and science grades, and College chemistry, math, physics, and english grades.	57
Mayer [112]	1986	College	Basic	57	Exam score.	Word problem translation, word problem solution, and following directions, could explain 50% of the variance in performance.	50
Corman [42]	1986	University	Cobol	51	Average numerical score on course assessments.	Age, major GPA, and Myers-Briggs personality type indicator (sensing/intuition dimension).	45.3

Continued on next page

Table 2.1 – *Continued from previous page*

Researcher	Year	Level	Language	n	Predicting	Predictors	R^2
Austin [6]	1987	College	Pascal	76	Composite score of programming achievement.	High school achievement, quantitative and algorithmic reasoning abilities, vocabulary and general information abilities, iconic pattern recognition abilities, inverse of negative computer association measures, self-assessed math ability, cognitive style: basic level introvert/extrovert and cognitive style: primitive level extrovert.	64
McCoy [115]	1988	Middle School	Basic	73	Numerical score obtained on a 40 item Basic exam.	General problem solving ability.	20.3
Sall [159]	1989	University	Basic Pascal	615	Overall letter grade.	Year level, number of different programming languages, secondary English, all 22 variables (excluding gender, current help in programming, current computer access, and previous computer access).	68
Scanlan [162]	1990	University	Basic	45	Overall numerical score.	Spatial ability, logical inference, visual memory, integrative process, flexibility of closure, algorithm comprehension, computer science GPA, and flowchart / pseudocode preference.	65.6
Schute [166]	1991	College	Pascal	260	Average score obtained by a student on three separate tests of programming skills.	Working memory factor, word problem solving abilities (problem identification and sequencing of elements), and the number of hints requested from an intelligent tutoring system.	68
Henry [74]	1993	University	Pascal	45	Overall numerical score.	Attribution of success to ability, task difficulty, effort, and luck.	31.2

Continued on next page

Table 2.1 – *Continued from previous page*

Researcher	Year	Level	Language	n	Predicting	Predictors	R^2
Subramanian [180]	1996	High School	Basic	39	Overall numerical score.	Computer aptitude literacy and interest profile test scores (series, graphical patterns, logical structures, interest, estimation).	31
Subramanian [180]	1996	University	Basic	46	Overall numerical score.	Computer aptitude literacy and interest profile test scores (series, graphical patterns, logical structures, interest, estimation).	13
Goold [65]	2000	University	C	32	Overall numerical score.	Average score in other units, whether the student disliked programming, problem solving ability, raw secondary score, and gender.	43
Wilson [210]	2001	University	C++	105	Score on the mid-term exam.	Had previous programming experience, had previous non-programming experience, rating of attribution of success to task difficulty, luck, effort, ability, rating for self-efficacy and comfort level, gender, work style preference, units of high school math completed, and whether the student was encouraged to persue computer science.	44.4
Ramalingam [147]	2004	University	C++	75	Overall letter grade.	Mental models (measured in terms of program comprehension and recall) and post-course self-efficacy.	30
Bergin [15]	2005	University	Java	80	Overall numerical score.	Gender, previous programming and non-programming experience, cognitive test score, previous academic experience, encouragement from others, work-style preference and hours spent working an a part time job.	30
Bergin [14]	2005	University	Java	34	Overall numerical score on coursework.	Cognitive, metacognitive, and resource management strategies.	45

Continued on next page

Table 2.1 – *Continued from previous page*

Researcher	Year	Level	Language	n	Predicting	Predictors	R^2
Bergin [16]	2005	University	Java	33	Numerical score obtained on an early lab exam.	Intrinsic motivation, self-efficacy and comfort level.	60
Ventura [189]	2005	University	Java	380	Overall numerical score.	Comfort level, SAT math score, and percentage of time spent in the lab.	52.9
Wiedenbeck [205]	2005	University	C++	120	Final letter grade in the course represented numerically.	Post-course self-efficacy and knowledge organisation (program recall).	27
Wiedenbeck [207]	2007	University	Unknown	121	Score on six debugging questions on final exam.	Software-self efficacy, programming self-efficacy, computer playfulness, and computer interest.	28
Lau [95]	2009	High School	Pascal C	217	Numerical score on a programming test.	Partial Gregorc learning style delineator scores (concrete/abstract sequential).	14.4
Milic [124]	2009	High School	Pascal	79	Score obtained on two programming exams.	Program comprehension ability, general intellectual ability (KOG-3), and attitude towards computers (Selwyns).	39.7
Lau [96]	2011	High School	Pascal C	131	Score on a programming exam.	Partial Gregorc learning style delineator (concrete/abstract sequential), mental models, secondary school ability group, and language the course was taught in (English / Chinese).	43.6

2.4 Predictors based upon Programming Behaviour

Exploring the relations between the kinds of mistakes which students make, and how they go about the process of developing programs has been the subject of research for some time. However the recent upsurge in data-driven approaches for automatically gathering data and analysing behavioural patterns has provided researchers with new insights into the novice programming process. In this section we briefly review previous studies which have examined the relations between programming behaviour and programming performance. An overview of studies is shown in Table 2.3.

2.4.1 Stoppers, Movers, Tinkerers

Much of the early research into programming behaviour was conducted through observational studies. Piech et al., [141] note several studies which took place to examine the relations between novice programmers and specific programming constructs such as variables [160], looping [171], and methods [88].

In a prominent example, Perkins et al., [138] observed novice students writing LOGO programs. They found that when writing programs to solve basic problems, students' exhibited three types of programming behaviour. These were:

- *Stoppers* are students who simply stop working on a problem when they lack a clear direction of how to proceed. When faced with a problem to which they have no solution they feel at a complete loss and are unwilling to explore the problem any further.

- *Movers* are students who will keep trying, experimenting, and modifying their code in the hope of eventually solving the problem. Unlike stoppers they can use feedback about errors effectively and have the potential to solve the current problem by gradually working towards the correct solution.

- *Tinkerers* are students which could be labelled as ineffective movers. They are not easily frustrated by a program which fails the first time, they like to experiment with their code by making changes, and they also hold the belief that a problem is solvable. However, they differ from movers as they will try to solve a programming problem by writing some code and then making small changes in the hopes of getting it to work. These changes are not always systematic and almost random in nature, which can make the problems students are facing worse.

They also found that students' attitudes to mistakes are an important factor in their progress, and those who are frustrated by their mistakes or have a negative emotional reaction to making errors are more likely to become stoppers.

The main shortcoming of observational studies of this nature is that they take considerable effort to run. Nowadays, many researchers have elected to explore the programming behaviour of students by augmenting various systems to gather data by using on-line protocols. The data logging typically involves adding various extensions to IDE's and operating systems to capture and relay data which describe students' programming processes. This usually includes collecting snapshots of source code, either by a defined interval, or when students' perform actions such as saving the project they are currently working on, which then enables the creation of data-driven approaches for finding characteristics that contribute to students' success. The main benefit of data-driven approaches based upon how students solve programming errors, or how they schedule their time, and whether they pay attention to code quality, is that, they are directly based on the regular programming activities of students, and therefore can directly capture changes in their learning progress over time. This is not the case for the more traditionally explored predictors, e.g. learning styles or personality traits.

Possibly one of the earliest studies was performed by Spohrer and Soloway [177] who augmented the operating system of the VAX 750 that students used to store copies of each syntactically correct Pascal programs which they compiled. Exploring the traces of 158 students over three problems, they found that just a few types of bugs accounted for the majority of mistakes in students programs, implying that educators can most effectively improve their students' performances by changing instruction to address and eliminate the high frequency bugs. Many bugs arise as a result of plan composition problems, i.e. difficulties in putting problems together, and not as a result of construct based problems, i.e. misconceptions about different language constructs [176].

2.4.2 The Errors that Students Make

Since then, considerable effort has been placed into applying data-driven techniques to explore how the types of syntax errors which students make relate to their programming performance. On the most basic level, many studies have attempted to identify the types of errors which students make (e.g. [79, 81, 82, 181]). The most common errors identified by these studies are shown in Table 2.2.

It is interesting to note that the types of errors which students make appears to be consistent across different teaching contexts, and it is common knowledge as to which types of error occur most frequently [51]. Additionally, the bulk of errors which students make could be classified as "syntax errors", relating to unknown variables, classes, packages, and methods. These could easily be attributed to basic typos, or the fact that students are not taking sufficient care when coding [123].

		Previous Researchers			
	Error	[82]	[81]	[79]	[181]
1	unknown variable	●	●	●	●
2	; expected	●	●	●	●
3	unknown method	●	●	●	●
4	unknown class	●	●	●	.
5	illegal start of expression	●	●	.	●
6	) expected	●	●	●	●
7	incompatible types	●	●	●	●
8	missing return statement	.	●	.	●
9	unknown constructor	.	.	.	.
10	<identifier> expected	●	.	●	●

Table 2.2: Most common errors identified by previous researchers.

2.4.3 Compilation Behaviour

Moving on from basic analysis into the types of different errors which students make, other researchers have explored automatically gathered datasets to identify patterns of novice programming behaviour. As this thesis explores the programming behaviour of Java students, this section will briefly discuss some of the research which has focussed upon exploring Java programming behaviour at the compilation level.

In an early study, Jadud [81] explored the edit-compile cycle of 63 novice programming students. Each time students compiled their code within the BlueJ IDE, a snapshot of the code being compiled would be collected, along with various metadata (e.g. timestamp, username, location). They found the five most common errors accounted for 58% of all errors encountered by students whilst programming: missing semicolons (18%), unknown variable (12%), bracket expected (12%), illegal start of expression (9%), and unknown class (7%). They also found that students had a tendency to recompile rapidly following an unsuccessful compilation.

In a follow on study, Jadud [82] proposed an algorithm designed to quantify several aspects of programming behaviour into a performance predictor. This was known as the Error Quotient (EQ), and is an algorithm which scores the programming behaviour of students based upon the frequency of errors encountered, and how successive compilation failures over a session compared in terms of error message, location, and edit location. The scoring scheme is applied to sets of consecutive compilation event pairings, which is then averaged based upon the total number of pairings logged from students. Initially data was collected from 161 students. This sample was then reduced to 96 students by removing those who had not used the public laboratories at least three times for working on their programs. The error quotient was then run on a further reduced sample of 56 students taken from the 2004/05 academic year only. Two regressions showed that the error quotient scores were a significant, but weak predictor of average assignment scores ($R^2 = 11\%$) and final exam scores ($R^2 = 25\%$). No verification regressions using data gathered from the remaining 40 students were presented, although a histogram showed that the error quotient scores were normally distributed for the entire sample. Jadud suggests that the poor quality fits can be attributed to student cheating, missing assignment data, or an incomplete representation of programming behaviour. In this thesis, we will argue that the third reason is correct, but also that there are several flaws associated with the methodology utilized by the error quotient algorithm which limits its ability to accurately reflect the programming behaviour of students.

The main methodological flaw of the Error Quotient is that the compilation pairings scored by the algorithm are constructed using the natural order that compilation events occurred during a session. However, this approach is flawed as it assumes that either students only work on a single source file, or work on multiple files in a linear manner. Suppose that a student received an error when compiling file 'A', and in their next action, they successfully compiled file 'B'. As the error quotient constructs pairings based upon the order that events occurred during a session, it would incorrectly mark the error in file 'A' as resolved. In reality, the student simply compiled a different file. As a student's Error Quotient is averaged by using the sum of every pair from a session, having a large proportion of invalid 0 scoring pairings like this, can lower their Error Quotient, inaccurately reflecting their performance.

Tabanao, Rodrigo, and Jadud [182] attempted to determine whether at-risk Java programming students could be identified based upon aspects of their programming behaviour. Data was gathered from 124 students who used the BlueJ IDE over five lab sessions. Based upon their mid-term exam scores, students were classified into three groups: high performing ($n = 25$), average ($n = 76$) and at-risk ($n = 23$). Significant differences were found between the three groups in terms of: the total number of errors encountered (more errors for lower performing), the total counts of a few specific types of the most common errors (more occurrences for lower performing), the time spent between two successive compilations (higher performing spent longer between compilations), the error quotient scores (higher for lower performing). A set of regressions were performed by using these four factors. Although the error quotient emerged as an influential predictor, it could only account for a moderate amount of the variance in performance ($R^2 = 29.7\%$, $p < .01$) and all of the regression models failed to accurately predict the at-risk students.

Rodrigo et al., [155] conducted an interesting experiment where data gathered from the BlueJ IDE was supplemented with human observations, describing the affective and behavioural states of 40 programming students. Based upon the mid-term exam scores, significant relations were found for the percentage of observations where students exhibited confusion (r = -.43), boredom (r = -.39), and IDE related on-task conversation ($r = .32$). A regression based upon these factors could account for a moderate amount of the variance in performance ($R^2 = 34.7\%$, $p < .05$). They also found that several aspects of programming behaviour were related to performance, including: the number of pairs of successive compilation errors ($r = .33$), the number of pairs with the same edit location ($r = .34$), the number of pairs with the same error location ($r = .30$) and the number of pairs with the same error ($r = .30$). However, a marginally significant regression based upon these factors could only account for a small amount of the variance in performance ($R^2 = 12.0\%$, $p = .09$).

As a follow on experiment from [155], Rodrigo and Barker [154] attempted to detect student frustration based upon aspects of their programming behaviour. These aspects included the average number of successive compilations with the same edit location, average number of successive pairs with the same error, the average time between compilations, and average number of errors. Although achieving significant results, the strength of the correlation was weak ($r = .31$). Additionally the model failed to detect frustration on a per-lab basis, suggesting that the approach required a substantial amount of data before it could yield accurate results.

2.4.4 Supplementing Compilations with Testing Data

To build a fuller picture of students programming behaviour, other researchers have supplemented snapshots gathered at compile time, with data gathered while students performed debugging tasks (e.g. performing invocations or unit tests).

Norris et al., [133] developed the ClockIt extension for the BlueJ IDE which extends the data collection which was performed by Jadud [82] by also logging package and invocation events. Package events are logged when students open or close packages. Invocation events are logged when students instantiate objects and invoke methods of objects which are on the interactive workbench. These events can be considered as evidence of students performing runtime testing on their code. The main purpose of ClockIt was to provide visualizations which show instructors and students statistics relating to project development. An initial case study which used the snapshots gathered from three students (strong, average, weak) suggested that the percentage of compilation errors and types of compilation errors were related to performance. The weak student had 15% errors which were based upon invalid method declarations, whereas the other students did not encounter this error. The number of compilation attempts also appeared to be related to performance, with the most compilations made by the strongest student and the fewest by the weakest student.

Fenwick et al., [59] continued to explore the data logged from ClockIt. Using data which described the programming activities of 110 students, they found that more than 50% of the time students would generate a follow up compilation within 30 seconds, which confirms the result of [81]. The percentage of most commonly occurring errors logged from students was found to decrease over time as they were replaced by more advanced errors. Students who started working on assignments early outperformed students who started close to the deadline. Additionally those students who worked over frequent short sessions outperformed those who worked a single long session.

Murphy at al., [130] developed the Retina tool which performs similar data logging as ClockIt, except it can integrate with BlueJ, Eclipse, and the javac compiler. Retina distinguishes itself from ClockIt by providing recommendations to students on how they can improve their programming based upon the behaviours that the rule-based tool identifies. For example, advising students to work in smaller intervals when a high number of errors are recorded per compilation. Examining the snapshots which were gathered from a sample of 48 students, it was found that students who made fewer compilation errors were awarded higher semester scores. Students who spent less time working on the assignments tended to score higher than the students who spent longer on the assignments. Additionally they found that the highest rates of errors per compilation occurred between 1am and 4am, which suggests that those students who work later at night tend to make more frequent mistakes than those working earlier.

2.4.5 Massive Datasets and Data Mining

The main limitation of studies into programming behaviour is that they are often conducted on a small scale nature, and only use participants at a single institution. To address these shortcomings, researchers have proposed the use of multi-institutional massive datasets and data mining techniques, to enable the identification of behaviours that are related to performance across different contexts.

Utting et al., [187] describe a modification to the BlueJ IDE which enables the collection of data from students throughout the world. The initial results of this project, called Blackbox are presented by Brown et al., [25] which verified the already well established research into the most common types of errors which students make.

In a very recent study, Brown et al., [24] surveyed 76 educators on their beliefs of the most common mistakes which students make, and verified these beliefs using data gathered from the Blackbox project. Over 14 million compilation events were screened for evidence of 18 instances of students misunderstanding syntax, type errors, and semantic errors. They found that educators only had a weak consensus over the types of errors which students made, and that years of teaching experience did not appear to have any effect on these perceptions. This finding corroborates with the already discussed work of Milne and Rowe [126] and Lahtinen et al., [94].

Although impressive in terms of the amount of data which has been captured by the Blackbox project, the types of analysis which can be performed by using this dataset are currently limited by the lack of an outcome measure. In other words, whilst a great amount of data has been collected, there is no indication as to whether the data is collected from a particularly *poor* or *strong* programming student. This makes applying standard statistical analysis into the correlations between different programming behaviours and performance difficult. It is possible that data mining techniques could be applied to address this limitation, as there have been attempts to utilize data mining techniques to identify different types of programming behaviour.

For instance, Piech et al., [141] used data mining techniques to analyse how programming students' progress through a Karel the Robot assignment. Snapshots were gathered from 238 CS1 students whenever they saved or compiled code in Eclipse. To measure progress over successive snapshots, a program distance metric based upon the weighted sum of abstract syntax tree changes and application program interface call dissimilarity was developed. This metric was then applied to cluster different coding states into high-level milestones by using Hidden Markov Models (HMM). Clustering the paths which students took through the HMM by using a k-means algorithm, they found that the development paths were predictive of students' midterm grades. But, there was no indication that the paths generalised to more complex assignments.

Vihavainen [192] applied data mining techniques to data gathered from 152 Java students who programmed using the Netbeans IDE over a six week period. A total of 200 measures of programming behaviour were considered. For each measure, the overall median, average, minimum, maximum, deviation were caculated, in addition to: the remaining time to deadline, time between snapshots, edit distance between snapshots and coding quality metrics taken from PMD and Checkstyle and FindBugs. Students were classified into three categories: pass, fail, excellent. Applying a non-parametric Bayesian classifier (B-Course) they found that "Minutes to Deadline" was a positive factor - students who started to work late on projects were more likely to fail the course than students who started early. Other positive factors were found for amount of indentation errors in the code. After two weeks of programming, the students classification (pass, fail, excellent) can be identified with 68% accuracy. At the end of the course (6 weeks), the final accuracy is 78%.

2.4.6 CVS Repositories

Instead of collecting data at compile time through IDE's, other researchers have analysed data gathered in Concurrent Versions System (CVS) Repositories. These approaches typically explore project level snapshots (i.e. complete snapshots of the entire project, rather than snapshots of the specific file being compiled).

Spacco et al., [174] developed Marmoset, which is an automated project snapshot, submission, and testing system which can integrate with Eclipse. Marmoset collects snapshots at a finer level of granularity than ClockIt or Retina, by committing a project snapshot to an individual CVS repository each time students save their projects. Feedback is generated for students based upon JUnit tests, but the system also supports static code analysis and bug detection by using FindBugs, PMD, and CheckStyle. An initial analysis used data gathered from 73 students to examine the accuracy of the bug detection tools. However, when attempting to explore the programming behaviour of students they found that the CVS representation was inadequate for encapsulating program histories and proposed a relational schema to address these issues.

Continuing this work, Spacco et al., [173] explored data which described the programming activities of 96 CS2 students over the duration of six assignments. They found that students preferred to work between the hours of 4pm and 6pm. They also found that the majority of assignment work was done within 48 hours of the deadline. A regression revealed a significant, although very weak relationship between the time of the first logged snapshot and final score in the assignment ($R^2 = 9.0\%$, $p < .01$). A second regression revealed a borderline relationship between the total estimated time spent working on each assignment and final score ($R^2 = 1.0\%$, $p < .05$).

Mierle et al., [123] have also used CVS repositories to identify programming behaviours. 166 features were extracted from the projects which students submitted, and were grouped into three categories: CVS repository data (e.g. average number of revisions per file), source code metrics (e.g. number of while loops), and higher level bad practices computed by PMD (e.g. empty if statements). Of the 166 features examined, only 3 had a significant correlation with performance. These included the total number of characters in the diff text between successive revisions, and the total number of times that a comma was followed by a space. The most significant correlation was for the total number of lines of code written. How early a student starts assignments, and how close to the deadline that they submitted was not predictive of performance. As the extra metrics based upon coding style and quality (PMD) were unrelated to performance, the authors concluded that contrary to the beliefs of many instructors that student work habits have very little effect on their performance, so long as they eventually do the work.

2.4.7 Online Submission Systems

Other researchers have explored the projects which students upload (either manually or automatically on specific events) to project submission systems. These systems differ to the CVS repositories by providing students with feedback on their work, such as the number of automated test cases their projects passed.

Edwards et al., [57] have used data gathered from the Web-CAT system to explore the programming behaviours of students. Web-CAT is an automated grading system which provides students with feedback on coding assignments based upon static code analysis and test coverage analysis. Data was gathered from three programming courses over a five year period. The submissions of 633 students were analysed by using a binary classification based upon ability: high achieving (grades A/B), and low achieving (grades C/D/F). Several statistically significant differences were found between the two groups. The total number of submission attempts was found to be slightly lower for the higher group (average 13.7 attempts) than the lower group (average 14.9 attempts). The time of the first submission attempt was found to be substantially earlier for the higher group (average 65.5 hours before deadline) than the lower group (average 27.9 hours). The time of the last submission attempt was found to be substantially earlier for the higher group (average 29.5 hours before deadline) than the lower group (average 12.5 hours). The amount of time spent between the first and last submissions was found to be slightly lower for the higher group (average 36.1 hours) than the lower group (average 40.2 hours). No significant differences were found between the non-commented lines of code written by either group.

Expanding on this work, Allevato and Edwards [2] have applied data mining techniques to the data gathered from the Web-CAT system. Continuing to explore the differences between high and low achieving students they found that both groups of students are characterised by increasing the number of methods and complexity of their code between successive submissions. But, the low achieving students were also found to remove entire methods from their solution, and then removing entire test cases as well. This may suggest that when weaker students encounter errors, they find it easier to start coding again from scratch.

Bosch et al., [20] have also examined the code submissions which students made to an online grading system. As with [2, 57] the correctness of students solutions was determined by verifying that the correct output is generated for specific and randomly generated test cases. Data was collected from 192 students from six semesters of a C/C++ CS1 course, completing 75-80 assignments each. Programming behaviour was measured using both process metrics and source code metrics. Process metrics included work sessions (time students spent programming over a session), iteration interval (time between submissions) and work time (time students spent working on an assignment). Source code metrics included the number of non-commented lines of code written, cyclomatic (McCabe) complexity (number of linearly independent paths of execution in a program) and declared variables (count of the number of variables which are common to CS1 programs, int, float, double, boolean, long, unsigned, char, String). They found that half of all submissions occurred within two minutes of the previous submission, and approximately one third of all submissions occurred within one minute of the previous submission. Applying the source code metrics they found that students tended to submit solutions which were of greater complexity than they actually required. This excess possibly stems from the iterative approach which they adopted to problem solving, identifying and correcting one problem at a time and persisting until no problems are apparent. The other explanation is that it indicates that novice students lack a complete understanding of concepts like State Space and Boolean Algebra. When trying to repair solutions which didn't quite work, students would increase the complexity of their solutions by introducing additional control flow structures.

Helminen et al., [72] explored the programming behaviour of students using online submission systems at a finer granularity than previous approaches. Instead of analysing the programming behaviour of students as they made complete submissions, they examined traces of students direct interactions with a web based Python IDE. Exploring the traces of students over three assignments, they found that students used automatic feedback provided by the system sparingly, and made active use of the console for both testing their code and exploring language features and libraries. Two main testing strategies were identified. Either students would attach test code at the end of their program and run it each time they ran their code, or they would just run their program and then test its functionality via the console. However concerns were raised over the loss of information by using this approach, as one third of the students confirmed that they almost always first wrote and ran their code in other tools (e.g. Eclipse) before submitting via the web based environment.

Helminen et al., [73] in a related study explored the programming behaviour of students working in an online environment to solve Parsons programming problems. In these assignments students had to correctly order and indent a given set of code fragments in order to build a functioning program that met the requirements. Aggregate graphs were constructed to illustrate all the possible solution paths for each of the five assignments. Analysing the data gathered from students studying two programming courses, they found that students had a tendency to add code fragments in a linear manner and follow a top-down approach but that there were a lot of small variations in the solution paths suggesting that some guesswork must be present.

Table 2.3: Overview of programming behaviours identified by recent studies (2005-2014).

Researcher	Year	Language	Tool	Observed Programming Behaviours
Jadud [81]	2005	Java	BlueJ	Students had a tendency to recompile rapidly following an unsuccessful compilation.
Spacco [174]	2005	Java	Marmoset	Found that the CVS representation was inadequate for encapsulating programming behaviours and proposed a relational schema to address these issues.
Mierle [123]	2005	Java	CVS	How close to the deadline that they submitted was not predictive of performance. Only 3 out of 166 metrics based upon coding quality were significantly related to performance, suggesting that work habits have very little effect on programming performance.
Jadud [82]	2006	Java	BlueJ	Proposed purpose built algorithm (Error Quotient) which struggled to predict average assignment scores ($R^2 = 11\%$) and final exam scores ($R^2 = 25\%$)
Norris [133]	2008	Java	ClockIt	The percentage of compilation errors and types of compilation errors were related to performance.
Rodrigo [155]	2009	Java	BlueJ	Marginally significant regression based upon the decoupling of the Error Quotient components could account for a small amount of the variance in performance ($R^2 = 12.00\%$, $p = .09$).
Rodrigo [154]	2009	Java	BlueJ	A model based upon programming behaviour failed to detect frustration on a per-lab basis, suggesting that the approach required a substantial amount of data before yielding accurate results.
Fenwick [59]	2009	Java	ClockIt	More than 50% of the time students would generate a follow up compilation within 30 seconds. The percentage of most commonly occurring errors logged from students were found to decrease over time as they were replaced by more advanced errors. Starting assignments early and incremental working related to stronger programming performance.

Continued on next page

Table 2.3 – *Continued from previous page*

Researcher	Year	Language	Tool	Observed Programming Behaviours
Murphy [130]	2009	Java	Retina	Students who made fewer compilation errors were awarded higher semester scores. Students who spent less time working on the assignments tended to score higher than the students who spent longer on the assignments
Edwards [57]	2009	Java	Web-CAT	No significant differences were found between the non-commented lines of code written by either group. Students starting and submitting assignments early outperformed those who did not.
Allevato [2]	2010	Java	Web-CAT	When weaker students encounter errors, they find it easier to start coding again from scratch.
Tabanao [182]	2011	Java	BlueJ	Statistically significant differences were found between the three groups in terms of the total number of errors encountered, the total counts of a few specific types of the most common errors, the time spent between two successive compilations, the error quotient scores. All regression models based on these factors failed to accuratley predict the at-risk students
Piech [141]	2012	Java (Karel)	Eclipse	Data mining revealed that certain development paths were predictive of students midterm grades.
Bosch [20]	2012	C / C++	Online Submission	Half of all submissions occurred within two minutes of the previous submission. Source code metrics revealed that students often submitted problems with greater complexity than required. When trying to repair solutions which didn't quite work, students would increase the complexity of their solutions by introducing additional control flow structures.
Helminen [73]	2012	Python	Web Based IDE	Students had a tendency to add code fragments in a linear manner and follow a top-down approach but that there were a lot of small variations in the solution paths suggesting that some guesswork must be present.
Vihavainen [192]	2013	Java	Netbeans	200 metrics based mainly upon coding quality could predict students programming performance with 78% accuracy.

Continued on next page

Table 2.3 – *Continued from previous page*

Researcher	Year	Language	Tool	Observed Programming Behaviours
Vihavainen [194]	2013	Java	Netbeans	200 metrics based mainly upon coding quality could predict students mathematics performance with 100% accuracy.
Spacco [173]	2013	Java	Marmoset	The majority of assignment work was done within 48 hours of the deadline. Significant, although very weak relationship between the time of the first logged snapshot and final score in the assignment ($R^2 = 9.00\%$, $p < .01$). Borderline relationship between the total estimated time spent working on each assignment and final score ($R^2 = 1.00\%$, $p < .05$).
Helminen [72]	2013	Python	Web Based IDE	Students adopted two main types of testing strategies, either console testing or running all tests on compilation.
Brown [25]	2014	Java	Blackbox	Confirming previous research [79, 81, 82, 181] on the types of errors which students make.
Brown [24]	2014	Java	Blackbox	Confirming previous research [94, 126] who found that found that educators only had a weak consensus over the types of errors which students made.

2.5 Summary

In this chapter, we have presented background material relating to the studies that are presented in this thesis.

The difficulties of learning to program from a theoretical perspective were discussed, which highlighted eight possible reasons as to why students can struggle to learn to program. These reasons ranged from an inappropriate teaching environment to difficulties in establishing clear mental representations of the underlying machine students are trying to program.

Following this, evidence of the conflicting findings surrounding a range of predictors based upon traditional learning theories were presented. These predictors form a subset of the 116 predictors which are examined in detail in Chapter 7.

Finally data-driven predictors which are based upon programming behaviour were reviewed. This highlighted the main types of behaviour identified by previous research, such as the relations between programming performance and aspects of coding quality, types of errors made, and descriptive data relating to the time of submissions and hours spent coding.

The Error Quotient [82] currently remains the only predictive algorithm which has been designed to predict programming performance based upon quantifying multiple aspects of programming behaviour. However, there are several methodological limitations with the algorithm which we will attempt to address as part of our work. An interesting aspect of programming behaviour which appears to have been overlooked by previous research is to consider how much time students take to resolve different types of errors compared to their peers. We hypothesise that such a measure would provide a natural way of relatively ranking students and possibly be predictive of their performance.

In the next chapter, the research method is presented which will discuss the methods employed to answer the four research questions of this thesis.

Chapter 3

Research Method

This chapter presents an overview of the research methods which are used in this thesis. Section 3.1 presents an overview of the three research objectives which were previously detailed in the Introduction chapter. Section 3.2 presents an overview of the four research questions that were explored in this thesis. Section 3.3 describes the teaching context of our University from which participants for the studies were gathered. Section 3.4 describes the motivations for including each research question. An overview of the method used to answer each question is presented, along with research sub-questions which were used to guide each study. As four considerably different techniques and samples were used to answer each of the research questions, the research methods are detailed alongside the experimental results within each corresponding chapter. Section 3.5 outlines statistical techniques which are used in this thesis.

3.1 Research Objectives

This thesis investigates factors that can be used to predict the success or failure of students taking an introductory programming course. Both factors that can be measured prior to, and during a course are examined. The broad research question which will be explored in this thesis is:

> *Which factors can be used to predict the success or failure of students taking an introductory programming course?*

Consequently the three research objectives (RO) of this thesis, are defined as follows:

> *RO1: To explore which aspects of the external teaching context can influence the performance of students taking an introductory programming course.*
>
> *RO2: To explore which internal factors derived from traditional learning theories can be predictive of the performance of students taking an introductory programming course.*
>
> *RO3: To explore which data-driven metrics based upon analysing the programming behaviour of students within an IDE, can be predictive of the performance of students taking an introductory programming course.*

The motivations behind each of these objectives were described in the Introduction chapter of this thesis. To briefly refresh the reader, the first category of factors that will be explored are those from the teaching context. These factors can include the country in which the course was taught, the programming language that was taught to students, and the grade level of the institution.

The second category of factors that will be explored are derived from traditional learning theories and students' academic backgrounds. Although such predictors have been the focus of research over the past fifty years, there are two major shortcomings of previous research concerning both a lack of verification and the lack of generalisability of such predictors to a range of different teaching contexts.

The third category of factors that will be explored are data-driven predictors which are based upon analysing data that is directly gathered from an IDE describing the programming behaviours of students. Such approaches have shown more promise than the traditional predictors, but research into these predictors is still in its infancy.

To satisfy the three research objectives, this thesis conducted four successive quantitative studies which formed the basis of the four research questions. These research questions are described in Section 3.2 and a visual overview showing how these questions related to the research objectives is presented in Figure 3.1.

3.2 Research Questions

In order to ensure that each of the three research objectives (RO) are satisfied in this thesis, this section summarises four research questions (RQ) that will be explored. The four questions which will be answered in this thesis are as follows:

RQ1 To what extent are students' programming performances influenced by aspects of the teaching context, including: year, country, grade level of the institution, cohort size, and the programming language taught in the course? *(RO1)*

RQ2 Which traditional learning theories describing the psychological and cognitive aspects of learning, and which aspects of students' academic backgrounds are predictive of their programming performances? *(RO2)*

RQ3 Which data-driven metrics derived from data describing students' programming behaviours are predictive of their programming performances? *(RO3)*

RQ4 How do factors based upon traditional learning theories, academic background, and programming behaviours, compare when they are used to predict students' programming performances across different teaching contexts? *(RO2, RO3)*

To present an overview to the reader RQ1 was answered by performing a systematic review of the literature on programming education. From the available literature, data describing the worldwide pass and failure rates of programming courses was extracted, and analysed by grouping the data based upon aspects of the teaching context.

RQ2 was based upon examining the relations between 34 predictors based upon traditional learning theories with the programming performance of one sample ($n = 39$) of students studying the 2012/13 introductory programming course at our University. These predictors were selected either as previous research had yielded inconsistent results, or, because no researcher had attempted to verify previous findings.

RQ3 was based upon exploring the data gathered from the BlueJ IDE which described the programming activities of three samples of students taking the programming course at our university. Samples were taken from 2011/12 cohort ($n = 37$), 2012/13 cohort ($n = 45$), and 2013/14 cohort ($n = 59$). Based upon our experiments, several aspects of programming behaviour which predicted performance were identified.

RQ4 was based upon performing a thorough meta-analysis to integrate the results from previous research with the results of this thesis. Based upon this study, 482 individual results describing 116 predictors were statistically synthesised, illustrating to researchers precisely which factors are the most critical for programming success.

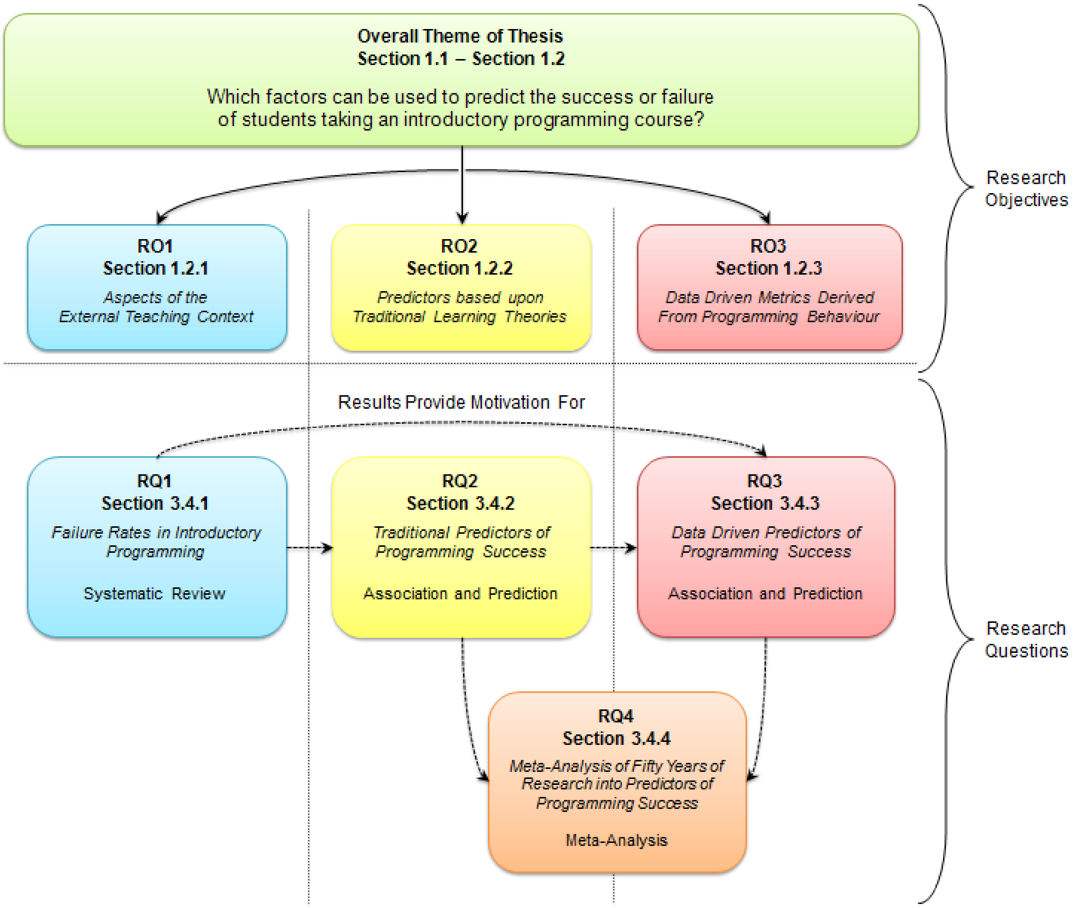

Overall Theme of Thesis
Section 1.1 – Section 1.2
Which factors can be used to predict the success or failure of students taking an introductory programming course?
RO1
Section 1.2.1
Aspects of the External Teaching Context
RO2
Section 1.2.2
Predictors based upon Traditional Learning Theories
RO3
Section 1.2.3
Data Driven Metrics Derived From Programming Behaviour
Research Objectives
Results Provide Motivation For
RQ1
Section 3.4.1
Failure Rates in Introductory Programming
Systematic Review
RQ2
Section 3.4.2
Traditional Predictors of Programming Success
Association and Prediction
RQ3
Section 3.4.3
Data Driven Predictors of Programming Success
Association and Prediction
RQ4
Section 3.4.4
Meta-Analysis of Fifty Years of Research into Predictors of Programming Success
Meta-Analysis
Research Questions

Figure 3.1: Overview of the research conducted within this thesis, showing how the three research objectives related to the overall research theme, and how each of the four research questions were related to the research objectives.

3.3 Teaching Context

As the four samples of students that were used in this thesis were taken from three different cohorts studying the programming course at our university, it is first necessary to discuss the similarities and differences between the teaching contexts of each year. To briefly summarise, the learning content and materials provided to all three cohorts were mostly identical. The main differences between each year of the course were in terms of assessments, with the largest difference being the replacement of the final exam from the 2011/12 course with a final project for 2012/13 and 2013/14 courses. As a result of these changes, and as the use of exams has been widely criticised as an accurate means to measure programming ability [46, 50, 101, 102, 105, 118, 140], we use the students overall coursework performance as the criterion variable of the studies.

3.3.1 Learning Content and Materials

The introductory programming module at the University of Durham was designed to teach Java to students of varying abilities, and assumed no prior knowledge. Teaching took place over nineteen weeks and covered variables, I/O, methods, conditionals, loops, lists, arrays, along with inheritance and exception handling. The majority of students were CS majors, although increasing numbers of elective students came from other disciplines, such as Physics. The course was delivered through a traditional approach. Students were supported through two weekly lectures that introduced concepts (optional attendance), and a weekly two hour lab session (compulsory attendance) where students would practice solving problems using the BlueJ IDE. Students only received one problem sheet per week, usually consisting of three progressively difficult tasks. We note that the in-lab demonstrating staff who were responsible for marking and supporting students were the same two postgraduates for all three years.

In both the 2011/12 and 2012/13 courses, the lectures were shared by two members of academic staff. However, one member of staff left following the completion of the 2012/13 course, and the 2013/14 course was taught solely by the remaining lecturer. As a result, there are differences between the examples used in the lectures for 2013/14 course the other two courses. We note that although the examples used in the course were not the same for each year, the underlying concepts that they were designed to teach were identical for all three cohorts.

3.3.2 Assessment Components

An overview of the assessments used in the course are presented in Table 3.1. The composition of assessments which formed the overall course marks varied over the years. All three cohorts were required to complete a first term written exam, a practical project, and a second term practical exam. Examples of each of the assessments are included in the Appendix.

Lab Assignments

Table 3.2, presents an overview of the assignments which students completed during their weekly two hour lab sessions. As can be seen from this table, there was considerable overlap in the assignments that students of all three cohorts completed. 10 of the assignments were identical for all three courses, and 13 of the assignments were identical for the 2012/13 and 2013/14 courses. Across all three courses, there were only five assignments which were never repeated for another year, these were: 2011/12 (Scanner, GUI), 2012/13 (Cryptography, Further Inheritance), and 2013/14 (Biogram).

Table 3.1: Table showing the weightings of the various assessment components for all three cohorts. All of the assessments were summative unless stated otherwise, a dash (-) indicates that an assessment was not used in the course during a given year.

	Component	2011/12	2012/13	2013/14
Coursework (60% module)	T1 Bench Test	25%	25%	33%
	T1 Project	25%	25%	33%
	Collection Exam	Formative	Formative	-
	T2 Bench Test	40%	40%	33%
	Weekly Labs	10%	10%	Formative
Finals (40% module)	Final Exam	100%	-	-
	Final Project	-	100%	100%

Table 3.2: Table showing the assignments/concepts that were covered in the weekly lab sessions for all three cohorts. Lab assignments that were not reused are shown in italics. Labs reserved for course assessments are shown in bold.

Lab	2011/12	2012/13	2013/14
1	No Practical	No Practical	No Practical
2	No Practical	Intro to BlueJ	Intro to BlueJ
3	Intro to BlueJ	Java Turtle	Java Turtle
4	Java Turtle	HiLo Game	*Biogram*
5	HiLo Game	Cyber Pets	HiLo Game
6	Cyber Pets	*Cryptography*	Free (Catch Up Lab)
7	**T1 Project**	ArrayLists	ArrayLists
8	**T1 Project**	Expressions	Expressions
9	ArrayLists	**T1 Project**	**T1 Project**
10	Christmas Exercises	Christmas Exercises	Christmas Exercises
11	**Collection Exam**	**Collection Exam**	**T1 Project**
12	Static	Static	HashMaps
13	Palindrome	Hash Maps	Palindrome
14	*Scanner*	Palindrome	Static
15	Inheritance	Inheritance	Inheritance
16	Abstract Classes	*Further Inheritance*	Abstract Classes
17	Exceptions	Abstract Classes	Exceptions
18	*GUI*	**T2 Bench Test**	**T2 Bench Test**
19	**T2 Bench Test**	Exceptions	**Final Project**

Term 1 Bench Test

The first term bench test was usually conducted half way through the first term (typically around week 5/6), and forms the basis of a one hour written exam that takes place during the usual lecture slot. For all three years, the exam was closed book. The tasks involved and concepts assessed in this component were:

- 2011/12: *Temperature monitoring program.* Two classes were provided. Students were required to identify the scope of variables (16%), draw an object diagram (24%), write code to calculate an average using a loop (32%), and design a new class based upon the two provided classes (28%).
- 2012/13: *Dice roll game.* Two classes were provided. The tasks involved and marks awarded per question were identical to the 2011/12 bench test but with the theme of a dice roll game.
- 2013/14: *Bank.* Unlike previous years, students were required to identify and correct errors in a provided Java class (50%). The remaining tasks were similar to previous years, and required students to design a new class with appropriate fields and constructors (10%), a method to deposit cash (10%), a method to withdraw cash (20%), and a method to calculate the average value of withdraws since cash was last added to the machine (10%).

Term 1 Project

The first project was usually released towards the end of the first term (typically around week 7/8), and forms the basis of a small project in which students are typically designed to implement 4-5 classes and create a small program based upon concepts covered in the first term labs. The objectives of these projects were for students to demonstrate their knowledge of defining classes, fields, constructors and methods in Java; using appropriate types, including collections; implementing basic algorithms using collections; and devising appropriate test cases. The tasks involved in this component were:

- 2011/12: *Fault Injection.* This project required students to intentionally propagate Java programs with errors, in order to explore two hypotheses: "The Java type system helps to identify programming errors at compile-time instead of run-time", and, "It is difficult for compilers to identify correctly what programming errors have been made and where". The project involved students writing programs that could automatically mutate code (such as removing random lines using the Scanner object), and then writing a report on their findings in relation to the two hypotheses (100%).

- 2012/13: *Phone Company.* This project required students to implement three classes that represented a phone company. For each of the three classes (Phonecall, Phone, and Person), the students were provided with a specification describing key fields and methods, which they then were required to complete (75%). Finally, students were required to implement appropriate testing methods (25%) to demonstrate that their programs satisfied the specifications.

- 2013/14: *Earthquake Monitoring.* This project required students to implement three classes that represented an earthquake monitoring system. The project was largely similar to the one that was used in 2012/13, and again required students to implement three classes (75%) and develop appropriate testing methods (25%).

Term 2 Bench Test

The second term bench test was usually conducted at the end of the second term (typically around week 18/19), and forms the basis of a two hour open book practical exam that takes place within the weekly lab session. The bench test was practically identical for each year, with the bulk of marks (85%) being awarded for identical tasks each year. The tasks involved and concepts assessed in this component were:

- 2011/12: *Online Learning Environment.* The student was provided with a single class representing a Person, and they were required to implement: two subclasses of Person (Staff and Student) using inheritance (40%), a custom list class to store instances of the subclasses (30%), a hashcode function (10%), a search function for an array (10%), and a method to to ensure that their custom list contained no duplicated entries (10%).

- 2012/13: *Online Learning Environment.* The questions were identical to the ones used in 2011/12, however the final question on writing a method to to ensure that their custom list contained no duplicated entries was removed, and the marks redistributed to implementing a hashcode function (15%), a search function for an array (15%).

- 2013/14: *Online Sales Environment.* The questions were identical to the ones used in 2012/13, however the scenario was changed to an online sales environment. The writing a search function for an array question was replaced by writing a method to to ensure that their custom list contained no duplicated entries (10%) from the 2011/12 bench test.

3.4 Research Methods

In this section the motivations for including each of the research questions is presented. Research sub-questions which were used to guide each of the studies are stated and the methods employed to answer each of the questions are briefly discussed. As a range of different quantitative methods are used in this thesis, for clarity, the detailed description of the research methods used to answer each question, are presented alongside the results in the corresponding chapters of this thesis.

3.4.1 RQ1: Failure Rates in Introductory Programming

RQ1: To what extent are students' programming performances influenced by aspects of the teaching context, including: year, country, grade level of the institution, cohort size, and the programming language taught in the course?

Motivation

Before exploring predictors of programming performance with the intention of identifying at-risk students, it was important to establish the proportion of struggling programming students that existed on a worldwide scale.

Therefore the aim of this study was two fold. Firstly, this study aimed to provide a solid motivation for our further research into predictors of programming performance. Secondly, this study aimed to explore whether certain aspects of the teaching context were influential on the programming performance of students.

Over the past half decade, the perceived high failure rates of programming courses have provided a staple motivation for research into programming education. An abundance of generalizations of high failure rates within the literature reinforces the notion, and research is now at the point where the high failure rates in programming courses have transformed from folklore into a widely accepted fact.

However, whilst high failure rates are an often cited motivation for research into programming education, only a single study to date [13] has attempted to provide any quantitative evidence to support the claim that high programming failure rates exist on a worldwide scale. This is problematic, and a lack of hard facts on the outcomes of programming courses can have implications for both instructors and students. Instructors of failing courses may accept their shortcomings as "that's just the way programming courses are", and make no attempt to alter their practice to improve the pass rates. Likewise, potential students may be easily put off from taking the course to start with, which will not help to satisfy future labour demands of a digital economy [186].

In short, when asked whether there is worldwide failure rate problem our response is that we simply do not know. Evidence of high failure rates [53, 92, 144, 152, 159, 214] and low failure rates [4, 32, 61, 170, 183, 195] can easily be located in the literature, and the only study to date that has attempted to quantify programming failure rates is limited by only reflecting course outcomes at a single point in time.

There is still a need to further examine other sources of evidence on the failure rate phenomenon. Only by expanding the work of [13] can a more accurate picture on the worldwide state of programming courses be drawn. If it could be established that the failure rates of programming courses are high, have not changed over time, and that aspects of the teaching context do not have a substantial moderating effect on failure rates, then this may suggest that the internal characteristics of students are more influential on their ability to acquire programming skills. This would serve as a motivation for thesis research questions: RQ2, RQ3, and RQ4, which explore the variations in programming performance based upon student factors.

Research Sub-Questions

To answer RQ1, the following sub-questions were answered:

1. What are the worldwide pass rates of introductory programming courses?

2. How do the pass rates of introductory programming courses compare over time?

3. Are pass rates moderated by any aspects of the teaching context, including:

    (a) Country in which the course was taught.

    (b) Programming language that was taught in the course.

    (c) Size of the cohort: small ($<$ 30 students), large ($\geq$ 30 students).

    (d) Grade level of the institution: university or other.

Method

In order to answer these research questions, we performed a systematic review of research on programming education conducted over the past fifty years to identify as many reported pass and failure rates of programming courses possible.

A statistical analysis was then performed on the pass rate data that was extracted from relevant articles. The first research question was answered by computing the mean pass rate on all the extracted data. The second and third research questions were answered by grouping pass rates into bins based on the moderating variable under investigation, and then performing a one-way ANOVA on the grouped data.

3.4.2 RQ2: Traditional Predictors based upon Learning Theories

RQ2: Which traditional learning theories describing the psychological and cognitive aspects of learning, and which aspects of students' academic backgrounds are predictive of their programming performances?

Motivation

Based upon the results from RQ1, we were motivated to start exploring the possible student characteristics which may be predictive of programming performance. For almost fifty years, researchers have explored how traditional learning theories and aspects of the students' academic backgrounds relate to their programming performance.

But despite over fifty years of research, almost all of the characteristics based upon traditional learning theories have failed to consistently predict the programming performance of students across a range of different teaching contexts. This is problematic, as without understanding which factors enables students to become good programmers, it becomes difficult to design and implement any effective pedagogical tools or strategies which could be applied to identify and better support the weakest students.

The aim of this study was to determine whether any relationships existed between performance and a range of characteristics which have only been evaluated in a small number of teaching contexts. This allowed us to explore whether these characteristics were also limited by the same context dependency as their predecessors, and to evaluate their wider applicability as enabler of programming ability. The second purpose of this study was to determine whether a subset of these predictors could be combined to form a context independent multivariate model, that could be used to predict performance across a range of different teaching contexts.

Research Sub-Questions

To answer RQ2, the following sub-questions were answered:

1. Are there any correlations between the predictors examined in this study and programming performance? If applicable, are these findings consistent with previous research, or inconsistent, suggesting a context dependency?

2. Are any categories of predictor more strongly correlated with programming performance than others?

3. Are any predictors either singularly or additively useful for predicting the performance of students studying an introductory programming module?

Method

We first had to identify a set of predictors that had only been evaluated in a small number of teaching contexts. The motivation behind this decision was two-fold. Firstly, it is reasonably straightforward to find a pair of conflicting results for any of the predictors previously trialled in multiple teaching contexts. Therefore, we were interested in exploring whether the mostly unverified predictors also suffered from the same context dependency as their predecessors.

Secondly, the thesis RQ4 is answered by performing a thorough meta-analysis of previous quantitative research. In order to statistically combine studies to estimate how a predictor would perform when it is applied across a range of different teaching contexts, at least two results are required. Verifying a selection of previously unverified predictors meant that we could include a greater range of predictors in the meta-analysis, which allows us to explore how a wider range of characteristics relates to programming performance.

In total 34 predictors were selected and grouped into seven different categories under three broad headings. These included: academic predictors (previous programming experience, previous academic experience), psychological predictors (attributional style, behavioural characteristics, self-esteem) and cognitive predictors (learning styles, learning strategies and motivations). Six instruments were used to collect data from the participants. These included: an in-house designed background questionnaire, Weiner's attributional style questionnaire, Rosenberg's Self-Esteem Scale, Kolb's Learning Style Inventory, Gregorc Style Delineator, and the Motivated Strategies for Learning Questionnaire. These instruments were completed by 39 students (36 male) sampled from the 2012/13 introduction to programming cohort at our University.

The first research question was answered by calculating Pearson's correlations between the students' programming performances and the scores they obtained on each of the six instruments. These results were compared to previous research if available to judge the context dependency of a predictor. This was evaluated by using a vote counting approach based upon how the strengths of the correlations obtained by this study compared to those reported by previous research.

The second research question was answered by determining whether there were any significant differences between the mean correlation strengths of the seven types of predictor that were explored in this study.

The third research question was answered by performing two stepwise regressions. The first regression was used to evaluate whether any combination of the characteristics explored in this study were predictive of performance. The second regression was performed to evaluate combinations of the context independent predictors only.

3.4.3 RQ3: Data-Driven Predictors based upon Programming Behaviour

RQ3: Which data-driven metrics derived from data describing students' programming behaviours are predictive of their programming performances?

Motivation

The main shortcoming of predictors derived from traditional learning theories is that they are rarely found to consistently predict the programming performance of students across a range of different contexts. We argue that the reason for such large variations in results is because the predictors examined to date have not specifically been designed for this purpose. Additionally the predictors are static in nature, and therefore cannot reflect changes in the students programming knowledge over time. The traditionally measured attributes (e.g., learning styles) also cannot easily be changed.

Possibly inspired by the push towards big data analysis, researchers have begun to explore data-driven predictors which are based upon analysing data directly logged from an IDE, which describes the programming activities of students. The data gathering involves collecting snapshots of source code and error messages, usually when students perform actions such as saving the project they are currently working on. This data then enables the utilization of data-driven approaches for identifying programming behaviours that are predictive of student performance.

The main benefit of data-driven approaches based upon how students' solve programming errors, or how they schedule their time, and whether they pay attention to code quality, is that, they are directly based on the regular programming activities of a student, and therefore can directly reflect changes in their learning progress over time. This is not the case for more traditional predictors explored over the past fifty years, such as age, gender, which remain static within the context of a course.

Whilst recent research has explored the possibility of displaying statistics on behavioural aspects to instructors so that a manual invention can be made, only Jadud [82] has attempted to quantify several aspects of programming behaviour into a performance predictor. This was known as the Error Quotient (EQ), and is an algorithm which scores the programming behaviour of students based upon the frequency of errors encountered, and how successive compilation failures over a session compared in terms of error message, location, and edit location. Although previously used by several studies, the EQ was shown to be a weak predictor of performance. This could be due to several flaws concerning the incompleteness and inaccuracy of the method which we attempt to address and expand upon in this work.

Research Sub-Questions

To answer RQ3, the following research sub-questions were answered:

1. How do the datasets gathered in the Durham teaching context compare to those used in previous research?

2. Which aspects of the students ordinary programming behaviour are associated with their programming performance?

3. Which aspects of programming behaviour can be combined, and scored, into an overall predictive measure of programming performance?

4. How do the measures proposed in this chapter compare in terms of accuracy and explanatory power over the duration of the entire course?

5. How do the measures based on programming behaviour compare to the predictors based upon traditional learning theories explored in Chapter 5 of this thesis?

Method

Data describing the programming activities of three different cohorts of students taking the programming module at our university (2011/12, 2012/13 and 2013/14) was gathered through the use of an on-line protocol added to the BlueJ IDE. Each time students compiled code on a university PC, the extension would log a snapshot of the code being compiled, along with the students username, a timestamp, event type (compilation success or failure), and the error message reported with line number (if applicable). Similar data was logged for invocation and package events.

The first four research sub-questions were answered by exploring how different aspects of the students' programming behaviour were related to performance, by calculating correlations and performing multiple regressions. The datasets were first cleaned, and a procedure was developed to construct a set of consecutive compilation pairings, which could be used to analyse the students programming behaviour in terms of successive compilations. For instance, if in one compilation the student received an error, in the next compilation, did the student receive the same error? Had they solved the error? How long did it take them? How many changes did they make to the source code between compilations? Specifically, measures based upon the percentage of different compilation pairings, and resolve times were considered in this research. The final research question was answered by comparing the correlation strengths of the predictors identified in this study to those we identified in thesis RQ2.

3.4.4 RQ4: Meta-Analysis of Fifty Years of Research on Factors that can Predict Programming Performance

RQ4: How do factors based upon traditional learning theories, academic background, and programming behaviours, compare when they are used to predict students' programming performances across different teaching contexts?

Motivation

The obvious question concerning any research conducted into predictors in this thesis is: how do these findings compare to the research on predictors that were not examined within this thesis? To address this, and to synthesise over fifty years of conflicting results on predictors of programming performance, we performed a meta-analysis.

Meta-analysis is a form of systematic review, in which the quantitative results from several studies are statistically combined in order to generate a more precise estimate of an effect under investigation. In an individual study, the units of analysis are individual observations, whereas in a meta-analysis the units of analysis are the results of individual studies (i.e. predictors in different teaching contexts). Compared to a simple narrative review, a meta-analysis offers several advantages. These include:

1. Increased statistical power of detecting the magnitude of an effect when compared to the power of individual studies.
2. Improved precision of the measurement of an effect.
3. Combining data from conflicting studies to determine whether an effect exists.
4. Less prone to bias, based upon a systematic review process.

Given the inconsistent findings of studies that have explored predictors of programming performance over the past fifty years, meta-analysis would seem to be an appropriate technique to resolve such conflicts. Only by statistically combining research across different teaching contexts, can researchers develop a true understanding of precisely which factors are most important for making successful programmers.

In this study, meta-analysis techniques are applied to synthesize the findings of multiple studies that have examined the same predictor of programming performance across different teaching contexts, such that conclusions on the general effectiveness of different predictors can be made. Although commonly applied within the medical and psychological domains, to our knowledge, this study represents the first attempt to apply meta-analysis techniques to statistically synthesise over fifty years of conflicting research into factors predictive of programming performance.

Research Sub-Questions

To answer RQ4, the following sub-questions were answered:

1. Which factors have researchers examined as predictors of programming performance over the past fifty years?
2. Which factors are the most predictive of programming performance?
3. Which factors are the least predictive of programming performance?
4. Which classes of factors have the strongest and weakest effects on programming performance?
5. What conditions moderate the effectiveness of the predictors?

Method

The research sub-questions were answered by applying the random-effects meta-analysis procedure presented in [31, 78]. The main advantage of this procedure is that it allows inferences to be made about the larger set of studies from which the studies included in the meta-analysis are assumed to be sampled from (whether such studies currently exist or not). Like many meta-analyses, this study followed several steps:

1. Locating all possible studies.
2. Screening potential studies for inclusion using preset criteria.
3. Coding all qualifying studies based upon their methodological and substantive features.
4. Calculating effect sizes for all qualifying studies for further combined analyses.
5. Carrying out comprehensive statistical analyses covering both average effects, and the relationships between effects and study moderators.

After completing the systematic review, we performed knowledge transformation by classifying the 116 identified predictors into a six-class theoretical framework of factors predictive of programming performance. The results of the meta-analysis were then considered within the context of the proposed framework. Based upon these results, we suggest which skills and characteristics are the most important to make a successful programmer, and suggest which skills instructors should focus upon teaching students to improve their programming performance. Thus, answering the original overall question of this thesis *which factors can be used to predict the success or failure of students taking an introductory programming course?*

3.5 Statistical Techniques

Analysis of the data involved employing a number of commonly used statistical techniques, including Pearson correlation coefficients, stepwise linear regression, analysis of variance, and t-tests for independent samples. In order to satisfy some of the underlying assumptions of these techniques equality of variance and normality tests were also performed. Our meta-analysis involved a number of statistical techniques which are not commonly used, such as the computation of an effect size, and sample heterogeneity. The techniques employed in this thesis, included:

1. Assumption Testing
    - Shapiro Wilk test of normality (W)
    - Levene Test for Equality of Variances
2. Association
    - Pearson's Product Moment Correlation Coefficient (r)
    - Spearman's Rank Correlation Coefficient (r_s)
3. Prediction
    - Multiple linear regression
    - Stepwise regression
4. Differences Between Groups
    - Independent samples t-test
    - One-Way Analysis of Variance (ANOVA)
5. Meta-Analysis
    - Effect Size Calculations
    - Effect Size Corrections
    - Meta-Analytic Model Fitting
    - Statistical Analysis Procedure

For readers interested in more detail on these techniques, definitions and discussions are presented in Appendix 1.

3.6 Summary

This chapter has presented an overview of the four research questions which will be answered in this thesis, and has shown how these research questions related to the original three objectives which were stated in the Introduction. The motivations for including each question were discussed, and the methods employed to answer each question briefly discussed.

Chapter 4

Failure Rates in Introductory Programming

A modified version of this chapter appears in the following peer-reviewed publication:

- C. Watson and F.W.B. Li. Failure Rates in Introductory Programming Revisited. In *Proceedings of the 19th Annual Conference on Innovation and Technology in Computer Science Education* (ITiCSE '14), pages 39-44, 2014, ACM.
  Best Paper Award.

Despite several studies citing the high failure rates of introductory programming courses as a motivation for research, the majority of available evidence on this phenomenon to date is anecdotal in nature, and only a single study has attempted to provide any quantitative evidence on the worldwide failure rates of programming courses.

In this chapter we contribute substantial quantitative evidence on the often cited high failure rates of introductory programming courses, by performing a systematic review of introductory programming literature, and analysing pass rate data extracted from relevant articles. Pass rates describing the outcomes of 161 CS1 courses that ran in 15 different countries were analysed. A mean worldwide pass rate of 67.7% was found. Moderator analysis based on aspects of the teaching context revealed significant, but not substantial, differences based upon grade level of the institution, country, and size of the cohort. However, pass rates were found not to have improved over the past ten years, or based upon the programming language taught in the course. This chapter suggests that aspects of the teaching context are not substantial moderators on student success in programming, and possibly the root cause of the failure rate phenomenon may be related to internal characteristics of students.

4.1 Introduction

Learning to program can be an incredibly difficult task, to the point where the phrases "failure rate" and "programming course" are almost synonymous [13]. Indeed, a brief review of the literature highlights widespread concerns alluding to the perceived high failure rates of programming courses. It is not uncommon for researchers generalize these concerns into a worldwide problem. For instance:

- Mancy [108]: *"Nonetheless, such low pass rates indicate that students experience difficulties with programming and anecdotal evidence confirms this."*

- Ma [107]: *"The high failure rates in programming courses are not surprising if students still do not understand these basic programming concepts at the end of courses."*

- Bergin [14]: *"It is well known in the Computer Science Education (CSE) community that students have difficulty with programming courses and this can result in high drop-out and failure rates."*

- Bornat [19]: *"Substantial failure rates plague introductory programming courses the world over and have increased rather than decreased over the years."*

- Shuhidan [165]: *"High attrition and high failure rates in foundation-level programming courses undertaken at tertiary level in Computer Science programs, are commonly reported."*

- Porter [145]: *"Many Communications readers have been in faculty meetings where we have reviewed and bemoaned statistics about how bad attrition is in our introductory programming courses for computer science majors (CS1). Failure rates of 30% to 50% are not uncommon worldwide."*

- Guzdial [67]: *"Rumors of high failure rates in introductory computing courses (typically referred to as CS1 in reference to an early curriculum standards report) are common in the literature and in hallway discussions at conferences such as the ACM Special Interest Group in Computer Science Education (SIGCSE) Symposium."*

- Hoskey [76]: *"Numerous studies document high drop-out and failure rates for students in computer programming classes."*

- Guzdial [68]: *"Start asking around - we're hearing about drop-out/failure rates in CS1 courses in the 15 to 30% range."*

However, whilst high failure rates are an often cited motivation for research into programming education, only a single study to date [13] has attempted to provide any quantitative evidence to support the claim that high programming failure rates exist on a worldwide scale. This is problematic, and a lack of hard facts on the outcomes of programming courses (henceforth CS1) can have implications for both instructors and students. Instructors of failing CS1 courses may accept their shortcomings as “that’s just the way programming courses are”, and make no attempt to alter their practice to improve the pass rates. Likewise, potential students may be easily put off from taking the course to start with, which will not help to satisfy future labour demands [186].

There are also implications for researchers of programming education, and for the research that is conducted in this thesis. Specifically, establishing the worldwide percentage of failing programming students so that a solid motivation could be provided for looking into the characteristics that were affecting students’ performances. It also was important to determine whether aspects of the external teaching context, such as the language taught in the course, were influential on performance. If it could be established that worldwide programming failure rates have not drastically improved over time, and that aspects of the teaching context do not have a substantial moderating effect on the failure rates, then this may suggest that internal characteristics of students are more influential on their ability to acquire programming skills.

Chapter Contributions

The purpose of this study was to establish a solid motivation for our research, and to explore whether factors of the external teaching context were moderators on students’ programming performance. This chapter answers the following thesis research question:

> *RQ1: To what extent are students’ programming performances influenced by aspects of the teaching context, including: year, country, grade level of the institution, cohort size, and the programming language taught in the course? (RO1)*

The contributions of this chapter are:

1. To expand the findings of the only study to date which has attempted to provide quantitative evidence on the failure rates of CS1. Our results suggest that the failure rates of CS1 are 32.3%, and have not substantially changed over time.

2. A moderator analysis of 161 failure rates, from 15 different countries based upon five key aspects of the teaching context. The results suggest that only the grade level of the institution, country, and cohort size moderate the failure rates of CS1.

4.2 Related Work

Over the past half decade, the perceived high failure rates of CS1 has provided a staple motivation for research into programming education. An abundance of generalizations of such failure rates within the literature serves to reinforce the notion, and research is now at the point where the high failure rates of CS1 have transformed from folklore, into a widely accepted fact. Scholars are very willing to generalize high failure rates into a *worldwide* problem, without considering evidence from beyond their own institutions.

As part of our work within this chapter, we have been able to locate articles reporting high CS1 failure rates for institutions based in: Australia 58% [125], Brazil 55% [53], Canada 53.5% [159], China 18.9% [195], Finland 62.0% [92], Germany 55.3% [144], Indonesia 51.4% [214], New Zealand 48.0% [152], Portugal 76.9% [120], South Africa 50.0% [148], Spain 53.0% [60], Taiwan 22.6% [37], UK 57.6% [66], and USA 65.0% [134]. These findings support the notion of high CS1 failure rates on a worldwide scale.

On the other hand, we found it equally straightforward to identify evidence of low CS1 failure rates, for institutions based in: Australia 4%, 7% [183], Canada 9%, 11%, 12% [61], China 5.9%, 7.0%, 7.5% [195], Denmark 4.4% [33],UK 4.0%, 6.0%, 9.7% [4], and USA 5.6%, 6.7%, 10% [170]. These findings reject the notion of high CS1 failure rates on a worldwide scale, yet researchers continue to selectively cite this as a motivation for work (e.g. [14, 49, 67, 68, 76, 107, 108, 145, 165]).

The question therefore arises, as to what exactly are the worldwide pass and failure rates of CS1, and whether any aspects of the teaching context moderates these rates? To date, only Bennedsen and Caspersen [13] have attempted to answer this question. Around 2005, [13] sent a short survey to the authors and panel participants of five CS educational conferences: Koli Calling '04, ICALT '04, ACEC '04, SIGCSE '05 and ITiCSE '05. The survey was designed to collect data on the outcomes of the CS1 courses at each respective researcher's institution. A total of 63 usable responses from researchers in 15 different countries were received, representing a response rate of 12.3%. The main findings of their study were:

1. The worldwide average pass rate of CS1 was estimated to be 67%.
2. Pass rates vary by cohort size: small cohorts 82%, large cohorts 69%.
3. Pass rates vary by grade level of the institution: colleges 88%, universities 66%.
4. Pass rates are independent of the type of language taught: object-orientated, imperative, and functional were found to have an almost identical pass rate.

However, there were several limitations of the study which we attempt to address and expand upon in our work. Firstly, the Bennedsen and Caspersen study provided a useful snapshot on the worldwide state of CS1 outcomes. But this snapshot only covered a single point in time, representing courses that ran around 2005. The study provides no evidence as to whether or not the pass rates of CS1 have improved over time, possibly in response to newer languages or tools that can lower the learning curve for novice programmers. e.g. IDE's targeted specifically at novices, such as BlueJ, or game-based learning tools, such as Greenfoot. If the pass rates have not improved over time, then this may imply that there are certain fundamental (threshold) concepts that students must gain an understanding of, and that adjusting context factors does not improve the likelihood of the students mastering these concepts.

Secondly, the authors acknowledge that their sample may be insufficient to make generalized conclusions on the worldwide state of CS1. Outcomes of only 63 courses taken from 62 different institutions were analyzed, and it is perhaps interesting to consider why 87.7% of the contacted authors failed to respond. One possibility is that the non-responding authors had higher failure rates than they wished to report. This would mean that the worldwide failure rate of CS1 could be higher than [13] estimated. The reverse is also a possibility.

Thirdly, although researchers from 15 different countries responded, the sample of 63 responses was heavily dominated by institutions from the United States. 66% of responses came from USA institutions, with the remaining 14 countries providing (mainly) 1-2 responses each. It is well known that educational practices can considerably vary by country, and using such a dominated sample makes generalization of the findings to a worldwide scale difficult.

Fourthly, the findings were based upon a survey that was sent to selected authors of only five conferences on CS education. This is a narrow target group, and inevitably omits a great deal of evidence on pass rates that remains unexplored, published in the proceedings of other conferences and journals. By exploring these additional sources of data, a fuller picture on the worldwide outcomes of CS1 courses may be obtained.

In short, when asked whether or not aspects of the teaching context influence the success rates of CS1, our response is that we simply don't know. Evidence of high and low failure rates can be located in the literature, and the only study to date that has attempted to estimate the worldwide failure rates has several shortcomings. There is still a need to further examine other forms of quantitative evidence on this phenomenon. Only by expanding the work of [13] can a more accurate picture on the worldwide state of CS1 be drawn, and if the failure rates are found to be high, then a solid motivation can be provided for further research into the cause.

4.3 Research Design

Our work is based upon performing a systematic review of the available literature on CS1 education, and then performing an analysis of the course outcomes extracted from relevant articles. The ideal approach for performing a study of this nature would be to consider both failure rates published within the literature, and to perform direct a survey of institutions and professional bodies. However, the low response rate of 12.3% that Bennedsen and Caspersen reported suggests that performing similar survey may be unsuccessful. Nevertheless, by only seeking failure rate data from the CS1 literature in this study, it allows us to analyse a previously unexplored source of data which can be considered complementary to the previous study by Bennedsen and Caspersen.

4.3.1 Research Questions

To answer thesis RQ1, the research questions that will be explored in this chapter, are defined as follows:

1. What are the worldwide pass rates of introductory programming courses?
2. How do the pass rates of introductory programming courses compare over time?
3. Are pass rates moderated by any aspects of the teaching context, including:
    (a) Country in which the course was taught.
    (b) Programming language that was taught in the course.
    (c) Size of the cohort: small (< 30 students), large (≥ 30 students).
    (d) Grade level of the institution: university or other.

4.3.2 Data Collection Method

The motivation for the work in this chapter arose whilst we were working on an upcoming meta-analysis which synthesized fifty years of research on predictors of performance (Chapter 7). Whilst we were collating literature for this study, we found that the actual evidence on the widely cited high failure rates of CS1 was sparse at best. As such, a proportion of the data that was used for the analysis in this study, was extracted from articles that were identified for use in the meta-analysis. However, simply using articles that were identified for this specific purpose would only represent a narrow range of research on CS1 education. We therefore carried out a series of supplementary searches in order to extract additional data from the wider literature base of CS1 research.

In this section, we outline the thorough search and inclusion procedure that was followed to identify as many pass and failure rates from the CS1 literature as possible.

Study Coding

To answer the research questions, the following data was coded from each study:

1. Course details: year, programming language, language class.
2. Institution details: name, grade level, country.
3. Totals and percentages: n, pass, fail, withdraw, fail/withdraw.
4. Article details: publication source, year, name, author.

Exclusion of the Teaching Context

Details on the possible teaching approaches used in the courses were not explored as part of this study. There were two reasons for this. Firstly, the main objective of thesis RQ1 was to validate the findings of the moderating factors which were previously examined by Bennedsen and Caspersen (which did not include the teaching context). Secondly, the actual teaching approaches used by different programming instructors were not widely reported, which made conducting a study into the effects of different interventions on pass rates difficult. We felt that it was more important to analyse a larger sample where information on the teaching methodology was not coded, instead of using a reduced sample where information on the teaching methodology was coded.

It is conceivable to hypothesise that the teaching methodology used may have an impact on the performance of students, we attempted to examine the impact of different teaching approaches on improving programming pass rates in a follow on study from the work presented in this thesis (see [193]).

In conjunction with the University of Helsinki, we performed a quantitative systematic review on articles describing introductory programming teaching approaches, and analysed the effect that various interventions can have on *improving* the pass rates of introductory programming courses. This study took a different approach to the one presented in this thesis, as we examined pre- and post- intervention pass rates (i.e. the improvement in pass rates as a result of changing the teaching methodology).

Out of the studies examined, on average, the pre-intervention pass rate was 61.4%, and post-intervention 74.4%. No statistically significant differences between the effectiveness of the teaching interventions were found. However, marginal differences between approaches were identified. The courses with relateable content (e.g. using media computation) with cooperative elements (e.g. pair programming) were among the top performers with CS0-courses, while courses with pair programming as the only intervention type and courses with game-theme performed more poorly when compared to others. Although not significant, these interventions were still able to improve pass rates by a minimum of 10%, suggesting that although they were not as strong as the other interventions in this study, they were still beneficial when compared to the traditional lecture and lab approach they replaced.

Initial Search Using The Meta-Analysis Articles

The search criteria and process used to perform the meta-analysis is presented in Section 7.3.2. Disappointingly, only 12 of the articles included in the meta-analysis provided quantitative data on the failure rates of their respective CS1 courses, which were extracted for analysis in this study.

Secondary Searches

As our meta-analysis was focused upon a narrow area of CS1 research (predictors of performance), it was necessary to conduct supplementary searches in order to identify other articles that provided quantitative evidence on the pass and failure rates of CS1. We hypothesized that an abundance of such data would be found within articles that described interventions designed to improve the performance of CS1 students.

As such, the initial search process was repeated with the same restrictions and repositories as the meta-analysis, but the following search criteria was used: (Pass Rate OR Failure Rate OR Success Rate OR Withdraw OR Completion OR Dropout OR Improving) AND (Programming OR Programming Course OR Introductory Programming OR CS1). After removing articles which did not provide the quantitative data required by this study, 42 additional articles were identified and added to the sample of 12 articles identified from the initial search.

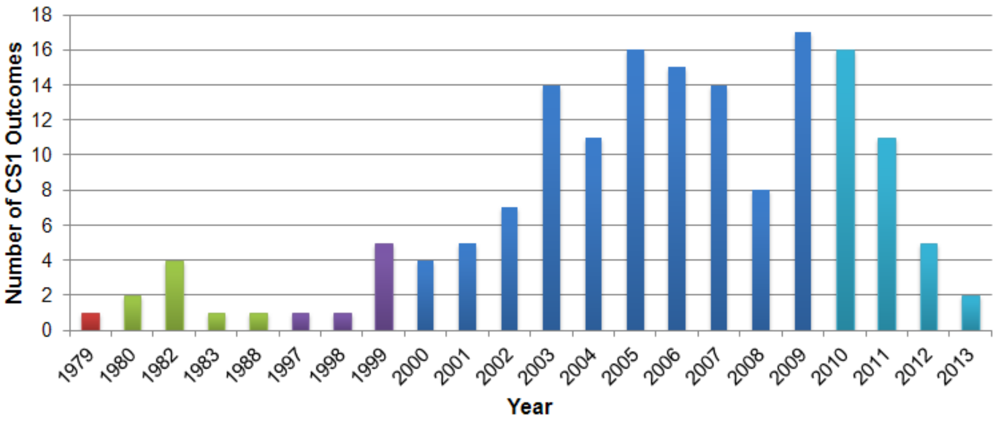


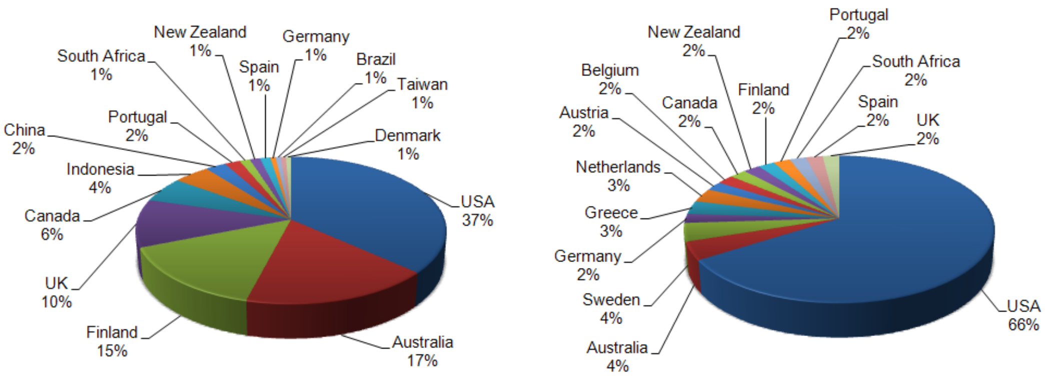
Figure 4.1: Bar chart showing the number of CS1 outcomes coded for each year. Different decades are indicated by different colors.

4.3.3 Description of the Sample

After verifying that data describing the same course had not been double coded, the final sample consisted of 54 articles. These were in the form of: 37 conference papers (68.5%), 11 journal articles (20.3%), 3 theses (5.6%), 2 unpublished reports (3.7%), and 1 book chapter (1.9%). From these 54 articles, the outcomes of 161 CS1 courses were coded using the coding scheme previously presented. A distribution of the number of outcomes coded for each year is shown in Figure 4.1. As can be seen, the outcomes describe CS1 courses that ran between the years 1979-2013, but the majority of outcomes (129, or 80%) were for courses that ran from 2003 onwards.

The outcomes used in this study described courses that ran in 51 different institutions, across 15 different countries. The geographical distribution of outcomes used in this study, compared to Bennedsen and Caspersen [13] is shown in Figure 4.2. To compare both samples, the sample used by this study was less dominated by a single country. In the sample used by [13], 66% of outcomes were from US institutions, and the remaining 33% were from institutions across 14 other countries.

Our sample was less dominated by a single country, with 63% of outcomes coming from 14 different countries. USA institutions contributed 37% of the outcomes, followed by Australia 17%, Finland 15%, and the UK 10%. Both samples had 10 countries in common, with the difference being pass rates from Indonesia, China, Brazil, Taiwan, Denmark included in our sample, but pass rates from Netherlands, Greece, Germany, Belgium and Sweden were not included in our sample. However, as these differences in countries represent less than 15% of the pass rates used by either samples, their presence or absence is unlikely to be influential on the overall results.



(a) This Study

(b) Bennedsen and Caspersen [13]

Figure 4.2: Geographical distribution of the CS1 outcomes used by both studies.

4.4 Results

In this section, the results in relation to the research questions are presented.

4.4.1 Worldwide Pass Rates of Programming Courses

The first question addressed by this study, was to determine the worldwide mean pass and failure rates of CS1. Figure 4.3 shows the distribution of the pass rates used by this study, alongside the pass rates reported by [13]. As can be seen from this figure, the distribution of pass rates used by this study followed a normal distribution, which was confirmed by a Shapiro Wilk test, $p > .05$. The proportions of each pass rate range was similar to those reported by [13]. The modal pass rate range of 61-70% was found to be comparable to [13], and the majority of pass rates (61%) were concentrated in the range of 50-80%. As with [13], the pass rates of CS1 were found to vary considerably by institution, ranging from a low of 23.1% to a high of 96%.

The mean worldwide pass rate of CS1 found by this study was 67.7%, which is practically identical to the 67% mean pass rate reported by Bennedsen and Caspersen [13]. It can be debated as to whether or not a sample that was based on the outcomes of only 161 CS1 courses across 15 different countries is representative of the worldwide state of CS1. In response, we suggest that when this finding is considered in conjunction with the similar result found by the independent [13] study, that an average CS1 pass rate of 67% may be close to the mean figure across other countries. Using our sample to estimate a 95% confidence interval for the population mean, suggests that the true worldwide pass rate of CS1 lies somewhere between $\mu = 65.3\%$ to 70.1%.

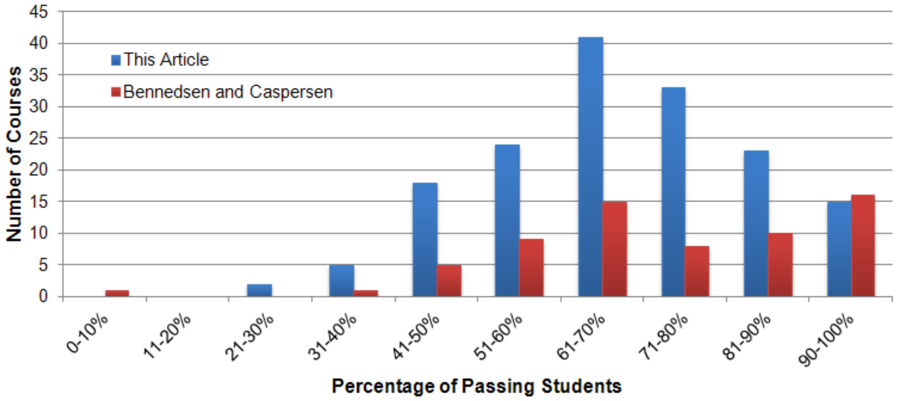


Figure 4.3: Pass rates of this study compared to Bennedsen and Caspersen [13]

The natural question which follows, is what happens to the remaining 32.3% of students who do not pass CS1? To answer this question we attempted to extract and analyze explicitly stated failure, withdrawal, and failure/withdrawal rates that were stated by articles. Disappointingly, this information was largely unstated, and only 42 failure rates were explicitly stated as such. Analyzing these 42 failure rates only, and comparing them to the overall sample of 161 rates, a comparable mean failure rate of 30.5% and comparable pass rate of 67.2% were found. The remaining 2.3% presumably represented students who withdrew or failed to complete the course.

Whilst we can state that 32.3% of students did not pass CS1, we cannot say whether this figure represents failures, withdrawals, or a combination of both. In the remainder of this study, we therefore assume this figure to represent a notion of *failure to pass.*

4.4.2 Pass Rates of Programming Courses over Time

The second question explored in this study was to determine whether the pass and failure rates of CS1 have changed over time. Grouping the 161 pass rates by the year in which the course was run, a one-way ANOVA was performed. There were no outliers in any of the groups, as assessed by the inspection of a box plot. Shapiro-Wilk tests confirmed that the pass rates were normally distributed for each year, $p > .05$, and homogeneity of variances was confirmed by Levene's test, $p = .11$. The one-way ANOVA showed that there were no statistically significant differences in pass rates of CS1 for any of the years that were covered by this study, $F(21, 139) = .486$, $p = .97$.

As shown in Figure 4.4, the mean percentage of non-passing students has remained generally constant for the years that were examined by this study. The mean percentage of non-passing students ranged from 53.5% to 17.4%. Two thirds of the years examined had a rate of non-passing students of between 25% to 33%. Given the increased amount

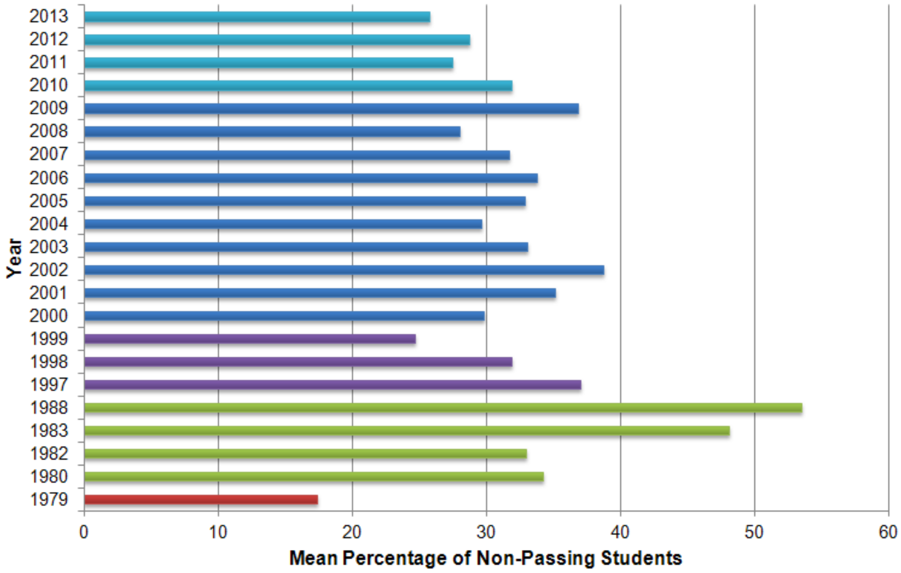


Figure 4.4: Bar chart showing the mean percentage of non-passing students for each year that was covered by this study. Different decades are indicated by different colors.

of tools that are available to support novice programming students, it is interesting to see that there have been no significant changes in the pass rates of CS1 over time. This finding may suggest that internal, rather than external factors are more influential on students programming performance, in that no matter how well supported students are in their learning by advances in pedagogy and technology, programming failure rates have remained practically constant over time.

4.4.3 Moderating Aspects of the Teaching Context

We were therefore motivated to consider whether any external aspects of the teaching context moderated the pass and failure rates of CS1 over time.

Country

We first examined the pass and failure rates in relation to the country in which the course was taught, as it is well known that educational practices and assessment criteria can vary across different continents. Grouping the 161 pass rates by country, a one-way ANOVA was performed. The previously stated assumptions were satisfied. Shapiro-Wilk tests confirmed that the pass rates were normally distributed for all countries, $p > .05$, with the exception of Canada, $p = .03$. However, as violations of normality do not substantially effect the Type I error rate, an ANOVA is considered to be rel-

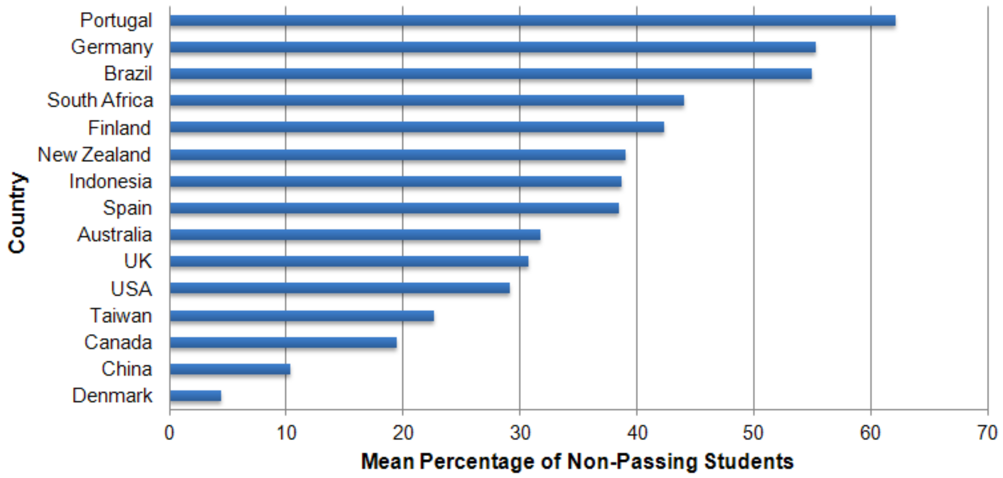


Figure 4.5: Bar chart showing the mean percentage of non-passing students by the different countries that were covered by this study.

atively robust against such violations [110]. Homogeneity of variances was confirmed by Levene's test, $p = .15$. The one-way ANOVA revealed statistically significant differences in pass rates of CS1 based upon the country in which the course was taught, $F(14,\ 146) = 4.58$, $p < .001$. To confirm that the non-normality of the Canada pass rates had not impacted on the ANOVA result, we also ran a Kruskal-Wallis H test, which confirmed that the pass rates of CS1 were statistically significantly different based upon the country in which the course was taught $\chi^2(14) = 46.23$, $p < .001$.

Post-hoc analysis was conducted using the largest five groups, due to a minimum size restriction to run the test. The conclusions previously reported from the ANOVA were unaffected. Tukey's HSD test showed significant differences between pass rates in Finland, with: Australia, $p = .04$, Canada, $p < .001$, and USA, $p = .01$.

Examining Figure 4.5 it can be seen that the mean percentage of non-passing students varies considerably by country. Portugal was found to have the lowest mean pass rate (37.9%), followed by Germany (44.7%), and Brazil (45%). China (11%) and Denmark (6%) were found to have the lowest percentage of non-passing students. But, these findings were obtained using a small sample, $n \leq 5$, and cannot be generalized.

In terms of the four countries which made 80% of the sample, Finland was found to have the lowest pass rate ($n = 24$, $M = 57.7\%$) and the pass rates of the USA ($n = 59$, $M = 70.9\%$), UK ($n = 17$, $M = 69.3\%$), and Australia ($n = 28$, $M = 68.3\%$) were comparable. We hypothesized that on larger samples, the pass rates for each country would tend towards a comparable range, as was the case with the pass rates reported from our five largest country samples.

Programming Language

There has been much debate among CS1 educators as to which language is the most appropriate for introducing students to programming. Considerable discussion has taken place within the literature on whether adopting an objects-first, or imperative first approach, can lower the learning curve for novice students (e.g. [26, 58]). It is also accepted that different languages have different syntactical complexities. For example, novice-specific languages such as Scratch do not require students to develop an understanding of pointers, fundamental to learning C. We therefore hypothesized that the programming language taught to students would moderate the pass rates of CS1.

The 161 pass rates that were identified in this study were grouped into 9 different categories. Our sample was dominated by Java courses, which represented almost half of the pass rates (46.6%). The remaining pass rates were grouped as follows: C (4.3%), Python (10.6%), C++ (6.8%), Visual Basic (1.9%), Fortran (1.9%), Novice Languages (Scratch, Karel, 6.2%), Other/Standalone Languages (8.1%), and Not Stated (13.7%).

A one-way ANOVA was performed. The previously stated assumptions were satisfied. Shapiro-Wilk tests confirmed that the pass rates were normally distributed for all languages, $p > .05$, with the exception of Python, $p = .02$. Levene's test however showed that the variances were heterogeneous, $p = .01$. As a result, an ANOVA was performed using Welch's test to take into account the unequal variances among each group of languages. The one-way ANOVA revealed no statistically significant differences in pass rates of CS1 based upon the programming language that was taught in the course, Welch's $F(8,\ 18.74) = 1.26$, $p = .31$. To confirm the violation of normality had not impacted on the result, we ran a Kruskal-Wallis H test, which confirmed that the pass rates of CS1 were not statistically significantly different based upon the programming language that was taught in the course $\chi^2(8) = 11.01$, $p = .20$.

Examining Figure 4.6 it can be seen that the mean percentage of non-passing students does not seem to vary by programming language taught in the course. The percentage of passing students appears to be lower for C ($n = 7$, $M = 61.1\%$) and C++ ($n = 11$, $M = 56.2\%$) however the differences are not significantly different to the other languages whose pass rates were all in the range 65% to 75%.

Cohort Size

The size of a cohort will have a natural impact on the level of support that a student receives. We therefore explored whether the size of the cohort acted as a moderator. Out of the 161 pass rates used in this study, data on the size of the cohort was only available for 101 (62.7%) of the outcomes that were included in this sample.

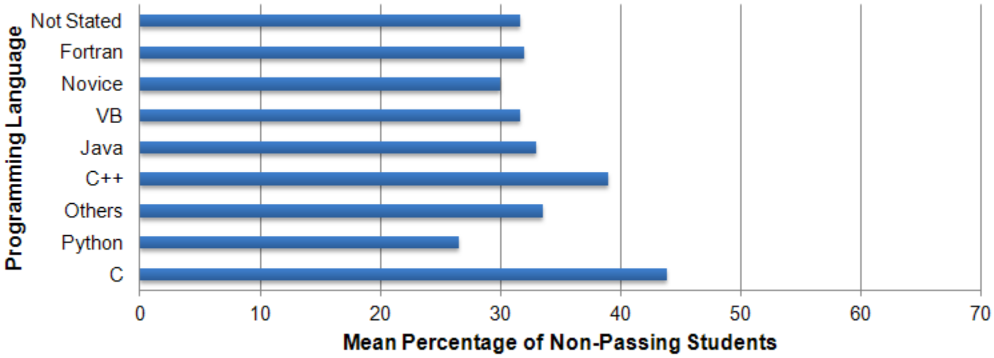


Figure 4.6: Bar chart showing the mean percentage of non-passing students by the different programming languages taught by courses covered by this study.

An independent-samples t-test was run to determine if there were any differences in the pass rates between studies that reported the number of students on the course, and those studies that did not report this. The t-test showed no statistically significant differences between the pass rates of the two groups, $t(159) = .97$, $p = .33$.

To verify previous work, we replicated the binary classification used by [13]. The 101 pass rates were divided into two groups based on cohort size into: small, $n < 30$ students, $k = 10$, and large, $n \geq 30$ students, $k = 91$.

A one-way ANOVA was performed. The previously stated assumptions were satisfied. Shapiro-Wilk tests confirmed that the pass rates were normally distributed for both groups, $p > .05$. Levene's test however showed that the variances were heterogeneous, $p = .02$. As a result, an ANOVA was performed using Welch's test to take into account the unequal variances among each group of pass rates. The one-way ANOVA revealed statistically significant differences in pass rates of CS1 based upon the size of the cohort, Welch's $F(2, 27.23) = 6.78$, $p < .01$.

A practical difference of almost 15% of the mean pass rate between small cohorts ($n = 10$, $M = 80.1\%$) and large cohorts ($n = 91$, $M = 65.4\%$) was found. This finding confirms the result of [13], who also found higher CS1 pass rates for smaller cohorts (82%) than larger cohorts (69%). The implication is that small group teaching may be the most effective way of delivering CS1.

The weakness of these results is that they depend upon the binary classification that was used to partition pass rates into two distinct groups. Correlations on the other hand would show whether or not there were any direct associations between the number of students enrolled on a course, and the overall pass rate. To explore this hypothesis, a Spearman's rank-order correlation was used to assess the relationship between these two variables. Spearman's was chosen in this instance over Pearson's to handle the non-normality of the number of students enrolled on the course, as was confirmed by a Shapiro Wilk test, $p < .05$. Preliminary analysis using a scatterplot showed a weak linear relationship between the two variables. This was confirmed, when a weak correlation between the number of students enrolled on a course and the pass rate of CS1 was found (n = 101, r_s = -.17, p = .10). This suggests that whilst the size of the cohort is associated with the pass rate of CS1, it is only a weak moderator.

Grade Level of Institution

Finally, we explored whether there were any significant differences between the pass rates of universities and other educational institutions (colleges, high school). Our sample consisted of 145 university courses and 16 from other institutions.

A one-way ANOVA was performed. The previously stated assumptions were satisfied. Shapiro-Wilk tests confirmed that the pass rates were normally distributed for Universities, $p > .05$, but not for the pass rates from other grade levels, $p = .01$. Homogeneity of variances was confirmed by Levene's test, $p = .20$. The one-way ANOVA revealed statistically significant differences in pass rates of CS1 based upon the grade level of the institution, $F(1, 159) = 11.62$, $p < .001$. To confirm that the non-normality of the pass rates from other grade levels had not impacted on the ANOVA result, we also ran a Mann-Whitney U test, which confirmed that the pass rates of CS1 were statistically significantly different based upon the grade level of the institution U = 543.01, z-score = -3.49, $p < .001$.

The results on our sample for universities (n = 145, M = 66.4%) and other grade levels (n = 16, M = 79.9%) confirm the findings of [13], who found the pass rates for universities to be lower than other institutions (66% and 88% respectively).

4.5 Discussion

The findings of this chapter confirm the results of the small study conducted by Bennedsen and Caspersen [13]. Compared to [13], this study found an almost identical mean worldwide pass rate of CS1 courses of 67.7%, and comparable results were found based upon cohort size, and institutional grade level. The additional contributions of this study have been to show that CS1 pass rates can vary by different countries, that pass rates have not changed considerably over time, and that pass rates are largely unaffected by the programming language taught in the course.

The implications for practice are that the best teaching approach for CS1 may be one based upon using small groups, and replacing traditional university approaches (e.g. lectures) with classroom based instruction.

The natural question that follows on from this study, is whether or not an average pass rate of 67.7% in CS1 is considered to be low? When considering the figure on its own, we share the sentiments of [13], in that 67% is not an *alarmingly* low pass rate. On the other hand, when considering this figure within the wider context of CS education, we have a different view. Enrollment and retention of CS majors are well known problems [12]. Within the UK for instance, statistics provided by the higher education funding council (HEFCE) show that out of all the available STEM degrees, Computer Science is the only subject where enrollment has consistently declined between the academic years 2001/2002, and 2011/12. A decline in enrollment numbers from 67,896 to 45,158, or 33% [75]. American institutions reported similar declines [217].

Part of this decline may stem from the reputation of Computer Science being a difficult course. If the pass rates can be improved and struggling students provided with assistance, then this may lead to an increase in enrolment numbers, and possibly increased retention. If an estimated 2 million students are currently enrolled worldwide in computing courses worldwide [13], then an improvement in the pass rate of only 5%, would lead to an additional 100,000 students graduating with the skills required to satisfy the future labour demands of a digital economy. Therefore whilst we do not believe that a 67.7% pass rate is *alarmingly* low, we also believe that there is considerable potential for improvement and a need to understand which internal factors are hindering the ability of so many students from understanding programming concepts.

4.5.1 Threats to Validity

Finally we discuss the threats to validity of this study. Whilst the sample of outcomes used by this study was over double the size of the sample used by [13] (98 more outcomes), it is still debatable as to whether or not it is representative of CS1 courses on a worldwide scale. [13] reported that in 1999, there were over one million students enrolled in computing courses across 72 different countries. In this study, we were only able to identify outcomes from 15 different countries, which means that we lack data from the majority of the countries in the world. Only by collecting such data can our results be further validated on a worldwide scale. On the other hand, when considering the results in conjunction with the findings of [13], our findings are consistent, and therefore may be close to the actual results in the population of CS1 courses.

The second threat to validity concerns the sources of the data used. Whilst [13] surveyed authors of selected conference papers and panel attendees directly via email, our data has come from a systematic review process. It is possible that the data used by [13] was already published in the articles that we have used in this study, in which case, 63 of our outcomes may represent data that has already been analyzed by earlier work. However we believe this is unlikely, as 83% of our sample came from articles published during a different time period to the authors contacted by [13], and only 2 articles were included from conference authors that [13] contacted. Also whilst [13] was based entirely on *grey* literature, ours was based entirely on published works. It is possible that our results may suffer from publication bias - as there may be a reluctance among authors and institutions to publish high failure rates, and the actual pass rate of CS1 may be lower than our study has indicated. This is also a limitation of the [13] study. It can be argued however that the reverse is also true.

The third threat to validity concerns the pass criteria of the individual courses that have been included in this study. Studies within the UK generally defined “pass rate” as consisting of those students who had scored over 40% in the course. However, other studies defined “pass rate” as consisting of those students who had scored at least a ‘C’, and others defined “pass rate” as consisting of those students who had scored any grade apart from an ‘F’. Other studies did not supply details at all. Therefore this study unavoidably has to assume that a consistent notion of “pass rate” exists and holds valid across the different teaching contexts covered by this study.

From an interesting discussion which followed the presentation of this study at the ITiCSE ’14 conference, two more threats were identified. The fourth threat to validity concerns the assessment criteria used in the course. The assessments used to measure programming performance will differ by context, and it is possible that failing students in one context would have passed if completing assessments taken from another. However, there is also the possibility of instructors treating CS1 as a “CS filter course” where intentionally difficult assessments are set, which are designed to remove those students who are not committed to majoring in CS before they completely failed the entire course.

The fifth threat to validity concerns the differing educational practices throughout the world. For example, in Greece, several resits of a module are permitted, whereas in our UK institution, only a single retake of a failed module is allowed. Germany which was found to have the second highest failure rate in this study (55.3%) allows students to enrol in higher education courses to receive benefits such as free transport. Such students rarely attend courses and expectedly fail or withdraw before the final exam. These drop-outs are classified as “failing” in our study, but they may not have actually taken any assessments. However until researchers begin to provide more details on the actual break down of non-passing students, we cannot explore the percentage of students who have outright failed compared to the percentage of students who withdrew. We can however state that 67.7% did pass.

4.6 Conclusion

Despite several studies citing a motivation for research as the "high failure rates of introductory programming courses", the evidence on this phenomenon to date, was at best anecdotal in nature. Before conducting any research which is intended to assist, or identify failing students in programming courses, it is important to determine the proportion of such students that actually exist, and whether any aspects of the teaching context act as moderators on programming performance. This provides us with a solid motivation for further research into predictors of programming performance.

In this chapter, we answered the call of for more substantial quantitative evidence on the often cited high failure rates of introductory programming courses, by performing a systematic review of introductory programming literature, and analysing the data extracted from the identified articles. Pass rates describing the outcomes of 161 different CS1 courses, that ran across 15 different countries were extracted from the literature and analysed. The mean worldwide pass rate of CS1 that was found by this study was 67.7%, which is practically identical to the 67% mean pass rate reported by Bennedsen and Caspersen. Further analysis revealed significant, but not substantial, differences in pass rates by moderators including the country in which the course was taught, grade level of the institution, and size of the cohort, $p < .01$. But, neither the programming language taught, or the year in which the course ran were found to moderate the pass rates. Although we have identified aspects of the teaching context that can have a significant moderating effect on the pass rates of CS1, the differences themselves were not substantial. As such, the fundamental question as to why two thirds of students can acquire programming skills, whilst one third of students endlessly struggle remains.

There is a need to identify and assist such students as early as possible. If factors that can influence programming performance can be identified, then they could be applied for the purpose of predicting weaker students before any assessments have taken place. This would allow such students to withdraw, or to be provided with additional support to assist their learning progress, without the need to complete formal exams which can take a considerable amount of both time and effort for an instructor to process. As aspects of the external teaching context do not seem to substantially effect course outcomes, in the remainder of this thesis we will explore whether internal characteristics of students can influence their programming performance.

Chapter 5

Traditional Predictors based upon Learning Theories

A modified version of this chapter appears in the following peer-reviewed publication:

- C. Watson, F.W.B. Li, and J.L. Godwin. No Tests Required: Comparing Traditional and Dynamic Predictors of Programming Success. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education* (SIGCSE '14), pages 469-474, 2014, ACM.

Factors derived from traditional learning theories and academic background has been the focus of research into predictors of programming performance for over fifty years. But previous research efforts have struggled to reveal factors which can consistently predict programming performance across a range of different teaching contexts. Even in the case of the often cited predictors such as math background, results vary considerably. As such, there is still no consensus on which factors influence students' abilities to learn programming.

In this study, the relations between the programming performance of 39 students at our University, and a set of 34 mainly unverified predictors based upon traditional learning theories are examined. Using stepwise regression, several predictors were combined to produce a context-tuned regression model, which could explain 35.9% of the variance in performance. However, when only considering the predictors that performed consistently across a range of contexts, the model could only explain 21.4% of the variance. This chapter argues that predictors based upon traditional learning theories yield inconsistent results when they are applied in different contexts because they are unrelated to the regular programming activities of students, and therefore cannot reflect changes in the students' programming knowledge over time.

5.1 Introduction

In the previous chapter, we presented the widespread concerns among researchers over the worldwide high attrition and failures rates of CS1, and we showed that on average, one third of a cohort fail to pass an introductory programming course. We also showed that fundamental aspects of the teaching context, such as the programming language taught to students, do not moderate the overall failure rates of this course. The question which follows on from these findings has been the subject of much debate among researchers of computing education for the past fifty years, and will form the subject of the remainder of this thesis. That is, if aspects of the external teaching context, such as the programming language taught to students, does not have a moderating effect on the failure rates of CS1, and the failure rates of CS1 have not improved over time, then which internal characteristics of certain students enables them to easily acquire programming skills, whilst other students endlessly struggle?

Identifying struggling programming students can be challenging. Introductory programming courses generally have a high student-to-lecturer ratio, an average of 50:1 or greater in the case of our University, and often lecturers do not know how well students are performing until after they have completed the first formal assessment. This may not take place until several weeks after the course has started, and given potentially high enrolment numbers, it can take an instructor a considerable amount of time and effort to process these assessments. Even if an assessment was indicative of overall performance, by the time it was processed, it may be too late for struggling students to withdraw, or for instructors to intervene to prevent students from failing. This is a cause of great concern for computer science educators, and unsurprisingly over the past fifty years these concerns have led to an abundance of research focusing on identifying predictors of programming performance. As well as identifying characteristics that can be used to predict struggling students without the need to use formal assessments, other motivations for exploring such internal predictors include [33]:

1. Exploring relationships between programming and other cognitive abilities.

2. Providing automated interventions for students based upon their characteristics, such as different compilation feedbacks based upon prior knowledge.

3. Advising students on major selection.

4. Improving programming classes for non-computing majors.

5. Determining the importance of different characteristics that influence the ability of students to acquire programming skills.

Whist an abundance of predictors have been explored, the limitation of research to date is a distinct lack of verification across different teaching contexts. Researchers examining predictors of programming performance have a tendency to judge the value of a predictor, based upon the statistical significance of how the predictor performed for a single sample of students, working within a single teaching context. However, this approach is problematic, as verification studies of the same predictor in a different teaching context have a tendency to yield inconsistent results [200].

Consider the often cited predictor of Math performance. Using the SAT Math instrument to measure this ability, [203] reported a strong correlation of $r(88) = .51$ between the math and programming performance on a sample of college students. [91] reported a similar moderate correlation of $r(32) = .48$. On the other hand [3] reported a weak correlation of $r(50) = .13$, and [80] reported a weak correlation of $r(45) = .13$.

These varying findings stress the importance of verifying predictors across a range of different teaching contexts, but also raises questions over the generalisability of any research into predictors. No matter how well designed individual studies are, the results of such individual studies are often insufficient to provide conclusive answers to questions of general importance. Only by evaluating the performance of different predictors across a range of teaching contexts, can the context independence of a predictor be judged, and thus, its wider usefulness as an enabler of programming ability be judged. It is also possible that no characteristic when considered singularly is predictive of performance. If a set of context independent predictors could be identified, then they could be combined to construct a predictive multivariate model, which could be applied for the purpose of pre-screening potentially weaker students without the need for formal assessments, or, applied for training purposes (e.g. improve Math ability to improve programming ability).

Chapter Contributions

The purpose of this study was to explore the relations between programming performance and a set of 34 mainly unverified predictors based upon traditional learning theories. Based upon our results, a context-tuned regression model was constructed to predict the performance of programming students across a range of different teaching contexts. This chapter answers the following thesis research question:

> *RQ2: Which traditional learning theories describing the psychological and cognitive aspects of learning, and which aspects of students' academic backgrounds are predictive of their programming performances? (RO2)*

The contributions of this chapter are:

1. To evaluate the context dependence of 34 predictors that are based upon traditional learning theories, demographic, or academic background.

2. To present a predictive model based upon the context independent predictors identified in this study. This model could explain a moderate amount of the variance in performance ($R^2 = 21.4\%$) of 39 programming students in our context.

5.2 Related Work

Unsurprisingly over the past fifty years concerns over the high failure rates of programming courses have led to an abundance of research focusing on identifying predictors of programming performance.

Numerous predictors have been explored. The majority of early work conducted during the 1960's and 1970's focussed upon using standardized programming aptitude tests as predictors of performance. Popular instruments included the IBM Programmer Aptitude Test (IBM PAT), and the Computer Programmer Aptitude Battery (CPAB). Predictors researched during these eras included: arithmetic reasoning, letter series reasoning, and figure classification [3, 8, 30]. However, despite measuring a student's level of *programming aptitude*, no single instrument or characteristic emerged as a conclusive predictor of performance.

This led to an expansion of the types of characteristics explored, and during the 1980's and 1990's, researchers expanded the search for predictors to include academic, cognitive, demographic, personality, and psychological traits. These included: performance in academic courses (with an emphasis on math [161] and science [159]), spatial ability [91], intellectual development, gender [113], personality style (Myers-Briggs [202]) and learning style (Kolbs [42]). As with the previous two decades, no single instrument or characteristic emerged as a conclusive predictor of performance.

During the 2000's, researchers mainly repeated efforts of the previous two decades and continued to explore various academic, cognitive, and psychological predictors of performance. These included: comfort level [210], self-efficacy [189] and learning strategies [16]. Again, efforts were largely unsuccessful and the search continued.

During the 2010's, researchers in the area followed the increasing trend of exploring "big data", and began to apply data-mining and statistical techniques to analyse various forms of autonomously gathered data, describing aspects of a student's ordinary programming behaviour. Predictors in this class have shown more promise than previous methods, and will be further explored in detail later in this thesis.

5.3 Research Design

There were two main purposes of this study. The first purpose was to determine whether any relationships existed between the programming performance of students and a range of predictors based upon traditional learning theories and academic background. These included: previous programming experience, previous academic experience, attributional style, behavioural traits, self-esteem, learning styles and learning strategies. These predictors were selected as the majority had only been evaluated in either a single or small number of contexts, or there was considerable variance in the results reported by previous researchers. Evaluating these predictors in our teaching context allows us to explore whether they are also limited by the same context dependency as their well explored predecessors.

The second purpose of this study was to determine whether the context independent predictors we had identified could be applied to form a multivariate predictive model, capable of accounting for a large percentage of the variance in performance. Combining the predictors which have performed consistently across a range of different teaching contexts supports the likelihood of the model generalising to further teaching contexts.

5.3.1 Research Questions

To answer thesis RQ2, the following sub-questions were answered:

1. Are there any correlations between the predictors examined in this study and programming performance? If applicable, are these findings consistent with previous research, or inconsistent, suggesting a context dependency?

2. Are any categories of predictor more strongly correlated with programming performance than others?

3. Are any predictors either singularly or additively useful for predicting the performance of students studying an introductory programming module?

5.3.2 Participants

In total 39 students (36 male) from the 2012/13 cohort completed the instruments used by this study. A priori analysis was carried out to verify that no significant differences existed between the mean overall scores of the class on the reference criterion, and those who agreed to participate in this study. Test assumptions of normality (Shapiro-Wilk test) and equality of variance (Levene's test) were satisfied, and a t-test showed no significant differences between the performance of those who participated in this study ($t(55) = 1.21$, $p = .26$) and the remainder of the cohort.

5.3.3 Instruments

Six instruments were used to collect data from the participating students. These included: an in-house questionnaire designed to gather data on students prior academic experiences, an attributional style questionnaire based upon Weiner's model, Rosenberg's Self-Esteem Scale, Kolb's Learning Style Inventory, Gregorc's Style Delineator, and the Motivated Strategies for Learning Questionnaire (MSLQ).

The background questionnaire gathered data on students' genders, high school GPA, college GPA, lectures attended per week, previous math courses and grades, previous science courses and grades, and prior programming experience (in terms of the years of experience and longest program written for each language they reported).

Attributional styles were measured by replicating the method used by Cantwell-Wilson [211], which used an instrument measuring students based upon the four dimensions of Weiner's attributional style model [201].

Self esteem was measured by using Rosenberg's Self-Esteem Scale. The scale consists of ten questions where respondents answer how their self-esteem relates to a specific situation using a four point scale, ranging from strongly agree to strongly disagree. To replicate the only study to date that used this instrument we also reworded the standard questions to be based upon a programming context [14].

Kolb's Learning Style Inventory (LSI) was the first learning style instrument used in this study. The questionnaire consists of a set of twelve sentences where respondents rank order four completions on a scale of one to four. The LSI provides scores (range 12 to 48) for an learner's predisposition toward concrete experience, reflective observation, abstract conceptualization, and active experimentation. The Gregorc Style Delineator was the second learning style instrument we used. The instrument consists of a set of ten sentences where individuals rank order four completions on a scale of one to four. The highest score among the four dimensions determines the dominant learning style.

The Motivated Strategies for Learning Questionnaire was used to measure the motivations and learning strategies of students based upon fifteen different dimensions: six based upon motivational aspects and nine scales based upon learning strategies. The scales can be used together, but given their modular design, they can also be administered individually, using a scale of one to seven.

5.3.4 Criterion Variable

We used the students overall coursework mark as the measure of programming performance. This consisted of a weighting of marks obtained on a mid-term exam (25%), project (25%), a practical exam (40%), and weekly programming exercises (10%).

5.3.5 Predictor Variables

The relationships between 34 predictors and the programming performance of students from our context were examined. These included:

1. Previous Programming Experience: has prior experience, number of languages previously studied, longest program written, years of experience.
2. Previous Academic Experience: college grades: physics, chemistry, maths; university grades: discrete math, calculus, GPA: college, high school.
3. Attributional Style: scores for 4 scales corresponding to Weiner's model.
4. Behavioural Characteristics: lectures attended, hours worked in a part time job.
5. Self-Esteem: overall score on Rosenberg's Self Esteem Scale.
6. Learning Styles: 8 scores taken from Kolb's and Gregorc's Instruments.
7. Learning Strategies and Motivations: 12 scales taken from the MSLQ.

Definition of Context Dependence

The correlations were classified based upon [47], who defined five strength categories:

- no association ($r < .10$)
- weak ($.10 \leq r < .30$)
- moderate ($.30 \leq r < .50$)
- strong ($.50 \leq r < .80$)
- very strong ($.80 \leq r \leq .1.0$)

In this study, the context dependence of a predictor was judged by considering the strengths of the correlations based upon the classification above. We considered both vote counting and variance to make a judgement. If a clear majority strength existed across the results for a predictor then it was classified context independent. For example, if the correlations for one predictor were: .63, .62, .57, .40, then the predictor would be classified as context independent on the basis that three of the results suggested this predictor was strong, and only a single result disagreed.

On the other hand, if no clear majority strength exists across the results for a predictor, then it was classified as context dependent. For example, if the correlations for one predictor were: .02, .10, .23, .47, .48, .61, .71, then there is no clear majority class, and the results range from no association to strong association.

5.4 Results

A Shapiro-Wilk test confirmed that the scores obtained by the sample on the criterion variable were normally distributed, $p > .05$, with a mean score of 66.7% ($SD = 10.3\%$). In the remainder of this section, the findings on the relationships between each of the predictors and programming performance is presented, and compared to the results of previous researchers.

5.4.1 Performance of the Cognitive Predictors

Overall results showing the correlations of the cognitive predictors explored by this study with programming performance are presented in Table 5.1.

Learning Styles

In total 38 participants completed both of the learning style instruments. No significant correlations, $p > .10$, were found between any of the four Kolb LSI dimensions and programming performance. The results on each dimension were: concrete experience (accomodator) r = -.18, $p = .29$, reflective observation (assimilator) r = -.07, $p = .69$, abstract experimentation (diverger) $r = .14$, $p = .39$, and abstract conceptualization (converger) $r = .10$, $p = .53$. As can be seen from Table 5.1, these results are within the range of previous work [29, 36, 42], suggesting that there is little to no relation between the dimensions of Kolb's LSI with programming performance.

Results for Gregorc's Style Delineator were more interesting. No significant correlations between scores on the concrete/random dimension and performance were found, r = -.14, $p = .39$. But moderate, marginally significant correlations were found for each of remaining dimensions, including: abstract/random r = -.33, $p = .05$, concrete/sequential $r = .27$, $p = .10$, and abstract/sequential $r = .29$, $p = .08$. Our findings are consistent with the previous two studies [95, 96] that have explored the use of Gregorc's Delineator as a predictor, who reported similar moderate correlations for the concrete/sequential dimension $r(218) = .35$, and an identical moderate correlation $r(131) = .30$ for abstract/sequential dimension. This suggests that certain dimensions of Gregorc's Delineator may perform as a reasonable predictor of performance.

Learning Strategies

All 39 students completed the MSLQ. Only 2 studies to date [14, 16] have explored the relations between programming performance and scores on the various motivational and learning strategies scales on the MSLQ. Compared these studies an identical strong

Table 5.1: Table showing the correlations (r) of the cognitive predictors explored by this study with programming performance, and compared to prior research if applicable. (* $p < .10$, ** $p < .05$, *** $p < .01$).

Category and Predictor	Previous Research, (r)	Our Study, (r)	Context Dependent	References
Learning Styles				
Kolb's LSI				
Converger (AC)	.15, .15, .26	.10	No	[29,36,42]
Diverger (AE)	.02	.14	No	[42]
Accomodator (CE)	-.16, -.23	-.18	No	[36,42]
Assimilator (RO)	-.36, -.13, .06	-.07	Yes	[29,36,42]
Gregorc's Style Delineator				
Concrete/Sequential	.35	.27 *	No	[95]
Abstract/Sequential	.13, .30	.29 *	No	[95,96]
Learning Strategies (MSLQ)				
Critical thinking	.57	.28 *	Yes	[16]
Total metacognitive	.54	.14	Yes	[16]
Resource strategy; Effort	.62	.28 *	Yes	[16]
Resource strategy: Peer	.37	-.06	Yes	[16]
Total resource strategy	.56	.04	Yes	[16]
Task value	.44, .54	.06	Yes	[14,16]
MSLQ total	.49	.22 *	Yes	[14]
Intrinsic goal orientation	.51	.33 *	Yes	[14]
Total self-efficacy	.54	.54 ***	No	[14]

correlation for the self-efficacy dimension was found, $r = .54$, $p < .01$. Marginally significant correlations were found for: intrinsic goal orientation, $r = .33$, $p = .04$, critical thinking $r = .28$, $p = .08$, resource strategies: effort $r = .29$, $p = .08$, and MSLQ total score $r = .22$, $p = .09$. These findings confirm the research by [14,16] who suggested that students who perform well in programming courses have high levels of intrinsic motivation and self-efficacy.

However the findings of this study differed on a number of dimensions. No significant correlations were found between the total scores on the resource strategies scale, $r = .05$, $p = .76$, task value scale, $r = .06$, $p = .70$, and the metacognitive strategies scale, $r = .15$, $p = .37$, was found to have a significantly lower correlation than previous researchers reported.

These findings suggest that whilst certain aspects of the learning strategies employed by students are related to programming performance, further research is required to identify precisely what the most significant dimensions are across a range of different contexts.

5.4.2 Performance of the Psychological Predictors

Overall results showing the correlations of the psychological predictors explored by this study with programming performance are presented in Table 5.2.

Affective Characteristics

All 39 students completed Rosenberg's self-esteem scale. A weak, but not significant, relation between score obtained on the instrument and programming performance was found, $r = .13$, $p = .42$. Only one other study to date [14] used Rosenberg's instrument to examine the relationship between self-esteem and programming performance. However a moderate correlation between these two variables, $r(54) = .36$, was reported. The differing results between this study and prior research suggests that self-esteem may be a context dependent predictor.

All 39 students completed the background questionnaire section on their attributions of success. To date only 3 studies [74,190,211] have explored relationships between attributions and performance. Significant, $p < .05$, but weak, correlations were found between performance and attributions of success to task difficulty, $r = -.10$, and attributions to effort, $r = .07$. A moderate and marginally significant correlation was found for attribution of success to luck, $r = -.31$, $p = .05$, and a moderate, significant correlation was found for attribution of success to ability, $r = .40$, $p < .05$.

The correlations reported by this study are consistent with, and within the range of correlations reported by the previous three studies on attributions to: task difficulty, $r = -.20$ to $.20$, and effort, $r = .07$ to $.16$. However much stronger relations for both attributions to ability, $r = .07$ to $.16$, and attributions to luck, $r = -.22$ to $.05$, were found. These conflicting results suggest that further research on how attributions of success relate to performance is required.

Behavioural Characteristics

All 39 students completed the section of the background questionnaire designed to gather data on behavioural characteristics. Interestingly, no relationship was found between the number of lectures attended and the performance of students, $r = .02$, $p > .10$. This may suggest that lectures do not provide a good means to teach programming, assuming that students had accurately reported their attendance.

Only 7 students had a part time job. A strong negative correlation was found between the hours students worked in a part time job and programming performance, $r = -.64$, $p < .01$. Although significant, this result must be interpreted with caution due to the very small number of students who actually had a part time job.

Table 5.2: Table showing the correlations (r) of the psychological predictors explored by this study with programming performance, and compared to prior research if applicable. (* $p < .10$, ** $p < .05$, *** $p < .01$).

Category and Predictor	Previous Research, (r)	Our Study, (r)	Context Dependent	References
Self Esteem				
Rosenberg's Self Esteem	.36	.10	Yes	[14]
Attributional Style				
Luck	-.22, -.19, .05	-.31 *	No	[74, 190, 211]
Effort	.06, .19, .22	.07 **	No	[74, 190, 211]
Task Difficulty	-.20, -.05, .25	-.10 **	Yes	[74, 190, 211]
Ability	.08, .08, .16	.40 **	No	[74, 190, 211]
Behavioural Characteristics				
Lectures attended	.12	.02	No	[1]
Part time job hours	.05	-.64 **	Yes	[202]

5.4.3 Performance of the Academic Predictors

Overall results showing the correlations of the academic predictors explored by this study with programming performance are presented in Table 5.3.

Previous Programming Experience

All 39 students completed the background questionnaire section on prior programming experience, of which 15 students reported prior experience.

A t-test revealed significant differences in the performance those students who had prior programming experience prior to enrolling on the course (n = 15, M = 71.7, SD = 10.5) and those students who did not (n = 24, M = 64.3, SD = 10.7), ($t(37) = 2.12$, $p < .05$). These findings are consistent with previous research such as [205], but contradict research such as [14, 190].

Further analysis showed more interesting relations between prior experience and performance; however none of the following measures were significant, $p > .05$. The number of languages that a student had previously studied weakly correlated with performance, $r = .24$, the longest program that a student had written prior to enrolment on the course also weakly correlated, $r = .15$, and surprisingly years of programming experience negatively correlated with performance, $r = -.20$.

An explanation for these findings could be that self-taught students have developed bad practices, and when formally assessed are penalised. The other explanation is that students have overrated their own experiences on the questionnaire and that years of experience was not the best measure of programming experience.

Table 5.3: Table showing the correlations (r) of the academic predictors explored by this study with programming performance, and compared to prior research if applicable. (* $p < .10$, ** $p < .05$, *** $p < .01$).

Category and Predictor	Previous Research, (r)	Our Study, (r)	Context Dependent	References
Prior Programming				
Has Prior Experience	.03, .06, .10, .23, .25, .27, .28, .32, .44	.13 **	Yes	[16, 28, 70, 84, 164, 189, 190, 205, 210]
Number of Languages	.05, .07, .29	.24	Yes	[87, 159, 190]
Longest Program	.25	.15	No	[147]
Years Experience	.06	-.20	Yes	[15]
Longest Java Program	-.10	.01	No	[189]
Academic Background				
College Physics	.40, .50, .61	.31	Yes	[91]
College Chemistry	.31, .45, .52, .54, .55, .56, .62	.27	No	[91]
College Math	.19, .40, .44, .43, .50, .51, .55, .61	.20	No	[52, 91]
University Discrete	.37	.06	Yes	[204]
University Calculus	.21, .33, .58	.21 *	No	[30, 178, 204]
College GPA	.20, .21, .25, .30, .37, .68	.37	No	[8, 42, 77, 132, 159, 202]
High School GPA	.07, .26, .39, .46, .73	.27 *	Yes	[65, 159, 189, 202, 203]

Academic Background

To establish the relationship between previous academic experience in mathematics and science, the achievable grades for each subject were ranked, with the highest rank given to the highest possible grade, and the lowest rank given to the lowest possible grade. No significant correlations were found, $p < .05$, between either: grades in college physics, $r(24) = .31$, chemistry, $r(13) = .27$, math, $r(26) = .20$, university discrete math grade, $r(13) = .06$, or college GPA, $r(33) = .21$. But, marginally significant correlations were identified between performance and university calculus grade, $r(24) = .37$, $p = .06$, and high school GPA, $r(36) = .27$, $p = .10$.

These findings are consistent with previous research that suggest generally academic background factors can be weakly correlated with programming performance [14, 28] and that grades obtained in calculus courses are more strongly related to programming performance, than grades obtained in discrete courses [178].

5.4.4 Overall Performance of the Predictors

To answer the first and second research questions, correlations were found between programming performance and all but 8 of the predictors that were examined by this study. These predictors were found not to belong to any particular category, and included 2 academic, 4 cognitive, and 2 psychological predictors.

Out of the remaining 26 predictors that were found to correlate with performance, 19 were found to only weakly correlate ($.10 \leq r < .30$), 5 were found to moderately correlate ($.30 \leq r < .50$), and 2 were found to strongly correlate ($.50 \leq r < .70$). The average strength of correlations were comparable across all groups, with mean correlations of: academic $M = .20$, cognitive $M = .20$, and psychological $M = .23$. When considering these results along with related work on the predictors that were not examined by this study, these results suggest that in general, traditional predictors are only at best weakly related to the programming performance of students, and that their general influence on programming performance to be doubtful.

Whilst the majority of predictors were only found to weakly correlate with performance, it is more important to consider whether any predictors yielded consistent results across a range of different teaching contexts, and therefore, would suggest their usefulness in constructing a predictive model that could potentially be used to predict weaker students across a range of different teaching contexts. When considering the context dependence of the predictors examined using our results and the available literature, 18 predictors were found to yield correlations of varying strengths with programming performance, and therefore were classified as being context dependent. The most notable example found by this study was the learning strategies and motivations predictors, where only a single predictor (self-efficacy) was found to be consistent with previous research. This emphasises our original motivation for work, in that, no matter how statistically significant a set of results are, unless the performance of traditional predictors are verified across a range of different teaching contexts, the wider applicability of a predictor cannot be judged.

Table 5.4 shows the top context independent predictors identified by this study. Disappointingly, only five of the predictors examined yielded an average moderate correlation strength, and whilst the remaining 14 predictors were determined to perform consistently across a range of teaching contexts, the average strength of their correlations were weak at best. We note that no particular category of predictor significantly outperformed another, although, more psychological predictors were found to be context independent, but, consistently unrelated to performance across the range of contexts examined, with average correlations in the range of $.07 \leq r \leq .19$.

Table 5.4: Table showing the strongest five context independent predictors identified by this study, compared to the average correlation strength reported by previous research.

Predictor	Category	Previous Research, ($\bar{r}$)	This Study, (r)
MSLQ Total self-efficacy	Cognitive	.54	.54
College Physics	Academic	.42	.31
University Calculus	Academic	.36	.21
College GPA	Academic	.34	.37
Gregorc Concrete/Sequential	Cognitive	.31	.27

5.4.5 Regression Analysis

Two stepwise regression analyses were performed to determine whether any combinations of the predictors examined were indicative of performance.

The first regression was performed to evaluate whether any combination of predictors from the six instruments used by this study (including new predictors which were not compared to previous research) were indicative of programming performance. Consideration was given for all the predictors examined in this study, with the exceptions of: academic background (apart from GPA) and hours worked in a part time job, due to the small number of students with results for each of these predictors.

The full model of MSLQ Self Efficacy, MSLQ Elaboration, and MSLQ Help Seeking (Model 3) was found to be statistically significant and could account for a moderate amount of the variance in performance, adjusted $R^2 = 35.9\%$, $F(3, 29) = 6.97$, $p <.01$. The addition of MSLQ Elaboration to the model (Model 2) led to a statistically significant increase in R^2 of 10.8%, $F(1, 30) = 4.64$, $p < .05$. The addition of MSLQ Help Seeking (Model 3) also led to a significant increase in R^2 of 14.6%, $F(1, 29) = 7.26$, $p < .05$. These results suggest that a model based on three dimensions of the MSLQ are a moderate predictor of performance in our context.

However it is more important to construct a generalisable model that could predict the programming ability of students across a range of different teaching contexts. Therefore, a second regression was performed to evaluate whether any combination of the context independent predictors identified by this study were indicative of programming performance. A model consisting of only MSLQ Self Efficacy was found to be statistically significant, but could only account for a smaller amount of the variance in performance than the previous model, adjusted $R^2 = 21.4\%$, $F(1, 38) = 11.35$, $p < .05$.

This result suggests that whilst we can construct a fine-tuned model for predicting performance in our context, that it is difficult to construct a generalisable model based on traditional predictors alone, even when those predictors appear to perform consistently across different contexts.

5.5 Discussion

For almost fifty years, researchers have examined how factors based upon traditional learning theories can be predictive of the students' programming performance. However, out of the previously unverified predictors that were examined by this study, only self-efficacy, $r = .54$, $p < .01$, was found to significantly strongly correlate with performance.

In this study, we constructed a context-tuned regression model that could explain 35.9% of the variance in the programming performance of our students. However, when only considering the predictors that performed consistently across a range of contexts, the model could only explain 21.4% of the variance. Accurately predicting weaker students with such a large proportion of unexplained variance would be difficult.

We believe that the reason for such varying results is that the predictors based on traditional learning theories examined over the past fifty years are fundamentally limited by their static nature, and therefore, they cannot reflect changes in students' learning progress over time. The predictors examined to date cannot change in response to increases in the students' programming knowledge, and therefore are incapable of predicting programming performance across a range of different teaching contexts. e.g. improvements in programming does not lead to changes in knowledge of chemistry.

It can be argued that the timing of applying some of the measures is important. For example, a students programming self-efficacy may be higher at the end of a course after they have mastered programming concepts. However the bulk of measures explored are static, and cannot directly tell an instructor whether a student has mastered the threshold concept of object-oriented programming.

Nevertheless, possibly inspired by the push towards big data analysis, researchers have begun to explore more data-driven predictors, which instead of being based upon data gathered from static tests, are based upon analysing data which is gathered from an IDE, describing the programming behaviours of students. The data gathering involves collecting snapshots from students source code, either by a defined interval, or when students' perform actions such as saving the project they are currently working on, which then enables the creation of data-driven approaches for finding characteristics that contribute to students' success. For instance, the number of errors that students have made, or the time which they take to resolve errors when compared to their peers.

The main benefit of data-driven approaches based upon how students' solve programming errors is that, they are directly based on the regular programming activities of a student, and therefore can directly reflect changes in their learning progress over time. This is not the case for more traditional predictors explored over the past fifty years, such as age, gender, and high-school performance, which remain static within the context of a course, whereas changes can be observed in students' programming behaviours. Such predictors could be also applied to drive an expert system, so that students can be automatically provided with appropriate interventions when required. Such interventions would be difficult to provide using traditional predictors based upon learning theories, and would be incapable of automatically adapting to changes in students' learning progress over time.

In defence of the predictors based upon traditional learning theories which researchers have previously explored, it can be argued that the majority fail to predict programming performance because they were never actually designed for this purpose. For instance, it is a reasonable expectation that learning style models would not be strongly associated with programming as their main motivation was to influence teaching methods, rather than predict the programming knowledge of students. On the other hand, even if a test based predictor was relevant, then the static limitation still remains. Tests would have to be repeated consistently overtime which is both cumbersome to students and instructors alike. The automated nature of the data-driven predictors however would mean they would not suffer from the same limitation.

5.5.1 Threats to Validity

Finally we acknowledge the limitations of this study. There are numerous difficulties associated with identifying predictors of programming performance.

The main threat to validity of this study is that the results are drawn on predictors that have only been examined in a small number of contexts, most in less than four. Should a predictor be trialled across a greater number of contexts, it is possible that eventually, it will be found to be context independent. However, we note that when examining the previous body of research on predictors which have been trailed across a greater number of contexts, the inconsistencies remain, and we have no reason to believe that this will not hold true for any of the predictors that this study has examined.

Additionally, although we have proposed a regression model in this study which was based upon the factors we determined to be context independent, there is no certainty that this will map to different teaching contexts until it its trailed within them. There is also the limitation of over-training the model based upon our chosen criterion variable. Furthermore, there are numerous predictors which were not examined as part of this study. If a different set of context independent predictors could be identified from the literature and incorporated into the model, then a greater percentage of the variance may have been accounted for, than the model developed on the limited number of predictors that were examined in this study.

5.6 Conclusion

In the previous chapter, we showed that aspects of the external teaching context do not moderate the overall success rates of programming courses, and that the failure rates have remained constant over time. In this chapter, we began to explore the possible internal characteristics of students based upon traditional learning theories which may be predictive of their programming performance.

For almost fifty years, researchers have examined how such characteristics are predictive of programming performance. However previous research struggled to identify any factors which are predictive of programming performance across a range of different contexts, and on the rare occasion when a predictor is found to perform consistently, it is usually only found to be weakly associated with performance at best.

In this study, we constructed a context-tuned regression model that could explain 35.9% of the variance in the programming performance of our students. However, when only considering the predictors that performed consistently across a range of contexts, the model could only explain 21.4% of the variance. The question remains as to which factors are accounting for such a large proportion of the unexplained variance.

We believe that the reason for such varying results is that the predictors based on traditional learning theories examined over the past fifty years are fundamentally limited by their static nature, and therefore, they cannot reflect changes in students' learning progresses over time. The predictors examined to date cannot change in response to increases in students' programming knowledge, and therefore are incapable of predicting programming performance across a range of different teaching contexts.

In the next chapter of this thesis, we begin to explore data-driven predictors that are based upon analysing data which is directly gathered from an IDE describing the programming behaviour of students.

Chapter 6

Data-Driven Predictors based upon Programming Behaviour

A modified version of this chapter appears in the following peer-reviewed publication:

- C. Watson, F.W.B. Li, and J.L. Godwin. Predicting Performance in an Introductory Programming Course by Logging and Analyzing Student Programming Behavior. In *Proceedings of the 13th IEEE International Conference on Advanced Learning Technologies* (ICALT '13), pages 319-323, 2013, IEEE.
  Outstanding Paper Award.

The purpose of this study was to identify data-driven metrics, which could illuminate the *symptoms* of struggling students based upon aspects of their programming behaviour. Unlike previous research, our metrics are validated by using three independently gathered datasets from students taking a first programming course, which supports the generalisability of the findings to other teaching contexts.

Initial results showed that weaker students are characterised by spending a greater percentage of their lab time resolving errors than their peers and by making a greater percentage of successive errors than their peers. Based upon these results, we developed the Robust Relative algorithm, which quantifies several metrics into a performance predictor. The novel relative penalizing of students based upon how their resolve times for different types of error compares to the resolve times of their peers resulted in a stronger predictor than any of the previously identified metrics when considered singularly. The regression analysis showed that the Robust Relative scores could significantly ($p < .01$) explain a large amount of the variance in the performance of all three samples of students, $R^2 = 42.19\%$, 43.65% and 44.17%, accounting for almost double the variance in performance than explained by the Error Quotient.

6.1 Introduction

The main shortcoming of predictors derived from traditional learning theories is that they are rarely found to consistently predict the programming performance of students across a range of different contexts. We argue that the reason for such large variations in results is that the predictors examined to date have not specifically been designed for this purpose. Additionally the predictors are static in nature, and therefore cannot reflect changes in the students' programming knowledge over time. The traditionally measured attributes (e.g., learning styles) also cannot easily be changed.

Another shortcoming is that the traditionally explored predictors also require students to complete batches of lengthy test in order to gather predictive data. For instance, the Motivated Strategies for Learning Questionnaire from which predictors based upon learning strategies are derived, requires students to answer 80 questions on a 7 point scale. The problem with these tests is that neither students want to complete them, nor do instructors want to process them. It is therefore common to administer a test only on a single occasion reflecting the state of students at a single point in time. But even if a test was repeated the fundamental characteristics which it is designed to measure are still static in nature and unrelated to programming.

Possibly inspired by the push towards big data analysis, researchers have begun to explore driven predictors which are based upon analysing directly logged from an IDE, which describes the programming activities of students. The data gathering involves collecting snapshots of source code and error messages, usually when students' perform actions such as saving the project they are currently working on. This data then enables the utilization of data-driven approaches for identifying programming behaviours that are predictive of student performance.

The main benefit of data-driven approaches based upon how students' solve programming errors, or how they schedule their time, and whether they pay attention to code quality, is that, they are directly based on the regular programming activities of a student, and therefore can directly reflect changes in their learning progress over time. This is not the case for more traditional predictors explored over the past fifty years, such as age, gender, which remain static within the context of a course.

As well as being able to dynamically identify struggling students, such predictors can be also applied to drive an expert system, so that students can be provided with appropriate interventions when required. Such interventions would be difficult to provide using traditional predictors and would not be capable of adapting to changes in a student's learning progress over time.

To date, only Jadud [82] has attempted to quantify several aspects of programming behaviour into a performance predictor. This was known as the Error Quotient (EQ). Although previously used by several studies, the EQ was shown to be a weak predictor of performance. This could be due to several flaws concerning the incompleteness and inaccuracy of the method which we attempt to address and expand upon in this work.

Chapter Contributions

The purpose of this study was to identify data-driven predictors of programming performance, which could yield consistent results when applied to multiple datasets. Unlike previous research, our findings are validated by using three independently datasets gathered from students taking the programming module at our university (2011/12, 2012/13 and 2013/14). This increases the likelihood of the predictors generalising to different teaching contexts. This chapter answers the following thesis research question:

> *RQ3: Which data-driven metrics derived from data describing students' programming behaviours are predictive of their programming performances? (RO3)*

The contributions of this chapter are:

1. Validation of the previous research into the most common types of error which novice programmers generate.

2. An exploration of the relations between the resolve times of different types of errors and student performance.

3. Identification of 10 new data-driven predictors. Unlike previous research 9 of these predictors yielded significant consistent results on all three datasets.

4. Development of the Robust Relative algorithm, which quantifies several aspects of programming behaviour into a predictive score. Unlike previous research the algorithm incorporates scoring based upon both the frequency and resolve times of different types of errors. The novelty of the algorithm is to relatively penalize students based upon how their resolve times for different types of errors, compares to the resolve times of their peers. Unlike previous methods (such as the Error Quotient), the algorithm yielded consistently strong results on three independently gathered datasets (R^2 = 42.19%, 43.65% and 44.17%).

5. Verification of the performance of the Error Quotient on the three datasets.

6. Comparison of the data-driven predictors identified in this study with the predictors based upon traditional learning theories which were explored in RQ2.

6.2 Related Work

Researchers have explored automatically gathered datasets to identify patterns of novice programming behaviour. As this thesis explores the programming behaviour of Java students, this section will briefly discuss some of the previous research into Java programming behaviour.

Jadud [82] proposed an algorithm designed to quantify several aspects of programming behaviour into a performance predictor. This was known as the Error Quotient (EQ), and is an algorithm which scores the programming behaviour of students based upon the frequency of errors encountered, and how successive compilation failures over a session compared in terms of error message, location, and edit location. The scoring scheme is applied to sets of consecutive compilation event pairings, which is then averaged based upon the total number of pairings logged from students. Initially data was collected from 161 students. This sample was then reduced to 96 students by removing those who had not used a public laboratories at least three times for working on their programs. The error quotient was then run on a further reduced sample of 56 students taken from the 2004/05 academic year only. Two regressions showed that the error quotient scores were a significant, but weak predictor of average assignment scores ($R^2 = 11\%$) and final exam scores ($R^2 = 25\%$). No verification regressions using data gathered from the remaining 40 students were presented, although a histogram showed that the error quotient scores were normally distributed for the entire sample. Jadud suggests that the poor quality fits can be attributed to student cheating, missing assignment data, or an incomplete representation of programming behaviour. In this thesis, we will argue that the third reason is correct, but also that there are several flaws associated with the methodology utilized by the error quotient algorithm which limits its ability to accurately reflect the programming behaviour of students.

Tabanao, Rodrigo, and Jadud [182] attempted to determine whether at-risk Java programming students could be identified based upon aspects of their programming behaviour. Data was gathered from 124 students who used the BlueJ IDE over five lab sessions. Based upon their mid-term exam scores, students were classified into three groups: high performing ($n = 25$), average ($n = 76$) and at-risk ($n = 23$). Statistically significant differences were found between the three groups in terms of: the total number of errors encountered (more errors for lower performing), the total counts of a few specific types of the most common errors (more occurrences for lower performing), the time spent between two successive compilations (higher performing spent longer between compilations), the error quotient scores (higher for lower performing). A set of regressions were performed by using these four factors. Although the error quotient emerged as an influential predictor, it could only account for a moderate amount of the variance in performance ($R^2 = 29.71\%$, $p < .01$) and all of the regression models failed to accurately predict the at-risk students.

Rodrigo et al., [155] found that several aspects of programming behaviour were related to performance, including: the number of pairs of successive compilation errors ($r = .33$), the number of pairs with the same edit location ($r = .34$), the number of pairs with the same error location ($r = .30$) and the number of pairs with the same error ($r = .30$). However, a marginally significant regression based upon these factors could only account for a small amount of the variance in performance ($R^2 = 12.00\%$, $p = .09$).

As a follow on experiment from [155], Rodrigo and Barker [154] attempted to detect student frustration based upon aspects of their programming behaviour. These aspects included the average number of successive compilations with the same edit location, average number of successive pairs with the same error, the average time between compilations, and the average number of errors. Although achieving significant results, the strength of the correlation was weak ($r = .31$). Additionally the model failed to detect frustration on a per-lab basis, suggesting that the approach required a substantial amount of data before yielding accurate results.

The main limitation of studies into programming behaviour is that they are often conducted on a small scale nature, and only using the participants at a single institution. This can make it difficult to identify patterns of programming behaviour that are predictive of performance. To address these shortcomings, Utting et al., [187] describe a modification to the BlueJ IDE which enables the collection of data from students throughout the world. The initial results of this project, called Blackbox are presented by Brown et al., [25] which basically verified the already well established research into the most common types of errors which students make. Although impressive in terms of the amount of data which has been captured by the Blackbox project, the types of analysis which can be performed by using this dataset are limited by the lack of an outcome measure. In other words, whilst a great amount of data has been collected, there is no indication as to whether the data is collected from a particularly *poor* or *strong* programming student. This makes applying conventional statistical analysis into the correlations between different quantitative behaviours and performance difficult.

Therefore, in this work we will explore programming behaviour using three independently gathered datasets within the same teaching context. This allows us to correlate different types of behaviours against different student performances. We also note that our experiments and data collection began before the Blackbox dataset was available.

6.3 Research Design

The first purpose of this study was to identify aspects of students' programming behaviours which are predictive of their programming performance. In particular this thesis focusses upon exploring both frequency and time based metrics as predictors.

The second purpose of this study was to determine whether a subset of the predictors could be combined to form a multivariate scoring algorithm, which could be applied for the purpose of automatically predicting programming performances. Such a predictor would have the advantage of not requiring students or instructors to process batches of static tests, and would be capable of describing how desirable, or undesirable a student's programming behaviour is over the duration of a course. The algorithm performs well on limited data, performs consistently on multiple datasets, and explains more of the variance in performance than the predictors based upon traditional learning theories.

This brings us onto the third purpose of this study, which was to compare the predictors based upon programming behaviour to the predictor based upon traditional learning theories which were explored in Chapter 5. Additionally, the performance of the programming behaviour predictors in terms of explanatory power and correlation strengths will be examined over the duration of the entire course.

To explore programming behaviour, three datasets describing the programming activities of three different cohorts of students taking the programming module at our university (2011/12, 2012/13 and 2013/14) were gathered through the use of an on-line protocol added to the BlueJ IDE (further described in Section 6.3.2).

6.3.1 Research Questions

To answer thesis RQ3, the following research sub-questions were answered:

1. How do the datasets gathered in the Durham teaching context compare to those used in previous research?
2. Which aspects of students' programming behaviour are associated with programming performance?
3. Which aspects of programming behaviour can be combined, and scored, into an overall predictive measure of programming performance?
4. How do the measures proposed in this chapter compare in terms of accuracy and explanatory power over the duration of the entire course?
5. How do the measures based on programming behaviour compare to the predictors based upon traditional learning theories explored in Chapter 5 of this thesis?

6.3.2 Criterion Variable

When we performed our original study [200] only two datasets sampled from the 2011/12 and 2012/13 cohorts had been gathered. In this thesis, we have chosen to update our experiments by using an additional dataset which we have recently gathered during the 2013/14 academic year. Due to the differences in the assessment components that were undertaken by each cohort of students (Table 3.1), it was necessary to redefine our criterion variable to take into account only those assessments that were completed by all three cohorts. Assuming that task difficulty and marking were also consistent, this would allow us to explore the relations between programming behaviour and a consistent measure of performance for three cohorts.

The weighting of assessment components was selected based upon the 2013/14 cohort, as they were the only cohort to only undertake the three assessments that have formed the criterion variable exclusively. Therefore, the measure of programming performance in this study is defined as a weighting of the students performance on the: Term 1 bench test (33%), Term 1 project (33%), and Term 2 bench test (33%). This allows the programming behaviour of all participants to be measured at the start, mid-point, and end of the course.

6.3.3 Data Collection Method

The exploration of programming behaviour that was carried out in this study relied upon the automatic collection of source code written by students while programming. In the past, such a method has been referred to as following an on-line protocol [82], as the gathered data did not rely upon a researcher directly observing students, but rather a computer which automatically reported students interactions with it.

We chose to gather this data automatically from the BlueJ IDE. Using the BlueJ extensions framework, a plugin was developed to gather data describing students' programming behaviours in a non-invasive and non-visible way. Listeners were added to capture the following five types of events when they occurred:

- *Compilation*: type of event (success or fail), line (if error), message (if error), class, filepath, source code snapshot.

- *Invocation*: class, filepath, method, invocation status (normal exit, exception exit, user terminated exit), object name, package name, parameters, results.

- *Package*: package name, event type (open or close).

- *Startup* and *End*: event type (start or end).

Compilation events were captured when students compiled either an individual file, or entire project, via the keyboard shortcut or clicking on the compile button in BlueJ. In addition to the event specific data, meta-data was gathered for each event. This included the date, timestamp (to the nearest second), students username, and project name.

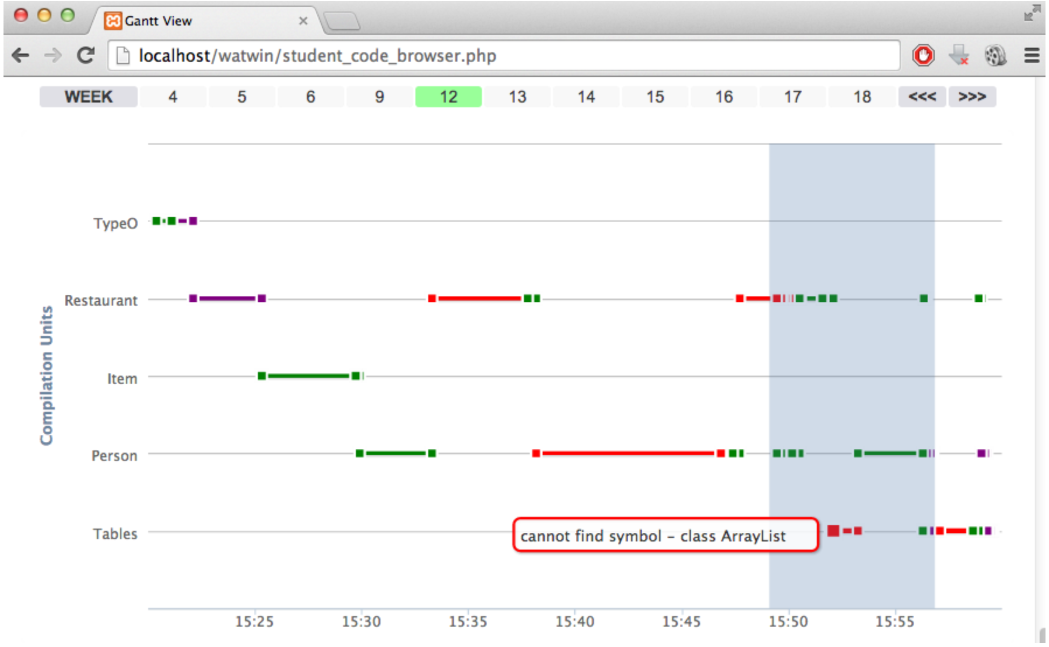
One of the challenges involves selecting the most appropriate level of granularity which data is to be captured at. At the highest level, data could be captured at the project level, such as only capturing events when students choose to build the entire project. However, it is known that novices can compile code quickly [82] and capturing events at this level of granularity would fail to capture a large number of events. At the other end of the extreme is to capture events on the keystroke level. The disadvantage of this approach however is the extreme levels of noise that are inherent to a dataset of this nature. As such, and in line with prior research [82,182] we have chosen to capture data on the snapshot level. Only when students triggered one of the five previously listed types of events was a snapshot logged for analysis.

Before collecting data, students were first required to complete a written faculty approved ethics form which outlined the nature of the study. Participants were free to opt-out or opt-in at any time through the extension menu in BlueJ. If consent was provided, then a student's username was added to the extension which would start logging their programming activities as an encrypted xml file in a student's personal network space. This file described their programming activities over a session, and would be instantly copied over to a departmental file store on the starting or closing of BlueJ. The file would then be decrypted and processed into a departmental database.

6.3.4 Data Preparation and Cleaning

We initially had to determine how the data that we had logged from students could be meaningfully analysed. We first adopted a traditional document analysis approach by developing a Code Browser tool, to visualise the students activities in BlueJ during a session (Figure 6.1). The tool was useful in assisting us to identify aspects of programming behaviour that were undesirable - i.e. failing to move students towards a compilable solution. But, manually screening logs was a time consuming process, and our objective was to derive *automatically distillable* measures of behaviour.

Previous research [82, 182] has considered this problem by analysing a student's programming behaviours in terms of a set of successive compilation event pairings. In this work, we also utilize pairs of events because they are more interesting than events taken singularly. Considering pairs of events for instance allows us to explore whether the student could resolve an error and how their code evolves over time.



cannot find symbol – class ArrayList

Figure 6.1: Screenshot of the Code Browser Tool which we developed to assist us in visualizing students' programming activities logged through our BlueJ extension. The classes the student interacted with during a session are shown on the left hand side, and squares represent events being executed. Red indicates a compilation error, Green indicates a compilation success, and Purple indicates an invocation. The researcher could obtain more detailed information on the events by hovering over them, and could zoom into an area of interest by selecting an area of the graph.

The previous approaches [81] to constructing these pairings however had several shortcomings which are addressed by our data preparation method, outlined as follows:

INPUT: Set of compilation events $\{c_1, c_2, ..., c_n\}$ and invocation events $\{h_1, h_2, ..., h_k\}$ logged from the student's programming session.

1. *Pair Construction.* For each file that a student compiled during a session, order the logged compilation events by timestamp and construct a tuple of successive compilation event pairings $\{\{c_1, c_2\}, \{c_2, c_3\}, ..., \{c_{n-1}, c_n\}\}$. Previous researchers [82,182] constructed compilation pairings based upon the natural order that compilation events occurred during a session. However this fails to take into account the possibility of a student working on multiple files simultaneously, and can lead to an inaccurate representation of their programming behaviour. For instance, suppose we had two compilation events c_1, c_2 from two distinct files. If the event type of c_1 was "fail" and the event type of c_2 was "success", then

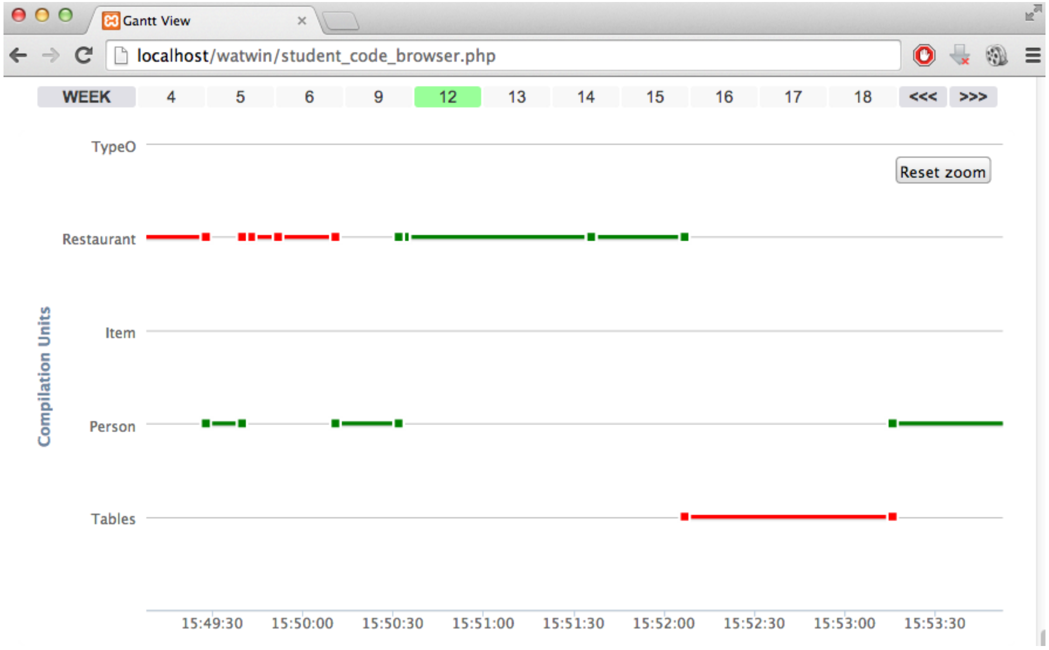


Figure 6.2: Example cases where errors would be inaccurately marked as resolved by previous approaches of building compilation pairings.

the compilation pairing $\{c_1, c_2\}$ would incorrectly convey that the error associated with c_1 was resolved. In reality, the student simply compiled a different file. Three examples of this can be seen in Figure 6.2. If we constructed pairings by using the same technique of previous research, the errors in the Restaurant class at 15:49:30, the Restaurant class at 15:50:00, and the Tables class at 15:53:00 would be incorrectly marked as resolved, due to the student successfully compiling a different file following these compilation errors. Constructing pairings on a per-student, per-session, per-file basis allows us to address this issue.

2. *Pair Pruning.* Remove each event pairing $\{c_x, c_y\}$ where the source code snapshots of c_x and c_y are identical as these pairings consist of identical compilation events, and can artificially inflate the total number of pairings produced. These pairings can arise from either a "compile all files" feature of an IDE, or from the student repeatedly attempting to compile the same file without making any changes. To take into account superficial changes that may have been made by the student between compilations, such as adding comments, modifying layout, or renaming identifiers, rather than performing string matching directly on the code snapshots, we first tokenize the snapshots using the BlueJ lexical analyser. Tokenized snapshots are then compared with comments removed.

3. *Filtering Commented and Deletion Fixes.* Remove each event pairing $\{c_x, c_y\}$ which is a *commented* or *deletion* fix. Although deleting and commenting large blocks of code to resolve errors can yield compilable code, these actions also provide little evidence that a student understands how to correct the underlying error. These actions can also be performed quickly, and therefore the time taken to resolve an error in this manner may not be representative of the time taken to correct the underlying error. Deletion fixes were detected and removed by computing the diff score between the tokenized code snapshots of c_x and c_y. If the count of insertions and changes excluding comments are 0, and the number of deletes are > 5 [199], then we consider the pair to be a deletion fix. This threshold was chosen as it allows genuine fixes such as removing an extra bracket to still be recognised as a valid fix. Commented fixes were detected by extracting the region of code surrounding the error location of c_x, and using a regex expression to determine if the same fragment has only become commented in c_y.

4. *Error Message Generalization.* Generalize the error messages within each compilation pairing $\{c_x, c_y\}$ to remove identifier information. This has the advantage of allowing different classes of error to be profiled, rather than focussing on specific messages. For example, "cannot find symbol - variable pet" is generalised to "cannot find symbol - variable". Using the data gathered from all three cohorts, regex expressions to generalize over 120 different errors were created and applied.

5. *Time Estimation.* Estimate the amount of time that a student has spent working on each compilation pairing $\{c_x, c_y\}$. As we construct compilation pairings on a per-file basis, we need to take into account the possibility of a student spending time working on other files between the compilation events c_x and c_y. Therefore, for each file in a session we first construct a combined sequence of invocation and compilation events ordered by timestamp. For all $\{c_x, c_y\}$, if there exists an event l_i, such that the timestamp of $c_x > l_i > c_y$, then we estimate the time spent on $\{c_x, c_y\}$ as the difference between the timestamps of c_y and l_i. The assumption is that a student has stopped working on the source code of c_x, and has either instead worked on the source code of l_i or spent time invocating the code of l_i.

   For instance, consider the example logged data shown in Table 6.1. These logs would yield the following compilation pairings:

   Square: $\{\{c_1, c_3\}, \{c_3, c_4\}\}$ Pet: $\{\{c_2, c_5\}, \{c_5, c_6\}, \{c_6, c_7\}\}$.

   Estimating the time spent on $\{c_3, c_4\}$ is performed by calculating the difference between the two timestamps the student has no events logged for other files between the two compilations. The same is true for the $\{c_5, c_6\}$ pairing.

Table 6.1: Example Logs of Two Files to Illustrate Pair Construction

Event ID	Event Time	Square.java	Pet.java
c_1	15:31:30	Fail	
c_2	15:32:45		Success
c_3	15:33:15	Fail	
c_4	15:34:00	Success	
c_5	15:35:15		Fail
c_6	15:36:00		Fail
h_1	15:40:30	Invocation	
c_7	15:41:00		Success

When calculating the time the student spent working on the $\{c_6, c_7\}$ pairing, we need to take into account the time the student spent performing the invocation i_1. Therefore, the estimated time the student spent on $\{c_6, c_7\}$ is calculated as 15:41:00 - 15:40:30 = 30 seconds, rather than 15:41:00 - 15:36:00 = 300 seconds.

OUTPUT: Set of compilation pairings for the student's programming session.

The motivation behind the majority of the steps included in our preparation process is intuitive. For example, the removal of duplicate pairings makes sense as they do not add any additional information to a students behavioural profiles. The filtering of commented and deletion fixes is less intuitive. This was primarily motivated by our own direct observations of the students programming activities in the lab - in that we noticed that students (or demonstrators intervening) would sometimes comment out non-compiling code in order to work on a different problem with a compilable program. It is possible that some behaviours have been lost by this preparation process, although the actual percentage of pairings removed by this stage was minimal ($< 5\%$).

In addition to the data cleaning which was performed, we also performed two further acts of data cleaning on the sets of compilation pairings. Firstly, all compilation pairings where the student was estimated to have spent more than 300 seconds between events were removed. The rationale behind this decision was that it was unlikely for the student to have spent 5 minutes working to resolve an error, when previous research has suggested that most errors are resolved in under two minutes [51, 82, 182]. We therefore assumed that students who have spent longer than five minutes to have gone off task and were no longer programming.

Secondly, although our extension was able to capture data whenever the student performed actions in BlueJ, we have restricted the data used in our analysis to only those logs which were gathered during the students regular assigned lab sessions. As can be seen from Table 3.2, the content of the lab sessions was mostly identical for all cohorts, and we hypothesised that students of similar ability in different cohorts would elicit similar programming behaviours when they were required to complete the same tasks. By restricting the data used to those gathered from lab sessions only, it meant that programming behaviour of students would be analysed on the same tasks each week, i.e. from one introductory programming lab session per week, rather than two, three, or four labs depending on the number of elective modules the student studied.

6.3.5 Datasets and Participants

In addition to data cleaning, we also removed students from the samples of each cohort for two reasons. Firstly, any student who failed to complete any of the assessment components which composed the criterion variable were excluded from this study. This decision was taken to ensure that we were exploring programming behaviour using a measure that was consistent to all students (i.e. performance on three assessments).

Secondly, any student who failed to attend a minimum of three lab sessions was excluded from this study. The rationale behind this decision was to follow the recommendation of previous research [82] in that attempting to profile students in cases of data scarcity would be difficult, and that a minimum of three observations would provide a more meaningful analysis. We note that all of these cases arise in a missing completely at random nature (MCAR) [103], as the students who had data logged from less than three sessions had elected to use their own laptops during the lab sessions and logging data on their personal hardware was not possible.

Following the exclusion of 7 students for the above reasons, our final sample consisted of 141 students spread across three cohorts. An overview of the performance of each cohort on the criterion variable is shown in Table 6.2 and a distribution of the marks is shown in Figure 6.3. Performance on the criterion variable was normally distributed (Shapiro-Wilk test, $p > .05$), ranging from a low of 42.0 to a high of 97.7 ($M = 71.6$, $SD = 12.3$). 82 students scored a 1st class (≥70%) or above on the criterion variable, 53 scored a 2nd class (50-69%), and 6 students scored a 3rd (40-49%).

In terms of the data which was used for this study, we made the decision to only use the data which was gathered up to and including the final assessment forming the criterion variable (Term 2 Bench Test). The rationale behind this decision was to exclude any additional data which was gathered after the final assessment had taken place, such that data describing the students future programming behaviours was not used to predict their performance in the past. Due to the nature of lab tasks further data exclusions were applied, which will be discussed in the context of each year.

Table 6.2: Descriptives of the participating students from all cohorts

Cohort	Consent	Missing Criterion	< 3 Labs	Sample n	Mean Mark	SD	1sts (70%)	2nds (50%)	3rds (40%)
2011/12	41	1	3	37	66.4	10.8	16	19	2
2012/13	47	0	2	45	72.2	14.3	28	13	4
2013/14	61	0	2	59	74.4	10.6	38	21	0
All	149	1	7	141	71.2	12.3	82	53	6

2011/12 Dataset

Performance on the criterion variable was normally distributed (Shapiro-Wilk test, $p > .05$), ranging from a low of 45.0 to a high of 82.3 ($M = 66.4$, $SD = 10.8$).

The logging plugin was added to BlueJ at the start of week 4, and data was logged from consenting students until week 19. Data gathered from weeks 7 and 8 while the students were completing the fault injection assignment (Section 3.3.2) was discarded. No data was gathered during week 11 as the collection exam was conducted during the lab. After exclusions, this leaves data from 13 weeks for analysis (4-6, 9-10, 12-19). On average, students attended a total of 11.3 ($SD = 2.2$) lab sessions.

2012/13 Dataset

Performance on the criterion variable was normally distributed (Shapiro-Wilk test, $p > .05$), ranging from a low of 42.0 to a high of 95.3 ($M = 72.2$, $SD = 14.3$).

Data was logged from consenting students from week 2 until week 19. No data was gathered during week 11 due to the basis of the collection exam taking place. Data gathered from week 19 was discarded as the final assessment took place during week 18. After exclusions, this leaves data from 16 weeks for analysis (2-10, 12-18). On average, students attended a total of 12.3 ($SD = 3.4$) lab sessions.

2013/14 Dataset

Performance on the criterion variable was normally distributed (Shapiro-Wilk test, $p > .05$), ranging from a low of 52.0 to a high of 97.7 ($M = 74.4$, $SD = 10.6$).

Data was logged from consenting students from week 2 until week 19. Data gathered from week 19 was discarded as the final assessment took place during week 18. After exclusions, this leaves data from 17 weeks for analysis (2-10, 10-18). On average, students attended a total of 14.9 ($SD = 3.2$) lab sessions.

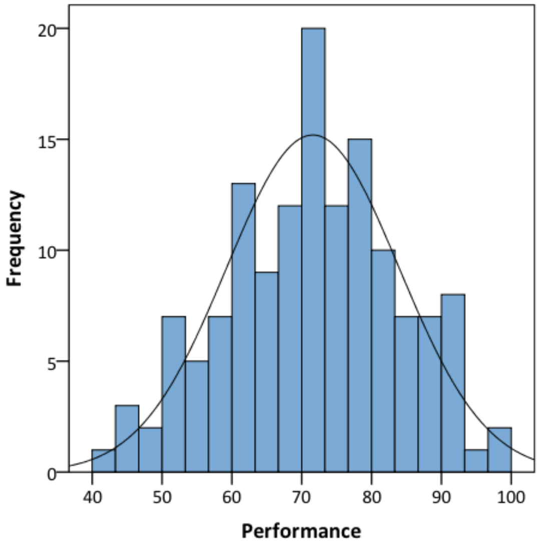


Figure 6.3: Distribution of the performance of all participating students $n = 141$.

Summary

In order to explore whether these were any significant differences in the performance of each cohort, a one-way ANOVA was performed. There were no outliers in any of the groups, as assessed by the inspection of a box plot. Shapiro-Wilk tests confirmed that the pass rates were normally distributed for each year, $p > .05$, and homogeneity of variances was confirmed by Levene's test, $p = .08$. The one-way ANOVA showed that there were statistically significant differences between the performances of each cohort, $F(2, 138) = 5.26$, $p = .01$. Tukey post-hoc analysis revealed that only performance of the 2011 and 2013 cohorts was statistically significant, $p = .01$.

Considering the activities which students were performing whilst their data was being logged, it can be seen from Table 3.2 that data was logged while all three cohorts completed at least 11 identical activities. In other words, the behaviours that are explored in this study are derived from data that was gathered while all three cohorts were completing identical activities at least 70% of the time.

As previously discussed, the assessments also showed a similar degree of consistency for all three cohorts, and students received similar learning materials each year. The high degree of consistency in terms of activities, assessment, and learning materials across all three cohorts would allow us to rule out the possibility of these factors explaining differences in the programming behaviours of each cohort. Additionally, it supports us in concluding whether or not the factors explored in this chapter are derived from programming behaviour, rather than the result of students completing different activities, assessments, or studying different materials.

6.4 Results

6.4.1 Descriptive Statistics of the Datasets

To answer the first research question, descriptive statistics of all three datasets were derived and compared. In a study of this nature, it was important to consider both internal consistency between the datasets, and external consistency of the datasets we gathered with those used in previous research. Descriptive statistics are considered in terms of both the frequency of different types of errors, and the time that students have taken to resolve different types of error. In this study, resolving an error is determined by using the compilation pairings previously constructed. If the generalized error of the first event is different in the second event, or if the second event was a successful compilation, then the error is classified as resolved.

Error Frequency

In total 56,046 compilation events were logged from the 2013/14 cohort, 40,802 compilations from the 2012/13 cohort, and 26,598 compilations from the 2011/12 cohort. The number of compilations which resulted in errors were 26,117 (46.6%) for the 2013/14 cohort, 20,700 (49.2%) for the 2012/13 cohort, and 13,889 (52.2%) for the 2011/12 cohort. These figures are comparable across our datasets, and consistent with previous research on the percentage of compilation errors encountered by students (e.g. 49% errors [51], 41% errors [155] and 50% errors [79]).

Figure 6.4 shows the most commonly occurring errors found in all three datasets. As can be seen from this figure, there was a good degree of internal consistency between the three datasets, with comparable percentages of each error recorded for all three cohorts. The most commonly occurring error in all three datasets was an unknown variable, followed by missing semicolon, and unknown method. Errors relating to unknown identifiers (variables, methods, classes, constructors) could account for approximately one third of the errors in all datasets. As more complex IDE's such as Netbeans provide auto corrective features to avoid these errors, we note that the most commonly occurring errors found in our datasets may be different to those gathered from a more complex IDE. Nevertheless, the most commonly occurring errors showed a good level of consistency in our context, possibly arising from the fact that data was logged whilst all cohorts were completing identical activities during at least 70% of the labs.

Table 6.3 compares the most commonly occurring errors in the three datasets used by this study to previous research. Again, a high degree of consistency was found, with at least 7 of the commonly occuring errors found by this study having been found in the errors most commonly found by previous researchers.

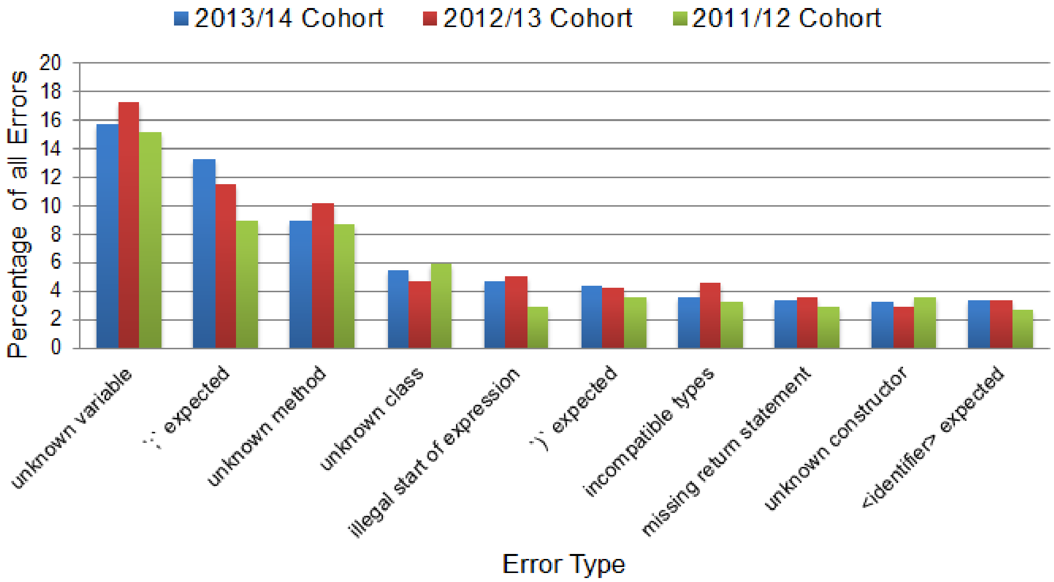


Figure 6.4: Most common errors found in the three datasets used by this study.

Table 6.3: Most common errors found in the three datasets used by this study, compared to most common errors found by previous researchers.

		Previous Researchers				This Study		
	Error	[82]	[81]	[79]	[181]	2011/12	2012/13	2013/14
1	unknown variable	●	●	●	●	●	●	●
2	; expected	●	●	●	●	●	●	●
3	unknown method	●	●	●	●	●	●	●
4	unknown class	●	●	●	.	●	●	●
5	illegal start of expression	●	●	.	●	●	●	●
6	) expected	●	●	●	●	●	●	●
7	incompatible types	●	●	●	●	●	●	●
8	missing return statement	.	●	.	●	●	●	●
9	unknown constructor	.	.	.	.	●	●	●
10	<identifier> expected	●	.	●	●	●	●	.
	Errors in Common	8	8	7	8	10	10	9

Table 6.4: Median (Md) and Interquartile Range (IQR) of the resolve times in seconds.

		2011/12			2012/13			2013/14		
	Error	n	Md	IQR	n	Md	IQR	n	Md	IQR
1	unknown variable	1301	13	20	1849	12	19	2307	14	21
2	‘;‘ expected	933	5	5	1458	5	6	2300	5	4
3	unknown method	570	18	30	892	17	37	1079	17	30
4	unknown class	482	12	15	518	12	16	795	13	15
5	illegal start of expression	231	14	30	430	16	33	523	15	31
6	‘)‘ expected	309	10	15	412	11	13	487	11	16
7	incompatible types	282	14	23	510	16	21	558	16	28
8	missing return statement	265	14	30	450	15	34	497	15	29
9	unknown constructor	216	25	43	267	28	52	328	28	57
10	<identifier> expected	247	12	23	320	14	27	433	13	29

Resolve Times

Table 6.4 shows the median resolve times for each of the most common errors by cohort. We selected the median as a robust measure of central tendency as the distributions of resolve times were left skewed. The resolve times for each of the errors were comparable across all three cohorts. To confirm that the resolve times of each of the errors were consistent across all three cohorts, ten Kruskal-Wallis H tests were performed.

Significant differences were found for two cases. Firstly, there were significant differences in the distributions of resolve times for the <identifier> expected error ($\chi^2(2) = 7.14$, $p = .03$). Post-hoc analysis showed the differences were between the 2011/12 and 2012/13 cohorts. Secondly, significant differences were found in the distributions of resolve times for the ‘;‘ expected error ($\chi^2(2) = 7.14$, $p = .03$). Post-hoc analysis showed the differences were between the 2012/13 cohort and both other cohorts. In the remaining 27 pairwise comparisons, no significant differences were found between the distributions of the resolve times of the most common errors. This suggests a high level of consistency in terms of resolve time across all three datasets.

We next explored whether there were any significant differences between the time students took to resolve different types of errors. Three Kruskal-Wallis H tests showed significant differences in error resolve times for the 2011/12 cohort, ($\chi^2(9) = 944.24$, $p < .001$), 2012/13 cohort, ($\chi^2(9) = 1317.97$, $p < .001$), and 2013/14 cohort, ($\chi^2(9) = 2191.25$, $p < .001$). Post-hoc analysis revealed significant differences in the resolve times of each type of error with at least four other types of error. If we consider the amount of time that a student takes to resolve an error as the measure of the errors difficulty, then this would suggest that different errors are more difficult for students to resolve than others, and these difficulties are consistently observed in all three datasets.

6.4.2 Relations between Different Error Types and Performance

To answer the second research question, based upon the descriptive statistics of our datasets we hypothesised that two types of programming behaviour would be related to student performance - error frequency and error resolve time. We chose to start our exploration into programming behaviour by considering these two metrics.

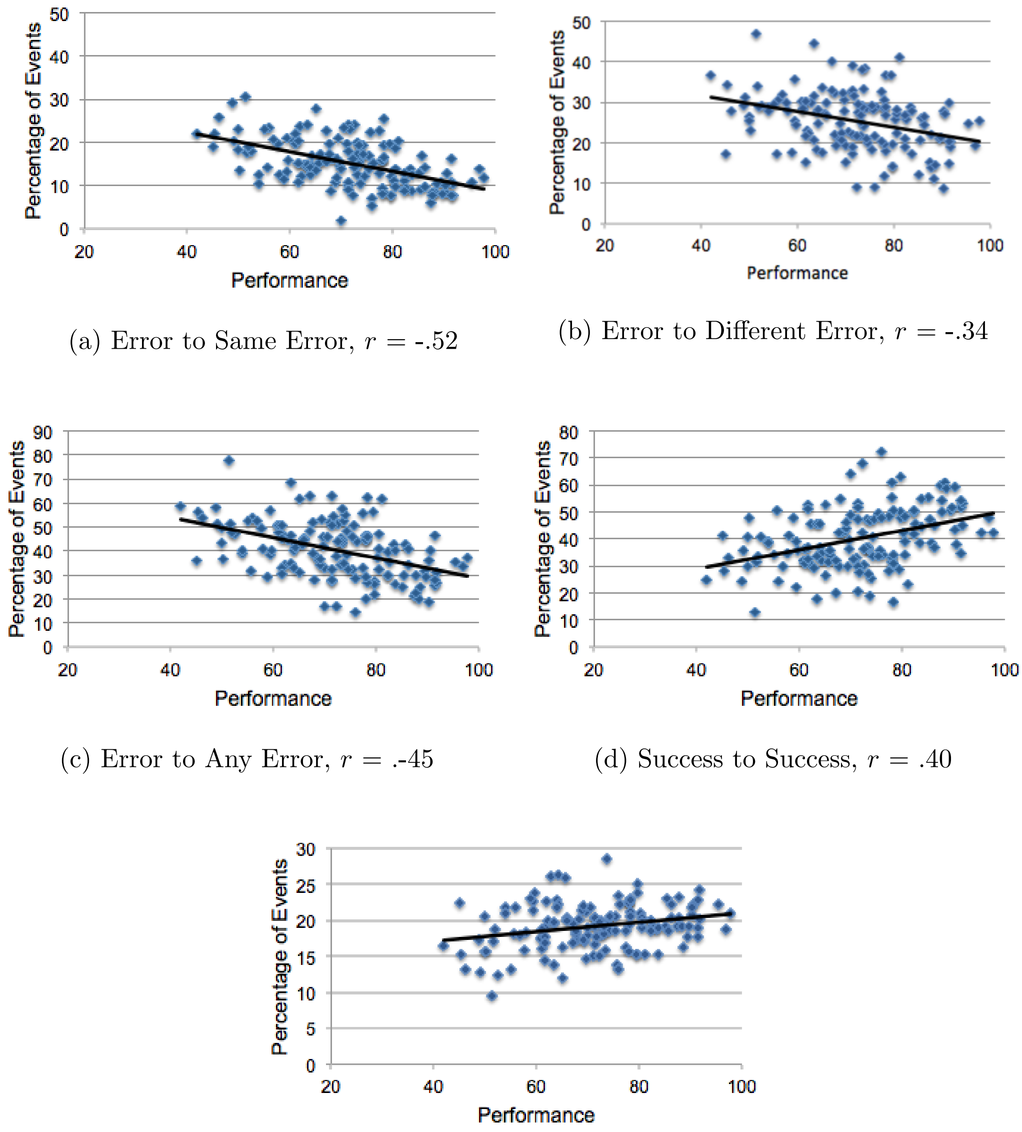
Prior research by [82] suggests weaker programming students are characterised by producing a high number of compilation pairings where both events are errors, and a high number of pairings where both events are the same type of error. [155] also explored these aspects, and found weak-moderate correlations for the total number of compilation pairings where both events are errors, $r = .32$, the total number of pairings with the same error, $r = .30$, and for the average number of errors produced $r = .27$.

The weakness of these results is that they are not standardized against the total number of pairings which have been logged for a student. Unlike previous research, we found no relation between total number of errors with student performance, $r = .07$. However, the students in each of the three cohorts had data logged from a different number of sessions. Also, from the sessions in which data was logged, some students will have attended, and compiled more often than others, and factors such as this need to be taken in to account when exploring programming behaviour.

In our study we also explore the relations between performance and the different types of pairings which are logged for students. However, unlike prior research the counts of these pairings are standardized as a percentage of the total number of pairings logged for each student. This allows factors such as students attending a different number of sessions, or compiling more frequently than others to be taken into account.

We first explored frequency based metrics, based upon the percentage of different types of pairings which were logged for each student. Table 6.5 shows correlations between these metrics and the performance of all three cohorts. Figure 6.5 shows scatter plots of these metrics on the combined sample of all three cohorts ($n = 141$).

In contrast to the traditional predictors which were explored in Chapter 5, an abundance of moderate-strong correlations were found. Using the full sample, $n = 141$, a strong negative correlation was found between performance and the percentage of error to same error pairings r = -.52, $p < .01$, and a moderate negative correlation was found for the percentage of error to any error pairings r = -.45, $p < .01$. The percentage of events in the form of two successive successful compilations was found to moderately positively correlate with performance, $r = .40$, $p <. 01$.



(a) Error to Same Error, r = -.52

(b) Error to Different Error, r = -.34

(c) Error to Any Error, r = .-45

(d) Success to Success, r = .40

(e) Error to Success, r = .26

Figure 6.5: Scatter graphs showing the combined correlations between percentage of different types of pairings with the performance of all three cohorts, n = 141.

Table 6.5: Table showing the correlations (r) of performance with the percentage of different types of pairings logged for students (* $p < .10$, ** $p < .05$, *** $p < .01$)

		2011/12		2012/13		2013/14		Combined	
From	To	$r(35)$		$r(43)$		$r(57)$		$r(139)$	
Error	Same Error	-.56	***	-.52	***	-.51	***	-.52	***
Error	Different Error	-.30	**	-.38	***	-.29	**	-.34	***
Error	Any Error	-.49	***	-.46	***	-.40	***	-.45	***
Error	Success	.22	*	.29	**	.17	*	.26	***
Success	Success	.36	**	.52	***	.36	***	.40	***

Examining the results for all three cohorts, it can be seen that the results are consistent across all three years in terms of correlation strengths, with the exception of the the percentage of events in the form of two successive successful compilations, which showed a slightly higher correlation for the 2012/13 cohort than the other two cohorts. Nevertheless, these results reinforce previous research, and demonstrate that weaker students are associated by making a greater percentage of repeated errors than their peers, and having a smaller percentage of successive successful compilations.

Regression Analysis

Encouraged by the strengths of the correlations, we next explored whether factors based upon the percentages of different types of pairings could be used to predict performance. A series of linear regressions were performed and the results are presented in Table 6.6. As can be seen from this table, in terms of the combined sample, the percentage of pairings which corresponded to two successive identical errors could explain a moderate amount of the variance in performance ($F(1,\ 139) = 51.91$, $p < .01$, $R^2 = 27.19\%$). This was closely followed by the percentage of pairings which corresponded to any two successive errors ($F(1,\ 139) = 35.43$, $p < .01$, $R^2 = 20.31\%$), the percentage of pairings which corresponded to any two successive successful compilations ($F(1,\ 139) = 26.32$, $p < .01$, $R^2 = 15.92\%$), and the percentage of pairings which corresponded to error to a different error ($F(1,\ 139) = 18.01$, $p < .01$, $R^2 = 11.47\%$). These regressions also appeared to yield comparable results on all three individual datasets.

Whilst these regression models could account for a moderate amount of the variance, the error rate was higher than anticipated, and RMSE's of above 10 were found across the board. It is possible that on more data, the RMSE's will lower. Nevertheless, six of these models could account for more than the 21.4% of the variance in performance than the model based upon context independent predictors presented in Chapter 5.

Table 6.6: Table showing the results of the regressions based upon the percentage of different types of event pairings logged for students.

		2011/12		2012/13		2013/14		Combined	
From	To	R^2	RMSE	R^2	RMSE	R^2	RMSE	R^2	RMSE
Error	Same Error	31.03	9.32	26.90	11.80	26.50	9.05	27.19	10.49
Error	Different Error	8.93	10.42	14.33	12.82	8.15	10.11	11.47	11.57
Error	Any Error	24.12	10.52	21.36	11.39	16.34	10.39	20.31	10.98
Error	Success	4.99	9.77	8.59	12.22	3.05	9.65	6.73	11.18
Success	Success	12.80	10.09	27.51	13.08	12.86	9.73	15.92	11.28

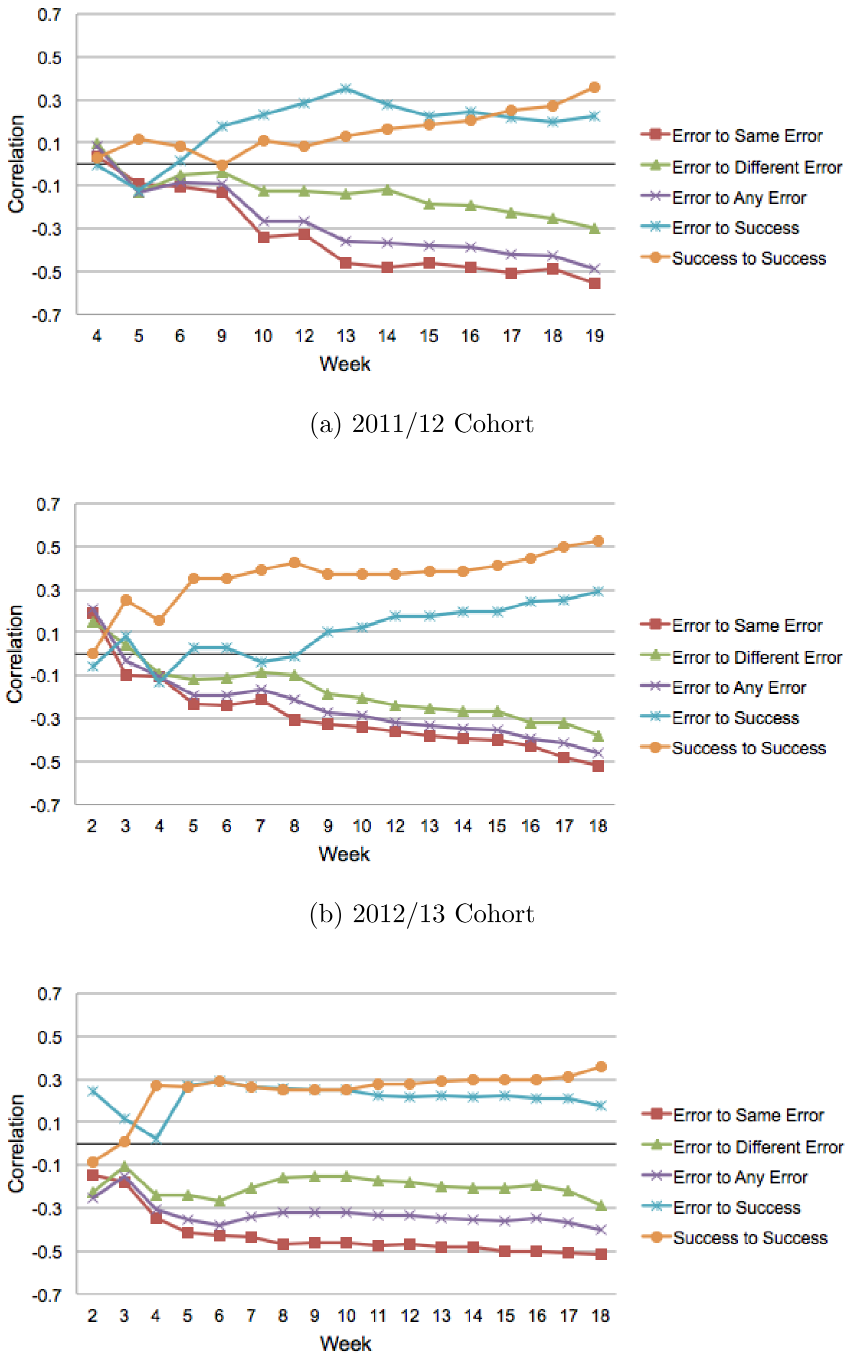
Performance Over Time

To answer the fourth research question, we next considered how the strengths of the correlations and regressions changed over time, and whether metrics based upon pairing frequency could be used to predict student performance using limited data. As data was logged from a number of different labs for each cohort, results are presented on a per-cohort basis. Figure 6.6 displays the strengths of the correlations between percentage of different pairings and performance, using all available data up to and including the current session (i.e. the predictions made for week six are made using all data gathered during the previous six weeks).

For the 2011/12 cohort, none of the correlations reached moderate levels until 5 labs of data was logged. After the 5th lab, the percentage of error to same error pairings began to moderately correlate with performance, r = -.34, and steadily rose to within its peak correlation after a further 2 labs. The percentage of error to any error pairings took 7 labs to reach a moderate correlation of r = -.36. The percentage of error to success pairings peaked at this time, r = .35, but declined over the remainder of the course. The remaining metrics did not peak until the final 2 labs.

For the 2012/13 cohort, the percentage of success to success pairings reached a moderate correlation after 4 labs, r = .35, and continued to slowly rise over the remainder of the course. None of the remaining four factors reached a moderate correlation until after 7 labs. The percentage of error to same error pairings reached a moderate correlation after 7 labs, and the percentage of error to any error pairings reached a moderate correlation after 8 labs. As with the 2011/12 cohort, the percentage of error to success, and percentage of error to different errors rose slower over the course, and did not peak until the final two labs of the course.

For the 2013/14 cohort, the correlations rose more rapidly than the other two cohorts. All of the correlations reached moderate levels after 4 labs, and showed no substantial increases in the strength of the correlations over the duration of the course.



(a) 2011/12 Cohort

(b) 2012/13 Cohort

(c) 2013/14 Cohort

Figure 6.6: Correlations between the percentage of different types of error pairings and performance over the duration of all three courses.

6.4.3 Resolve Times as a Predictor

One aspect of programming behaviour which previous research has not considered as a predictor is the amount of time that students take to resolve errors. Research by [51] showed that the resolve times of different types of error can vary based upon student ability, but, they did not use this variable as a predictor of performance.

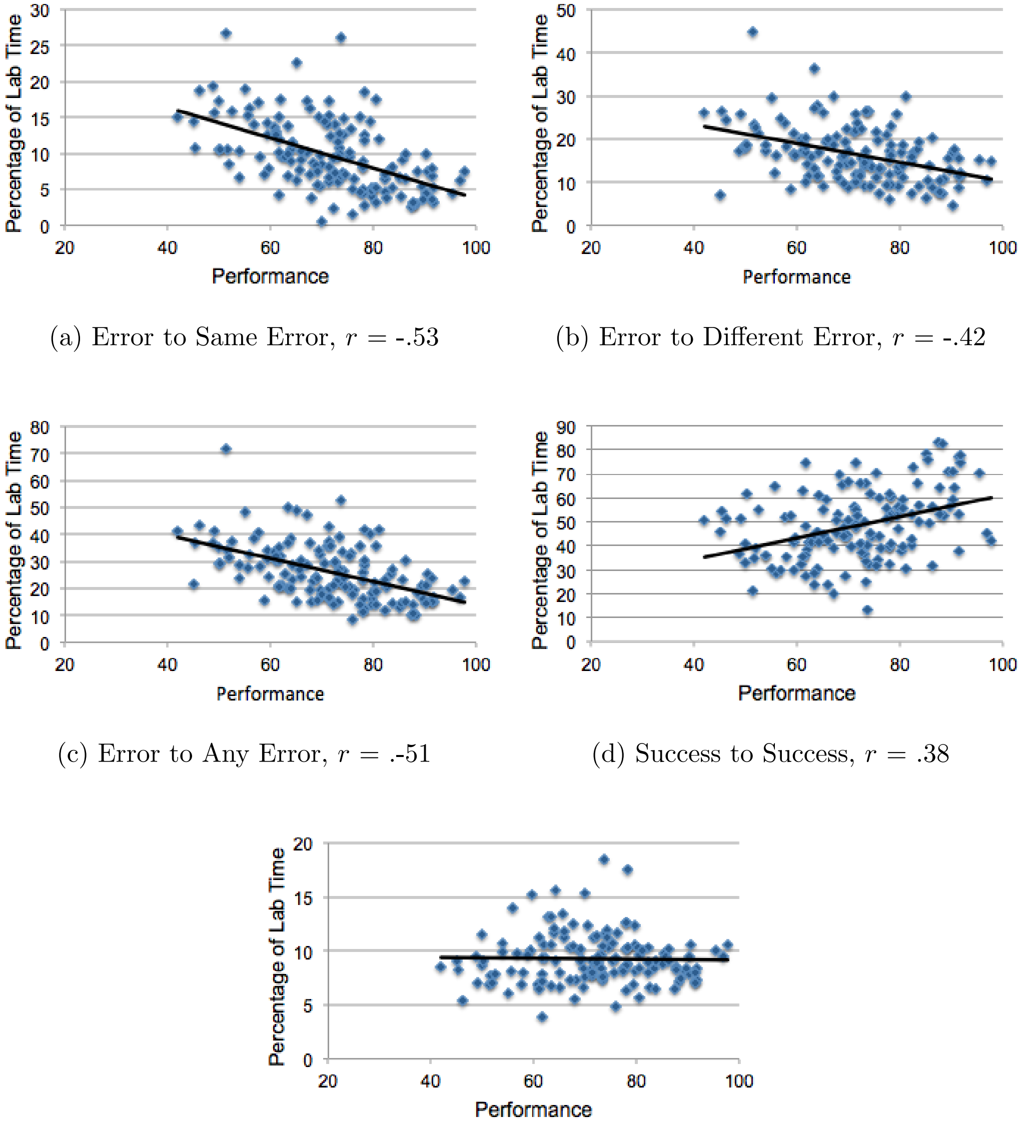
We next explored the percentage of lab time which students spent working on different types of pairings. As with the frequency based metrics, we elected to use percentages as a means to standardize the data logged for students against the number of sessions they attended. This was calculated by summing the total time students spent working on all pairings to estimate total lab time, and then calculating the percentage of time spent working on different types of pairings as a percentage of this total. Table 6.7 shows correlations between these metrics and the performance of all three cohorts. Figure 6.7 shows scatter plots of these metrics on the combined sample of all three cohorts.

Similar to the results on the relationships between the percentage of different types of pairings and performance, a number of strong correlations were found for the percentage of time that students spent working on different pairings. The percentage of lab time that students spent on error to same error pairings was found to strongly negatively correlate with performance $r(139)$ = -.53, $p < .01$. This correlation was consistent across all three cohorts, yielding correlations for the three datasets of $r(35)$ = -.59, $r(43)$ = -.56, and $r(57)$ = -.48. Percentage of lab time spent working on error to any error pairings was found to moderately negatively correlate with performance, $r(139)$ = -.42, $p < .01$. The percentage of lab time spent on two successive successful compilations was found to moderately positively correlate with performance, $r(139) = .38$, $p < .01$. This implies that not only are stronger students characterised by making a greater percentage of successive errors than their peers, but they also are characterised by spending a greater percentage of their lab time working on these errors.

Regression Analysis

We also considered performance of the time based metrics over time. A set of linear regressions were performed and the results are presented in Table 6.8.

As can be seen from this table, in terms of the combined sample, the percentage of lab time spent working on error to same error pairings could explain a moderate amount of the variance in performance ($F(1,\ 139) = 56.65$, $p < .01$, $R^2 = 27.85\%$). This was closely followed by the percentage of lab time spent working on error to



(a) Error to Same Error, r = -.53

(b) Error to Different Error, r = -.42

(c) Error to Any Error, r = .-51

(d) Success to Success, r = .38

(e) Error to Success, r = -.02

Figure 6.7: Scatter graphs showing the combined correlations between the percentage of lab time students spent working on different types of pairings with the performance of all three cohorts, $n = 141$.

Table 6.7: Table showing the correlations (r) of performance with the percentage of lab time students spent working on different types of event pairings. (* $p < .10$, ** $p < .05$, *** $p < .01$)

		2011/12		2012/13		2013/14		Combined	
From	To	$r(35)$		$r(43)$		$r(57)$		$r(139)$	
Error	Same Error	-.59	***	-.56	***	-.48	***	-.53	***
Error	Different Error	-.30	**	-.52	***	-.40	***	-.42	***
Error	Any Error	-.51	***	-.57	***	-.47	***	-.51	***
Error	Success	.10		.06		-.10		-.02	
Success	Success	.26	*	.43	***	.41	***	.38	***

Table 6.8: Table showing the results of the regressions based upon the percentage lab time students spent working on different types of pairings.

		2011/12		2012/13		2013/14		Combined	
From	To	R^2	RMSE	R^2	RMSE	R^2	RMSE	R^2	RMSE
Error	Same Error	35.21	9.11	31.08	11.56	22.89	9.27	27.85	10.45
Error	Different Error	8.93	10.36	26.99	11.59	16.04	9.67	17.50	11.17
Error	Any Error	26.12	27.66	31.94	10.65	21.64	28.45	26.19	20.23
Error	Success	0.94	9.53	0.35	11.17	0.96	9.34	0.04	12.29
Success	Success	6.94	10.60	18.69	14.11	16.93	10.50	14.67	11.36

any error pairings ($F(1,\ 139) = 49.32$, $p < .01$, $R^2 = 26.19\%$), percentage of lab time spent working on error to different error pairings ($F(1,\ 139) = 29.49$, $p < .01$, $R^2 = 17.50\%$), and percentage of lab time spent working on two successive successful compilations ($F(1,\ 139) = 23.88$, $p < .01$, $R^2 = 14.67\%$). No relations were found between performance and the percentage of lab time students spent working on error to success pairings ($F(1,\ 139) = .04$, $p = .81$, $R^2 = .07\%$).

Whilst these regressions could account for a moderate amount of the variance, the regression results appeared to vary based upon the cohort. For instance, the percentage of lab time spent working on two successive errors could explain 35.21% of the variance performance of the 2011/12 cohort, but only 22.89% of the variance in the performance of the 2013/14 cohort. Nevertheless, nine of these models could account for more than the 21.4% of the variance in performance than the model based upon traditional predictors presented in Chapter 5. Additionally three metrics - percentage of lab time spent working on error to any error pairings, percentage of lab time spent working on an error to different error pairings, and percentage lab time spent working on error to same error pairings, could explain a greater amount of variance than their frequency based counterparts. This may suggest that percentage of lab time spent working on different pairings is a stronger predictor than those based upon frequency of events.

Performance Over Time

To answer the fourth research question, Figure 6.8 displays the strengths of the correlations between the percentage of lab time spent working on different types of pairings and performance, for all the labs from which data was logged.

For the 2011/12 cohort, none of the correlations reached moderate levels until 5 labs of data was logged. After the 5th lab, the percentage of lab time spent working on error to same error pairings began to moderately correlate with performance, yielding r = -.33, and steadily rose to within its peak correlation after a further 2 labs. The percentage of lab time spent working on error to any error pairings also took 5 labs to reach a moderate correlation of r = -.34, and continued to rise over the duration of the course. The remaining metrics did not peak until the final 2 labs, although the percentage of lab time spent working on error to different error pairings reached a correlation of r = -.27 after 6 labs.

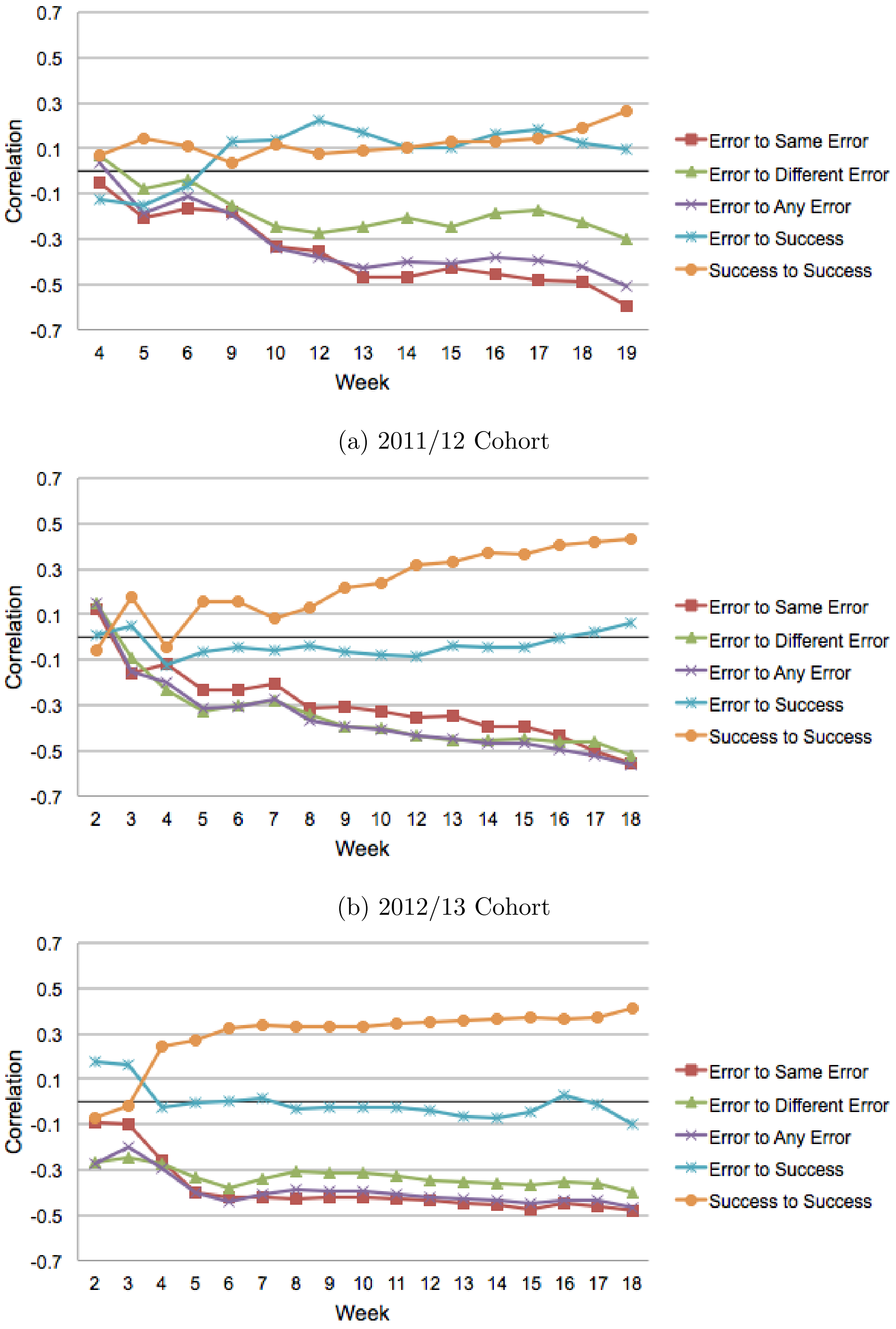
For the 2012/13 cohort, the percentage of lab time spent working on error to same error pairings took 7 labs to reach a moderate correlation, r = -.35, whereas it only took 4 labs for moderate correlations to be found for the percentage of lab time spent working on error to any error, r = -.33, and error to different error pairings, r = -.32. The percentage of lab time spent working on two successive successful compilations did not begin to moderately correlate with performance until the second term (9 labs).

For the 2013/14 cohort, the percentage of lab time spent working on error to any error began to moderately correlate with performance after 3 labs, r = -.30, error to different error after 4 labs, r = -.34, error to same error after 4 labs, r = -.40, and any two successive successful compilations after 5 labs r = -.32.

Comparison of Frequency and Time Based Metrics

Both the frequency and time based metrics correlated moderately with the performance of students across all cohorts, which raises the question as to whether one dimension is more important for predicting programming performance than the other.

Examining the correlations of all three cohorts in Tables 6.5 and Table 6.7, both time and frequency based metrics performed similarly for all five types of pairings. But, the metrics based upon time reached a moderate correlation earlier in the course, and yielded stronger correlations than the frequency based metrics on 13 results (65%). This suggests that time is a more important factor, when the metrics are considered singularly. In the remainder of this chapter, a hybrid algorithm which combines several of these metrics into a single predictor of performance will be presented.



(a) 2011/12 Cohort

(b) 2012/13 Cohort

(c) 2013/14 Cohort

Figure 6.8: Correlations between the percentage of time spent working on different types of error pairings by students and performance, over the duration of all three courses.

6.4.4 Developing a Predictive Algorithm

We had identified that both the frequency of different types of pairings (e.g. repeated errors), and the time spent working on different types of pairings (e.g. resolving errors) were moderately to strongly associated with programming performance.

To answer the third research question, we next decided to explore whether a hybrid model based upon several metrics could account for a greater proportion of the variance, than when the same metrics were considered on a singular basis. But, different forms of programming behaviour can be considered to be more *undesirable* than others. For example, constantly repeating the same error could be considered to be worse than making and resolving an error, as repeating errors shows no learning on the students part. Also, if students are spending a large percentage of their lab time resolving errors, then it provides evidence that they are struggling to overcome the syntax barrier.

If several factors could be algorithmically combined and scored to yield a single measure of how undesirable students' programming behaviours are, then it may provide a more effective means of predicting performance than considering the metrics on a singular basis. In the remainder of this section we present an algorithm which we have developed which is designed to fulfil this requirement by applying a scoring scheme to penalize students based upon different aspects of their programming behaviour.

Component Selection

In the first instance, we had to decide which aspects of *undesirable* programming behaviour to incorporate into our scoring algorithm. We were interested in the Error Quotient [82] which proposed a scoring algorithm based upon the frequency of different types of events. Based upon this research, and in conjunction with our own results on the relations between different types of pairings and performance, we selected the following frequency based components for our algorithm: repeated full error message, repeated generalised error, and repeated error location. These factors will be scored by applying a set penalty for their occurrence.

Our research in the previous section demonstrated that metrics based upon the time spent working on different types of pairings yielded stronger correlations than the frequency based metrics. We were keen to exploit these findings as we believed that including a scoring dimension based upon resolve time would yield a considerably stronger algorithm than one based on the frequency of events alone.

However, two factors needed to be taken into consideration when determining how to penalise students. Firstly, certain errors are more difficult for students to resolve than others. Significant differences in resolve times were found for the most commonly occurring errors (Table 6.4), and it is important to consider the difficulty to resolve an error when penalizing students based upon their resolve time. Secondly, students of different abilities can take longer to resolve different types of errors than others [51]. It follows that high performing students can take a considerable amount of time to resolve relatively straightforward errors, and in some cases, low performing students can resolve certain types of error quicker than high performing students. If we were to simply apply an arbitrary penalty based upon whether the student has resolved an error in an arbitrary good (e.g. ≤ 4 seconds) or arbitrary bad (≥ 7 seconds) amount of time, it would fail to take into account both the students ability to resolve an error, and the relative difficulty to resolve different types of errors.

To take these factors into account in our algorithm we included a scoring dimension based upon error resolve times, where students are relatively penalized based upon how their resolve times for different types of error compare to the resolve times of their peers. As time distributions are generally skewed, we penalize the students resolve times based upon the use of robust median-based statistics [208].

For each type of error, a distribution of resolve times is constructed by using all the available pairings data for the student and their peers. This distribution is used to define what resolve times are to be considered "normal" for a particular type of error, to which the students resolve time will be compared against. Outlying resolve times are removed by applying the robust $2MAD_e$ (Median Absolute Deviation) approach [208].

A penalty is then selected based upon how the students resolve time compares to the inter quartile range of the distribution. If the students resolve time is within the lowest quartile of the distribution, then a student resolved an error faster than their peers - so we select a low penalty. If a student's resolve time is in the upper quartile, then they resolved an error slower than their peers the highest penalty is selected. Otherwise, a student's resolve time is within the inter quartile range, so a medium penalty is selected. The advantage of scoring students in this manner is that we can implicitly take into account the relative difficulty of resolving different types of error, and judge the students ability by comparing their resolve times to their peers. Additionally it allows the bounds for penalties to be derived from the actual behaviours of the students, rather than specifying an arbitrary penalty for different types of errors which may not hold across different teaching contexts.

For example, consider the penalty bounds of the most common errors displayed in Table 6.9. Suppose that a student from the 2011/12 cohort took 4 seconds to resolve an unknown variable error. Compared to their peers, this time ranks in the lowest quartile, and a low penalty would be selected. However, if the student took 30 seconds, then this time ranks in the highest quartile, and a higher penalty would be selected.

Table 6.9: Low (L) and High (H) penalty bounds (seconds) of the most common errors. Medium penalties are applied for times between the low and high bounds.

		2011/12		2012/13		2013/14	
	Error	L	H	L	H	L	H
1	unknown variable	8	21	7	18	7	22
2	';' expected	4	7	4	7	4	6
3	unknown method	9	26	8	26	9	31
4	unknown class	7	17	7	19	7	19
5	illegal start of expression	7	22	8	27	7	21
6	')' expected	6	16	7	17	5	13
7	incompatible types	9	25	9	23	8	19
8	missing return statement	8	25	9	28	8	21
9	unknown constructor	12	43	12	52	12	47
10	<identifier> expected	7	19	8	25	6	18

Deriving Fair Penalties

Common to machine learning techniques we first classified our three datasets into the roles of training (2013/14 dataset), validation (2012/13 dataset), and testing (2011/12 dataset), based upon their sample sizes. To derive appropriate penalties, we used a genetic algorithm. The objective was to identify a set of penalties which could maximise the variance explained on the training and validation datasets, by building a regression model based upon the scoring of students' programming behaviours and their course performance. The specific conditions of the algorithm are described as follows:

1. *Population.* The initial population consisted of 250 randomly generated arrays of six integers (representing one penalty for each component in the algorithm).

2. *Crossover.* Single point crossover. One crossover index was randomly selected. The array from index 0 to the crossover index was copied from one parent, the remaining penalties were copied from the second parent to form the child.

3. *Mutation.* Chromosome switching. Two indexes were randomly selected, and the penalties from these indexes switched if mutation occurred (10% chance).

4. *Fitness.* Fitness was defined as the variance explained by applying the penalties to the training dataset + the variance explained on the validation dataset (i.e. ${R^2}_{\text{training}} + {R^2}_{\text{verification}}$).

Our aim was to identify a set of penalties which could maximise the variance explained on the validation dataset, but if the same penalties could also explain a substantial proportion of the variance on the previously unseen test dataset, then it may suggest the generalization of our algorithm to other teaching contexts.

Multiple runs of the genetic algorithm were conducted, generally for 300 generations each time. After this number of generations, we noticed no substantial increases in the fitness of the population. A number of optimization techniques were applied. We initially experimented by ensuring at least 25% of the population met a minimum level of fitness ($R^2_{\text{training}} + R^2_{\text{verification}} \geq 40$) which ensured there were fit parents available for the first generation. At the end of each generation, the top 10% of parents from the previous generation were also retained and seeded into the child population. This combined population was sorted by fitness, and the instances with the worst fitness dropped until the new population was the same size as the initial population.

After our initial runs of the genetic algorithm, several possible combinations of penalties were identified which all yielded comparable fitness levels on both training and validation datasets. To select the penalties for use in our algorithm, we explored how each of the fittest penalty combinations performed at an early stage in the course when limited data was available. The final penalties selected for the algorithm could explain the most amount of variance in the training and validation datasets after 5 labs of data had been gathered (the typical amount of labs that students completed before the first assessment took place).

Differences to our Original Watwin Algorithm

The algorithm which is presented in this thesis improves upon our previously published version [200] in several ways. The reason we have elected to revise aspects of our algorithm was to address some of the limitations which we identified, and to take advantage of the availability of the larger dataset gathered from the 2013/14 cohort.

The components included in the algorithm (Figure 6.9) remain identical to our original version. The first difference concerns the penalties which are applied. Originally, penalties were selected through the use of a brute force search of a restricted parameter space using a single dataset. This approach however has the limitation of over fitting penalties to a single dataset. By using a genetic algorithm to search for penalties which hold on two independent datasets, we reduce the likelihood of this limitation occurring.

Secondly, in the original algorithm, the resolve time penalty bounds were based upon the use of a mean resolve time and using standard deviations to define the bounds. But this approach was sensitive to deviations from normality, and using percentiles to specify the penalty bounds allows us to address this shortcoming. For example, on smaller datasets where the standard deviation is larger than the mean resolve time, the lower penalty bound could be a negative time, which is impossible. By changing our approach to use robust median based statistics, rather than mean based, it increases the likelihood that our approach will not be effected by this shortcoming.

Robust Relative: A Predictive Algorithm of Programming Performance

Combining the work of the previous sections, our revised algorithm (named *Robust Relative*) for predicting performance in a programming course is described as follows:

INPUT: Set of compilation events $\{c_1, c_2, ..., c_n\}$ and invocation events $\{h_1, h_2, ..., h_k\}$ logged from the student's programming session.

1. Prepare a set of pairings by applying the procedure presented in Section 6.3.4. Pairings where the first event is a compilation success should be removed.
2. Quantify Programming Behaviour
    - *Score* each compilation pairing by using scoring algorithm (Figure 6.9).
    - *Normalize* each score by dividing by 16 (the maximum possible score).
    - *Average* the normalized scores of all pairings.

OUTPUT: The mean average of all pairing scores, that is taken as the student's *Robust Relative* score for the session. A score of 0 indicates that the student encountered no errors over a session. A score of 1 indicates that every compilation ended in an error, and that the student spent longer than their peers between different compilation events. The closer the score to 0, the stronger the students ability.

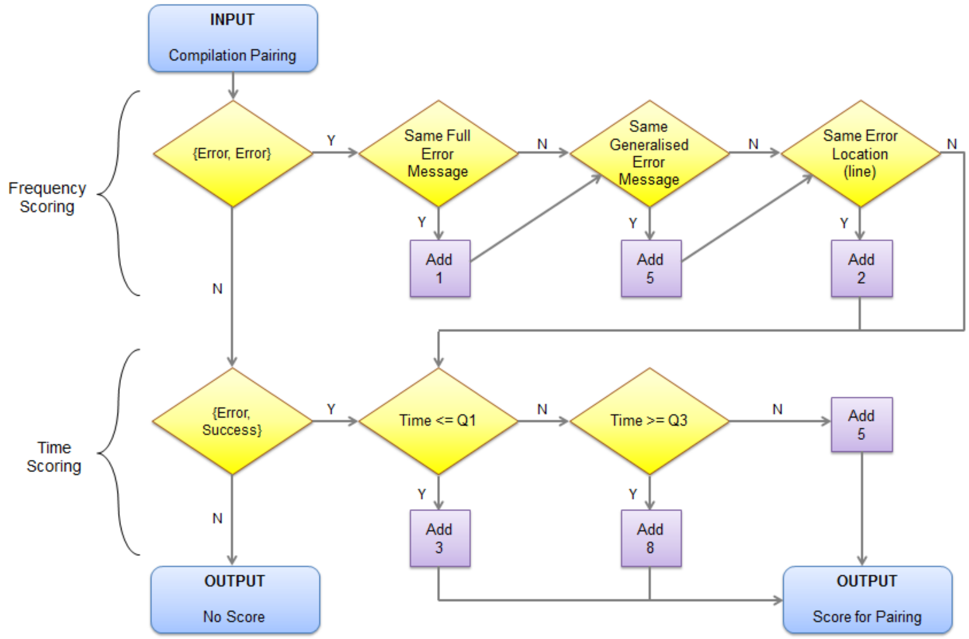


Figure 6.9: The scoring algorithm used by Robust Relative algorithm, showing the penalties which were identified through applying the genetic training algorithm.

Table 6.10: Table showing the results of the regressions based upon the Robust Relative and Error Quotient algorithms.

	2011/12		2012/13		2013/14		Combined	
Algorithm	R^2	RMSE	R^2	RMSE	R^2	RMSE	R^2	RMSE
Robust Relative	43.65	8.23	44.17	10.83	42.19	8.16	41.21	9.73
Error Quotient	24.49	9.53	25.43	12.52	23.59	9.38	25.10	10.80
Difference R^2	19.16		18.74		18.60		16.11	
Improvement (%)	78.23		73.69		78.82		64.18	

Overall Results

To evaluate the predictive power of our algorithm, linear regressions were performed. As the penalties used in the scoring algorithm were derived by using the 2013/14 training and 2012/13 verification datasets only, the strength of our algorithm can be judged based upon its performance on the independent 2011/12 test dataset.

Overall results are presented in Table 6.10. As can be seen from this table, the scoring of programming behaviour by the Robust Relative algorithm could account for a large proportion of the variance in the performance of all three cohorts.

For the 2013/14 dataset used for algorithm training, we found that a linear regression could explain a large amount of the variance ($F(1,\ 57) = 41.60$, $p < .01$, $R^2 = 42.19\%$). The final RMSE of the model was 8.16. As this dataset was used for the purposes of training and identifying appropriate penalties, strong results are to be expected.

For the 2012/13 dataset used for algorithm verification, we found that a linear regression could explain a large amount of the variance ($F(1,\ 43) = 33.74$, $p < .01$, $R^2 = 44.17\%$). The final RMSE of the model was 10.83. Finding consistent results on a second dataset is encouraging, suggesting the further generalisability of the algorithm.

Using the 2011/12 dataset for algorithm testing, we found that a linear regression could again explain a large amount of the variance ($F(1,\ 35) = 27.25$, $p < .01$, $R^2 = 43.65\%$). The final RMSE of the model was 8.23. The comparability of the results on an independent dataset suggests the generalizability of our algorithm to similar teaching contexts. This notion is strengthened by the consistency of our datasets both internally, and externally against those gathered by previous researchers, suggesting that certain elements of programming behaviour encapsulated by our algorithm may consistently hold across different teaching contexts.

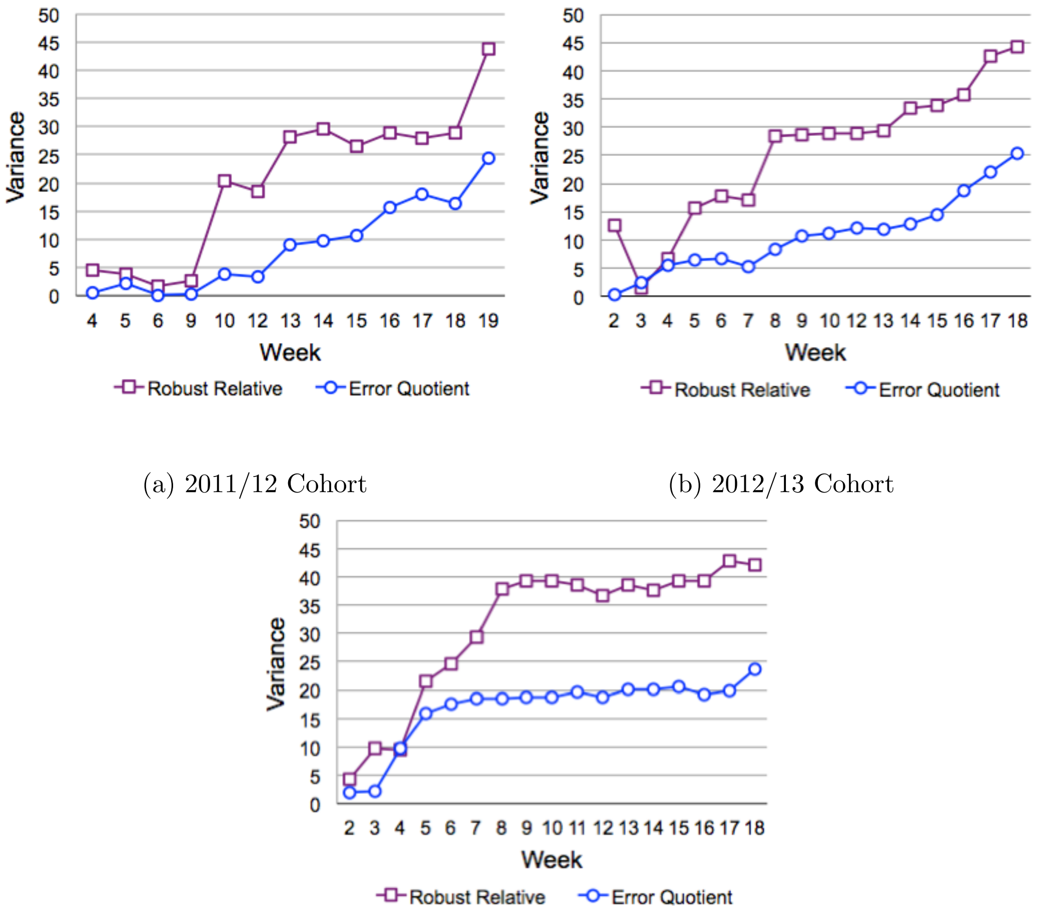
We also ran the Error Quotient algorithm on our datasets. Consistent with previous research, we found the Error Quotient to be a weak-moderate predictor of performance [82, 155]. The explanatory power ranged from a low of 23.59% to a high of 25.43%. We note that similarly to our Robust Relative algorithm, the Error Quotient yielded comparable results across all three datasets. However, our Robust Relative algorithm was able to explain an additional 19.16% of the variance in the performance of the 2011/12 cohort, an additional 18.74% of the variance for the 2012/13 cohort, and an additional 18.60% of the variance for the 2013/14 cohort. This suggests that incorporating a dimension based upon resolve time can account for substantially more of the variance in performance, than an algorithm that is based upon error frequency alone.

Results Over Time

It is important to consider how our algorithm performs over the duration of a course, as predicting performance at the end of a course leaves little room for an instructor to provide an intervention to prevent a struggling student from failing.

Results are presented in Figure 6.10. As can be seen from this figure, both of the algorithms appear to be data-driven, explaining less of the variance in performance across all three cohorts at the early stages in the course when data is scarce.

For the 2013/14 cohort, the variance explained by Robust Relative rose sharply, accounting for approximately 40% of the variance after six labs. The Error Quotient rose sharply over the first four labs, however showed no substantial improvements in explanatory power over the remainder of the course, remaining around 20% until the final lab. For the 2012/13 cohort, the variance explained by Robust Relative rose more steadily over the entire duration of the course, and could account for 30% of the variance by the sixth lab. The Error Quotient rose at a comparable rate to Robust Relative, however, consistently could only account for approximately 20% less of the variance over the duration of the course. For the 2011/12 cohort, both algorithms struggled to explain any of the variance over the first four labs. During the fifth lab, Robust Relative jumped to 20% explanatory power which increased to 30% after another two labs. The algorithm held steadily around the 30% mark for a further six labs, only rising to above 40% in the final labs. The Error Quotient struggled to explain any of the variance until the seventh lab, when it could account for 10% of the variance in performance. The explanatory power then steadily increased over the remainder of the course. Considering that the 2011/12 dataset consists of fewer logged lab sessions than the other two datasets, the increases in explanatory power are comparable to the other two datasets after a similar number of labs had been conducted.



(a) 2011/12 Cohort

(b) 2012/13 Cohort

(c) 2013/14 Cohort

Figure 6.10: Variance explained by the Robust Relative and Error Quotient algorithms over the duration of the course.

Table 6.11: Results of running the Error Quotient using the Robust Relative Penalties.

	2011/12	2012/13	2013/14
Algorithm	R^2	R^2	R^2
Robust Relative	43.65	44.17	42.19
Error Quotient	24.49	25.43	23.59
Error Quotient using Relative Penalties	26.32	25.31	25.38
Improvement	1.83	-0.12	1.79
Difference in R^2 to Robust Relative	17.33	18.86	16.81

6.4.5 Comparative Analysis

Finally, we compared the variance explained by the algorithms to the frequency and time based metrics explored earlier in this chapter. We also answer the fifth research question by comparing the correlation strengths of the metrics explored in this chapter to the predictors based upon traditional learning theories explored in Chapter 5.

Comparing Robust Relative to the Error Quotient

The main methodological flaw of the Error Quotient is that the pairings scored by the algorithm are constructed using the natural order that compilation events occur during a session. As previously discussed, this approach is flawed as it assumes that either students only work on a single source file, or work on multiple files in a linear manner. But, we have found that students do not work like this. Applying the pair construction process used by the Error Quotient, we found that 7,927 (35.4%) pairings for the 2011/12 dataset, 10,937 (29.9%) pairings for 2012/13 dataset, and 16,322 (32.7%) pairings for the 2013/14 dataset, contained compilation events from two different files.

This has serious implications for the validity of the approach. For instance, when examining pairings corresponding to fully resolving an error (i.e. having event types in the form {error, success}) we found that in 742 (20.3%) pairings for 2011/12 dataset, 911 (16.5%) pairings for the 2013/13 dataset, and 1057 (15.5%) pairings for 2013/14 dataset, the events were actually from two different files. All of these cases were scored as if a student had resolved an error, whereas in reality, they had simply compiled a different file. As a student's Error Quotient is averaged by using the sum of every pair from a session, having a large amount of possibly invalid 0 scoring pairings can lower their Error Quotient, inaccurately reflecting their performance.

The Error Quotient also removes all pairings where the source code between events is unchanged. but by constructing pairings on a per-session basis, it is possible for the source code similarity to be invalidly calculated by using two different files. Additionally there are no measures taken to check superficial changes made to source code can be incorrectly flagged as semantic changes. There are also the fundamental differences between the two algorithms to consider, such as the inclusion of metrics based upon the time spent between different events. The Error Quotient also fails to take the difficulty of resolving different error into account, and scores all errors equally.

The Robust Relative algorithm could account for a larger percentage of the variance in performance in the early stages of the course when data was scarce, and by the end of the course the algorithm could explain almost double the variance of the Error Quotient. An interesting question arises as to whether this improvement is the result of incorporating a dimension based upon error resolve times or the result of applying different penalties which may fit our teaching context better. To explore this, we reran the Error Quotient using the penalties used in Robust Relative. As can be seen from the results presented in Table 6.11, Robust Relative still comfortably outperformed the Error Quotient. The additional source of variation concerns the manner in which the compilation pairings were constructed. However, in previous work we demonstrated that this also yields no significant improvement in the Error Quotients performance [200]. With these two factors seemingly having no moderating effect on the Robust Relative performance, it appears that the inclusion of the time dimension is the contributing factor for the difference in the performance of the two algorithms.

Comparing the Frequency Based Metrics to the Algorithms

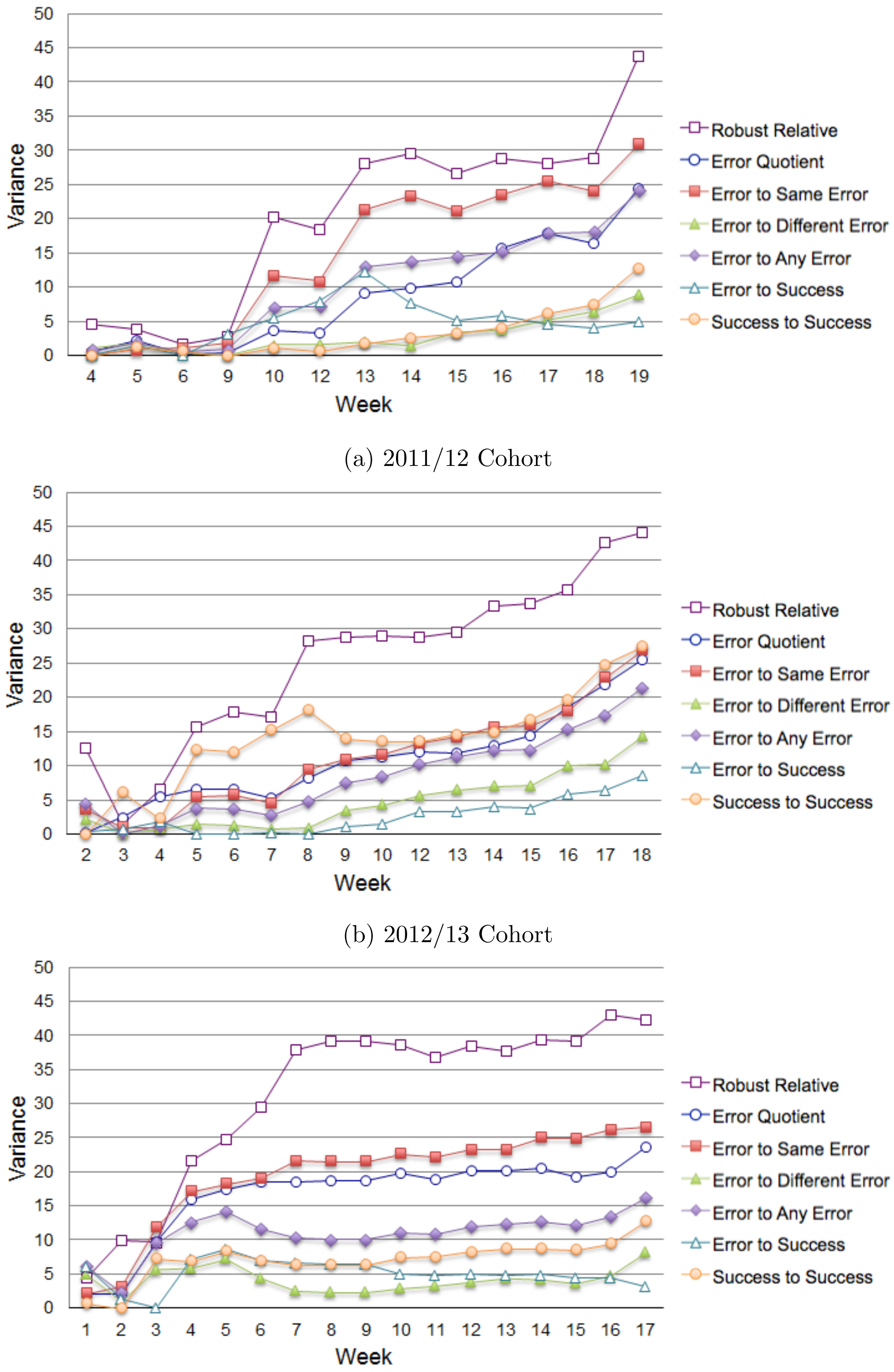
Figure 6.11 shows the variance explained by both Robust Relative and the Error Quotient compared to the frequency based metrics previously explored in this chapter.

As can be seen from this figure, for all cohorts the Robust Relative algorithm could account for the largest amount of variance in performance after four sessions, and remained the top metric through to the end of the course. The interesting trend which can be observed from these graphs is that the Error Quotient is outperformed by the percentage of error to same error pairings on all three datasets. In addition, percentage of error error to any error outperforms the Error Quotient on the 2011/12 dataset, and the percentage of success to success pairings outperforms the Error Quotient on the 2012/13 dataset.

Comparing the Time Based Metrics to the Algorithms

Figure 6.12 shows the variance explained by both Robust Relative and the Error Quotient compared to the time based metrics previously explored in this chapter.

As can be seen from this figure, for all cohorts the Robust Relative algorithm could account for the largest amount of variance in performance after four sessions, and remained the top metric through to the end of the course. As with the frequency based metrics, the Error Quotient was outperformed by several metrics. For the 2011/12 cohort, the Error Quotient was outperformed by the percentage of lab time spent working on error to same error pairings, and error to any error pairings. This may confirm that time is a stronger factor than error frequency when exploring predictors of programming performance. Additionally, the Error Quotient was outperformed by all metrics apart from percentage of lab time spent working on error to success pairings and success to success pairings.

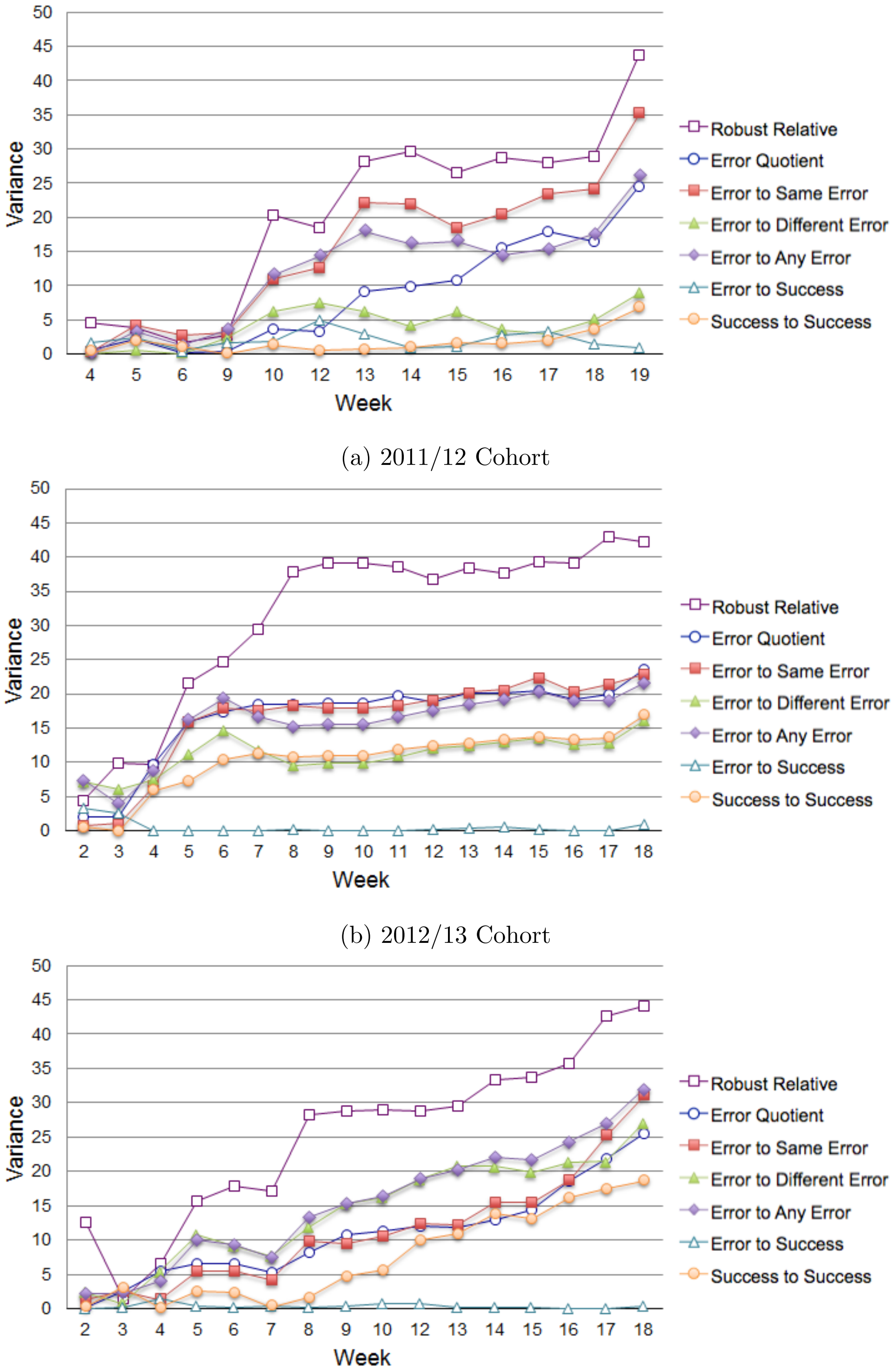


(a) 2011/12 Cohort

(b) 2012/13 Cohort

(c) 2013/14 Cohort

Figure 6.11: Figures showing the variance explained by both Robust Relative and the Error Quotient algorithms compared to the frequency based metrics previously explored in this chapter.



(a) 2011/12 Cohort

(b) 2012/13 Cohort

(c) 2013/14 Cohort

Figure 6.12: Figures showing the variance explained by both Robust Relative and the Error Quotient algorithms compared to the time based metrics previously explored in this chapter.

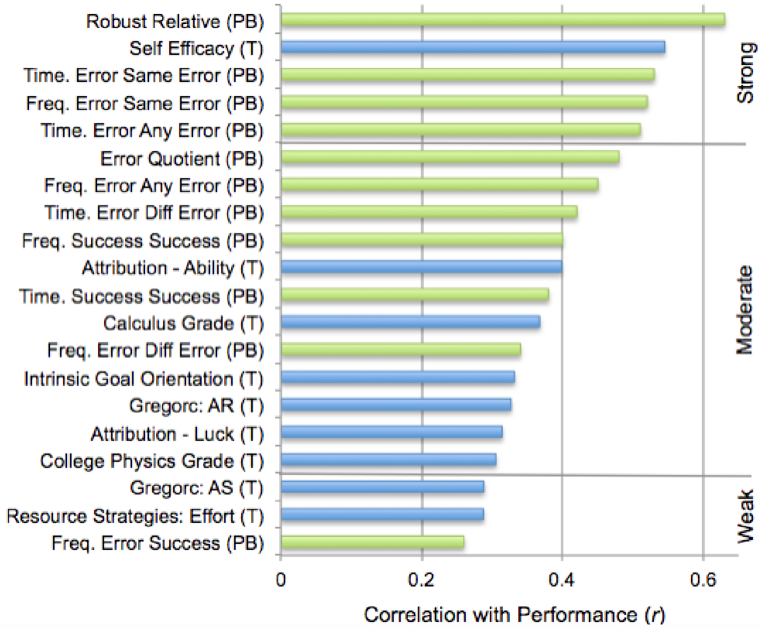


Figure 6.13: Comparing the correlation strengths of the predictors based upon programming behaviour (PB) against the predictors based upon traditional learning theories (T) which were explored in Chapter 5.

Comparing all Programming Behaviour Metrics to Traditional Predictors

Combining this research with the results from the previous chapter, Figure 6.13 compares the correlation strengths of the predictors based upon programming behaviour with the predictors based upon traditional learning theories which were explored in Chapter 5. The strongest 20 predictors are presented.

As can be seen from this figure, whilst 9 of the traditional predictors were found in the top 20, the strength of their correlations with performance were mostly in the weak-moderate range. In contrast, 11 of the 12 predictors based upon programming behaviour were found in the higher moderate-strong range, with 4 predictors strongly correlating and 6 predictors moderately correlating with performance.

The wider implication of this research is that traditional predictors are substantially less effective at reflecting the programming ability of students than the automated data-driven predictors explored in this chapter. The data-driven approaches developed in this chapter offer a more accurate method of prediction. Whereas the results for traditional predictors were found to be mostly inconsistent and context dependent, the programming behaviour metrics identified by this study performed consistently on three independent samples of students. But further research is required to validate these results in wider contexts (e.g. multi-national, multi-institutional studies).

6.5 Discussion

We briefly contrast the main benefits and drawbacks of using data-driven approaches for predicting programming performance.

The main advantage of using data-driven approaches to derive metrics based upon students' programming processes, is that these metrics are directly based upon the programming behaviours of students, and therefore can encapsulate changes in their programming knowledge over time. This is not the case for some of the more traditionally explored predictors over the past fifty years, such as gender, which remain static within the context of the course.

The second advantage of using predictors derived from programming behaviour is that they require no additional workload for either instructors or students. All of the metrics identified in this chapter require neither students to complete batches of lengthy tests, or for instructors to process them. Also as the identified metrics are based upon students' ordinary programming activities, any instructors wishing to trial data-driven metrics in their contexts do not have to change their curriculum to incorporate any specific exercises. Students can simply complete their regular programming tasks, and predictions formed from the data gathered from their IDE of choice.

The third advantage of using predictors based upon programming behaviour is that their automated nature allows them to be applied practically to drive an adaptive expert system, so that students can be provided with appropriate pedagogical interventions when required. For instance, by using the Robust Relative scores proposed earlier in this chapter, a system could be developed to provide different levels of compilation feedback to struggling students. Such interventions would be difficult to provide using traditional predictors and would not be capable of adapting to changes in a student's learning progress over time. The downside is that one would have to consider the impact of performing an intervention to change the behaviour of a struggling student on their future performance predictions.

On the other hand, the main strength of these predictors is also their main limitation. Although predictors based upon programming behaviour can explain a large proportion of the variance in performance, these predictors also required several sessions of data to be gathered before their explanatory power started to rise. We found that several of the time and frequency based metrics required at least six sessions of data before they moderately correlated with performance (12 hours). Even our Robust Relative algorithm which quantified several metrics into a single predictor required at least four sessions of data before moderately correlating with performance (8 hours). In Chapter 5, we found that several dimensions extracted from the MSLQ strongly correlated with students performance (e.g. self efficacy $r = .54$). This instrument only required students to complete 80 questions, and took us roughly four hours to process. But, we note that although yielding a strong correlation such instruments would be impractical to repeatedly issue to students over the duration of the course. Whilst the programming behaviour metrics are clearly data-driven and perform less well when data is scarce, we note that they still comfortably outperform the majority of metrics based upon traditional predictors explored in Chapter 5, most of which were shown to weakly correlate with performance (Figure 6.13). Essentially the timing of the application of either traditional or data-driven predictors is critical to their success, but this limitation is more prominent for the data-driven predictors.

6.5.1 Threats to validity

The first set of limitations concern the Robust Relative algorithm. Although we used a classic training, validation, and independent test approach for selecting penalties, there remains no certainty that the algorithm will perform comparably in different teaching contexts which use different measures of assessment. This issue is akin to the difficulties of mapping weighted multiple regression models based upon traditional predictors from one teaching context to another. Inevitably, local differences in contexts will result in variations in model performance, and even though our algorithm performed comparably on an independently gathered dataset from our context, there are no guarantees it will perform similarly in other contexts. We do however note that it would be straightforward for researchers to repeat our training process, and possibly by working with researchers in different contexts could a set of context independent penalties be identified (i.e. a set of penalties which yield strong results across different contexts despite of the differences in IDE or assessments). It is possible that due to the similarities between our datasets with ones used by previous researchers, the unweighed percentage based frequency and time metrics may perform more consistently in other contexts than the Robust Relative algorithm.

Secondly, there are no guarantees that the metrics examined in this thesis hold for other programming languages. The choice of language and IDE may indirectly influence the programming behaviours which are exhibited by the students in our context. For example, as BlueJ does not provide any automatic compilation or autocorrective features, the IDE may force students to manually compile their code more often than if they were using a full IDE such as Netbeans. An exploration of how these metrics perform in environments using different tools would be useful future work.

Thirdly, there was a lack of data from failing students in our sample. In the case of all three datasets, the majority of students performed strongly on the criterion variable, scoring $\geq 70\%$, and only 6 students scored $\leq 50\%$. This may have resulted in the training bias of the Robust Relative algorithm to successful students, and it may struggle to identify weaker students. The metrics identified in this study also could suffer from this limitation, and may perform differently when they are applied to a sample with a wider spread of marks. There is no way of exploring this limitation without data from failing students, however the significant strong strengths of the correlations support the notion that a general trend exists.

Finally, although several data-driven metrics have been identified which can illuminate the *symptoms* of struggling programming students, the actual underlying behavioural causes remain explored for future work. For example, what underlying behaviour caused them to repeat errors? This study would involve a new qualitative analysis, possibly exploring the mental models and misconceptions which students hold which cause them to exhibit certain programming behaviours.

6.6 Conclusion

In this chapter we have identified ten metrics based upon students' programming behaviours and their relations to performance in an introductory programming course. We also developed an algorithm designed to quantify programming behaviour into an overall score, measuring how desirable or undesirable aspects of students' programming behaviour has been over the duration of a course.

In contrast to previous work which predict performance by analysing indirect background, psychological or cognitive traits, our algorithm predicts a student's performance by quantifying directly logged data, describing aspects of their programming behaviour, which we have shown to correlate with performance. This supports predictions that can evolve over time reflecting changes in a student's learning progress, without the need to use traditional static tests, that often yield inconsistent results.

The novelty of this algorithm is to relatively penalize a student based upon the amount of time they take to resolve specific types of error compared to their peers. An evaluation showed our algorithm was a consistent predictor, explaining 43.65%, 44.17%, and 42.19% of the variance in performance of three independent samples of students, which was almost double the variance explained by the Error Quotient.

The results are encouraging, and the implication of this study is that predictors based upon aspects of programming behaviour may be one of the strongest predictors of performance. Researchers should continue to explore their potential further, and work is essential to verify the performance and applicability of such predictors across a variety of teaching contexts.

Chapter 7

Meta-Analysis of Fifty Years of Research into Predictors of Programming Performance

A modified version of this chapter appears in the following peer-reviewed publication:

- C. Watson et al., Meta-Analysis of Fifty Years of Research into Predictors of Programming Performance. *ACM Transactions on Computing Education* under review.

To synthesise fifty years of conflicting research on factors that are important for programming, in this chapter, we perform a meta-analysis of 482 individual results describing the relations of 116 distinct factors to programming performance. Based upon our results, a six class theoretical framework of factors predictive of programming is proposed, which highlights to researchers the most important factors for programming success that are known at this time. The results showed that factors based upon demographic, aptitude testing, and psychological factors had the weakest effects on programming. Factors based upon programming behaviour and cognitive factors, were found to have the strongest effects. The implication from this synthesis is that the most effective way of pre-screening at-risk students is through applying a test of self-efficacy, and then to continually monitor the students progress automatically based upon their programming behaviour.

7.1 Introduction

In Chapter 5 we explored predictors of programming performance that were based upon traditional learning theories. In line with previous research, the predictors were found to yield inconsistent results when applied in different teaching contexts. In Chapter 6, we explored data-driven predictors of programming performance - which unlike the traditional predictors that were explored in Chapter 5, were found to yield consistent results on three independent datasets gathered within the same teaching context. These predictors also outperformed the traditional predictors explored in Chapter 5.

Whilst these results are encouraging, the obvious question is how do these findings compare to the body of research on predictors that were not already explored in this thesis? Given that previous research contains many inconsistent findings, a narrative review would be inappropriate. Therefore, we performed a substantial meta-analysis of the past fifty years of research on predictors of programming performance.

Meta-analysis is a form of systematic review in which the quantitative results from multiple studies are statistically combined, in order to generate a more precise estimate of an effect under investigation. In an individual study, the units of analysis are individual observations, whereas in a meta-analysis the units of analysis are the results from individual studies (i.e. predictors in different teaching contexts). Compared to a simple narrative review, a meta-analysis offers several advantages, such as increased statistical power of detecting the magnitude of an effect, improved precision of the measurement of an effect, combining data from conflicting studies to determine whether an effect exists, and is less prone to bias due to the systematic review process. Only by statistically synthesising the conflicting findings of predictors applied in different teaching contexts, can researchers develop a true understanding of precisely which factors are most important for making successful programmers.

Chapter Contributions

This meta-analysis quantitatively integrates findings from the primary research on predictors of programming performance, for students taking a programming course in a school, college, or university. In this study, meta-analysis techniques are applied to synthesize the findings of multiple studies that have examined the same predictor of programming performance across different teaching contexts, such that conclusions on the general effectiveness of different predictors can be made. Although commonly applied within the medical and psychological domains, to our knowledge, this study represents the first attempt to apply meta-analysis techniques to statistically synthesise over fifty years of conflicting research into predictors of programming performance.

This chapter answers the following thesis research question:

> *RQ4: How do factors based upon traditional learning theories, academic background, and programming behaviours, compare when they are used to predict students' programming performances across different teaching contexts? (RO2, RO3)*

The contributions of this chapter are:

1. To systematically review and synthesize fifty years of research on predictors of programming performance, conducted between the years 1964-2014.
2. To perform knowledge transformation by classifying the predictors identified from the systematic review phase into a theoretical framework. This supports researchers to rapidly identify the most important factors for programming, and to derive practical applications.
3. To apply meta-analytical techniques to resolve conflicting results on different predictors, such that the importance on different factors for programming success across different teaching contexts can be determined.
4. To explore sources of moderation and heterogeneity in effect sizes, to verify our earlier findings on the influence of the teaching context on performance.
5. To provide a synthesized benchmark on the effects of different predictors on programming, which future researchers can compare their results with.

7.2 Related Work

In terms of identifying context independent predictors of programming performance, a multi-national, multi-institutional study was conducted in 2004 using students from 11 different institutions (based in Australia, New Zealand, and Scotland). Four diagnostic tests were used to measure different traits of the students. These included: paper folding test (spatial visualization and reasoning ability), map sketching (ability to articulate decisions), phone book searching (ability to articulate a search strategy), study process questionnaire (deep or surface approaches to learning). The findings of this study were that spatial visualization was associated with success, along with adopting a deep learning approach, being able to articulate a search strategy in detail, and map drawing style (landmark, route, or survey) [168,169,184]. Whilst these studies were interesting, they also take considerable effort to perform. As such the majority of experiments into

Table 7.1: Table showing examples of inconsistent findings of research into predictors of programming performance, that were not previously explored in this thesis.

	Original Result				Verification Result				
Class and Predictor	*n*	*r*	*ST*	Ref.	*n*	*r*	*ST*	Ref.	Diff.
Age	21	.53	S	[116]	93	-.01	N	[131]	.54
Gender	32	.35	M	[65]	83	.04	N	[42]	.31
IBM PAT Arithmetic	39	.75	VS	[30]	63	.23	W	[113]	.52
IBM PAT Figures	68	.43	M	[8]	46	-.04	N	[30]	.47
CPAB Reasoning	30	.56	S	[209]	49	-.03	N	[209]	.59
IBM PAT Overall	106	.56	S	[17]	46	.16	W	[30]	.40
CPAB Overall	30	.42	M	[209]	49	.08	N	[209]	.34
Wolfe PAT	93	.70	VS	[213]	33	.17	W	[213]	.53
SAT Math	92	.37	M	[97]	50	.05	N	[3]	.32
Ability (Attributional Style)	39	.40	M	[198]	105	.08	N	[210]	.32
Luck (Attributional Style)	39	-.31	M	[198]	45	.05	N	[74]	-.36
Programmer Self Efficacy	120	.37	M	[205]	247	.04	N	[189]	.33
Assimilator (Kolbs LSI)	74	-.36	M	[29]	83	.06	N	[42]	-.42
Active (Solomon-Felder ILS)	61	.34	M	[216]	84	-.02	N	[37]	.36
Group Embedded Figures	154	.40	M	[108]	83	.05	N	[42]	.35
Intellectual Development	23	.80	VS	[93]	353	.12	W	[7]	.68
Mental Model	75	.48	M	[147]	142	-.07	N	[32]	.55
High School English	32	.45	M	[91]	99	.08	N	[28]	.37
High School Science	35	.57	S	[28]	616	.18	W	[159]	.39

Notes
ST: Strength of effect on performance: VS: Very Strong, S: Strong, M: Moderate, W: Weak, N: No effect.

predictors of CS1 performance take place across considerably different contexts, often yielding considerably different results [198].

The applications of meta-analysis techniques within the computing domain are rare. More commonly systematic reviews are performed, most notably in software engineering (e.g. [27, 54, 85]). Two systematic reviews into CS1 education have been performed by ourselves, and focussed on exploring the worldwide CS1 pass rates [197], and the impact of different teaching interventions on improving CS1 pass rates [193]. In addition to the evidence already presented in RQ2 results, Table 7.1 presents examples of inconsistencies that are common within the literature on predictors of programming performance. The widespread nature of such inconsistencies would suggest that performing a meta-analysis would be worthwhile, to resolve such conflicting findings.

7.3 Research Design

There were three aims of this study. The first was to set the findings of this thesis on the relations between programming performance with traditional learning theories (Chapter 5) and programming behaviours (Chapter 6), within the greater context of quantitative research conducted over the past fifty years. The second aim was to apply unbiased meta-analysis techniques to statistically synthesise 482 results, enabling generalizable conclusions on the true effects and relevance of 116 different factors to programming. The third aim was to perform knowledge transformation by deriving a theoretical framework of interacting factors predictive of programming performance, from the results of the meta-analysis.

7.3.1 Research Questions

To answer thesis RQ4, the following sub-questions were answered:

1. Which factors have researchers examined as predictors of programming performance over the past fifty years?
2. Which factors are the most predictive of programming performance?
3. Which factors are the least predictive of programming performance?
4. Which classes of factors have the strongest and weakest effects on programming performance?
5. What conditions moderate the effectiveness of the predictors?

Like many previous meta-analyses, this study follows several key steps: (1) locating all possible studies; (2) screening potential studies for inclusion; (3) coding all qualifying studies based upon their methodological and substantive features; (4) calculating effect sizes for all qualifying studies for further combined analyses; (5) carrying out comprehensive statistical analyses covering both effects, and the relationships between effects and study moderators. In the remainder of this section, the systematic review and statistical analysis procedures employed are presented in detail.

7.3.2 Systematic Review and Data Extraction

An overview of the systematic review and data extraction procedure is presented in Figure 7.1. This will be discussed in further detail in the remainder of this subsection.

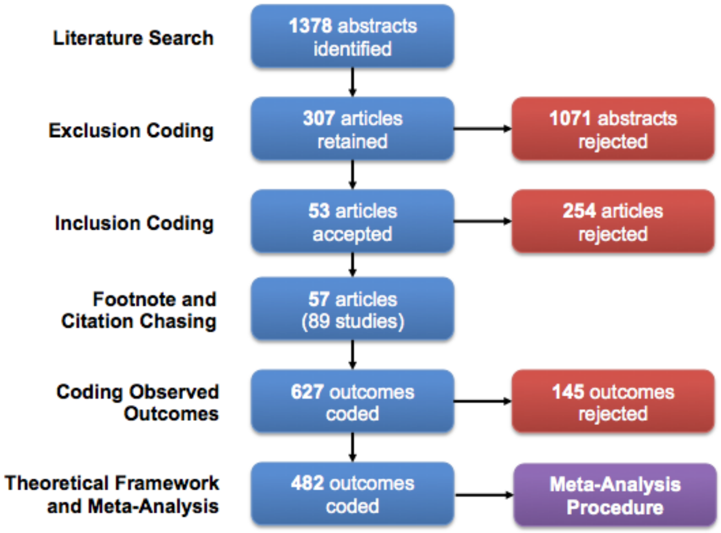


Figure 7.1: Overview of the Systematic Review and Data Extraction Procedure

Literature Search Procedure

Given the increased time and cost in identifying and retrieving grey literature, no researchers were directly contacted to request unpublished work (most meta-analyses are characterized by this limitation [114,175]). A search of all articles published between the years Mar. 1960 - Mar. 2014 was carried out. Initial electronic searches were made of the following databases, repositories, and websites: (1) ACM, (2) IEEE, (3) Science Direct, (4) Wiley Online, (5) Taylor & Francis, (6) JSTOR, (7) SAGE, (8) PsycNET, (9) EThOS, (10) ProQuest, (11) DART, (12) Trove. Following this, further searches were made using (1) Google Scholar, (2) ISI Web of Knowledge, (3) ERIC, in an attempt to identify both published and unpublished work which was not indexed by the initial 12 repositories. A final search was conducted by manually screening the indexes of selected publications, including: (1) Computers & Education, (2) British Journal of Educational Technology, (3) Transactions on Education, (4) SIGCSE, (5) ITiCSE, (6) ICER, (7) ICALT, (8) Review of Educational Research, (9) Journal of Educational Computing Research, (10) Journal of Educational Research.

Keywords were identified by two researchers, and a strategy using the operators AND and OR refined the searches, and ensured that an exhaustive search was conducted. Specifically the search criteria used was: (Predict OR Predictors OR Predicting OR Identifying OR Indicators OR Influence OR Correlates OR Factors OR Traits OR Tests OR Relationship OR Behaviour) AND (Performance OR Aptitude OR Ability OR Success OR Training OR Achievement OR Outcomes OR Learning) AND (Programming OR Programming Course OR Introductory Programming OR CS1).

Criteria for Exclusion

As the initial electronic searches identified a total of 1378 abstracts, the studies in this meta-analysis were subjected to a two-stage process of screening for inclusion. In the first stage exclusion coding was performed based upon abstract content. This was performed by two researchers, based upon the following three criteria:

1. The study appears to involve students taking a programming course.
2. The study appears to discuss traits/tests that relate to programming performance.
3. The study reports quantitative findings.

Articles that were not clearly outside the exclusion criteria were given the "benefit of the doubt", and retained for further screening. 1071 articles (78%) were eliminated based on the exclusion coding. During this initial screening, 864 articles (80%) were removed as they failed to discuss traits or tests that relate to the programming ability of students, and the remaining 157 (20%) articles were removed due to violations of the remaining criteria. The inter-coder reliability was strong (Cohens $\kappa = .88$), indicating a very good level of agreement between the two researchers.

Criteria for Inclusion

In the second stage, an inclusion coding was performed by two researchers using the full texts of the remaining 307 articles, and using the following ten criteria:

1. The participants were students taking a programming course.
2. The course must be taught in either a school, college, or university (CS1).
3. The course may use any generally available programming language, but should not use any study specific, or pseudo language.
4. The course should teach all students using the same method.
5. The study was quantitative in nature.
6. The study contained sufficient statistical data so that an effect size between a predictor and programming performance could be computed.
7. The study could have taken place in any country, but results reported in English.
8. The study should provide sufficient details of the instruments used to gather data on each predictor examined, such that a study could be replicated.

9. The study used programming performance as the criterion variable.

10. The study states the measure the programming performance of students.

The motivation for using the majority of the inclusion criteria above should be intuitive. Studies that did not use any generally available programming language were excluded due to the difficulties for researchers to replicate the study conditions. These languages are also limited as they are unlikely to be used by the general population.

In total, 254 articles (83%) were eliminated based upon the inclusion criteria. During the inclusion screening, 87 articles (35%) were eliminated as they examined the effect of an intervention on student performance, 64 articles (25%) were eliminated as they were not quantitative in nature, and 52 articles (20%) were eliminated as they failed to provide sufficient statistical data for effect size calculation. The final 51 articles (20%) were eliminated due to violations of the remaining six inclusion criteria.

Restricting a literature search to electronic databases can result in missing as many as 50% of published studies [41]. Therefore footnote and cited reference searching were performed on the 53 articles using Google Scholar. This lead to the identification of an additional 4 articles, yielding a total of 57 articles, covering 89 distinct studies.

Study Coding

Statistical data was extracted from each article so that an outcome effect for each predictor of programming performance could be calculated. Individual study features were coded to aid examination of methodological and substantive characteristics that may contribute to variations in the effect sizes among studies. Two researchers coded each study in unison according to the following coding scheme:

SC1 Types of publication: published and unpublished.

SC2 Publication period: 1960s, 1970s, 1980s, 1990s, 2000s, and 2010s.

SC3 Mean cohort size of each predictor examined: small ($n < 50$), medium ($50 \leq n < 100$) and large ($n \geq 100$).

SC4 Gender composition: male dominated ($\leq 45\%$ female), female dominated ($\leq 45\%$ male), balanced, or not stated.

SC5 Grade level: school, college, university.

SC6 Introductory or first year course: yes/no.

SC7 Programming language: Java, C, C++, C#, Python, Pascal, Fortran, others.

SC8 Criterion variable: overall letter grade, overall numerical score, exam performance, coursework performance, performance on standalone test(s), others.

SC9 Grade composition: coursework dominated ($\leq$ 45% exam), exam dominated ($\leq$ 45% coursework), balanced, or not stated.

Quality Assessment

As the bulk of articles used in this meta-analysis used a correlational design, the quality assessment indicators were developed to be consistent with this type of design, and formed a subset of the previously applied inclusion coding criteria. Specifically, the following 8 indicators of study quality were applied:

QC1 Research question(s) clearly stated.

QC2 Participants of each study were described, and information provided on at least one demographic or background trait.

QC3 Grade level of institution where the study took place was stated.

QC4 Instruments used to collect data on each predictor examined were described to the point that the study could be replicated.

QC5 Programming language taught by the course was stated.

QC6 Composition of materials used to measure programming performance was stated.

QC7 Criterion was based on assessments used in the course.

QC8 Sample size: $n \geq 25$, for each predictor examined.

Quality criteria was applied to measure whether any differences in study quality moderated the effect sizes. All studies satisfied at least 5 of the quality criteria, and none were removed based on quality assessment.

Description of the Sample

From the 57 articles that were identified during the search, a total of 627 observed outcomes were coded. Disappointingly 145 of these outcomes had to be removed, as they represented predictors which were only examined by single studies, and therefore meta-analysis techniques could not be applied to them. The final sample therefore consisted of 55 articles, describing 80 studies, and 482 observed outcomes.

Table 7.2: Summary Table of the Sample of Studies based upon the Coding Criteria

Study Coding				Study Coding			
Category	Class	n	%	Category	Class	n	%
Publication Status (SC1)	Published	80	100	Year (SC2)	1960s	2	2.5
	Unpublished	0	0		1970s	14	17.5
					1980s	28	35
					1990s	2	2.5
					2000s	28	35
					2010s	6	7.5
Cohort Size (SC3)	Small	24	30	Gender (SC4)	Male	20	25
	Medium	23	28.8		Female	3	3.8
	Large	33	41.2		Balanced	2	2.5
					Not Stated	55	68.8
Grade Level (SC5)	University	66	82.5	CS1 (SC6)	Yes	72	90
	Colleges	6	7.5		No	8	10
Language (SC7)	Java	19	23.4	Criterion (SC8)	Overall Letter	31	38.7
	Fortran	14	17.5		Overall Numerical	28	35
	Basic	7	8.8		Standalone Tests	8	10
	Pascal	7	8.8		Exam Score	6	7.5
	Multiple	7	8.8		Coursework Score	3	3.8
	C++	6	7.5		Average Letter	2	2.5
	Cobol	3	3.8		Average Numerical	2	2.5
	Others	17	21.4				
Grade Composition (SC9)	Exams	26	32.5	Study Quality (QC)	5 points	18	22.5
	Coursework	6	7.5		6 points	22	27.5
	Balanced	2	2.5		7 points	35	44.8
	Not Stated	46	57.5		8 points	5	5.5

Notes For an overview of the predictors explored by each of the 80 studies, see Table 7.3.

Summary statistics of the sample based upon the coding criteria are shown in Table 7.2. An overview of the data coded from each study and the predictors examined is shown in Table 7.3. As can be seen from Table 7.2, all of the studies included in the sample were published. The majority of studies were published during the 1980s and 2000s (70%), and the mean cohort size per predictor examined was found to be large (45.5%). The majority of studies were conduced in Universities (82.5%) at introductory level (90%). The majority of students were taught Java (23.4%) and performance was measured by either overall letter or numerical score (73.7%). The grade composition was stated by less than half of the included studies (42.5%), but when stated, the majority of grades were dominated by exams (32.5%). The average study quality was high, with the majority satisfying 7 quality criteria (44.8%).

Table 7.3: Overview of the data coded from each study included in this meta-analysis, and the predictors that each study examined in relation to the proposed theoretical framework of interacting internal factors (Figure 7.3).

Study		Sample and Context								Predictors Examined													
										Aptitude		Academic		Cognitive			Psychological			Prog. Behav.			
References	Year	Cohort Size	Gender	Grade Level	Introductory	Language	Criterion	Composition	Quality	PRO	ACA	SSP	GEN	LSY	LST	STS	AFF	PER	BEH	EBM	TBM	ALG	DEMO
1960s																							
[17]	1964	L	n/s	U	•	n/s	O/L	n/s	5	•	.	.	.	.	.	.	.	.	.	.	.	.	.
[8]	1968	M	n/s	U	•	Fortran	O/L	E	6	•	.	.	•	.	.	.	.	.	.	.	.	.	.
1970s																							
[131]	1971	M	n/s	C	•	n/s	O/L	n/s	6	•	.	.	.	.	.	.	.	.	.	.	.	.	•
[209] (a)	1971	S	n/s	U	•	n/s	O/L	n/s	5	•	•	•	.	.	.	.	.	.	.	.	.	.	.
[209] (b)	1971	S	n/s	U	•	n/s	O/L	n/s	5	•	•	•	.	.	.	.	.	.	.	.	.	.	.
[213] (a)	1971	M	n/s	S	•	n/s	O/N	E	6	•	.	.	.	.	.	.	.	.	.	.	.	.	.
[213] (b)	1971	S	n/s	S	•	n/s	O/N	n/s	6	•	.	.	.	.	.	.	.	.	.	.	.	.	.
[213] (c)	1971	L	n/s	C	•	n/s	O/N	n/s	6	•	.	.	.	.	.	.	.	.	.	.	.	.	.
[213] (d)	1971	S	n/s	C	•	n/s	O/N	n/s	6	•	.	.	.	.	.	.	.	.	.	.	.	.	.
[213] (e)	1971	S	n/s	C	•	n/s	O/N	n/s	5	•	.	.	.	.	.	.	.	.	.	.	.	.	.
[3] (a)	1972	M	M	U	•	Assembly	O/N	E	7	•	•	•	.	.	.	•	.	•	.	.	.	.	.
[3] (b)	1972	M	M	U	•	Fortran	O/N	E	7	•	•	•	.	.	.	•	.	•	.	.	.	.	.
[80]	1973	S	M	C	•	Fortran	O/N	n/s	7	.	•	.	.	.	.	•	.	.	.	.	.	.	.
[30] (a)	1975	S	n/s	U	•	Cobol	O/N	C	5	•	.	•	.	.	.	.	.	.	.	.	.	.	.

Continued on next page

Table 7.3 – *Continued from previous page*

Study		Sample and Context								Predictors Examined													
										Aptitude		Academic		Cognitive			Psychological			Prog. Behav.			
References	Year	Cohort Size	Gender	Grade Level	Introductory	Language	Criterion	Composition	Quality	PRO	ACA	SSP	GEN	LSY	LST	STS	AFF	PER	BEH	EBM	TBM	ALG	DEMO
[30] (b)	1975	S	n/s	U	•	Fortran	O/N	E	6	•	.	.	.	.	.	.	.	.	.	.	.	.	.
[132]	1975	L	n/s	U	•	Fortran	O/L	n/s	6	.	.	.	•	.	.	.	.	.	•	.	.	.	.
										1980s													
[93]	1980	S	F	U	•	Fortran	O/L	C	7	.	.	.	.	.	.	•	.	.	.	.	.	.	.
[113]	1980	M	n/s	U	•	Fortran	O/N	E	7	•	.	.	.	.	.	.	.	.	.	.	.	.	•
[97]	1982	M	n/s	U	•	n/s	O/L	n/s	5	.	•	•	•	.	.	.	.	.	.	.	.	.	.
[7]	1983	L	M	U	•	n/s	O/L	n/s	6	.	.	.	.	.	.	•	.	.	.	.	.	.	.
[77]	1983	M	B	U	•	Fortran	O/N	n/s	6	•	.	•	•	.	.	.	.	.	.	.	.	.	.
[203]	1984	M	n/s	C	•	Pascal	S/T	E	5	.	•	.	•	.	.	.	.	•	.	.	.	.	.
[91] (a)	1985	L	n/s	U	•	Fortran	O/L	n/s	7	.	•	•	•	.	.	.	•	.	.	.	.	.	.
[91] (b)	1985	L	n/s	U	•	Fortran	O/L	n/s	7	.	•	•	•	.	.	.	.	.	.	.	.	.	.
[91] (c)	1985	L	n/s	U	•	Fortran	O/L	n/s	7	.	•	•	•	.	.	.	.	.	.	.	.	.	.
[91] (d)	1985	L	n/s	U	.	Fortran	O/L	n/s	7	.	•	•	•	.	.	.	•	.	.	.	.	.	.
[91] (e)	1985	L	n/s	U	.	Fortran	O/L	n/s	7	.	•	•	•	.	.	.	.	.	.	.	.	.	.
[91] (f)	1985	L	n/s	U	.	Fortran	O/L	n/s	7	.	•	•	•	.	.	.	.	.	.	.	.	.	.
[91] (g)	1985	L	n/s	U	•	Pascal; Fortran	O/L	n/s	7	.	•	•	•	.	.	.	•	.	.	.	.	.	.

Continued on next page

Table 7.3 – *Continued from previous page*

Study		Sample and Context								Predictors Examined													
										Aptitude		Academic		Cognitive			Psychological			Prog. Behav.			
References	Year	Cohort Size	Gender	Grade Level	Introductory	Language	Criterion	Composition	Quality	PRO	ACA	SSP	GEN	LSY	LST	STS	AFF	PER	BEH	EBM	TBM	ALG	DEMO
[91] (h)	1985	L	n/s	U	•	Pascal; Fortran	O/L	n/s	7	.	•	•	•	.	.	.	.	.	.	.	.	.	.
[91] (i)	1985	L	n/s	U	•	Pascal; Fortran	O/L	n/s	7	.	•	•	•	.	.	.	.	.	.	.	.	.	.
[91] (j)	1985	S	n/s	U	•	Basic	O/L	n/s	7	.	•	•	•	.	.	.	.	.	.	.	.	.	.
[91] (k)	1985	S	n/s	U	•	Basic	O/L	n/s	7	.	•	•	•	.	.	.	.	.	.	.	.	.	.
[91] (l)	1985	L	n/s	U	.	Pascal	O/L	n/s	7	.	•	•	•	.	.	.	.	.	.	.	.	.	.
[91] (m)	1985	L	n/s	U	.	Pascal	O/L	n/s	7	.	•	•	•	.	.	.	.	.	.	.	.	.	.
[91] (n)	1985	L	n/s	U	.	Pascal	O/L	n/s	7	.	•	•	•	.	.	.	.	.	.	.	.	.	.
[42]	1986	M	n/s	U	•	Cobol	A/N	n/s	6	.	.	.	•	•	.	•	.	•	.	.	.	.	•
[52]	1986	L	n/s	U	•	Basic; Cobol; Pascal	O/L	n/s	5	.	.	•	.	.	.	.	.	.	.	.	.	.	.
[161]	1986	M	n/s	U	•	Cobol	S/T	E	5	.	•	.	.	.	.	.	.	.	.	.	.	.	.
[202]	1986	M	M	U	•	Pascal	O/L	C	7	.	.	•	•	.	.	•	.	•	•	.	.	.	•
[116] (a)	1988	M	n/s	S	•	Basic	S/T	E	6	.	•	.	.	.	.	.	.	.	.	.	.	.	.
[116] (b)	1988	S	n/s	S	•	Basic	S/T	E	6	.	•	.	.	.	.	.	.	.	.	.	.	.	.
[115]	1988	S	n/s	S	•	Basic	S/T	E	5	.	.	.	.	.	.	•	.	.	.	.	.	.	•

Continued on next page

Table 7.3 – *Continued from previous page*

Study		Sample and Context								Predictors Examined													
										Aptitude		Academic		Cognitive			Psychological			Prog. Behav.			
References	Year	Cohort Size	Gender	Grade Level	Introductory	Language	Criterion	Composition	Quality	PRO	ACA	SSP	GEN	LSY	LST	STS	AFF	PER	BEH	EBM	TBM	ALG	DEMO
[159]	1989	L	n/s	U	●	Basic; Pascal	O/L	n/s	7	.	.	●	●	.	.	.	.	.	.	.	.	.	●
1990s																							
[74]	1993	S	n/s	U	●	Pascal	O/N	n/s	6	.	.	.	.	.	.	.	●	.	.	.	.	.	.
[36]	1999	L	n/s	U	●	n/s	O/N	n/s	5	.	.	.	.	●	.	●	.	.	.	.	.	.	.
2000s																							
[65]	2000	S	n/s	U	●	C	O/N	E	6	.	.	.	●	.	.	.	.	.	.	.	.	.	●
[70]	2000	M	n/s	U	●	Java	O/N	E	7	.	.	●	.	.	.	.	.	.	.	.	.	.	.
[28]	2001	M	F	U	●	Basic	O/N	E	7	.	.	●	.	.	.	.	.	.	.	.	.	.	●
[210]	2001	L	n/s	U	●	C++	E	E	7	.	.	●	.	.	.	.	●	.	.	.	.	.	●
[178]	2002	L	n/s	U	.	Java	O/L	n/s	6	.	.	●	.	.	.	.	.	.	.	.	.	.	.
[204]	2003	L	n/s	U	●	Visual Basic	O/L	n/s	5	.	●	●	.	.	.	.	.	.	.	.	.	.	.
[1]	2004	L	n/s	U	●	n/s	O/N	n/s	5	.	.	●	.	●	.	.	.	.	●	.	.	.	.
[108]	2004	L	n/s	U	●	Ada	O/N	E	6	.	.	.	.	.	.	●	.	.	.	.	.	.	.
[147]	2004	M	n/s	U	●	C++	O/L	n/s	6	.	.	.	.	.	.	●	.	.	.	.	.	.	.
[189]	2004	L	n/s	U	●	Java	O/N	B	6	.	.	●	.	.	.	.	●	.	.	.	.	.	.

Continued on next page

Table 7.3 – *Continued from previous page*

Study		Sample and Context								Predictors Examined													
										Aptitude		Academic		Cognitive			Psychological			Prog. Behav.			
References	Year	Cohort Size	Gender	Grade Level	Introductory	Language	Criterion	Composition	Quality	PRO	ACA	SSP	GEN	LSY	LST	STS	AFF	PER	BEH	EBM	TBM	ALG	DEMO
[16]	2005	S	n/s	U	•	Java	C	n/s	7	.	.	.	.	.	•	.	.	.	.	.	.	.	.
[14]	2005	S	M	U	•	Java	E	E	8	.	.	.	.	.	•	.	•	.	.	.	.	.	.
[15]	2005	S	n/s	U	•	Java	O/N	E	7	.	.	•	.	.	.	.	•	.	.	.	.	.	•
[87]	2005	L	M	U	•	Java	O/L	n/s	6	.	•	.	.	.	.	.	.	.	.	.	.	.	•
[142] (a)	2005	S	M	U	•	Java	O/N	n/s	5	.	.	.	.	.	.	.	.	.	.	.	.	.	•
[142] (b)	2005	L	F	U	•	Java	O/N	n/s	6	.	.	.	.	.	.	.	.	.	.	.	.	.	•
[190]	2005	L	M	U	•	Java	O/N	B	8	.	•	•	•	.	.	.	•	.	•	.	.	.	•
[205]	2005	L	M	U	•	C++	O/L	n/s	7	.	.	•	.	.	.	.	•	.	.	.	.	.	.
[82]	2006	M	n/s	U	•	Java	A/N	C	7	.	.	.	.	.	.	.	.	.	.	.	.	•	.
[216] (a)	2006	M	M	U	•	C++	A/L	n/s	7	.	.	.	.	•	.	.	.	.	•	.	.	.	.
[216] (b)	2006	M	M	U	•	C++	A/L	n/s	7	.	.	.	.	•	.	.	.	.	•	.	.	.	.
[32]	2007	L	n/s	U	•	Java	E	E	7	.	.	.	.	.	.	•	.	.	.	.	.	.	.
[84] (a)	2008	S	n/s	U	•	Java	O/N	n/s	5	.	.	•	.	.	.	•	.	.	.	.	.	.	•
[84] (b)	2008	S	n/s	U	•	C++	O/N	n/s	5	.	.	.	.	.	.	•	.	.	.	.	.	.	.
[164]	2008	M	M	U	•	n/s	O/N	n/s	5	.	.	•	.	.	.	.	.	.	.	.	.	.	.
[95]	2009	L	M	S	•	Pascal; C	S/T	E	7	.	.	.	.	•	.	.	.	.	.	.	.	.	.
[124]	2009	M	B	S	.	Pascal	S/T	E	5	.	.	.	.	.	.	.	.	.	.	.	.	.	•
[155]	2009	S	M	U	•	Java	E	E	8	.	.	.	.	.	.	.	.	.	.	•	•	.	.

Continued on next page

Table 7.3 – *Continued from previous page*

Study		Sample and Context								Predictors Examined													
										Aptitude		Academic		Cognitive			Psychological			Prog. Behav.			
References	Year	Cohort Size	Gender	Grade Level	Introductory	Language	Criterion	Composition	Quality	PRO	ACA	SSP	GEN	LSY	LST	STS	AFF	PER	BEH	EBM	TBM	ALG	DEMO
										2010s													
[29]	2010	M	M	U	●	Java	O/N	E	7	.	.	.	.	●	.	.	.	.	.	.	.	.	.
[37]	2011	M	n/s	U	●	Java	E	E	6	.	.	.	.	●	.	.	.	.	.	.	.	.	.
[96]	2011	L	M	S	●	Pascal; C	S/T	E	7	.	.	.	.	●	.	●	.	.	.	.	.	.	.
[182]	2011	L	M	U	●	Java	E	E	8	.	.	.	.	.	.	.	.	.	.	●	●	●	.
[198]	2014	S	M	U	●	Java	C	C	7	.	.	●	●	●	●	.	●	.	●	●	●	●	●
[196]	2014	S	M	U	●	Java	C	C	8	.	.	.	.	.	.	.	.	.	.	●	●	●	.

Notes

Coding abbreviations. (see section 7.3.2 for full descriptions).

Cohort Size: S: Small, M: Medium, L: Large. *Gender*: M: Male Dominated, F: Female Dominated, B: Balanced, n/s: Not Stated. *Grade Level*: S: School, C: College, U: University. *Criterion*: O/L: Overall Letter, O/N: Overall Numerical, S/T: Standalone Tests, E: Exam Score, C: Coursework Score, A/L: Average Letter, A/N: Average Numerical. *Composition*: E: Exam Dominated, C: Coursework Dominated, B: Balanced, n/s: Not Stated.

Predictor abbreviations. (see section 7.4 for full descriptions).

PRO: Programming Aptitude; ACA: Academic Aptitude; SSP: Subject Specific Performance; GEN: General Performance; LSY: Learning Styles; LST: Learning Strategies; STS: Cognitive Style and Skills; AFF: Affective Traits; PER: Personality; BEH: Behavioural Traits; EBM: Event Based Metrics; TBM: Time Based Metrics; ALG: Algorithmic Based Metrics; DEMO: Demographic.

7.3.3 Meta-Analysis Procedure

A statistics consultant from the Durham Wolfston Institute was contacted for advice on an appropriate meta-analysis procedure to apply. An overview of the procedure applied in this study [31,78] is shown in Figure 7.2, and will be detailed in this section.

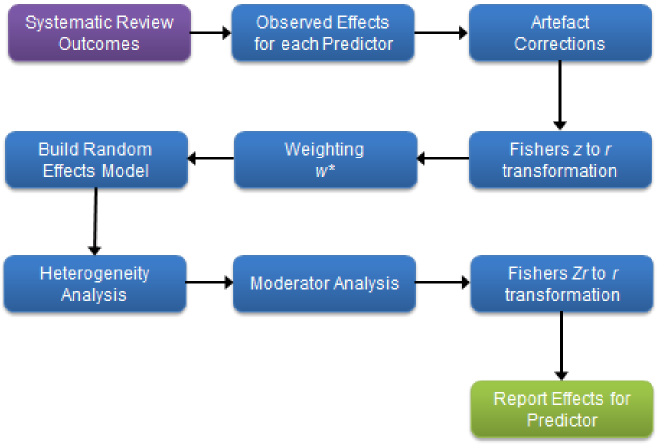

Systematic Review Outcomes
Observed Effects for each Predictor
Artefact Corrections
Build Random Effects Model
Weighting w^*
Fishers z to r transformation
Heterogeneity Analysis
Moderator Analysis
Fishers Zr to r transformation
Report Effects for Predictor

Figure 7.2: Overview of the Meta-Analysis Procedure Applied by this Study

Model Selection

Most meta-analyses are based on sets of studies that are not exactly identical in their methodology and/or the characteristics of their sample. Differences in the methods and sample characteristics can introduce variability (heterogeneity) among effect sizes that cannot be explained through random sampling fluctuations alone. One way to model this heterogeneity is to treat it as purely random. This leads to the random effects meta-analysis procedure [31,71,78,156] which is applied by this study.

The main difference when compared to the fixed effects procedure is that rather than assuming there is only one true effect size, the random effects procedure assumes that there is a distribution of true effect sizes (arising from differences in methods and sample characteristics), and that the overall true effect can be calculated by averaging the effects within this distribution. Similar to the fixed effects procedure, a weighting of effect sizes is applied. However the random effects weighting does not discount effects from the smaller studies by giving them a smaller weight, and the impact of larger studies is also less pronounced. The trade off of this weighting is larger variance. As long as the between studies variation is non-zero, the variance, standard error, and confidence interval of the true effect will always be larger than an estimate calculated by using the fixed effects model.

The main advantage of the random effects model is that it allows unconditional inferences to be made about the larger set of studies, from which the studies included in the meta-analysis are assumed to be sampled from (whether such studies currently exist or not). For this reason, the random effects model is most applicable when the objective of a researcher is to accumulate data from a series of studies that have been performed under a number of different conditions. i.e., supporting the generalizability of results with the trade-off of less precise effect estimates (wider confidence intervals) [149].

Effect Size Measure Selection

As this study mainly examines linear relationships between two continuous variables, or in some cases, a dichotomous and continuous variable, Pearson's r was selected as the most appropriate measure of effect size. This measure was selected as it provides a number of advantages over alternative measures that are based upon standardized mean difference (Cohen's d or Hedges' g), such as familiarity and flexibility [31]. When studies did not directly report their effects in terms of Pearson's r, a number of standard conversion formulae were applied based upon available data (see Appendix A.5.1).

Handling of Multiple Effects Reported by a Single Study

One of the assumptions of meta-analysis is that each of observed effects are independent from others. However, this assumption is often violated, and the non-independence of observations can lead to inflations in the calculated effect sizes [63]. This arises through two situations and is usually handled by exclusion or averaging techniques [157].

In the case of studies which reported the outcome effect of a predictor on multiple criterion variables, a single outcome effect was included in the meta-analysis. This was chosen based upon the amount of assessment that was used to measure the programming performance of students. Measures based upon overall course performance were given first preference, followed by performance on coursework, exams, then standalone test(s). The rationale behind this decision was that performance across a range of assessment components were more likely to be indicative of ability than measures which were based upon fewer assessments. Coursework assessments were given preference over exams due to the fact that exams have been widely criticised as an accurate means to assess programming performance and only focus upon surface level knowledge [118, 198].

In the case of studies which reported multiple outcome effects on a single criterion variable (e.g. math performance, prior experience, and spatial ability all correlated with overall performance), neither averaging or exclusion techniques were appropriate for this meta-analysis. Simply averaging the outcome effects for math performance and spatial ability would not make sense, as they are two unrelated traits. As the purpose of this study was to include as many different predictors as possible excluding one predictor over another also could not be justified. Therefore each pairwise combination of a predictor and criterion variable were treated as an independent study, and to ensure that the independence assumption was not violated, an individual meta-analysis was performed on a per-predictor basis.

Artefact Corrections

Artefact correction attempts to adjust outcome effects to take into account imperfections in instruments. This allows conclusions to be formed about the associations among particular predictors, rather than among the instruments that were used to measure the predictors [78]. In this meta-analysis two forms of artefact correction were applied: unreliability and artificial dichotomization (see Appendix A.5.2).

Statistical Approach

The statistical analyses were performed using the *metafor* package for the statistical software R [191]. To ensure that the independence assumptions of meta-analysis were not violated, observed outcomes were meta-analytically combined on a per-predictor basis, to calculate effects ($ES_{\bar{r}}$) for various predictors on programming performance.

Observed outcomes were initially coded from each article using r. For cases where r was not directly reported standard conversion formulas were applied to estimate r using the available statistics (see Appendix A.5.1). The reported dichotomous effects were adjusted to have consistent polarity (e.g. studies that reported effects for gender were adjusted based upon the coding used for males and females). Artefact corrections for unreliability and artificial dichotomization were applied along with Fisher's z to r transformation to normalize the distribution of effects and so that the sampling variance could be estimated (see Appendix A.5.2).

The true effect of each predictor was then estimated using the random effects model (see Appendix A.5.3). For each effect, if the test of heterogeneity (Q) was significant, and the number of studies, $k \geq 5$ [31], then moderator analysis was performed within the general regression framework using the 9 properties that were coded from each study and study quality score (Table 7.2). Finally, a Z_r to r back transformation was applied for reporting the true effect and confidence interval for each predictor.

In line with recommendations by [31,40], a small effect size is indicated by a $ES_{\bar{r}} \geq .10$, moderate effect by $ES_{\bar{r}} \geq .30$, strong effect by $ES_{\bar{r}} \geq .50$, and very strong effect by $ES_{\bar{r}} \geq$ of .60.

7.4 Theoretical Framework

To answer RQ1, such that researchers can develop an understanding of which factors are predictive of programming performance, we propose a six class theoretical framework of interacting factors. This is shown in Figure 7.3, and an overview of the predictors examined by each study is presented in Table 7.3.

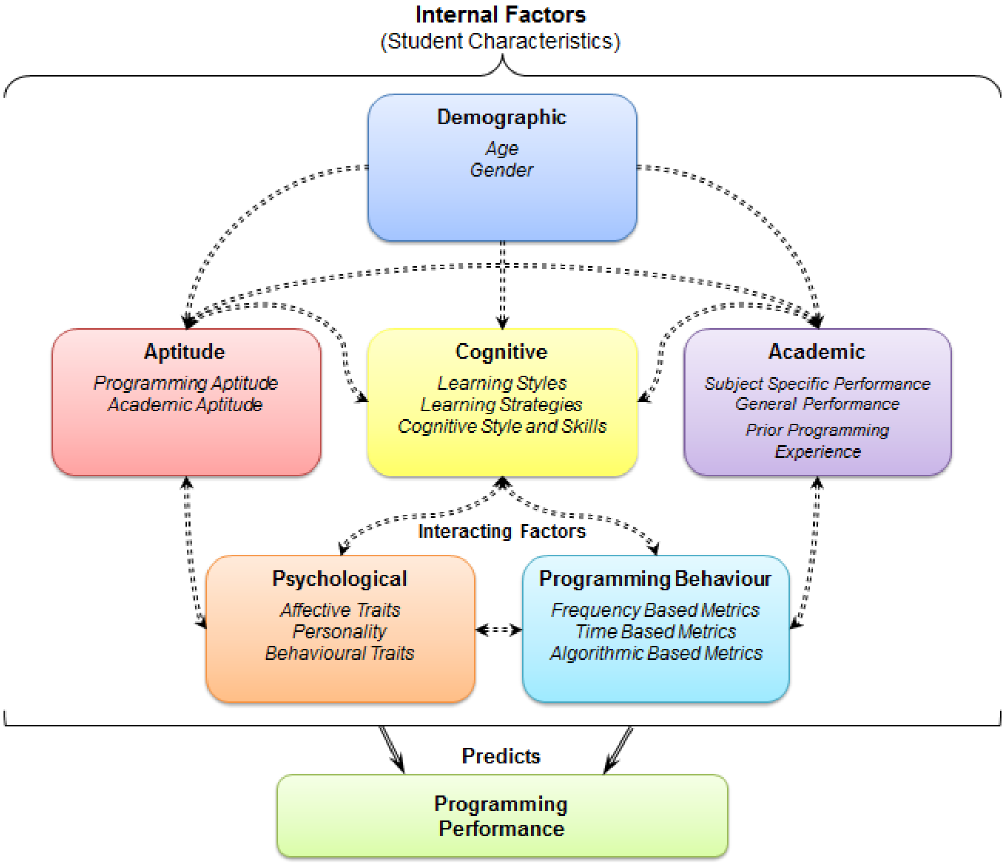


Figure 7.3: Framework of internal factors that are predictive of programming performance. Interacting factors were determined by related studies in the literature, and are shown on the diagram by using dashed arrows.

As can be seen from this figure, the theoretical framework consists of six classes of predictors, based upon: demographic, aptitude, cognitive, academic, psychological, and programming behaviour. The interacting factors were not added arbitrarily. In most cases, the interactions were intuitive and determined through related works on learning theories, and further detail of their derivation will be presented at the end of this section. In the remainder of this section the classes of the framework are defined in detail, and example predictors of each sub-class are provided.

7.4.1 Aptitude Predictors

This class consists of predictors that are based upon tests intended to measure the learners readiness to learn specific concepts or skills. For example, a high aptitude in math would suggest that a student is ready to learn more math concepts on the basis of demonstrating their knowledge of the basic precepts for learning math. Within our framework, aptitude is divided into two broad subclasses:

1. *Programming Aptitude.*
   Subclasses: *Arithmetic, Letter and Other Reasoning, Overall Measures.*
   Many early researchers used logic and arithmetic reasoning tests to predict programming potential. This class consist of predictors that are based upon such tests. Instruments include: IBM Programmer Aptitude Test (IBM PAT) and Computer Programmer Aptitude Battery (CPAB). Predictors include: performance on the IBM PAT letter reasoning, and CPAB diagramming scales.

2. *Academic Aptitude.*
   Subclasses: *Math Aptitude, English Aptitude.*
   This subclass consists of predictors that are based upon performance on tests of academic aptitude, specifically English and Math. The instruments used by predictors in this subclass include SAT and ACT tests. Example predictors include performance on the SAT Math and SAT Verbal tests.

7.4.2 Academic Predictors

This class consists of predictors that are based upon the academic performance of learners in various subjects. Within our framework, academic predictors are divided into two broad subclasses:

1. *Subject Specific Performance.*
   Subclasses: *Math, English, Science, Prior Programming, Other Subjects.*
   This class consists of predictors that are based upon performance in various secondary and further education courses. Instruments include: High School and College exams. Predictors include: High School Science performance, and Prior Programming Experience.

2. *General Performance.*
   Subclasses: *Overall College Performance, Overall High School Performance.*
   Whereas the first subclass was based upon subject specific performance, this subclass was based upon overall measures of academic performance. Predictors included: High School GPA and College GPA.

7.4.3 Cognitive Predictors

This class consists of predictors that are based upon the various mental processes that influence the ways in which learners think and learn, and the cognitive learning styles and strategies that learners employ to make knowledge and skill acquisition possible. The influence of different cognitive factors has been well explored over the past fifty years, however this is also one of the classes of predictors where researchers have reported a high number of inconsistent results. There were three cognitive subclasses:

1. *Learning Styles.*
   Subclasses: *Kolb LSI, Soloman-Felder ILS, Gregorc Style Delineator.*
   A learning style describes how learners perceive, interact with, and respond to different learning situations. The implication of learning styles is that different learners will respond uniquely to different teaching approaches, and in order for instruction to be most effective, the teaching approach employed should be as closely aligned with an individual's learning style as possible. Learning style models will usually describe the degree to which a learner ranks on a particular scale. This class consists of predictors that are based upon three instruments: Kolb LSI, Soloman-Felder ILS, and the Gregorc Style Delineator. Predictors include scores on LSI scales: sequential, visual, accomodator, and concrete-sequential.

2. *Learning Strategies.*
   Subclasses: *Expectancy components, Resource management components, Value components, Metacognitive strategies, Overall measures.*
   A learning strategy describes the techniques and methods that learners employ to ensure that they satisfy any required learning objectives. They differ from learning styles as they describe the specific techniques that a learner employs, rather than characteristics describing the learners general approach of knowledge acquisition. The instrument used by all predictors within this subclass was the Motivated Strategies for Learning Questionnaire (MSLQ). Predictors include: intrinsic goal orientation, critical thinking, and control of learning beliefs.

3. *Cognitive Style and Skills.*
   Subclasses: *Cognitive style, Specific cognitive skills.*
   A cognitive style describes the way in which learners obtain, organize, and apply knowledge. This subclass also consists of specific cognitive skills. Instruments used by predictors in this class include the Group Embedded Figures Test, and the Watson-Glaser critical thinking test. Predictors include: clarity of mental model, spatial ability, and level of intellectual development.

7.4.4 Psychological Predictors

This class consists of predictors that are based upon the various non-cognitive factors that influence the ways in which learners think and learn. They describe the personal characteristics of the learner, describing how they differ from, or are similar to others, in a non-cognitive sense, such as in terms of their personality, self esteem, or self efficacy. Within our framework, psychological predictors are divided into three subclasses:

1. *Affective Traits.*
   Subclasses: *Attributional style, Self perceived abilities, Self efficacy.*
   Affective traits generally describe the emotional state of a learner, and include factors that can reflect their feelings and attitudes during the learning process. Considerable research has been conducted over recent years on the influence of affective factors on the learning process, most notably on the trait of self-efficacy, or the belief in one's own abilities. Three subclasses of affective traits are examined. The instruments used by predictors in this subclass include: Rosenberg's Self Esteem Scale and the Computer Programmer Self Efficacy Scale. Predictors include: attribution of success to luck, self perceived problem solving ability, comfort level, level of self esteem, and self efficacy.

2. *Personality.*
   Subclasses: *Thurstone temperament schedule, Myers-Briggs type indicator.*
   Personality traits describe the frequency or intensity of the learners feelings, thoughts, or behaviours, relatively against other learners. Classically in Psychology there are five personality factors: openness, conscientiousness, extraversion, agreeableness, and neuroticism. In this study, studies used two instruments to measure the personality type of learners: Thurstone Temperament Schedule, and Myers-Briggs Personality Type Indicator (MBTI). Predictors in this category include scores on various personality scales, such as: the degree to which a learner was impulsive, and the degree to which a learner was orientated towards judgement or perception styles.

3. *Behavioural Traits.*
   Subclasses: *None.*
   This subclass consisted of predictors that were based upon the behavioural aspects of learners which can either directly, or indirectly, influence their learning progress. The instruments used by studies in this category were mainly in-house questionnaires. Predictors include the amount of lectures attended, hours worked in a part-time job, and the percentage of time spent in the lab.

7.4.5 Programming Behaviour Predictors

This class consists of predictors that are based upon analysing aspects of the programming behaviour of learners. Unlike the other classes in the framework, these classes do not require formal test instruments, and are based upon analysing data logged directly from an IDE describing the programming activities of learners. Within our framework, the programming behaviour predictors are divided into three subclasses:

1. *Event Based Metrics.*
   Subclasses: *Percentage based metrics, Frequency based metrics.*
   This subclass consists of predictors that are based upon the types of compilation pairings [200] that were logged for learners during a programming session. These pairings describe how the learners programming behaviour changes in response to a sequence of consecutive compilation events. Two subclasses are used to classify these event based metrics. The first is based on raw count of specific types of event pairings, and the second is based on the percentage of different events a learner encountered, relative to all the events they encountered. Predictors in this category include: percentage of successive errors and average number of errors encountered.

2. *Time Based Metrics.*
   Subclasses: *Percentage based metrics, Frequency based metrics.*
   This subclass consists of predictors that are based upon the percentage of lab time that learners spent working on different compilation pairings. Two subclasses are used. The first was based on the total amount of lab time learners spent working on different types of pairings, and the second was based on the percentage of lab time learners spent working on different types of pairings. Predictors include: percentage of lab time spent in error, and average time between compilations.

3. *Algorithmic Based Metrics.*
   Subclasses: *Overall quantification.*
   The third subclass consists of predictors which use algorithmic approaches to weight and combine different aspects of programming behaviour into an overall performance predictor. Predictors include: Error Quotient and Robust Relative.

7.4.6 Demographic Predictors

Many researchers in the field of education have considered the impact of demographic characteristics on the learning process. These include: age, race, and religion. Within this meta-analysis, two demographic factors: age and gender are explored.

Table 7.4: Example interacting factors within the theoretical framework classes. Many of these are hypothesised based upon the systematic review we performed.

Interacting Classes		Interacting Predictors	
Class A	Class B	Predictor A	Predictor B
Demographic	Aptitude	Age	SAT Math
	Cognitive	Age	Intellectual Development
	Academic	Age	High School GPA
Aptitude	Academic	SAT Math	College Math
	Cognitive	CPAB Diagramming	Spatial Ability
	Psychological	ACT English	Self Perceived Writing
Cognitive	Aptitude	Group Embedded Figures Test	CPAB Diagramming
	Academic	Intrinsic Goal Orientation	College GPA
	Psychological	Self Efficacy	CPSE
	Prog. Behaviour	Mental Model	Percentage of Errors
Academic	Aptitude	College English	ACT English
	Cognitive	College GPA	Self Efficacy
	Prog. Behaviour	Prior Prog. Experience	Time to Resolve Errors
Psychological	Aptitude	Self Perceived Math	ACT Math
	Cognitive	CPSE	Self Efficacy
	Prog. Behaviour	Problem Solving	Total Number of Successful Compilations
Prog. Behaviour	Psychological	Percentage of Successful Compilations	Self-Efficacy
	Cognitive	Total Repeated Errors	Mental Model
	Academic	Total Compilations	Prior Prog. Experience

7.4.7 Identifying Interacting Factors

Table 7.4 presents examples of the interacting factors in the theoretical framework. For the most part, these interactions were hypothesised based upon the literature we examined as part of the systematic review process. For instance, a level of aptitude measured using the SAT Math instrument was believed to infulence academic math performance, on the basis the level of math aptitude can suggest students are ready to learn more math concepts. As no study has examined interacting factors with programming behaviour, we hypothesized interacting factors with psychological traits (self-efficacy), academic traits (prior programming), and cognitive traits (mental models).

7.5 Results

To answer the remaining research questions, results from the meta-analysis are explored within the context of the theoretical framework. Note that for clarity, the Figures used to display the results of the meta-analysis for each class of the framework (Figures 7.4 to 7.9) have had negative effects inverted (i.e. absolute effect sizes are shown). This allows the strengths of different effect sizes to be visually compared on the same scale, rather than displaying both positive and negative effects. Different colours are used to represent negative effects. The actual effect sizes are shown in Tables 7.5 to 7.10.

7.5.1 Aptitude Predictors

Results are presented in Table 7.5 and Figure 7.4. In total 101 observed outcomes were coded from 38 studies which described 16 aptitude predictors. Of these predictors, 11 were classified into our theoretical framework as being based upon programming aptitude and 5 were classified as being based upon academic aptitude. The mean of the effect sizes which resulted from the meta-analysis suggests that aptitude factors are a weak predictor of programming performance ($\overline{ES_{\bar{r}}} = .239$, $SD = .136$). 10 predictors (63%) were found to have a weak effect on programming performance and 2 predictors (13%) were found to have no effect. Only 3 predictors (19%) were found to have a moderate effect and 1 predictor (5%) to have a strong effect.

Programming Aptitude

A total of 49 outcomes describing 11 predictors based upon programming aptitude were coded. The mean of the effect sizes which resulted from the meta-analyses of predictors in this subclass, suggests that programming aptitude has a weak effect on programming performance ($\overline{ES_{\bar{r}}} = .284$, $SD = .128$). The magnitude of effects were found to vary based upon the four programming aptitude subclasses, but the majority of the 95% confidence intervals for effects of this subclass suggests that factors based upon programming aptitude are weak predictors of performance (Figure 7.4).

The 2 predictors that were based upon arithmetic reasoning yielded mixed effects. Whilst the IBM PAT arithmetic dimension had a moderate effect on performance ($ES_{\bar{r}} = .396$, $p < .01$), the equivalent CPAB number ability dimension only had a weak effect on performance ($ES_{\bar{r}} = .163$, $p < .05$). The 2 predictors that were based upon letter reasoning both yielded weak effects on performance, although the IBM PAT letter series dimension ($ES_{\bar{r}} = .257$, $p < .01$) yielded a stronger effect than the equivalent CPAB letter series ($ES_{\bar{r}} = .147$, $p = .06$). The 4 predictors that were based upon other forms of reasoning (e.g. logical) also yielded mixed effects.

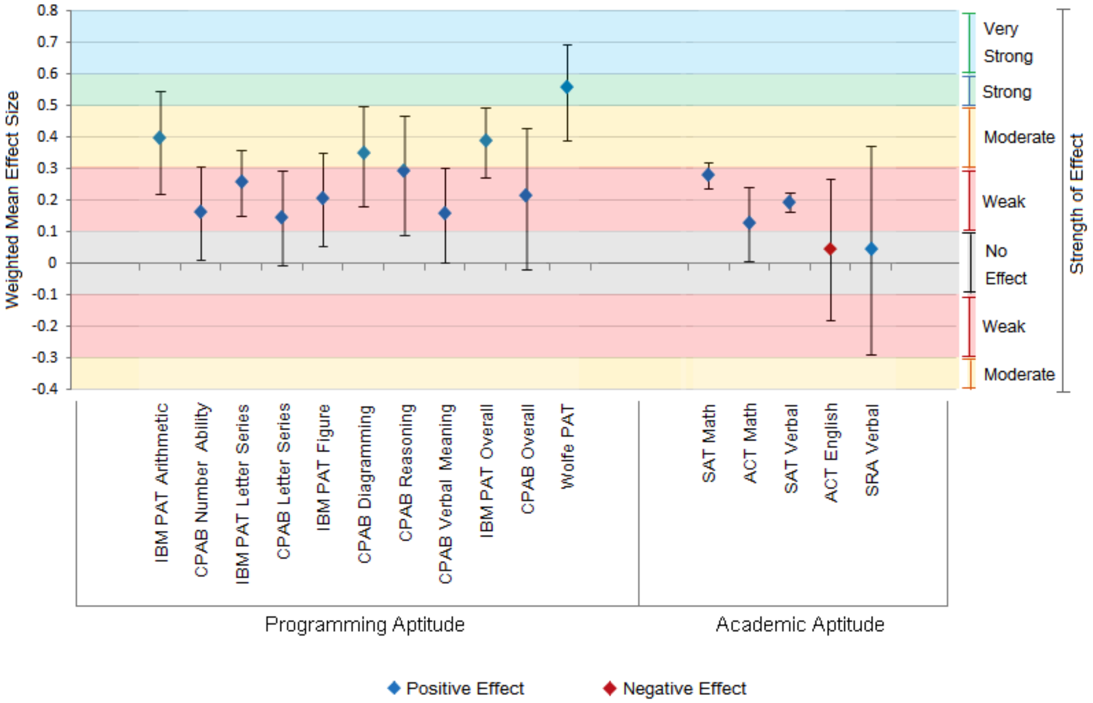


Figure 7.4: Chart Showing Meta-Analysis Results for the Aptitude Predictors

Out of the 3 predictors that were based upon overall score on programming aptitude tests, the Wolfe PAT was found to have a strong effect on performance ($ES_{\bar{r}} = .560$, $p < .01$), the IBM PAT was found to have a moderate effect ($ES_{\bar{r}} = .388$, $p < .01$), whereas the CPAB was found to have a weak effect ($ES_{\bar{r}} = .216$, $p = .07$).

It is unclear as to why the Wolfe PAT produced a stronger result than the other two aptitude tests we examined. One reason could be that the Wolfe PAT was specifically designed for students, whereas the CPAB and IBM PAT were designed for professionals. The tasks therefore may be more tailored towards students' abilities.

Academic Aptitude

A total of 49 outcomes describing 11 predictors based upon academic aptitude were coded. The mean of the effect sizes which resulted from the meta-analyses of predictors in this subclass, suggests that academic aptitude has a weak effect on programming performance ($\overline{ES_{\bar{r}}} = .138$, $SD = .101$).

Out of the 2 predictors that were based upon Math aptitude, performance on the ACT Math instrument yielded a weak effect on programming ($ES_{\bar{r}} = .126$, $p < .05$), and the performance on the popular SAT Math instrument ($ES_{\bar{r}} = .281$, $p < .01$)

Table 7.5: Meta-Analysis Results for the Aptitude Predictors

	Sample			Meta-Analysis				
Predictor	k	n	SM	$ES_{\bar{r}}$	CI_L	CI_U	z	p
Aptitude								
Programming								
Arithmetic Reasoning								
IBM PAT Arithmetic	6	316	52	.396	.222	.546	4.24	.01
CPAB Number Ability	3	172	57	.163	.011	.308	2.10	.04
Letter Reasoning								
IBM PAT Letter Series	6	316	52	.257	.149	.360	4.55	.01
CPAB Letter Series	3	172	57	.147	-.006	.293	1.89	.06
Other Reasoning								
IBM PAT Figures	6	316	52	.208	.055	.351	2.66	.01
CPAB Diagramming	4	236	59	.349	.179	.498	3.90	.01
CPAB Reasoning	4	236	59	.291	.090	.469	2.80	.01
CPAB Verbal Meaning	3	172	57	.157	.005	.302	2.02	.04
Overall								
IBM PAT Overall	7	422	60	.388	.271	.495	6.07	.01
CPAB Overall	2	79	39	.216	-.020	.430	1.79	.07
Wolfe PAT	5	360	72	.560	.388	.694	5.56	.01
Academic								
Math Aptitude								
SAT Math	23	4028	175	.281	.238	.322	12.29	.01
ACT Math	3	278	92	.126	.007	.241	2.08	.04
English Aptitude								
SAT Verbal	22	3678	167	.194	.162	.225	11.78	.01
ACT English	2	79	39	-.047	-.270	.180	-.40	.69
SRA Verbal	2	120	60	.046	-.290	.372	.26	.79

Notes
k: number of studies, n: number of participants, SM: mean sample size, $ES_{\bar{r}}$: weighted effect size, CI_L: 95% confidence interval (lower), CI_U: 95% confidence interval (upper), z: test of effect size, p: significance of effect size.

was found to have the strongest effect on programming performance out of any of the predictors that were based upon academic aptitude. The narrow 95% confidence interval of the SAT Math effect ($CI_L = .238$ to $CI_U = .322$) suggests that the true effect in the population is weak, with a high degree of confidence.

Out of the 3 predictors that were based upon English aptitude, only performance on the SAT Verbal instrument yielded a significant weak effect ($ES_{\bar{r}} = .194$, $p < .01$). As with the SAT Math instrument, the confidence interval was narrow ($CI_L = .162$ to $CI_U = .225$) suggesting that the true effect in the population is weak at best.

7.5.2 Academic Predictors

Results are presented in Table 7.6 and Figure 7.5. In total 144 observed outcomes were coded from 42 studies which described 19 academic predictors. Of these predictors, 16 were classified into our theoretical framework as being based upon subject specific performance and 3 were classified as being based upon general performance. The mean of the effect sizes which resulted from the meta-analysis suggests that academic factors are a weak predictor of programming performance ($\overline{ES_{\bar{r}}} = .275$, $SD = .143$). 8 predictors (42%) were found to have a moderate effect, 8 predictors (42%) were found to have a weak effect, 2 predictors (11%) were found to have no effect, and 1 predictor (5%) was found to have a strong effect.

Subject Specific Performance

A total of 114 outcomes describing 16 predictors based upon subject specific performance were coded. The mean of the effect sizes which resulted from the meta-analyses of predictors in this subclass, suggests that subject specific performance has a weak effect on programming performance ($\overline{ES_{\bar{r}}} = .270$, $SD = .153$). The magnitude of effects were found to vary based upon the five subject subclasses.

For the predictors which were based upon performance in Math, the strongest effect was found for overall college Math ($ES_{\bar{r}} = .481$, $p < .01$). The narrow 95% confidence interval for this predictor (range: $CI_L = .446$ to $CI_U = .515$) suggests that the true population effect of College Math on programming is moderate to strong. The weakest math based predictor was units of high school math completed ($ES_{\bar{r}} = .185$, $p < .01$). This may suggest that knowledge of specific higher level concepts encountered at college level and that a certain level of mathematical maturity beyond high school is beneficial for programming.

This is possibly confirmed by the results for English and Science performance which also found stronger effects for college based subject performance than high school performance. College chemistry ($ES_{\bar{r}} = .502$, $p < .01$) and college physics ($ES_{\bar{r}} = .484$, $p < .01$) both returned comparable moderate-strong effects. High school science ($ES_{\bar{r}} = .273$, $p < .01$) returned a weak effect. College English ($ES_{\bar{r}} = .282$, $p < .01$) returned a weak effect, but high school English ($ES_{\bar{r}} = .191$, $p < .01$) returned a weaker effect.

The interesting results for this section concern prior programming, which has often been cited as an enabler of success in programming courses. The presence of prior programming experience was found to have a weak effect on programming performance ($ES_{\bar{r}} = .178$, $p < .01$), whereas specific knowledge of Java and C++ were found to

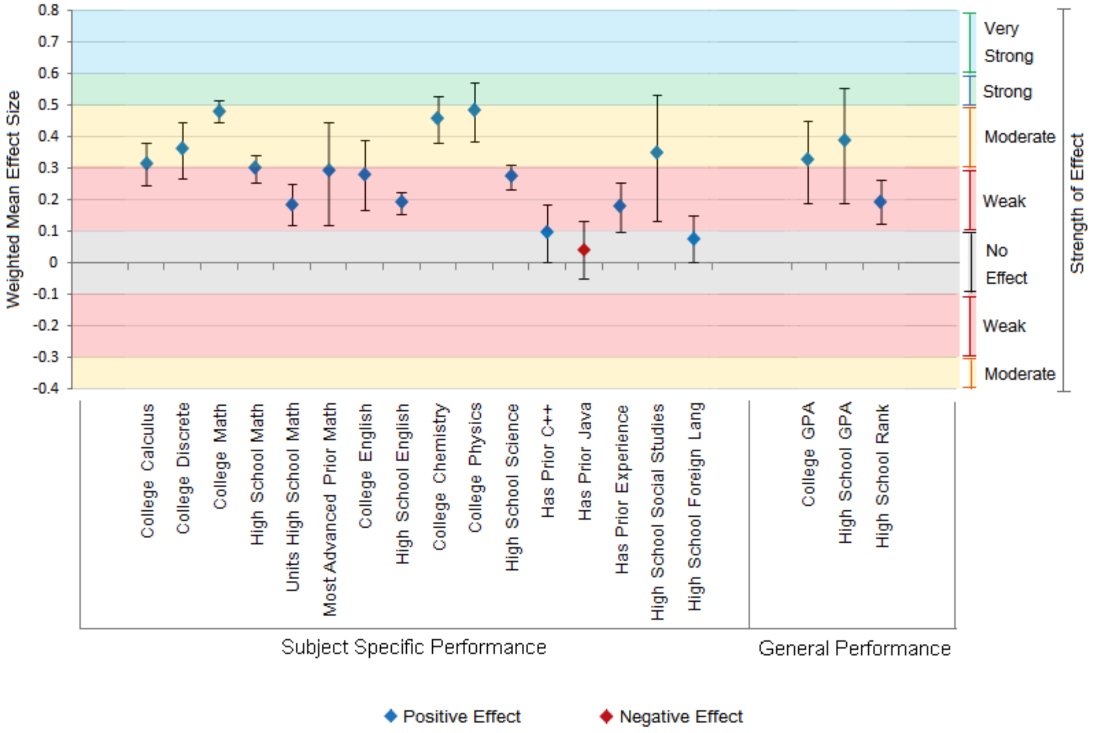


Figure 7.5: Chart Showing Meta-Analysis Results for the Academic Predictors

have no effect. This possibly suggests that the experience students had prior to taking a programming course was insufficient, or that prior experience in one language was not sufficient to transfer to the learning of another. Another explanation is that possibly self-taught programming students who have developed bad practices, and have been penalized on their performance when assessed within a formal course.

General Performance

A total of 30 outcomes describing 3 predictors based upon general subject performance were coded. The mean of the effect sizes which resulted from the meta-analyses of predictors in this subclass, suggests that general subject performance has a moderate effect on programming ($\overline{ES_{\bar{r}}}$ = .302, SD = .098).

High school GPA was found to have the strongest effect on programming ($ES_{\bar{r}}$ = .388, $p < .01$), and a similar moderate effect was found for College GPA ($ES_{\bar{r}}$= .326, $p < .01$). High school rank was only found to have a weak effect on programming ($ES_{\bar{r}}$ = .194, $p < .01$). This is possibly due to the fact this measure depends upon relatively ranking a single cohort, whereas High School GPA is based upon scores obtained on standardized exams which are consistent across different contexts.

Table 7.6: Meta-Analysis Results for the Academic Predictors

	Sample			Meta-Analysis				
Predictor	k	n	SM	$ES_{\bar{r}}$	CI_L	CI_U	z	p
Academic								
Subject Specific Performance								
Math								
College Calculus	4	679	169	.315	.245	.382	8.42	.01
College Discrete	2	363	181	.360	.267	.447	7.12	.01
College Math	15	3183	212	.481	.446	.515	23.06	.01
High School Math	18	3967	220	.301	.256	.343	12.72	.01
Units of High School Math	4	871	217	.185	.120	.249	5.49	.01
Most Advanced Prior Math	4	202	50	.291	.119	.446	3.26	.01
English								
College English	3	607	202	.282	.167	.389	4.68	.01
High School English	16	3484	217	.191	.156	.226	1.45	.01
Science								
College Chemistry	9	2049	227	.502	.435	.564	12.49	.01
College Physics	4	633	158	.484	.384	.572	8.40	.00
High School Science	17	3448	202	.273	.233	.313	12.74	.01
Prior Programming								
Has Prior C++	2	462	231	.096	.004	.186	2.05	.04
Has Prior Java	2	462	231	-.043	-.134	.049	-.92	.36
Has Prior Experience	10	1085	108	.178	.097	.256	4.26	.01
Other Subjects								
High School Social Studies	2	79	39	.349	.134	.532	3.11	.01
High School Foreign Lang.	2	715	357	.076	.002	.148	2.02	.04
General Performance								
College Performance								
College GPA	7	1055	150	.326	.189	.452	4.49	.01
High School Performance								
High School GPA	6	1085	180	.388	.189	.556	3.69	.01
High School Rank	17	3143	184	.194	.124	.262	5.35	.01

Notes

See Table 7.5 for list of abbreviations.

7.5.3 Cognitive Predictors

Results are presented in Table 7.7 and Figure 7.6. In total 76 observed outcomes were coded from 24 studies which described 27 cognitive predictors. Of these predictors, 10 were classified into our theoretical framework as being based upon learning styles, 12 were classified as being based upon learning strategies, and 5 were classified as being based upon specific cognitive style and skills. The mean of the effect sizes which resulted from the meta-analysis suggests that cognitive factors are a weak predictor of programming performance ($\overline{ES_{\bar{r}}} = .274$, $SD = .142$). 12 predictors (45%) were found to have a weak effect, 10 predictors (37%) were found to have a moderate effect, 3 predictors (11%) were found to have no effect, and 2 predictors (7%) were found to have a strong effect.

Learning Styles

A total of 35 outcomes describing 10 predictors based upon learning styles were coded. The overall results for this subclass suggests that learning styles have a weak effect on programming ($\overline{ES_{\bar{r}}} = .159$, $SD = .089$). Although we found the magnitude of the effects to vary based upon the three different learning style models from which the predictors were derived, we also found that the 95% confidence intervals of the 10 effects (Figure 7.6) were narrow, and that all but one of the intervals indicated that the true effects for predictors based upon learning styles are weak at best.

The 4 predictors that were based upon dimensions of Kolb's LSI showed weak effects for the students tendency towards the: accomodator ($ES_{\bar{r}} = \text{-}.167$, $p < .01$), assimilator ($ES_{\bar{r}} = .130$, $p < .01$) and converger dimensions ($ES_{\bar{r}} = .240$, $p < .01$). The 95% confidence intervals for these predictors were narrow (mean range: $CI_L = .119$ to $CI_U = .236$) suggesting that the true effects of learning styles from this model are weak. No effect was found for tendency towards the diverger dimension ($ES_{\bar{r}} = .059$, $p = .53$), but the result was not significant.

The 4 predictors that were based upon dimensions from the Solomon-Felder ILS showed comparable effects. Weak effects were found for the students tendency towards the active ($ES_{\bar{r}} = .198$, $p < .01$) and visual ($ES_{\bar{r}} = .112$, $p = .10$) dimensions. However, no effects were found for either tendency towards the sensing ($ES_{\bar{r}} = .075$, $p = .25$) or sequential dimensions ($ES_{\bar{r}} = .065$, $p = .25$). The 95% confidence intervals for these predictors were also narrow (mean range: $CI_L = .053$ to $CI_U = .227$) suggesting that the true effects in the population of programming students are weak at best.

The 2 predictors that were based upon dimensions from the Gregorc Style Delineator showed stronger effects than those based upon the previous two models. A weak effect was found for the students tendency towards the abstract-random dimension ($ES_{\bar{r}} = .208$, $p < .01$). The only moderate effect for any of the learning style predictors was found for the students tendency towards the concrete-sequential dimension.

Possibly learning styles struggle to effect programming performance as they have not been specifically designed to measure programming knowledge. Also, there has been criticism over the relevance of learning styles to any discipline, suggesting that most learning style models are flawed [39, 150].

Learning Strategies

A total of 25 outcomes describing 12 predictors based upon learning strategies were coded. The overall results for this subclass suggests that learning strategies have a moderate effect on programming ($\overline{ES_{\bar{r}}} = .356$, $SD = .108$). All of the predictors in this subclass were based upon dimensions of the MSLQ. Due to the small number of studies and samples that have explored predictors based upon learning strategies, the 95% confidence intervals of the 12 effects (Figure 7.6) are wider than the predictors that were based upon learning styles.

The predictors based upon expectancy components returned mixed effects. A weak effect was found for the students control of learning beliefs scale ($ES_{\bar{r}} = .223$, $p = .07$), and the strongest effect for all of the cognitive predictors was found for the students self efficacy ($ES_{\bar{r}} = .555$, $p < .01$). This is consistent with literature outside of programming which has suggested that self efficacy is a critical component for learning performance. However the the 95% confidence interval was wide (range: $CI_L = .367$ to $CI_U = .700$), suggesting that the true effect for self efficacy on programming could be anywhere between moderate to very strong.

The predictors that were based upon resource management returned mixed effects. Weak effects were found for peer ($ES_{\bar{r}} = .152$, $p = .34$) and time ($ES_{\bar{r}} = .278$, $p < .05$). Moderate effects were found for effort ($ES_{\bar{r}} = .465$, $p < .01$) and total resource management ($ES_{\bar{r}} = .319$, $p = .10$). This suggests that effectively scheduling, planning, and managing study time can have an effect on programming performance and that students who commit to completing their study goals, even when there are difficulties or distractions are more likely to succeed. This could be interpreted in terms of the students ability to persevere through programming threshold concepts [121].

The 5 predictors that were based upon the value components and metacognitive strategies scales all returned significant moderate effects (range: $ES_{\bar{r}} = .352$ to $ES_{\bar{r}} = .432$, $p < .05$). This suggests that the students perceptions of the course material in terms of interest, importance, and utility, along with the degree to which the student perceives herself to be participating in a task for reasons such as challenge,

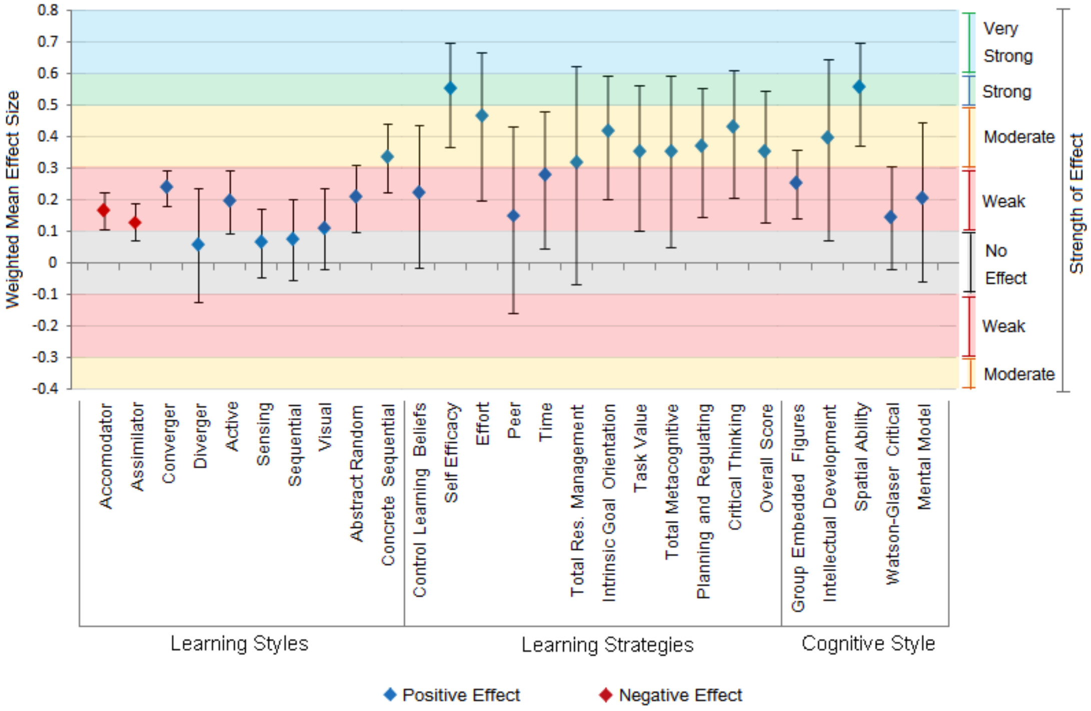


Figure 7.6: Chart Showing Meta-Analysis Results for the Cognitive Predictors

curiosity, mastery has an effect on their programming performance. Additionally the degree to which students report applying previous knowledge to new situations in order to solve problems, reach decisions, or make critical evaluations with respect to standards of excellence also has an effect. This makes sense if we consider that learning programming generally depends upon students mastering a set of progressively more difficult concepts in order to solve increasingly complex programming problems [200].

Cognitive Style and Skills

A total of 16 outcomes describing 5 predictors based upon cognitive styles and skills were coded. The overall results for this subclass suggests that these predictors have a moderate effect on programming ($\overline{ES_{\bar{r}}} = .312$, $SD = .163$). The 95% confidence intervals of the 5 effects (Figure 7.6) were comparable to the previous subclasses.

The predictors within this subclass were found to have a weak effect on performance, apart from level of intellectual development ($ES_{\bar{r}} = .397$, $p < .05$), and spatial ability ($ES_{\bar{r}} = .553$, $p < .01$) which returned a strong effect comparable to the self efficacy component of the learning strategies subclass.

Table 7.7: Meta-Analysis Results for the Cognitive Predictors

	Sample			Meta-Analysis				
Predictor	k	n	SM	$ES_{\bar{r}}$	CI_L	CI_U	z	p
Cognitive								
Learning Styles								
Kolb								
Accomodator	3	998	332	-.167	-.227	-.105	-5.29	.01
Assimilator	4	1072	268	-.130	-.188	-.070	-4.24	.01
Converger	4	1072	268	.240	.182	.296	7.97	.01
Diverger	2	121	60	.059	-.123	.237	.63	.53
Soloman-Felder								
Active	5	1208	241	.198	.095	.296	3.75	.01
Sensing	4	331	82	.065	-.045	.173	1.15	.25
Sequential	4	331	82	.075	-.054	.202	1.15	.25
Visual	4	331	82	.112	-.020	.239	1.66	.10
Gregorc								
Abstract Random	3	386	128	.208	.099	.312	3.70	.01
Concrete Sequential	2	255	127	.337	.223	.442	5.54	.01
Learning Strategies (MSLQ)								
Expectancy Components								
Control Learning Beliefs	2	72	36	.222	-.016	.436	1.83	.07
Self Efficacy	2	72	36	.555	.367	.700	5.08	.01
Resource Management								
Effort	2	73	36	.465	.200	.667	3.28	.01
Peer	2	73	36	.152	-.157	.433	.96	.34
Time	2	73	36	.278	.046	.481	2.33	.02
Total Res. Management	2	73	36	.319	-.069	.623	1.62	.10
Value Components								
Intrinsic Goal Orientation	2	72	36	.418	.201	.596	3.62	.01
Task Value	3	106	35	.356	.102	.566	2.71	.01
Metacognitive Strategies								
Total Metacognitive	2	73	36	.352	.049	.596	2.26	.02
Planning and Regulating	2	73	36	.370	.148	.556	3.18	.01
Critical Thinking	2	73	36	.432	.208	.613	3.61	.01
Overall Measures								
MSLQ Overall Score	2	72	36	.353	.127	.545	3.00	.01
Cognitive Style and Skills								
Cognitive Style								
Group Embedded Figures	4	1172	293	.253	.142	.357	4.39	.01
Specific Cognitive Skills								
Intellectual Development	4	455	113	.397	.073	.646	2.37	.02
Spatial Ability	2	76	38	.553	.375	.697	5.25	.01
Watson-Glaser Critical	3	145	48	.146	-.021	.305	1.72	.09
Mental Model	3	348	116	.207	-.060	.446	1.52	.13

Notes
See Table 7.5 for list of abbreviations.

7.5.4 Psychological Predictors

Results are presented in Table 7.8 and Figure 7.7. In total 99 observed outcomes were coded from 20 studies which described 35 psychological predictors. Of these predictors, 20 were classified into our theoretical framework as being based upon affective traits, 11 were classified as being based upon personality, and 4 were classified as being based upon behavioural traits. The mean of the effect sizes which resulted from the meta-analysis suggests that psychological factors are a weak predictor of programming performance ($\overline{ES_{\bar{r}}} = .151$, $SD = .103$). 19 predictors (54%) were found to have a weak effect, 13 predictors (37%) were found to have no effect, and only 3 predictors (9%) were found to have a moderate effect.

Affective Traits

A total of 64 outcomes describing 20 predictors based upon affective traits were coded. The mean of the effect sizes which resulted from the meta-analyses of predictors in this subclass, suggests that affective traits have a weak effect on programming performance ($\overline{ES_{\bar{r}}} = .137$, $SD = .116$). The magnitude of effects were found to vary based upon the three affective subclasses, with the 13 effects based upon self perceived abilities producing the narrowest confidence intervals of the entire subclass (Figure 7.7), indicating that the true effects for predictors based upon self perceived abilities are weak at best.

The 4 predictors that were based upon dimensions of Weiner's attributional style model showed little relations to programming. Weak effects were found for the: ability ($ES_{\bar{r}} = 115$, $p < .05$), effort ($ES_{\bar{r}} = 107$, $p < .05$), and luck ($ES_{\bar{r}} =$ -.184, $p < .01$) dimensions. No effect was found for attributions to task difficulty ($ES_{\bar{r}} =$ -.063, $p =$ .19) on programming performance. The 95% confidence intervals for these predictors were narrow (mean range: $CI_L = .039$ to $CI_U = .209$) suggesting that the true effects in the population of programming students are weak. Again, this instrument was not designed to predict programming performance, possibly resulting in these weak effects.

The 13 predictors that were based upon students self perceived abilities also showed little relations to programming. Of these 13 predictors, 9 predictors: leadership, artistic, management, mechanical, personal relations, public speaking, reading, spatial ability, and writing ability, all showed no effects on programming performance ($ES_{\bar{r}} <$ .10). The 95% confidence intervals for these predictors were narrow (mean range: CI_L = .090 to $CI_U = .177$) suggesting that the true effects in the population of programming students are weak at best. It is interesting to note that the result for self perceived spatial ability ($ES_{\bar{r}} = .084$, $p = .07$) contradicts the result for formally tested spatial ability in the cognitive predictors subsection ($ES_{\bar{r}} = .553$, $p < .01$). This could be due to a lack of student understanding of the meaning of the term. However, the result for self perceived spatial ability was not significant. The results for self perceived math ($ES_{\bar{r}} = .271$, $p < .01$), science ($ES_{\bar{r}} = .249$, $p < .01$), and problem solving ($ES_{\bar{r}} = .194$, $p < .01$) abilities are consistent with the results previously presented in the academic predictors subsection.

The 3 predictors which were based upon various instruments designed to measure self efficacy were found to show weak relations to programming, although these relations were stronger than the previous two affective subclasses. Both scores on the Computer Programmer Self Efficacy Scale ($ES_{\bar{r}} = .219$, $p < .01$) and Rosenbergs Self Esteem Scale ($ES_{\bar{r}} = .270$, $p < .05$) were shown to have a weak effect on programming performance. Comfort level ($ES_{\bar{r}} = .472$, $p < .01$) was shown to have a strong effect on performance. These results are consistent with the previous strong result on self efficacy and learning strategies that reported in the cognitive predictors subsection ($ES_{\bar{r}} = .555$, $p < .01$), suggesting that a certain element of self belief and confidence is a necessary attribute for success in programming.

Personality

A total of 26 outcomes describing 11 predictors based upon personality were coded. The mean of the effect sizes which resulted from the meta-analyses of predictors in this subclass, suggests that personality has a weak effect on programming performance ($\overline{ES_{\bar{r}}} = .179$, $SD = .099$). The magnitude of effects were found to vary based upon the two instruments used to measure personality attributes, with the 7 effects based upon the Thurstone Temperament Instrument showing stronger effects on programming ($\overline{ES_{\bar{r}}} = .229$, $SD = .089$) than those 4 effects that were based upon the Myers-Briggs Type Indicator ($\overline{ES_{\bar{r}}} = .090$, $SD = .026$).

For the predictors that were based upon dimensions of the Thurstone Instrument, only the students tendency towards the sociable ($ES_{\bar{r}} = -.303$, $p < .01$) and impulsive ($ES_{\bar{r}} = -.379$, $p < .01$) dimensions showed moderate effects on programming performance. The remaining dimensions all showed weak effects. This suggests that students who are patient, plan, and persevere with programming are likely to succeed.

For the predictors that were based upon dimensions of Myers-Briggs Type Indicator, only borderline weak effects were found for the students tendency towards the judgement ($ES_{\bar{r}} = -.121$, $p = .07$) and sensing dimensions ($ES_{\bar{r}} = .100$, $p = .14$). Neither of these effects were significant, but they do suggest that there are little relations between the personality aspects measured by Myers-Briggs Instrument and programming, given that all four dimensions only showed borderline weak effects.

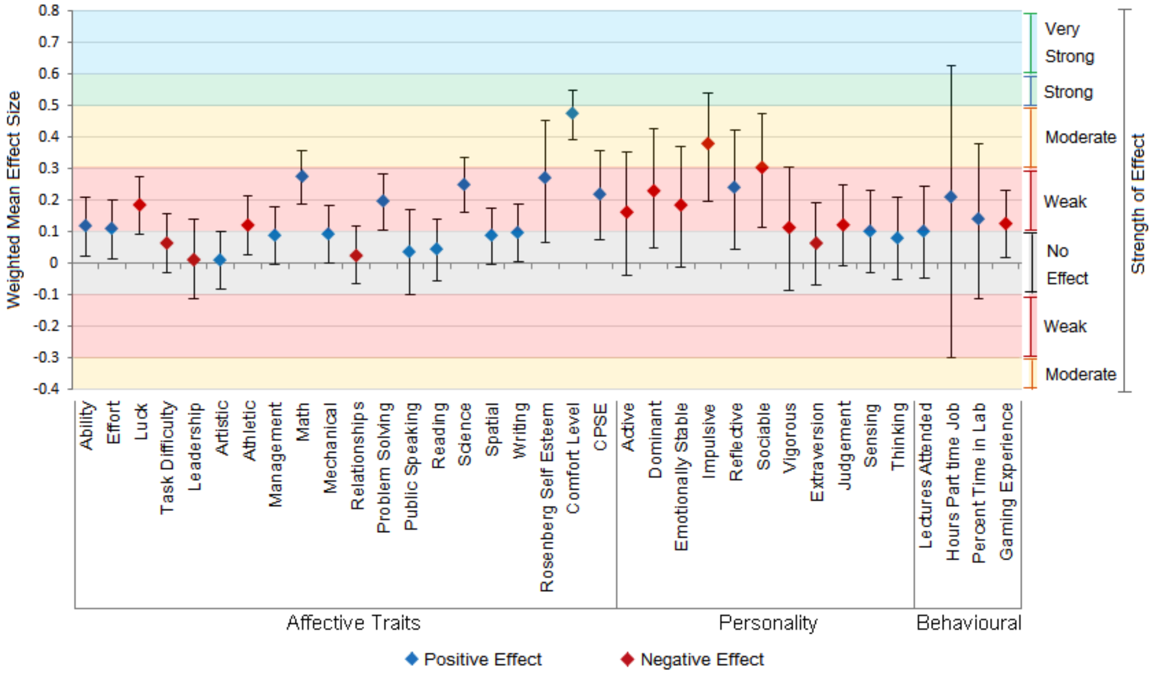


Figure 7.7: Chart Showing Meta-Analysis Results for the Psychological Predictors

These results suggest that personality traits are only weakly related to programming performance, but none of the personality instruments were designed explicitly for the purpose of measuring programming performance. It is also possible that students are able to succeed in programming regardless as to their personality type.

Behavioural Traits

A total of 21 outcomes describing 8 predictors based upon behavioural traits were coded. No effect was found for the number of lectures attended ($ES_{\bar{r}}$ = .099, p = .19) on programming performance. A weak effect was found for percentage of time spent in the lab ($ES_{\bar{r}}$ = .139, p = .28), however the 95% confidence interval of this result was wide, and this conclusion cannot be generalised to a true effect with reasonable confidence. Hours spent working in a part time job whilst indicating a weak effect on performance ($ES_{\bar{r}}$ = .210, p = .42) also suffered from a wide confidence interval. The only significant result of this subclass was found for computer gaming experience ($ES_{\bar{r}}$ = -.125, $p < .05$), which showed a weak negative effect on programming performance. However the 95% confidence interval suggests that there could be no true effects.

Table 7.8: Meta-Analysis Results for the Psychological Predictors

	Sample			Meta-Analysis				
Predictor	k	n	SM	$ES_{\bar{r}}$	CI_L	CI_U	z	p
Psychological								
Affective Traits								
Attributions								
Ability	4	438	109	.115	.020	.207	2.37	.02
Effort	4	438	109	.107	.012	.199	2.21	.03
Luck	4	438	109	-.184	-.274	-.091	-3.85	.01
Task Difficulty	4	438	109	-.063	-.157	.032	-1.30	.19
Self Perceived Abilities								
Leadership	3	465	155	-.010	-.136	.116	-.16	.88
Artistic	3	465	155	.009	-.083	.100	.19	.85
Athletic	3	465	155	-.120	-.214	-.024	-2.45	.01
Management	3	465	155	.086	-.006	.176	1.84	.07
Math	3	465	155	.271	.184	.354	5.94	.00
Mechanical	3	465	155	.091	-.001	.181	1.95	.05
Relations	3	465	155	-.023	-.115	.068	-.50	.62
Problem Solving	3	465	155	.194	.104	.280	4.18	.01
Public Speaking	3	465	155	.034	-.102	.169	.49	.62
Reading	3	465	155	.043	-.056	.140	.85	.40
Science	3	465	155	.249	.161	.333	5.43	.01
Spatial	3	465	155	.084	-.008	.174	1.80	.07
Writing	3	465	155	.096	.004	.186	2.05	.04
Self Efficacy								
Rosenberg Self Esteem	2	93	46	.270	.066	.452	2.58	.01
Comfort Level	3	385	128	.472	.390	.547	9.95	.01
CPSE	4	526	131	.219	.072	.356	2.91	.01
Personality								
Thurstone Temperament								
Active	2	100	50	-.161	-.349	.040	-1.58	.12
Dominant	2	100	50	-.229	-.410	-.031	-2.26	.02
Emotionally Stable	2	100	50	-.184	-.370	.016	-1.80	.07
Impulsive	2	100	50	-.379	-.538	-.194	-3.87	.01
Reflective	2	100	50	.240	.043	.419	2.37	.02
Sociable	2	100	50	-.303	-.474	-.110	-3.03	.01
Vigorous	2	100	50	-.112	-.305	.089	-1.09	.27
Myers Briggs Type Indicator								
Extraversion	3	229	76	-.063	-.192	.069	-.93	.35
Judgement	3	229	76	-.121	-.248	.011	-1.80	.07
Sensing	3	229	76	.100	-.032	.229	1.49	.14
Thinking	3	229	76	.077	-.055	.207	1.15	.25
Behavioural Traits								
Lectures Attended	2	178	89	.099	-.050	.244	1.30	.19
Percentage Time in Lab	2	65	32	.139	-.114	.376	1.08	.28
Part Time Job Hours	2	630	315	.210	-.300	.626	.80	.42
Gaming Experience	3	335	111	-.125	-.230	-.017	-2.27	.02

Notes
See Table 7.5 for list of abbreviations.

7.5.5 Programming Behaviour Predictors

Results are presented in Table 7.9 and Figure 7.8. In total 43 observed outcomes were coded from 5 studies which described 17 predictors based upon programming behaviour. Of these predictors, 8 were classified into our theoretical framework as being based upon event based metrics, 6 were classified as being based upon time based metrics, and 3 were classified as being based upon algorithmic based metrics.

The mean of the effect sizes which resulted from the meta-analysis suggests that factors based upon programming behaviour are a moderate predictor of programming performance ($\overline{ES_{\bar{r}}}$ = .378, SD = .165). 7 predictors (40%) were found to have a moderate effect, 4 predictors (24%) were found to have a weak effect, 5 predictors (29%) were found to have a strong or very strong effect, and 1 predictor (6%) was found to have no effect. This is perhaps unsurprising given that the predictors in this class have been explicitly designed for the purpose of predicting programming performance, and are based off data describing the students own programming process.

Event Based Metrics

The mean of the effect sizes which resulted from the meta-analyses of predictors in this subclass, suggests event based metrics have a moderate effect on programming performance ($\overline{ES_{\bar{r}}}$ = .351, SD = .101).

Considering the different types of pairings that students produced over a session (as a percentage of all pairings), a moderate negative effect on performance was found for the percentage of pairings representing two successive errors ($ES_{\bar{r}}$ = -.443, p < .01), and a strong negative effect was found for the percentage of pairings which represented repeated errors ($ES_{\bar{r}}$ = -.526, p < .01). The 95% confidence interval of this result (range: CI_L = -.392 to CI_U = -.639) suggested that the true effect in the population could be moderate to very strong. Similar effects were found when considering raw frequency counts. A moderate negative effect on performance for total number of repeated errors ($ES_{\bar{r}}$ = -.317, p < .01), and for the total number of pairings which contained at least one error ($ES_{\bar{r}}$ = -.313, p < .01).

In contrast, the percentage of pairings representing two successful compilations was found to have a moderate positive effect on performance ($ES_{\bar{r}}$ = .414, p < .01). This suggests that in general, weaker programming students are characterised by producing a greater percentage of repeated errors than stronger students, possibly indicating that they are unable to overcome the syntax barrier to produce compilable code. This is problematic, given that until novices can overcome the syntax barrier, they have no means of obtaining feedback on the semantic correctness of their code at runtime.

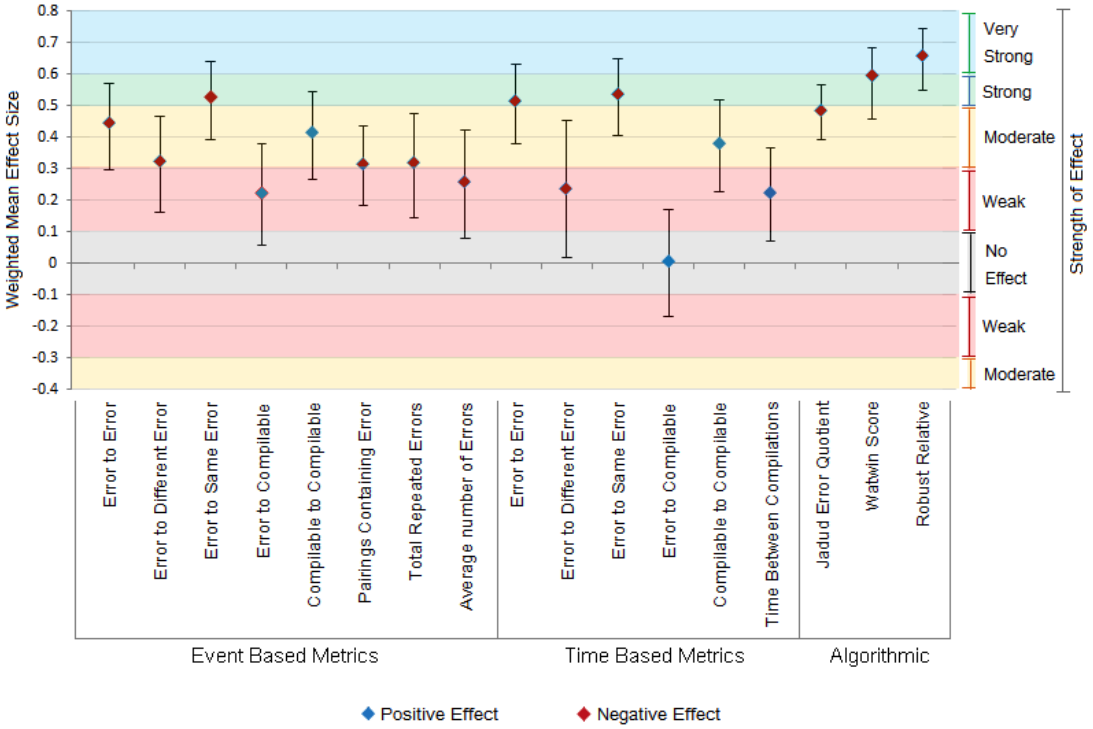


Figure 7.8: Chart Showing Meta-Analysis Results for the Programming Behaviour Predictors

Time Based Metrics

The mean of the effect sizes which resulted from the meta-analyses of predictors in this subclass, suggests time based metrics have a moderate effect on programming performance ($\overline{ES_{\bar{r}}} = .333$, $SD = .220$).

Considering the amount of time students spent working on different types of pairings (as a percentage of their total lab time), results comparable to the event based metrics were found. A strong negative effect was found for the percentage of lab time that students spent working on pairings in the form of two successive errors ($ES_{\bar{r}} =$ -.513, $p < .01$), and a strong effect was found for the percentage lab time students spent working on two repeated errors ($ES_{\bar{r}} =$ -.535, $p < .01$). As with the previous subclass, a moderate positive effect was also found for the percentage of lab time students spent working on pairings representing two successful compilations ($ES_{\bar{r}} = .380$, $p < .01$).

This suggests that not only are weaker students characterised by having a greater percentage of repeated errors than stronger students, but also that weaker students will spend a larger percentage of their lab time trying to overcome these errors.

Table 7.9: Meta-Analysis Results for the Programming Behaviour Predictors

	Sample			Meta-Analysis				
Predictor	k	n	SM	$ES_{\bar{r}}$	CI_L	CI_U	z	p
Programming Behaviour								
Event Based Metrics								
Percentage Based Metrics								
Error to Error	3	141	47	-.443	-.569	-.296	-5.47	.01
Error to Different Error	3	141	47	-.322	-.465	-.162	-3.83	.01
Error to Same Error	3	141	47	-.526	-.639	-.392	-6.72	.01
Error to Success	3	141	47	.222	.055	.377	2.59	.01
Success to Success	3	141	47	.414	.263	.545	5.06	.01
Frequency Based Metrics								
Pairings Containing an Error	4	246	61	-.313	-.432	-.183	-4.58	.01
Total Repeated Errors	3	122	40	-.317	-.472	-.143	-3.49	.01
Average number of Errors	3	122	40	-.257	-.420	-.078	-2.79	.01
Time Based Metrics								
Percentage Based Metrics								
Error to Error	3	141	47	-.513	-.628	-.377	-6.52	.01
Error to Different Error	3	141	47	-.235	-.453	-.017	-1.10	.27
Error to Same Error	3	141	47	-.535	-.646	-.403	-6.87	.01
Error to Success	3	141	47	.020	-.167	.171	.03	.98
Success to Success	3	141	47	.380	.225	.516	4.59	.01
Frequency Based Metrics								
Time Between Compilations	2	164	82	.221	.069	.363	2.83	.01
Algorithmic Based Metrics								
Overall Quantification								
Error Quotient	5	321	64	-.482	-.563	-.392	-9.20	.01
Watwin Score	3	141	47	-.582	-.683	-.458	-7.64	.01
Robust Relative	3	141	47	-.657	-.743	-.549	-9.05	.01

Notes
See Table 7.5 for list of abbreviations.

Algorithmic Based Metrics

The mean of the effect sizes which resulted from the meta-analyses of predictors in this subclass, suggests algorithmic scoring methods have a strong effect on programming performance ($\overline{ES_{\bar{r}}}$ = .573, SD = .088), possibly as they encapsulate several aspects of programming behaviour into a single performance predictor. The Error Quotient was found to have a moderate effect on programming ($ES_{\bar{r}}$ = -.477, $p < .01$), and Watwin Score was found to have a strong effect on programming ($ES_{\bar{r}}$ = -.630, $p < .01$). The Robust Relative scores were found to have a very strong effect on programming ($ES_{\bar{r}}$ = -.657, $p < .01$), yielding the strongest effect of any of the 116 predictors.

7.5.6 Demographic Predictors

Results are presented in Table 7.10 and Figure 7.9. In total 21 observed outcomes were coded from 17 studies which described 2 predictors based upon demographic factors. The mean of the effect sizes which resulted from the meta-analysis suggests that factors based upon demographic factors are not predictive of programming performance ($\overline{ES_{\bar{r}}}$ = .015, SD = .008). No effect was found for either gender ($ES_{\bar{r}}$ = -.009, p = .74) or age ($ES_{\bar{r}}$ = .020, p = .70) on programming. The 95% confidence intervals were narrow (range: CI_L = -.083 to CI_U = .123), suggesting that whilst the true effect of age may be borderline weak, gender has no effect on programming performance.

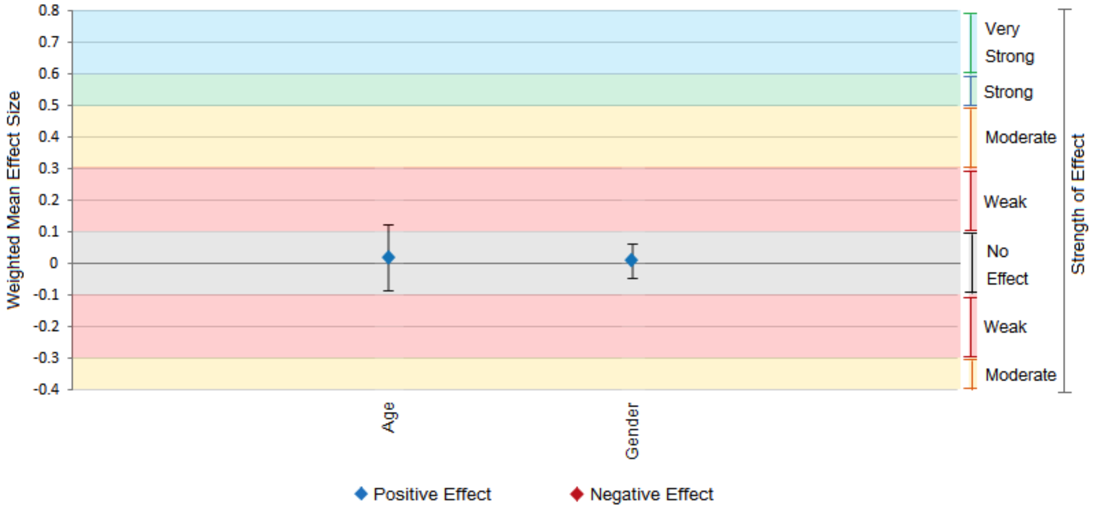


Figure 7.9: Chart Showing Meta-Analysis Results for the Demographic Predictors

Table 7.10: Meta-Analysis Results for the Aptitude Predictors

	Sample			Meta-Analysis				
Predictor	k	n	SM	$ES_{\bar{r}}$	CI_L	CI_U	z	p
Age	7	1005	143	.020	-.083	.123	.39	.70
Gender	14	2656	189	.009	-.045	.064	.33	.74

Notes
See Table 7.5 for list of abbreviations.

Table 7.11: Results of the Moderator Analysis

Heterogeneity Tests					Possible Moderators								
Predictor	k	Q_t	p	I^2	Year	Language	Gender	Grade Level	Introductory	Cohort Size	Criterion	Composition	Quality
Heterogeneous													
IBM PAT Arithmetic	6	16.4	.01	55.1	.	.	.	.	.	.	.	.	.
Wolfe PAT Overall	5	15.1	.01	71.8	.	.	.	.	.	.	.	.	.
Age	7	16.2	.01	4.3	.	.	.	.	.	.	.	.	.
High School Math	18	37.1	.01	49.2	●	.	.	.	.	●	●	.	.
SAT Math	23	43.6	.01	46.8	.	.	.	.	.	●	.	.	.
College Chemistry	9	38.5	.01	69.9	.	.	.	.	.	.	●	.	.
College GPA	7	23.6	.01	74.0	.	●	.	.	.	.	.	.	.
High School GPA	6	39.1	.01	78.4	.	.	.	.	.	.	.	.	.
High School Rank	17	51.9	.01	73.0	.	.	.	.	.	●	.	.	.

Notes

● Indicates that the effect is significantly moderated by the possible moderator, $p < .10$.

7.5.7 Moderator Analysis

Results of the moderator analysis are shown in Table 7.11. In total 18 predictors satisfied the $k \geq 5$ criteria, and of these predictors 9 were found to be heterogeneous (Q_t, $p < .10$). The results of the analysis showed that neither year, gender, grade composition, or grade level of the institution were found to moderate the effects.

In one case, the programming language was found to marginally moderate the effects ($p < .10$), and in three cases the cohort size was found to marginally moderate the effects ($p < .10$). These results are consistent with the findings presented in Chapter 4 of this thesis, which showed marginally significant differences in the pass rates of CS1 courses based upon these two moderators. The criterion variable was also found to moderate the failure rates in two cases ($p < .10$). The cause of variation for 4 of the 9 predictors however remains unknown, suggesting that there are additional uncoded aspects which may moderate the performance of predictors in different contexts.

It is interesting to note that none of the effects were significantly moderated ($p < .05$) by the coding criteria, which may suggest that other predictors are unaffected by the nine coded aspects. However, unless further studies are conducted, there is no method of confirming this hypothesis.

7.6 Discussion

Overall results presenting the strongest 15 predictors are shown in Table 7.12, the weakest 15 predictors are shown in Table 7.13, and the mean strengths of effect sizes found for each class are shown in Table 7.14.

To answer RQ2, out of the top 15 predictors which were identified by this study, we found that 7 predictors (47%) were based upon programming behaviour, 4 predictors (27%) were based upon cognitive factors, 3 predictors (20%) were based upon academic factors, and 1 predictor (6%) was based upon aptitude testing. This suggests that the strongest predictors of programming performance found over the past fifty years are those that are based upon programming behaviour. This is perhaps unsurprising, as these predictors are based directly upon the factors that are indirectly measured in programming assessments (e.g. whether the student can produce a compilable solution), and are capable of directly reflecting aspects of the students programming knowledge. Self efficacy and comfort level were also among the top predictors. This suggests that an element of self belief, or self concept is necessary for programming. This can be recognised in terms of students willingness to persevere in experimenting with their individual solutions to complex programming problems. The presence of college math, physics, and chemistry, among the top predictors suggests that a level of scientific maturity is required to be a successful programmer.

To answer RQ3, out of the weakest 15 predictors which were identified by this study, we found that 10 predictors (67%) were based upon either demographic or psychological factors. Additionally, 2 cognitive predictors based upon learning styles were among the weakest predictors. These results suggest that self perceived abilities, learning styles, and personality traits have little relevance for programming success. Age (ranked 111) and gender (ranked 114) both ranked among the worst predictors identified over the past fifty years, suggesting that neither of these factors have any influence on the students ability to understand programming.

To answer RQ4, we found that the factors which were based upon aspects of programming behaviour were the most predictive of performance. Out of the 17 predictors in this class, 11 predictors (65%) were found to have moderate to very strong effects on performance. The cognitive and academic predictors were found to have a comparable effect on programming. 82% of cognitive predictors were found to have a weak/moderate effect, and 84% of academic predictors were found to have a weak/moderate effect on programming. The majority of aptitude predictors (63%) were weak, over one third of the psychological predictors (37%) were found to have no effect, and no effects were found for any of the demographic predictors.

Table 7.12: The Overall Strongest 15 Predictors found by this Meta-Analysis

Predictor			Meta-Analysis					
	Predictor	Class	k	n	$ES_{\bar{r}}$	CI_L	CI_U	p
1	Robust Relative	Prog. Behav.	3	141	-.657	-.743	-.549	.01
2	Watwin Score	Prog. Behav.	3	141	-.582	-.683	-.458	.01
3	Wolfe PAT	Aptitude	5	360	.56	.388	.694	.01
4	Self Efficacy	Cognitive	2	72	.555	.367	.700	.01
5	Spatial Ability	Cognitive	2	76	.553	.375	.697	.01
6	Error to Error (Time)	Prog. Behav.	3	141	-.535	-.646	-.403	.01
7	Error to Same Error (Event)	Prog. Behav.	3	141	-.526	-.639	-.392	.01
8	Error to Same Error (Time)	Prog. Behav.	3	141	-.513	-.628	-.377	.01
9	College Chemistry	Academic	9	2049	.502	.435	.564	.01
10	College Physics	Academic	4	633	.484	.384	.572	.01
11	Error Quotient	Prog. Behav.	5	321	-.482	-.563	-.392	.01
12	College Math	Academic	15	3183	.481	.446	.515	.01
13	Comfort Level	Cognitive	3	385	.472	.390	.547	.01
14	Effort (Res. Management)	Cognitive	2	73	.465	.200	.667	.01
15	Error to Error (Event)	Prog. Behav.	3	141	-.443	-.569	-.296	.01

Notes
See Table 7.5 for list of abbreviations.

Table 7.13: The Overall Weakest 15 Predictors found by this Meta-Analysis

Predictor			Meta-Analysis					
	Predictor	Class	k	n	$ES_{\bar{r}}$	CI_L	CI_U	p
101	Sequential	Cognitive	4	331	.075	-.054	.202	.25
102	Sensing	Psychological	4	331	.065	-.045	.173	.25
103	Task Difficulty (Attribution)	Psychological	4	438	-.063	-.157	.032	.19
104	Extraversion	Psychological	3	229	-.063	-.192	.069	.35
105	Diverger	Cognitive	2	121	.059	-.123	.237	.53
106	ACT English	Aptitude	2	79	-.047	-.270	.180	.69
107	SRA Verbal	Aptitude	2	120	.046	-.290	.372	.79
108	Reading (Self Perc.)	Psychological	3	465	.043	-.056	.140	.40
109	Has Prior Java	Academic	2	462	-.043	-.134	.049	.36
110	Public Speaking (Self Perc.)	Psychological	3	465	.034	-.102	.169	.62
111	Relations (Self Perc.)	Psychological	3	465	-.023	-.115	.068	.62
112	Age	Demographic	7	1005	.020	-.083	.123	.70
113	Leadership (Self Perc.)	Psychological	3	465	-.010	-.136	.116	.88
114	Error to Success (Time)	Prog. Behav.	3	141	-.020	-.167	.171	.98
115	Gender	Demographic	14	2656	.009	-.045	.064	.74
116	Artistic (Self Perc.)	Psychological	3	465	.009	-.083	.100	.85

Notes
See Table 7.5 for list of abbreviations.

Table 7.14: Summary of effect sizes by class sorted by mean strength ($\overline{ES_{\bar{r}}}$)

Predictor Class					Predictor Strengths				
	Predictor Class	TN	$\overline{ES_{\bar{r}}}$	SD	None	Weak	Moderate	Strong	V. Str.
1	Prog. Behaviour	17	.378	.101	1 6%	4 24%	7 40%	4 24%	1 6%
2	Academic	19	.275	.143	2 11%	8 42%	8 42%	1 5%	. .
3	Cognitive	27	.274	.142	3 11%	12 45%	10 37%	2 7%	. .
4	Aptitude	16	.239	.136	2 13%	10 63%	3 19%	1 5%	. .
5	Psychological	35	.151	.103	13 37%	19 54%	3 9%	. .	. .
6	Demographic	2	.015	.008	2 100%	. .	. .	. .	. .

Notes
TN: number of predictors included in each class of the theoretical framework.

7.6.1 Practical Implications for Teaching

This study has provided much needed quantitative evidence on the influence of different factors on programming performance and has synthesised over fifty years of conflicting quantitative results. Contrary to folklore, neither gender nor age were found to have an effect on programming. Whilst performance in scientific subjects at college level were found to have an effect on programming, the corresponding performance in scientific subjects at high school level were found to have a much smaller effect. Commonly applied aptitude tests were found to be only weakly associated with performance. Factors based upon learning styles and personality yielded weak to no effects. The strongest effects were found for predictors based upon programming behaviour.

The practical implications for teaching are that successful programming students are characterised by high levels of self efficacy, have performed well in college level science subjects, and exhibit traits of desirable programming behaviour. This would imply that in order to rapidly identify at risk students before assessments take place, pre-course screening could be performed by using test based on self efficacy (e.g. MSLQ [14]), and then to monitor the students programming behaviour by using an overall algorithmic metric (e.g. Robust Relative Score [196]). We hypothesise that a regression model based upon these factors would provide a reasonable performance predictor.

7.6.2 Implications and Applications of the Framework

Figure 7.10 shows the resulting theoretical framework with the strongest fifteen predictors of programming performance which were identified by this meta-analysis. The primary purpose of the framework was to consolidate the meta-analysis findings into a logical hierarchy. We now close this discussion by considering possible practical applications for the framework.

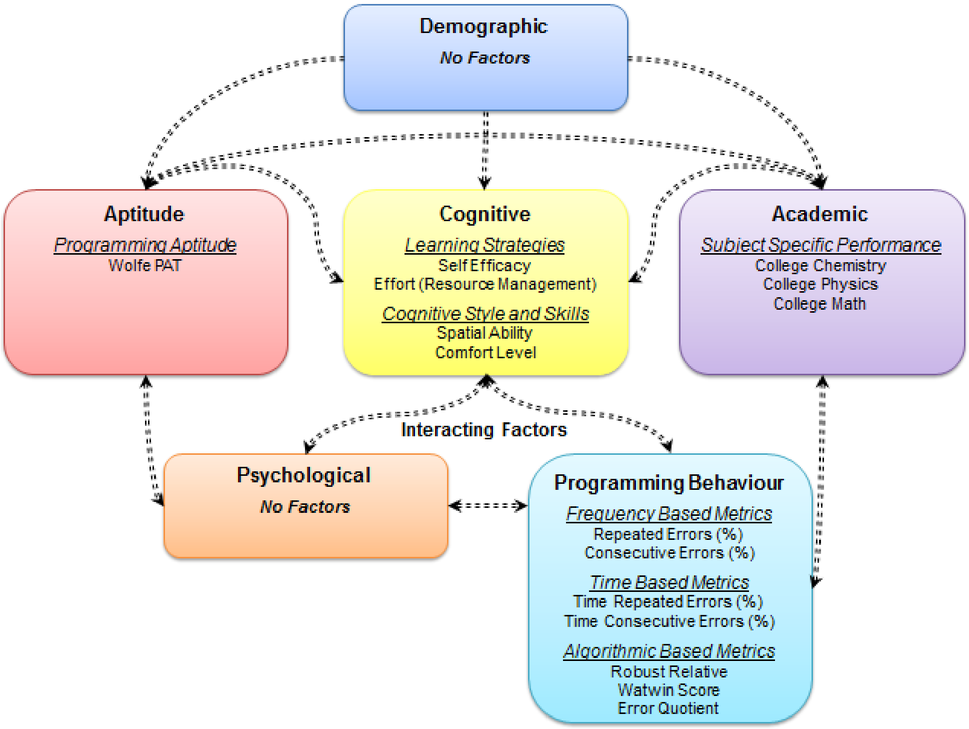


Figure 7.10: Updated theoretical framework, displaying the strongest fifteen predictors found by this meta-analysis. Interacting factors were determined by related studies in the literature, and are shown on the diagram by using dashed arrows.

One application could be to apply the theoretical framework to other computer science modules. Programming is not just a single skill, which is required by a single course. Many other modules as part of a computer science degree require programming skills, (e.g. Networks) and it would be interesting to determine whether the fifteen factors on display can be applied to predict performance in these modules.

Additionally, the framework could be applied by other researchers to continue with systematic development of their own predictors in the domain of computer programming. The framework has highlighted certain "key types" of predictors which are worthy of exploration, and it is likely there are undiscovered predictors in each of the strongest categories which may support acquisition of computing knowledge.

7.6.3 Threats to Validity

Although this study identified 482 observed outcomes for inclusion in the meta-analysis, a number of validity concerns can be raised, which are discussed in this section.

Firstly, we note the unavoidable limitation that the assessment criteria and learning objectives will vary between different studies. Studies within the UK generally defined a pass boundary at 40%, whereas other studies from the USA defined a pass boundary at 50%. As 31 of the included studies measured performance using overall letter grade, concerns can be raised over such differing grade boundaries of different studies. Therefore this study unavoidably has to assume that a consistent notion of programming performance exists and holds valid across the different teaching contexts. But, we note that this a common limitation of studies of this nature (e.g. [193, 197]) and that meta-analysis techniques are designed to take into account such limitations.

Secondly, the teaching approaches that were used for different studies were rarely explicitly stated in a very detailed fashion. For the most part, it was implied from the articles that the current approach was one based upon a traditional lecture and lab approach, but this is not certain for all studies. It is possible that the teaching approach could account unexplained variance in the moderator analysis.

Thirdly, although the number of observed outcomes is high at 482, the final number of selected studies, $k = 80$ is low, especially when considering that introductory programming courses have been studied for decades. A major concern in our case is the possibility of selective reporting, but despite looking for sources of *gray literature* via generalized searches, our efforts were largely unsuccessful. It is possible that this work suffers from a publication bias, a common limitation of meta-analyses [31, 71].

Fourthly, 145 observed outcomes were dropped from the dataset as they represented predictors which had only been examined by single studies, and therefore meta-analysis techniques could not be applied. It is possible that these predictors may perform similarly to some of the ones included in this analysis, but, there is no way of confirming this hypothesis until the predictors have been trialled in different contexts.

Fifthly, there is the fundamental limitation of meta-analysis in that qualitative evidence cannot be included. A number of studies were also dropped as they failed to provide sufficient statistical data (e.g. [168,169]). As with the previous case, if sufficient data was provided then these predictors could have been included in the analysis.

Sixthly, although the number of students contributing observations for each predictor was large ($n \geq 60$), the actual number of studies were mostly low, and only 18 predictors reported having greater than 5 observations. This resulted in wide confidence intervals for multiple predictors, but only by trialling predictors in more contexts can the bounds be tightened and a more precise estimate of the effect be calculated.

7.7 Conclusion

Previous research has been limited by both a lack of verification of predictors in different teaching contexts, and the conflicting results which are commonly reported when verifications take place. Due to these issues there was no general consensus among researchers on which factors support students in developing programming skills. Without understanding which factors are the most important for success, it becomes extremely difficult to design and implement any effective pedagogical tools or strategies which can be applied to identify and support the weakest students.

In this chapter, we have attempted to resolve over fifty years of conflicting findings on the relevance of different factors for programming performance by performing a substantial meta-analysis. This has enabled the synthesis of findings from multiple studies which have examined the same predictor in different teaching contexts, allowing researchers to make generalisable conclusions on the relevance of different factors for programming which are independent of aspects of their teaching contexts (e.g. programming language taught) and aspects of their students (e.g. learning styles).

In total, 482 previous research results describing the relations between 116 predictors and programming performance were identified and classified into a theoretical framework. These included predictors that were based upon aptitude, psychological factors, cognitive factors, academic background, demographic factors and programming behaviour. The results suggest that the strongest predictors of programming performance which have been found over the past fifty years are those based upon programming behaviour. However, several of the traditionally measured attributes also featured among the strongest predictors. The presence of self-efficacy and comfort level suggests that an element of self belief, or self concept is necessary for programming success. The presence of college math, physics, and chemistry among the top predictors reinforces anecdotal evidence that a level of scientific maturity, or scientific thinking, is a necessary skill to become a successful programmer. Two thirds of the weakest predictors were based upon demographic and psychological factors. These results showed that self perceived abilities, learning styles, personality traits, age, and gender, have little relevance for programming success.

The implication from this study is that to rapidly identify at risk students before assessments take place, an initial screening can be performed by considering the students academic background and results from a test based upon self efficacy. Following this, the predictors based upon programming behaviour, in particular the Robust Relative algorithm, could be applied to autonomously monitor the students programming behaviours, alerting instructors of at risk students or providing interventions.

Chapter 8

Conclusion and Future Works

This thesis has detailed four studies which were performed to identify factors which can influence programming success. These factors included aspects of the teaching context, traditionally explored learning theories, and data-driven predictors based upon aspects of programming behaviour. This thesis then detailed a meta-analysis of fifty years of research which was performed to synthesise the conflicting findings on the relevance of different factors for the learning of programming, and has identified a subset of predictors which have the largest influence on programming performance. This concluding chapter summarises the contributions made and provides suggestions for possible future directions of the work.

8.1 Contributions

The studies performed in this thesis make several novel contributions to the body of research on programming education, and predictors of programming performance.

Quantitative Evidence on the Failure Rates in Programming Courses

The first major contribution of this thesis was to provide substantial quantitative evidence on the worldwide pass and failure rates of programming courses. Before the research was conducted in this thesis, only a single study to date [13] had attempted to provide any quantitative evidence to support the often cited claim, that high failure rates plagued programming courses on a worldwide scale. This study was limited, as it only considered 63 failure rates from a single point in time.

To address these shortcomings we chose to exploit an unconsidered source of failure rate data, by performing a systematic review of the research on introductory programming education. Analysing this data, this thesis made three novel contributions. First, this thesis provided an estimate of the worldwide pass and failure rates of programming courses (Section 4.4.1). Secondly, this thesis then explored whether failure rates had changed over time (Section 4.4.2). Thirdly, this thesis explored the impact of different aspects of the teaching context on the failure rates (Section 4.4.3).

Investigation into Predictors based upon Traditional Learning Theories

The second major contribution of this thesis was to evaluate the context dependency of predictors which were based upon traditional learning theories. Verification studies are not performed very often, as researchers have a tendency to judge the value of predictors based upon the statistical significance of the results of a single sample of students. This is problematic, as when predictors are trailed in different teaching contexts they have a tendency to yield inconsistent results (Sections 5.1 and 7.2).

To address this shortcoming, we examined 34 predictors which had only been explored in a limited number of teaching contexts, or where previous research had yielded inconsistent results. These predictors included previous programming experience, previous academic experience, attributional style, behavioural characteristics, learning styles, and, learning strategies and motivations (Sections 5.4.1 to 5.4.3). By re-evaluating these predictors in our context, this thesis contributes to the knowledge on whether these predictors are context dependent like their predecessors, and evaluates their wider applicability as enablers of programming success. The additional contribution of this study was to perform a regression analysis to develop a predictive model based upon the context independent predictors (Section 5.4.5).

New Data Driven Predictors based upon Programming Behaviour

The third major contribution of this thesis was to identify a set of data-driven predictors which are based upon the programming behaviours of students. Throughout this thesis we have argued that almost all of the predictors which are based upon traditional learning theories perform inconsistently in different teaching contexts, as they are incapable of reflecting changes in a student's programming knowledge over time. Recently researchers have begun to explore predictors which are based upon analysing data which is gathered from an IDE describing the programming behaviours of students. These approaches are still in their infancy and very few studies on the relations between programming behaviour and programming performance had been conducted.

In this thesis we have identified 10 new data-driven predictors based upon aspects of programming behaviour. Five of these predictors were based upon the percentage of different types of compilation pairings that were logged from students (Section 6.4.2). A further five were based upon the percentage of lab time students spent working on different types of pairings (Section 6.4.3). Nine of these predictors were statistically significant, but more importantly, and unlike the traditional predictors, were found to yield consistent results on three independently gathered datasets.

Development of a Data Driven Predictive Algorithm

The fourth major contribution of this thesis was to develop a hybrid algorithm which could weight and combine several data-driven predictors to form an overall score, describing how *undesirable* a student's programming behaviour had been over the duration of a session. Previously the Error Quotient [82] was the only algorithm designed for this purpose. But there are several flaws concerning the incompleteness of the algorithm which limit its ability to accurately reflect the programming behaviour of students.

To address these shortcomings, we extended our previous work [200] by developing the Robust Relative algorithm (Section 6.4.4), which scores students based upon four aspects of their programming behaviour. These aspects take into account both error frequency and resolve times. The originality of our algorithm is to incorporate a novel scoring technique of relatively penalizing students based upon how their resolve times for different types of error, compares to the resolve times of their peers. This allows both the difficulty of resolving different types of errors, and the students own programming abilities to be taken into account. The algorithm was shown to yield consistently strong results when applied on three independently gathered datasets (Section 6.4.5) and the results from the meta-analysis placed the scores yielded by the algorithm among the strongest quantitative predictors identified over the past fifty years (Section 7.6).

Synthesising Fifty Years of Conflicting Quantitative Research

The fifth major contribution of this thesis was to statistically synthesise fifty years of conflicting research into predictors of programming performance. Previous research has been limited by both a lack of verification of predictors in different teaching contexts, and the conflicting results which are commonly reported when verifications take place (Section 7.2). Due to these issues there was no general consensus among researchers on which factors support students in developing programming skills. This is of greater concern nowadays given the introduction of programming into ordinary classroom environments. Without understanding which factors are the most important for success, it becomes extremely difficult to design and implement any effective pedagogical tools or strategies which can be applied to identify and support the weakest students.

To address this issue, we performed a substantial meta-analysis which aimed to resolve over fifty years of conflicting quantitative research into predictors of programming performance, by applying meta-analysis techniques to synthesize the findings of multiple studies that have examined the same predictor of programming performance across different teaching contexts. This enables researchers to make generalizable conclusions on the relevance of different factors for programming performance, that are independent of both aspects of the teaching context and characteristics of the students themselves. The comprehensiveness of the work can be highlighted from the number of predictors included in the analysis. From the initial review, 482 previous research results describing the relations between 116 predictors and programming performance were identified. These included predictors that were based upon aptitude, psychological factors, cognitive factors, academic background, demographic factors and programming behaviour (Sections 7.5.1 to 7.5.6). The additional contribution of this work was to provide a synthesized benchmark on the effects of different predictors on programming, which future researchers can compare their own results with.

Development of a Theoretical Framework

The sixth major contribution of this thesis was to perform knowledge transformation by classifying the 116 predictors which were identified from the meta-analysis into a six class theoretical framework (Section 7.4). This framework was applied to consolidate the meta-analysis findings by exploring which types of factors are more relevant to programming than others (Section 7.6). The extended framework (Figure 8.1) accounts for both factors which are internal to students (Chapters 5, 6, and 7), and factors from the teaching context (Chapter 4) and could be applied by future researchers to derive practical applications.

8.2 Research Questions

This thesis satisfied the three research objectives (Section 3.1) by answering all four of the research questions. The answers to the four research questions are briefly discussed in relation to the wider body of work and the proposed theoretical framework.

RQ1: Failure Rates in Introductory Programming

> *RQ1: To what extent are students' programming performances influenced by aspects of the teaching context, including: year, country, grade level of the institution, cohort size, and the programming language taught?*

Section 4.4.1 first estimated the worldwide pass rates of programming courses to be 67.7% (SD = 15.5%, 95% CI: 65.3% to 70.1%). This complements the main finding of the smaller survey performed by Bennedsen and Caspersen [13] who estimated the worldwide pass rate to be 67%. Examining the data based upon the moderating aspects of the teaching context (Sections 4.4.2 and 4.4.3) yielded the following results:

- *Year.* No significant differences were found in pass rates over time, $p = .97$. Failure rates were found to be consistently between 25% and 33%.
- *Country.* Three significant differences between the pass rates of Finland with Australia, $p = .04$, Canada, $p < .001$ and USA, $p < .01$. No significant differences were found between the pass rates of the remaining 15 countries.
- *Grade Level.* Significant differences were found between the pass rates of universities ($n = 145$, M = 66.4%) and other grade levels ($n = 16$, M = 79.9%)
- *Cohort Size.* Significant differences were found between the pass rates of small cohorts ($n = 10$, M = 80.1%) and large cohorts ($n = 91$, M = 65.4%). However a Spearman's correlation only found a weak association between the number of students enrolled in a course and pass rates, $r_s(101)$ = -.17, $p = .10$.
- *Programming Language.* No significant differences were found between the pass rates based upon programming languages, although the pass rates were lowest for courses which taught C ($n = 7$, M = 61.1%).

These results imply that aspects of the teaching context do not have a substantial moderating effect on the programming performance of students. The more interesting finding which was contributed by this thesis was that the pass rates of programming courses have not substantially changed over time. This implies that student level factors may be responsible for failing students.

RQ2: Traditional Predictors based upon Learning Theories

> *RQ2: Which traditional learning theories describing the psychological and cognitive aspects of learning, and which aspects of students' academic backgrounds are predictive of their programming performances?*

Section 5.4.1 explored the associations between performance and predictors which were based upon cognitive factors. These included learning styles and learning strategies. Examining learning styles, weak correlations were found for the scores obtained on the concrete/sequential $r = .27$ and abstract/sequential $r = .29$ dimensions of Gregorc's Style Delineator. For learning strategies, correlations were found for MSLQ total score $r = .22$, critical thinking $r = .28$, and resource strategy (effort) $r = .28$. A moderate correlation was found for intrinsic goal orientation $r = .33$. A strong correlation was found for total self efficacy $r = .54$.

Section 5.4.2 explored the associations between performance and predictors which were based upon psychological factors. These included attributional style and behavioural characteristics. Examining attributional style, a weak correlation was found for attribution of success to task difficulty, $r = -.10$. Moderate correlations were found for attribution to luck, $r = -.31$, and attribution to ability, $r = .40$. Considering behavioural characteristics revealed a strong negative correlation found for hours spent working in a part time job, $r = -.64$.

Section 5.4.3 explored the associations between performance and predictors which were based upon academic factors. These included subject specific performance, grade point average, and prior programming experience. Having prior programming experience appeared to weakly support programming performance, $r = .13$, and overall High School GPA revealed a similar weak correlation, $r = .27$. A moderate correlation was found for university calculus grade $r = .37$.

When considering the context dependence of the predictors, 18 predictors were found to be context dependent. Out of the context independent predictors identified by this study, only five of the predictors examined yielded an average moderate correlation strength. The remaining 14 context independent predictors were found to yield weak correlations at best. The regression analysis conducted in Section 5.4.5 showed that a moderate amount of the variance in performance could be accounted for ($R^2 = 21.40\%$, $p < .05$) when considering only context independent predictors. However when using all of the predictors examined in this study, a regression could explain a large amount of the variance in performance ($R^2 = 35.90\%$, $p < .01$). This suggests that it is difficult to construct a generalisable model upon traditional predictors possibly because they were not designed for the purpose of predicting programming performance.

RQ3: Data Driven Predictors based upon Programming Behaviour

> *RQ3: Which data-driven metrics derived from data describing students' programming behaviours are predictive of their programming performances?*

Section 6.4.2 explored the associations between performance and the compilation pairings which were logged from students. Five significant ($p < .01$) correlations were found (Figure 6.5) which showed that weaker students are associated by making a greater percentage of successive errors than their peers, and a smaller percentage of successive successful compilations. The regression analysis showed that the percentage of error to same error pairings which were logged from students could explain a moderate amount of the variance in performance ($R^2 = 27.19\%$, $p < .01$). This was closely followed by the percentage of error to any error pairings ($R^2 = 20.31\%$, $p < .01$).

Section 6.4.3 explored the associations between performance and the percentage of lab time students spent working on pairings. Four significant ($p < .01$) correlations were found (Figure 6.7) which showed that weaker students are characterised by spending a greater percentage of their lab time resolving errors than their peers. The regression analysis showed that the percentage of lab time spent working on error to same error pairings could explain a moderate amount of the variance in performance ($R^2 = 27.85\%$, $p < .01$). This was closely followed by the percentage of lab time students spent working on error to any error pairings ($R^2 = 26.19\%$, $p < .01$), error to different error pairings ($R^2 = 17.50\%$, $p < .01$) and two successful compilations ($R^2 = 14.67\%$, $p < .01$).

Section 6.4.4 presented the Robust Relative algorithm which computes a score describing how *undesirable* a student's programming behaviour was over a session. The novel relative penalizing of students based upon how their resolve times for different types of error compares to the resolve times of their peers, along with the hybridization of four metrics of programming behaviour, resulted in a stronger predictor than any of the singularly considered metrics. The regression analysis showed that the Robust Relative scores could significantly ($p < .01$) explain a large amount of the variance in the performance of all three samples of students, $R^2 = 42.19\%$, 43.65% and 44.17%.

As with many of the metrics explored in this study, the consistent results found on all three datasets suggests the generalisability of these predictors to different contexts. The comparative analysis conducted in Section 6.4.5 (Figures 6.11 and 6.12) showed that the Robust Relative algorithm could account for the largest amount of the variance in performance when limited data was available (four sessions). The metrics based upon the percentage of lab time students spent working on pairings explained a larger amount of the variance in performance than those based upon the percentage of different types of pairings. This suggests that time is a more important predictor than frequency.

RQ4: Meta-Analysis of Fifty Years of Research on Factors that Can Predict Programming Performance

> *RQ4: How do factors based upon traditional learning theories, academic background, and programming behaviours, compare when they are used to predict students' programming performances across different teaching contexts?*

Chapter 7 synthesised 482 results which described the relations between 116 predictors and programming performance across multiple teaching contexts. These predictors were classified into a six class theoretical framework and discussed in Section 7.4.

The majority (63%) of the 16 aptitude predictors explored in Section 7.5.1 were found to have a weak effect on performance. Over one third (37%) of the 35 psychological predictors explored in Section 7.5.4 and both of the 2 demographic predictors explored in Section 7.5.6 were found to have no effect on performance.The 27 cognitive predictors explored in Section 7.5.3 and the 19 academic background predictors explored in Section 7.5.2 were found to have a comparable effect on programming. 82% of cognitive predictors, and 84% of academic predictors were found to have a weak/moderate effect on programming. The 17 predictors based upon aspects of programming behaviour that were explored in Section 7.5.5 were found to have the strongest effect on programming performance. Out of the 17 predictors in this class, 11 predictors (65%) were found to have moderate to very strong effects on performance.

Section 7.6 summarised the strongest and weakest predictors found by this meta-analysis. Almost half (47%) of the strongest 15 predictors were based upon programming behaviour. Out of the remaining 8 strongest predictors, 4 (27%) were cognitive predictors, 3 (20%) were academic predictors, and 1 (6%) was based upon aptitude testing. These results suggest that the strongest predictors of programming performance which have been found over the past fifty years are based upon programming behaviour. Additionally, 5 of the strongest predictors were identified by the experiments conducted to answer RQ3 of this thesis. However, several of the traditionally measured attributes also featured among the strongest predictors. The presence of self-efficacy and comfort level suggests that an element of self belief, or self concept is necessary for programming success. The presence of college math, physics, and chemistry among the top predictors reinforces anecdotal evidence that a level of scientific maturity, or scientific thinking, is a necessary skill to become a successful programmer.

Two thirds (67%) of the weakest 15 predictors were based upon demographic and psychological factors. These results showed that self perceived abilities, learning styles, personality traits, age, and gender, have little relevance for programming success.

8.3 Implications for the Theoretical Framework

The framework of factors displaying the results of this thesis is shown in Figure 8.1.

External Factors

RQ1 examined the influence which factors from the teaching context had on programming performance. Only cohort size and grade level of the institution were found to significantly moderate performance. Based upon these results, the wider implication for practice is that the best approach for teaching programming may be one which is based upon using small groups, and replacing traditional university approaches with classroom based instruction. This ties in with research on small group teaching, and the use of pair programming to improve the performance of programming students [119].

Although not statistically significant, there were differences between the failure rates of courses which taught C and courses which taught Java (Figure 4.6). This ties into the literature on the difficulties of learning different types of languages from a conceptual perspective [45, 99, 127, 136]. The fact that the year in which the course was taught did not significantly moderate performance perhaps alludes to the presence of threshold concepts in programming [22, 106, 121] in that no matter how well supported students are by advances in pedagogy, there are still fundamental programming concepts for which the learning barrier cannot be easily lowered.

Internal Factors

RQ2, RQ3, and RQ4 examined the factors which are internal to students. Of the six classes proposed in the framework, the results from the meta-analysis showed that certain programming behaviours, cognitive, and academic predictors were relevant. Specifically, the implications from these findings is that successful programming students are characterised by high levels of self efficacy, have performed well in college level science subjects, and exhibit traits of desirable programming behaviour.

To rapidly identify at risk students before assessments take place, we suggest that an initial screening can be performed by considering the students academic background and results from a test based upon self efficacy. Following this, the predictors based upon programming behaviour which have been identified in this thesis, in particular the Robust Relative algorithm, could be applied to continually and autonomously monitor the students programming behaviours. Due to their automated nature, the use of programming behaviours could be applied to alert instructors when students appear to be at risk, or to provide an automated pedagogical intervention to support them, such as adapting the compilation feedbacks they receive [199].

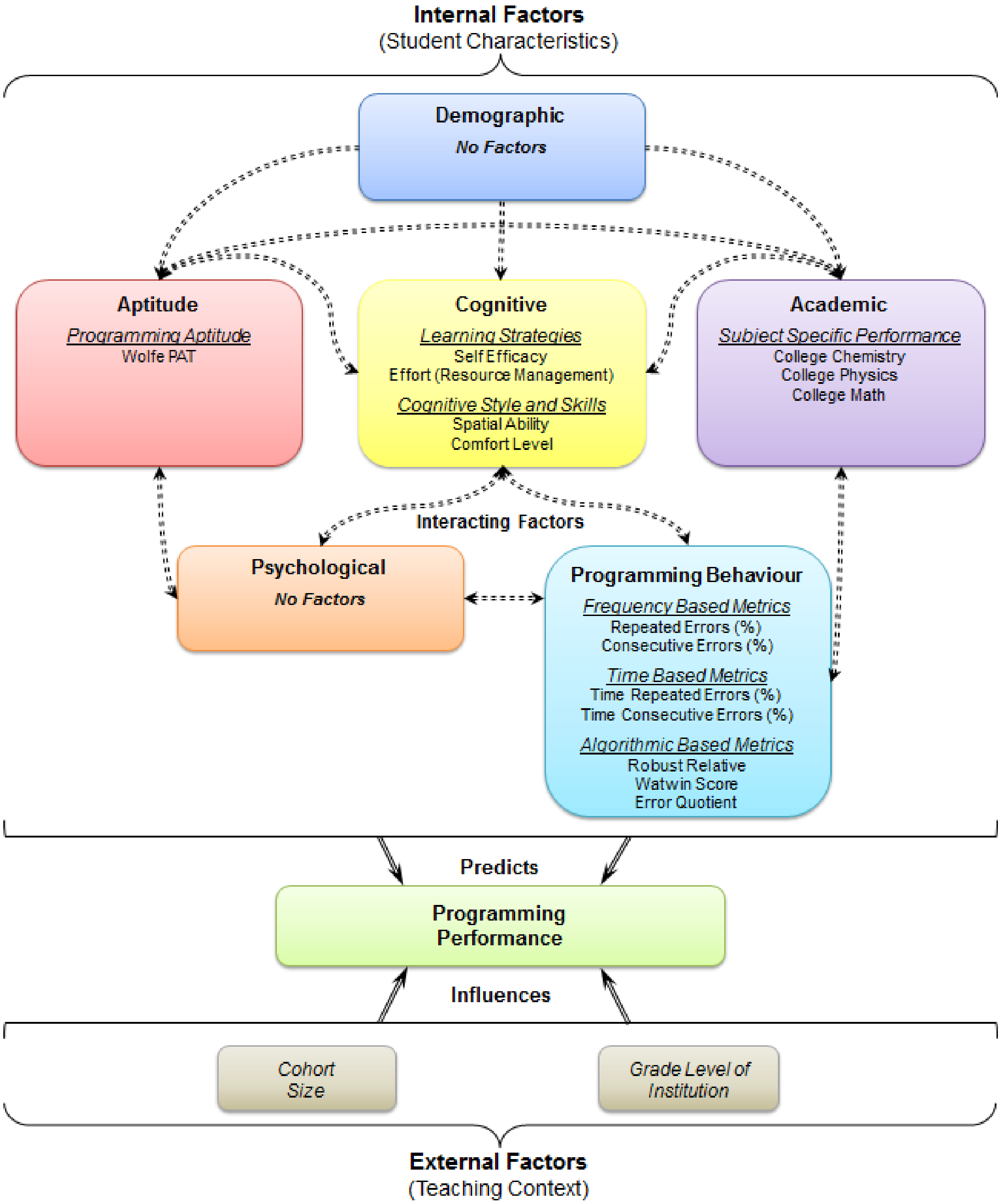


Figure 8.1: Updated theoretical framework, displaying the strongest fifteen internal factors from the meta-analysis, and the two influential external aspects of the teaching context identified by this thesis.

8.4 Limitations

The limitations of the four studies which have been performed in this thesis were discussed in Sections 4.5.1, 5.5.1, 6.5.1 and 7.6.3. We briefly discuss the limitations which run through the entire thesis.

The main limitation of this thesis is that it has explored quantitative evidence only, and no qualitative evidence has been included. However, this was an appropriate decision based upon the nature of this study. This thesis made use of substantial quantitative evidence from previous research in order to develop an understanding of which trends existed between different types of predictors and programming performance. Identifying such a trend would have been near impossible by using qualitative evidence only, due to the varying measurement techniques which would have been performed by different studies. Most of the quantitative predictors explored in this study were based upon standardised testing instruments which increases the reliability of the findings.

The second and unavoidable limitation of this thesis is that the the assessment criteria and learning objectives will vary between the different studies we have analysed. Studies within the UK generally defined a pass boundary at 40%, whereas other studies from the USA defined a pass boundary at 50%. Therefore this thesis unavoidably has to assume that a consistent notion of programming performance exists and holds valid across the different teaching contexts. But, we note that this a common limitation of studies of this nature and that meta-analysis techniques are designed to take into account such limitations.

8.5 Future Directions

There are numerous potential directions for future work in this area. In this section we provide suggestions for future work within the area.

Identifying Further Data Driven Predictors

The data-driven predictors which were based upon analysing aspects of programming behaviour were among the strongest predictors examined over the past fifty years (Section 7.6), and their automated nature provides several advantages over the traditionally explored predictors based upon learning theories. It would seem to be worthwhile for future researchers to continue to identify programming behaviours which are predictive of programming performance. Most research to date has only been conducted at single institutions on a small scale. In contrast the Blackbox project which collects data from worldwide users of the BlueJ IDE provides an opportunity to explore the programming behaviours of thousands of students working worldwide across different teaching contexts [25]. However one of the main limitations of Blackbox is the lack of outcome measure in the dataset, making it difficult for researchers to know whether the data they are analysing is taken from a high or low performing student. Until this issue is addressed, researchers will be limited in the depth of analyses which they are able to perform using the Blackbox dataset.

Exploring the Context Dependency of Data Driven Predictors

One of the weaknesses of the data-driven predictors which we identified in Chapter 6 is that they have only been explored in our teaching context. Whilst the results of the meta-analysis suggest that these predictors will perform comparably in similar teaching contexts, it would be worthy to explore how they generalise to different teaching contexts which use different teaching approaches, tools, and programming languages. We are currently working with the University of Helsinki to explore this hypothesis.

Developing Practical Applications

There are numerous possible practical applications of the data-driven predictors and algorithms which have been identified in this thesis. One possible application could be to apply the predictors in an intelligent tutoring system, to automatically adjust the levels of compilation feedbacks provided to students.

Feedback is regarded as having one of the most important influences on learning and motivation. When learning to program students are guided on the correctness of their syntax by compiler feedback. However standard compiler feedback is designed for experts - not novices, and often fails to match their current level of conceptual knowledge, making it difficult to understand. Although programmers can often encounter cryptic messages which are difficult to resolve, most related disciplines have not paid much attention to this aspect, because it is felt that programmers should adapt to compilers [185]. In contrast most pedagogical theory places a strong emphasis on adaptation to the individual to make instruction most effective [167]. Additionally as most compilers are context-sensitive, the same error feedback can be presented for a range of different underlying causes. In essence, if we consider the principles of effective feedback from a pedagogical perspective (such as precision) [167], the feedback which is supplied to novice programmers from the compiler offers the exact opposite of what they require [199]. By using the data-driven metrics identified in this thesis, it could be possible to design a feedback system which is capable of pre-emptively providing support to weaker students who are displaying undesirable programming behaviour.

Applications of the Predictors to other Computer Science Modules

It would be useful to determine how effective the strongest predictors identified in this thesis are at predicting performance in other computer science modules, for example, Introduction to Python or Discrete Mathematics. Programming is only one skill which students must master in order to be awarded a computer science degree. If a subset of predictors could be identified which predict performance in several computer science modules, then it may be possible to construct a model which can predict students who are at risk of failing their entire degree. Such a predictive model would be a highly significant contribution to the education community. It would highlight which factors and skills are the most important for "computer science thinking" and could possibly be applied for the purpose of identifying and improving the weak skill sets of weaker students which may cause them to drop out.

Impact of Different Teaching Approaches on Programming

One aspect of the teaching context which we did not explore in this thesis was the teaching methodology employed by instructors. Whilst numerous studies suggest approaches that provide effective means of teaching programming, no study had attempted to quantitatively compare the impact that these approaches can have on improving the pass rates of failing programming courses. Without any quantitative evidence on the relative strengths of different approaches, the research community as a whole will continue to lack a clear consensus of precisely which methodologies provide the most effective means of teaching programming and saving failing programming students.

In conjunction with the University of Helsinki and following on from the research presented in Chapter 4, we have published a quantitative systematic review on articles describing introductory programming teaching approaches, and analysed the effect that various interventions can have on the pass rates of introductory programming courses. The results showed that on average, teaching interventions can improve programming pass rates by nearly one third when compared to a traditional lecture and lab based approach. While no statistically significant differences between the effectiveness of teaching interventions were observed, marginal differences do exist. The courses with relatable content (e.g. using media computation) with cooperative elements (e.g. pair programming) were among the top performers with CS0-courses, whilst courses with pair programming as the only intervention type and courses with game-theme performed more poorly when compared to others [193].

Deeper examination of Programming Behaviour

Another good direction for future work would be to perform a qualitative analysis of programming behaviour to identify the true underlying causes of the *symptoms* of struggling students based upon aspects of their programming behaviour. For instance, although frequently repeating errors is data-driven metric which can be used to infer that a student is displaying signs of struggling behaviour, the actual underlying cause of this behaviour was not explored within this thesis. By developing a deeper understanding of the underlying cause of the symptoms of *undesirable* programming behaviour, a more accurate and practical intervention system could be developed.

Although the penalties selected within the Robust Relative algorithm were derived through using a machine learning approach and validated on multiple datasets, there remains the question as to why different penalties were selected for the components. e.g., the exact reason why making two successive errors with the same location was awarded a higher penalty than repeating the exact error type remains for future exploration.

8.6 Summary

This chapter has summarised the research presented in this thesis, and has discussed the contributions, limitations, and possible directions for future works. This thesis has provided much needed quantitative evidence on the worldwide outcomes of programming courses, explored the impact of aspects of the teaching context on performance, and has statistically synthesised over fifty years of research by applying unbiased meta-analysis techniques. The data-driven predictors which were identified in this thesis have shown considerable potential as predictors of programming performance by yielding consistent results on three independent datasets. This thesis can be considered to be a precursor to the valuable multi-national, multi-institutional studies which must follow. Only by performing such studies, can the relevant factors explored and metrics identified in this thesis be verified across a more varying set of contexts, and thus, truly generalisable predictors be identified.

This thesis argued that factors based upon traditional learning theories struggle to consistently predict programming performance across different teaching contexts because they were not intended to be applied for this purpose. In contrast, the main advantage of using data-driven approaches to derive metrics based upon students' programming processes, is that these metrics are directly based upon the programming behaviours of students, and therefore can encapsulate such changes in their programming knowledge over time. Researchers should continue to explore data-driven predictors in the future.

Appendix A

Statistical Tests

A.1 Assumption Testing

The **Shapiro Wilk test of normality** (W) tests the null hypothesis that a sample, $x^1, ..., x^n$ comes from a normally distributed population and is recommended for small or medium sized samples ($n \leq 100$). The test is given by:

$$W = \frac{(\sum_{i=1}^{n} w_i x_{(i)})^2}{\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{A.1}$$

where n is the number of observations, X is the original data, X' is the ordered data, and $\bar{X}$ is the sample mean of the data. The constants w_i are given by:

$$w_1, ..., w_n = MV^{-1}[(M'V^{-1})(V^{-1}M)]^{-\frac{1}{2}} \tag{A.2}$$

where M denotes the expected values of the standard normal order statistics for the sample, and V is the corresponding covariance matrix. When $W = 1$, the sample follows a perfect normal distribution. If the test statistic is non-significant, $p > .05$, then the distribution of the sample is not significantly different from a normal distribution, and parametric tests can be used. If the test statistic is significant, $p < .05$, then the distribution of the sample is significantly different from a normal distribution, and either data transformations, or non-parametric equivalent tests can be used.

The **Levene Test for Equality of Variances** tests the null hypothesis that the variances across k subgroups are equal. Given a variable Y with a sample size N divided into k subgroups, where N_i is the size of the ith subgroups, the Levene test statistic is defined as:

$$W = \frac{(N-k)}{(k-1)} \frac{\sum_{i=1}^{k} N_i(\bar{Z}_i - \bar{Z}_{ij})^2}{\sum_{i=1}^{k}\sum_{j=1}^{N_i}(Z_{ij} - \bar{Z}_i)^2} \tag{A.3}$$

where $Z_{ij} = |Y_{ij} - \tilde{Y}_{i.}|$, and $\tilde{Y}_{i.}$ is the median of the i-th subgroups, $\bar{Z}_i$ are the subgroup means of Z_{ij}, and $\bar{Z}_{..}$ is the overall mean of Z_{ij}. The median is recommended as the choice of $\tilde{Y}_{i.}$, to provide good robustness against many types of non-normal data without sacrificing power. If the test statistic is non-significant, $p > .05$, then there are no differences between the variances of the subgroups. If the test is significant, $p < .05$, then homogeneity of variances cannot be confirmed, and a correction needs to be applied to the results. In the case of an one-way ANOVA, this includes reporting the results of a Welch ANOVA and using a Games-Howell post-hoc test instead of the conventional Tukey post-hoc test.

A.2 Association

Pearson's Product Moment Correlation Coefficient (r) measures the strength of the linear relationship between two continuous variables. It is based on the assumption that both variables are interval or ratio based, and are samples from populations that follow a normal distribution. Given two n dimensional vectors, $X = X_1, ..., X_n$ and $Y = Y_1, ..., Y_n$, the Pearson's correlation coefficient is calculated by:

$$r = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{(\sum_{i=1}^{n}(X_i - \bar{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \bar{Y})^2}} \tag{A.4}$$

The value of r can range from -1.0 representing a perfect negative linear relationship, to 1.0 representing a perfect positive linear relationship. A value of 0 indicates no relationship between two variables. The strengths of the relationships between X and Y that are between these values are classified in this thesis based upon the guidelines of [47], who defined the following interpretations of the absolute value of r:

- no association, $|r| < .10$.
- weak association, $.10 \leq |r| < .30$.
- moderate association, $.30 \leq |r| < .50$.
- strong association, $.50 \leq |r| < .80$.
- very strong association, $.80 \leq |r| \leq .1.0$.

If the test statistic is non-significant, $p > .05$, then there is insufficient evidence to conclude that a linear relationship between X and Y exists in the population. If the test is significant, $p < .05$, then there is sufficient evidence to conclude that a linear relationship between X and Y exists in the population. If there are violations of assumptions, then the non-parametric Spearmans-Rank Correlation Coefficient (rho) may be calculated instead, which has a similar interpretation to r as above.

A.3 Prediction

Multiple linear regression is used to assess the relationship between a continuous dependent variable (Y) and a combination of several continuous independent variables ($X = X_1, ..., X_n$). It can be applied for two main purposes. Firstly, to determine how much of the variation in the dependent variable can be explained by the independent variables. Secondly, to predict new values for the dependent variable given the independent variables. Regression requires that a number of underlying assumptions are satisfied, including: a linear relationship between the dependent variable and each of the independent variables exists (scatter plot), no significant outliers (box plot), homoscedasticity (scatter plot), independence of residuals (Durbin-Watson), normally distributed residuals (Shapiro-Wilk) and no multicollinearity between independent variables ($r_{xy} < .70$). The regression equation is given by:

$$\hat{Y} = A + B_1X_1 + B_2X_2 + ... + B_nX_n \tag{A.5}$$

where $\hat{Y}$ is the predicted value on the dependent variable, the X values represent the independent variables, the B values are the regression coefficients, representing the amount the dependent variable $\hat{Y}$ changes when the corresponding independent variable changes one unit, A is the intercept with the y-axis, representing the value of $\hat{Y}$ when all the independent variables are 0. The regression line is commonly fitted through a least-squares method, where a line of best fit is calculated by minimising the sum of squares of the actual values Y and the predicted values $\hat{Y}$. The quality of a regression model can be judged based upon several aspects.

The *F-ratio* is used to determine whether the proposed regression model fits the data well, and is the ratio of the mean regression sum of squares, to the mean residual sum of squares. The null hypothesis of this test is that the multiple correlation coefficient, R, is equal to 0. What this also means is that at least one regression coefficient (except the intercept) is statistically significantly different to zero. If the test-statistic is not significant, $p > .05$, then it can be concluded that the model is unreliable, and that there is a lack of linear fit. If the test-statistic is significant, $p < .05$, then it can be concluded that the model is reliable, and that there is linear fit.

The *root mean squared error (RMSE)* describes the absolute fit of the regression model to the underlying dataset in terms of how close each of the observed data points (Y_i) are from the models predicted values ($\hat{Y}_i$), providing a good measure of how accurately the regression model can predict the actual response. The lower the RMSE the better the fit the model is to the dataset. But, the precise definition of what constitutes to be a good RMSE depends upon the purpose for which the model is developed.

The *coefficient of determination (R^2)* represents the proportion of variance in the dependent variable that can be explained by the independent variables. Generally the greater the percentage of variance that is accounted for by the model, the more accurate the predictions can be. However, there is a risk of over-fitting a model if too many independent variables are added. R^2 can be interpreted similarly to Pearson's r as follows:

- no effect ($R^2 \leq .02$)
- weak effect ($R^2 \geq .03$)
- moderate effect ($R^2 \geq .10$)
- strong effect ($R^2 \geq .25$)
- very strong effect ($R^2 \geq .65$)

Stepwise regression is a variation of multiple linear regression where independent variables are added to the model in sequence (rather than simultaneously), and then are assessed as to whether they have a significant role in explaining the variance of the dependent variable. If adding an independent variable contributes to the coefficient of determination, then it is retained, but all other variables in the model are then re-tested to see if they are still contributing to the success of the model. If they no longer contribute significantly they are removed. The advantage of this technique is that it usually results in smallest possible set of independent variables included in the model, without significantly impacting on the coefficient of determination. If the variable does not contribute, then it may be possible to use fewer independent variables in the regression model and still maintain a good level of prediction. Thus, this approach reduces the likelihood of overfitting a model to a large set of independent variables, which is a risk of multiple regression.

A.4 Differences Between Groups

The **independent samples *t*-test** is used to determine whether there are any differences between the means of two independent groups on a continuous dependent variable. It is based on the assumptions that each group's data is approximately normally distributed and that there is homogeneity of variances across the two groups. If these assumptions are satisfied, then the t-test can be calculated by using:

$$t = \frac{\bar{x_1} - \bar{x_2}}{\sqrt{\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}}} \tag{A.6}$$

where x_1 and x_2 are the given values of both groups, S_1 and S_2 are the standard deviations of both groups, and n_1 and n_2 are total number of values of both groups. If the test statistic is significant, $p < .05$, then the difference between the population means is zero. If the test statistic is non significant, $p > .05$, then the difference between the population means is not zero.

If assumptions are violated, then sensitivity analysis can be conducted by running an equivalent non-parametric test, Mann-Whitney U test, and providing that both tests lead to the same conclusion, the t-test will still be valid for the analysis.

The **One-Way Analysis of Variance (ANOVA)** is used to determine whether there are any differences between the means of two or more independent groups, and can be considered as an extension to the t-test. It is based on the assumption that each group's data is normally distributed, and that there is homogeneity of variances across the groups. As a rule of thumb, it is recommended that an ANOVA is only conducted when each group consists of at the very least six values. If these assumptions are satisfied, then an ANOVA F-ratio can be calculated by using:

$$F = \frac{\text{between group variance}}{\text{within group variance}} = \frac{\sum_i n_i(\bar{X}_i - \bar{X})^2/(K-1)}{\sum_{ij}(\bar{X}_{ij} - \bar{X}_i)^2/(N-K)} \tag{A.7}$$

where $\bar{X}_i$ is the sample mean of the ith group, n_i is the number of observations in the ith group, $\bar{X}$ denotes the overall mean of the data, and K denotes the number of groups, X_{ij} is the jth observation in the ith out of K groups and N is the overall sample size. If the test statistic is significant, $p < .05$, then at least one group mean is different, and a Tukey post-hoc test can be conducted to identify any significant differences between groups. If the test statistic is non significant, $p > .05$, then all group means are equal and no post-hoc tests need to be conducted.

If the assumption of homogeneity of variances is violated, then a robust version of the ANOVA, such as Welch's ANOVA needs to be performed, and post-hoc analysis performed using a Games-Howell test, rather than the conventional Tukey post-hoc test. If multiple assumptions are violated, then sensitivity analysis can be conducted by running an equivalent non-parametric test, Kruskal-Wallis H test, and providing that both tests lead to the same conclusion, the one-way ANOVA will still be valid for the analysis.

A.5 Meta Analysis

The following section outlines the statistical analysis procedures that were used to perform the meta-analysis that was conducted as part of this thesis. Other details of the methodology such as descriptions of the literature search methodology and description of the resulting sample are discussed in the relevant chapter.

A.5.1 Effect Size Calculations

As this study mainly examines linear relationships between two continuous variables, or in some cases, a dichotomous and continuous variable, Pearson's r was selected as the most appropriate measure of effect size in this thesis. This measure was selected as it provides a number of advantages over alternative measures that are based upon standardized mean difference (Cohen's d or Hedges' g), including:

- In the case of studies that use a t-test or F-ratio, calculating a corresponding Pearson's r only requires the inferential test value and degrees of freedom to be reported. This allows a converted effect size to be more accurately calculated for studies that fail to report this data.

- The same equations are used to compute r for independent sample, and repeated-measures inferential tests, whereas different formulas are necessary when computing g or d.

- r is already familiar to most students and researchers and as such, a number of studies already report their findings using r. In other words, less conversions of reported effect sizes are required than in the case of using g or d.

- r is more suitable for describing an association between two continuous variables than g or d. [31].

When studies did not directly report their effects in terms of Pearson's r, a number of conversion formulae were applied based upon available data. These included:

- Independent t-test

$$r = \sqrt{\frac{t^2}{t^2 + df}} \tag{A.8}$$

- Independent F-ratio

$$r = \sqrt{\frac{F_{(1,df)}}{F_{(1,df)} + df}} \tag{A.9}$$

- 2 x 2 (i.e. 1 *df*) contingency χ^2

$$r = \sqrt{\frac{\chi^2_{(1)}}{n}} \tag{A.10}$$

- Probability levels from significance tests

$$r = \frac{Z}{\sqrt{n}} \tag{A.11}$$

- Standardized mean difference, Cohen's d

$$r = \sqrt{\frac{d^2}{d^2 + 4}} \tag{A.12}$$

- Standardized mean difference, Hedges' g

$$r = \sqrt{\frac{g^2 n_e n_c}{g^2 n_e n_c + (n_e + n_c)df}} \tag{A.13}$$

where Z is the standard normal deviate, not Fisher's z-transform, n is the total sample size, n_c is the size of the control group, n_e is the size of the experimental group, df is the degrees of freedom.

A.5.2 Effect Size Corrections

Artifact correction attempts to adjust the effect sizes calculated from a study to take into account the imperfections in the measurement instruments that were used. This allows conclusions to be formed about the associations among constructs, rather than among the instruments that were used to measure constructs [78]. There are two common forms of artifact correction in addition to applying Fisher's z to r transformation. Generally, corrections to effect sizes take the following form:

$$\text{Effect Size (ES)} = \frac{ES_{\text{observed}}}{\alpha_{\text{correction}}} \tag{A.14}$$

$$\text{Standard Deviation (SD)} = \frac{SE_{\text{observed}}}{\alpha_{\text{correction}}} \tag{A.15}$$

Unreliability refers to non-systematic error that occurs during the measurement process that arise from sources of error that are beyond the control of instruments used to measure a construct. For example, the same instrument can be used to measure a construct across different teaching contexts or lab conditions, but the instrument itself does not take these different conditions into account [78]. Unreliability can be corrected by applying the following formula:

$$\alpha_{\text{unreliability}} = \sqrt{r_{xx}} \tag{A.16}$$

where r_{xx} is the reliability estimate of a variable xx, such internal consistency of a measure Cronbach's α, or inter-rater reliability, Cohen's κ.

Artificial dichotomization refers to splitting a variable that is naturally continuous into two distinct categories. For example, measuring programming ability in terms of pass or fail instead of using an overall numerical course mark. As artificially dichotomizing a variable that is naturally continuous can attenuate associations that this variable has with others, corrections for artificial dichotomization can be performed by using:

$$\alpha_{\text{dichotomization}} = \frac{\phi(c)}{\sqrt{PQ}} \tag{A.17}$$

where P and Q are the proportions of each group, and $\phi(c)$ is the normal ordinate at the point c.

Fisher's z to r transformation (Z_r) is commonly applied to satisfy two assumptions for meta-analytic methods. Firstly, many meta-analytic methods assume an approximately symmetrical sampling distribution of observed outcomes about a given population. However in general, the sampling distributions of correlations are skewed. Fisher's z to r provides an effective normalizing transformation, yielding an almost symmetrical distribution of transformed correlations, and satisfying one of the assumptions required by meta-analytic techniques. However, this is not the main purpose of applying the transformation. Another assumption of meta-analytic methods is that the sampling variances of the observed outcomes are approximately known. However, calculating the sampling variance using Pearson's r requires knowing the true correlation coefficient, which isn't always known. Fisher's z to r transformation provides a means of approximating the sampling variance without requiring knowledge of this population correlation coefficient, allowing the second assumption of meta-analytic methods to be satisfied [31]. Fishers z to r is calculated using the following formula:

$$Z_r = \frac{1}{2} \ln \left(\frac{1+r}{1-r} \right) \tag{A.18}$$

A.5.3 Meta Analytic Model Fitting

In general, there are three meta-analytic models that can be used to compute an overall effect size. These are the fixed, random, and mixed effects models [31, 71, 78, 191].

Fixed Effects Model

The fixed effects model is defined as:

$$ES_i = \theta + \epsilon_i \tag{A.19}$$

In the fixed effects model, the effect sizes for each study ES_i are assumed to be a function of two components, namely a single population effect size θ, and the deviation of this study from the population effect size ϵ_i. Generally, the population effect size is unknown, and is instead estimated by using a weighted average of the effect sizes, using the studies that are included in the meta-analysis. The weighting (w), that is applied to each study is defined as:

$$w_i = \frac{1}{{SE_i}^2} \tag{A.20}$$

where SE_i is the standard error of the effect size estimate, for study i. The fixed effects model assumes that all studies included in the analysis share a true common effect size, and that differences in effect sizes between different studies are due to random sampling fluctuations alone. It is appropriate to use this type of model if a sample of effect sizes is Heterogeneous. Whether a sample is heterogeneous can be determined by the following formula:

$$Q = \sum (w_i(ES_i - \overline{ES}^2) \tag{A.21}$$

and the magnitude of heterogeneity (I^2) can be found by:

$$I^2 = \begin{cases} \left(\frac{(Q-(k-1))}{Q}\right) \times 100 & \text{if } Q > (k-1) \\ 0 & \text{if } Q \leq (k-1) \end{cases} \tag{A.22}$$

The main limitation of the fixed effects model is that unconditional inferences cannot be made about the larger set of studies from which the studies included in the meta-analysis are assumed to be randomly sampled from. For this reason, the fixed effects model is most applicable when the objective of the researcher is to compute a common effect size, that is to be generalized to identical studies of the same population, both in terms of the characteristics of the sample, and methodology used.

Random Effects Model

Most meta-analyses are based on sets of studies that are not exactly identical in their methodology and/or the characteristics of their sample. Differences in the methods and sample characteristics can introduce variability (heterogeneity) among effect sizes that cannot be explained through random sampling fluctuations alone. One way to model this heterogeneity is to treat it as purely random. This leads to the random effects model, which is defined as:

$$ES_i = \mu + \vartheta_i + \epsilon_i \tag{A.23}$$

The main difference compared to the fixed effects model is that rather than assuming there is a single population effect size, there is actually a distribution of population effect sizes (due to differences in methods and sample characteristics), and that an overall effect size can be calculated by averaging the effects within this distribution. To reflect this, the single population effect size estimate used by the fixed effects model (θ) is decomposed into two parameters: a measure of the central tendency of the distribution (μ), and the reliable deviation of each study, from the mean of the distribution of population effect sizes (ϑ). As with the fixed effects model, a weighting of effect sizes is applied, however the weighting of the random effects model is defined as:

$$w_i^* = \frac{1}{\tau^2 - SE_i^2} \tag{A.24}$$

where τ^2 represents the population variability in effect sizes and is calculated by:

$$\tau^2 = \begin{cases} \dfrac{Q - (k-1)}{(\sum w_i) - \dfrac{(\sum w_i^2)}{(\sum w_i)}} & \text{if } Q \geq (k-1) \\ 0 & \text{if } Q < (k-1) \end{cases} \tag{A.25}$$

which can be applied to calculate the Random effects mean effect size:

$$\overline{Z_r} = \frac{\sum w^* Z_r}{\sum w^*} \tag{A.26}$$

with the Random effects mean standard error:

$$SE_{\overline{Z}_r} = \sqrt{\frac{1}{\sum w^*}} \tag{A.27}$$

Compared to the fixed effects weighting, when assigning weights to different studies in the random effects model, information in the smaller studies is not discounted by giving it a smaller weight. The goal is to estimate the effects in a range of populations, so a weighting is applied, but the overall estimate is not overly influenced by any one effect size. Larger studies still receive a larger weighting, but the weighting is more evenly distributed across the rest of the studies included. Larger studies are still influential, but the impact is much less pronounced. Similarly, smaller studies have a larger weighting than in the fixed effects model, and can influence the overall effect. The trade off of this weighting is that variance in the random effects model is defined as variance within a study, plus variance between studies. As long as the between studies variation is non-zero, the variance, standard error, and confidence interval will always be larger than an estimate using a fixed effects model. However, the main strength of the random effects model is that it allows unconditional inferences to be made about the larger set of studies, from which the studies included in the meta-analysis are assumed to be randomly sampled from (whether such studies currently exist or not). For this reason, the random effects model is most applicable when the objective of a researcher is to accumulate data from a series of studies that have been performed under a number of different conditions. The subjects or interventions have differed in ways that can impact on the results, and a single true effect size cannot be assumed. Essentially the strength of the random effects model is generalization to a population of studies, with the trade off of less precise point estimates (wider confidence intervals).

Mixed Effects Model

Mixed effect models extend random effect models by allowing the inclusion of one or more moderators (study level variables) in the model, that may account for at least part of the heterogeneity in the true effects. These models combine moderator analysis from the fixed effects model, with the generalizability of the random effects model. Analysis in the mixed effects model follows the logic of moderator analysis within a general regression framework. However, these models include additional terms representing population variability in effect sizes, above and beyond systematic variability accounted for by moderators as well as sampling fluctuations. The general equation for mixed-effects models can be represented by the following equation:

$$ES_i = \beta_0 + \beta_1(X_1) + \beta_2(X_2) + ... + \vartheta_i + \epsilon_i \tag{A.28}$$

where β_0 is the model intercept, $X_1, X_2...$ are moderator variables, and $\beta_1, \beta_2...$ are the regression coefficients of the moderator variables. In the mixed effects model, τ^2 denotes the amount of residual heterogeneity among the true effects. That is, variability among the true effects that is not account for by the moderators included in the model.

Appendix B

Example Assessment Materials

B.1 Example Term One Bench Test

Introduction to Programming Bench Test

Dr N.S. Holliman

10th November 2011

The Scenario

A max-min thermometer stores the current and the maximum and minimum temperatures over a given time interval.

This bench test concerns two Java classes called :
Thermometer and **Simulation**.
Code listings of these can be found at the end of the document.

Note: In the following questions, when asked to write or rewrite elements from the classes, you only need to write down the parts required, all unchanged code can be omitted.

Questions

1. The `resetMinMax()` method in the class `Thermometer` makes use of two variables `min` and `max`. What is the scope of each of these variables? [2 marks]

   How does this contrast with the scope of the variable `tempNow` used in the method `newSample()`? [2 marks]

2. Draw an object diagram using UML that depicts the object structure when the `testThermometer` method is running. [6 marks]

   NOTE: ensure your diagram clearly depicts all object instances that have been created and includes the attribute fields.

3. Extend the `Thermometer` class so that it calculates and stores the average temperature since the last reset, include an accessor method to return the current average. [8 marks]

4. Design a new class, `RainGauge`, that can accept regular readings of rainfall to the nearest 0.1 cm and keeps a total value since the last reset.

   Write source code that includes a constructor method to initialise instances of the class and a mutator method that can be used to update the total value. [4 marks]

   Add an accessor method that returns the total rainfall since the last reset. [3 marks]

Java source code for the min-max thermometer.

```
public class Thermometer
{
    private int min ;
    private int max ;
    private int current ;

    public Thermometer()
    {
        resetMinMax();
    }

    public void newSample(int tempNow)
    {
        current = tempNow ;
        if (current < min)
            min = current ;

        if (current > max)
            max = current ;
    }

    public void resetMinMax()
    {
        min = 0 ;
        max = 0 ;
    }

    public String toString()
    {
        return "Current temperature:  " + current + "\n" ;
    }
}
```

SOURCE CODE CONTINUES ON THE NEXT PAGE

```
public class Simulation
{
    private Thermometer minMaxThermometer ;

    public Simulation()
    {
        minMaxThermometer = new Thermometer() ;
    }

    public int takeMeasurement()
    {
        return (int)( Math.random() * 20.0 ) - 10;
    }

    public void testThermometer()
    {
        int current ;

        current = takeMeasurement() ;
        minMaxThermometer.newSample( current ) ;

        current = takeMeasurement() ;
        minMaxThermometer.newSample( current ) ;

        current = takeMeasurement() ;
        minMaxThermometer.newSample( current ) ;

    }
}
```

END OF SOURCE CODE.

B.2 Example Term One Project

IP: Coursework specification - assignment earthquake

Outline

This work is due for electronic submission through duo by 2.00 on Monday 27th January 2014. It contributes 20% to the final mark for this module.

Objectives

The objectives for this coursework are for you to demonstrate that you can

- Define classes, fields, constructors and methods in Java
- Use appropriate types, including collections
- Implement basic algorithms using collections
- Devise appropriate test cases

Scenario: Earthquake monitoring

Earthquakes can have devastating effects when they occur, and although it is known that they are more likely to occur in some places than in others they are almost impossible to predict. In order to help scientists understand earthquakes better, an international monitoring system is in place to record where earthquakes occur and how powerful they are. Earthquake strength is measured by the ‘magnitude moment’: earthquakes of magnitude three or lower usually imperceptible but those with magnitude seven and over can cause serious damage over large areas. National seismological observatories record earthquakes that occur, although different observatories were set up at different times, so the period over which historical data is available varies from place to place.

Problem Specification

When a method to find something is required, the value should be returned as a value from the method, not printed to the terminal.

(25%) Define a Java class Earthquake with appropriate fields, methods and constructors to store and retrieve information about the

- magnitude
- position (latitude and longitude)
- year of the event

(25%) Define a Java class Observatory with appropriate fields and constructors to store and retrieve

- name of the observatory
- the name of the country in which it is located
- the year in which earthquake observations started
- the area covered by the observatory (in square kilometres)
- a list of Earthquake events that it has recorded.

Include methods to find:

- The largest magnitude earthquake recorded by the observatory
- The average earthquake magnitude recorded at the observatory
- The average number of earthquakes recorded per year at the observatory
- A list of all earthquakes recorded at the observatory with a magnitude greater than a given number

(25%) Define a Java class Monitoring which holds information about all observatories. Include methods to find

- the observatory with the largest average earthquake magnitude
- the largest magnitude earthquake ever recorded
- a list of all earthquakes recorded magnitude greater than a given number
- the observatory with the fewest earthquakes per year on average

Testing

(25%) Each class chould include a testing method which

- test each method and checks that it executes it correctly
- reports to the user via the console (i.e. System.out) when tests are passed or failed

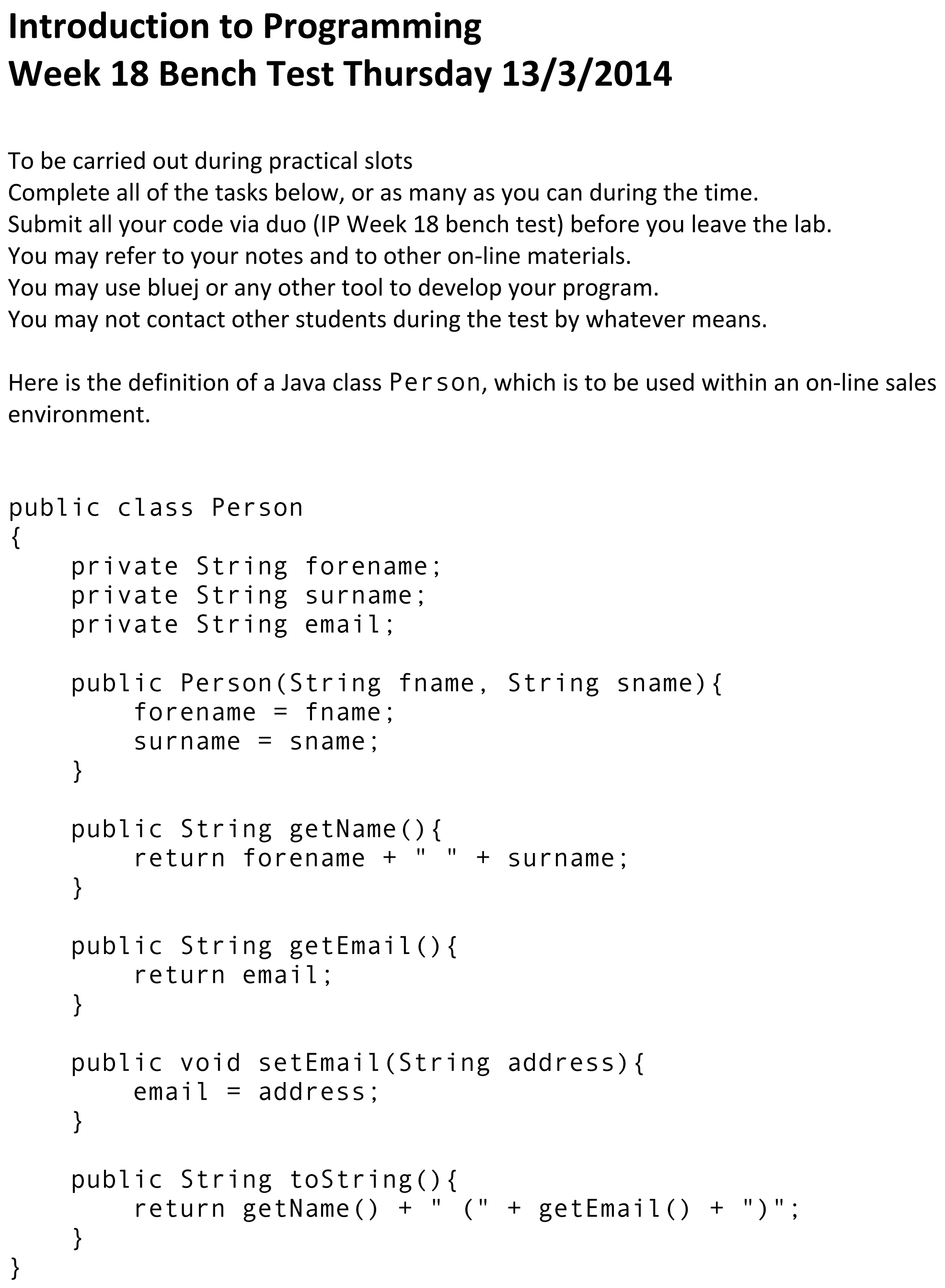
You do not need a separate test for each method: one test may cover two or more methods.

Submission

You need to submit, via duo, the source code (.java files) for your classes.

Steven Bradley 2013-12-03

B.3 Example Term Two Bench Test



Introduction to Programming
Week 18 Bench Test Thursday 13/3/2014

To be carried out during practical slots
Complete all of the tasks below, or as many as you can during the time.
Submit all your code via duo (IP Week 18 bench test) before you leave the lab.
You may refer to your notes and to other on-line materials.
You may use bluej or any other tool to develop your program.
You may not contact other students during the test by whatever means.

Here is the definition of a Java class Person, which is to be used within an on-line sales environment.

```
public class Person
{
    private String forename;
    private String surname;
    private String email;

    public Person(String fname, String sname){
        forename = fname;
        surname = sname;
    }

    public String getName(){
        return forename + " " + surname;
    }

    public String getEmail(){
        return email;
    }

    public void setEmail(String address){
        email = address;
    }

    public String toString(){
        return getName() + " (" + getEmail() + ")";
    }
}
```

Tasks

1. (40%) Define two new subclasses of `Person` called `Staff` and `Customer`. All staff have an alphanumeric staff ID and optionally a member of staff that manages them. All customers have an account number which is a positive whole number. Include for each class
   - A constructor, which takes the forename, surname and staff ID/customer account number as parameters.
   - Get methods for fields.
   - A `toString()` method which includes the name, email address and staff ID/customer account number.

   For the `Staff` class only, define:
   - A method for adding a customer to the client list of a member of staff.
   - A method for assigning a manager to a member of staff.
2. (20%) Each member of staff has a client list of up to 30 customers that they are responsible for looking after. Adapt your `Staff` class, defining field(s) and method(s) for adding a client (i.e. customer) to a member of staff including a check that hey have no more than 30 clients. Define a custom checked exception class which should be used to report if the condition is violated.
3. (20%) Write a main method for the `Staff` class which tests the behaviour of adding a client, including what happens when the client list size is too large and only just small enough.
4. (10%) The customer relationship management (CRM) system requires that everything it considers (staff, customer, products etc) should have its own unique code. Adapt the `Person`, `Staff` and `Customer` classes so that the `Person` class defines a new abstract method `uniqueCode()` which returns unique string for each `Staff` or `Customer` object. For example, if a customer has an account number of 123456 then the unique code for that customer should be "Customer123456".
5. (10%) Adapt the `Person` class so that when a `HashSet` of `Person` objects is made that there are no duplicate people in the set, even if two distinct `Person` objects with the same unique code are added to the set. Remember that `HashSet` uses the `hashCode()` method and the `equals()` method, and that any two objects that are equal must have the same hash code. E.g. after your modifications the following code should print "`Set size 1`"

```
HashSet<Person> s = new HashSet<Person>();
s.add(new Staff("S P", "Bradley", "007"));
s.add(new Staff("Steven", "Bradley", "007"));
System.out.println("Set size " + s.size());
```

When you have finished don't forget to submit all of your code via duo. Feel free to submit partially complete work if you run out of time.

B.4 Example Lab Worksheet

CS1011: Introduction to Programming:
Practical 4: Teaching Week 5

Instructions:
BlueJ is available on the ITS machines in the laboratories.

Level 1: Modelling a Class

Develop on paper a class diagram that represents the problem below. Your model should consider the following points:

What data do you want to store?
What operations do you want to perform?

HiLo Game In this game, the player has to guess a number. For each guess, they are told whether the number is higher or lower than their guess. The number of guesses may be limited, and the range of numbers to guess from could be altered. While the model should lead to a representation of the game that a user can play, you should also consider including a computer player. Think about the different styles of computer player, you could have a good player, or a poor player.

You should propose the following in your model of your game, making sure you understand what each of the parts within the declarations and signatures mean:

Field declarations
Method signatures
Constructor signature(s)

Level 2: Implementing Your Model

Now try to implement the class that you modelled in above. You should do this in a new BlueJ project, so on the BlueJ menu, select *Project* -> *New Project* and save it in your file space.

Create a new class by selecting *New Class* on the left hand side. This will give you a choice of different types of class types that can be implemented in Java, but for the time being we are just concerned with the first type. Give your class a name and click *Ok*.

When you open the editor for the new class you will see a lot of default code that BlueJ automatically includes to illustrate a field, constructor and method, along with the matching *JavaDoc* - comments describing what the class and methods do. While you may want to use these as a guide to get started, the best option is to clear out the contents of the class to start with a clean sheet.

Writing the class: Add in the class declaration with the curly brackets that will contain all the fields and methods for the class Add in the fields with their data types that you defined in your model. Pick the simplest method/constructor first to gradually build up the functionality in your class

Hint: To produce a random number, the following expression can be added into a method in your class:

```
int randomNumber = (int)(Math.random() * upperLimit);
```

This expression will give you a random number from 0 to upperLimit - 1. If you want the range to be from 1 to upperLimit, simply +1 onto the end of the expression.

Note: The method `Math.random()` returns a value of type double between 0 and 1. This is multiplied by the range we want to select from, and then **cast** into the int data type using `(int)`. This removes any numbers after the decimal place so will always round down.

Remember, when comparing two Strings use the boolean expression:

```
string1.equals(string2);
```

Level 3: Extending Your Implementation

Add in a GameManager class that records outcomes from the game you implemented above and can play repeated games (using the computer player). The two coins project used earlier in the course may help here. Think about the different statistics you can produce, e.g. average number of guesses in HiLo (can you beat the computer).

For your implementation, draw an object diagram that depicts the object structure that is created when part way through a game.

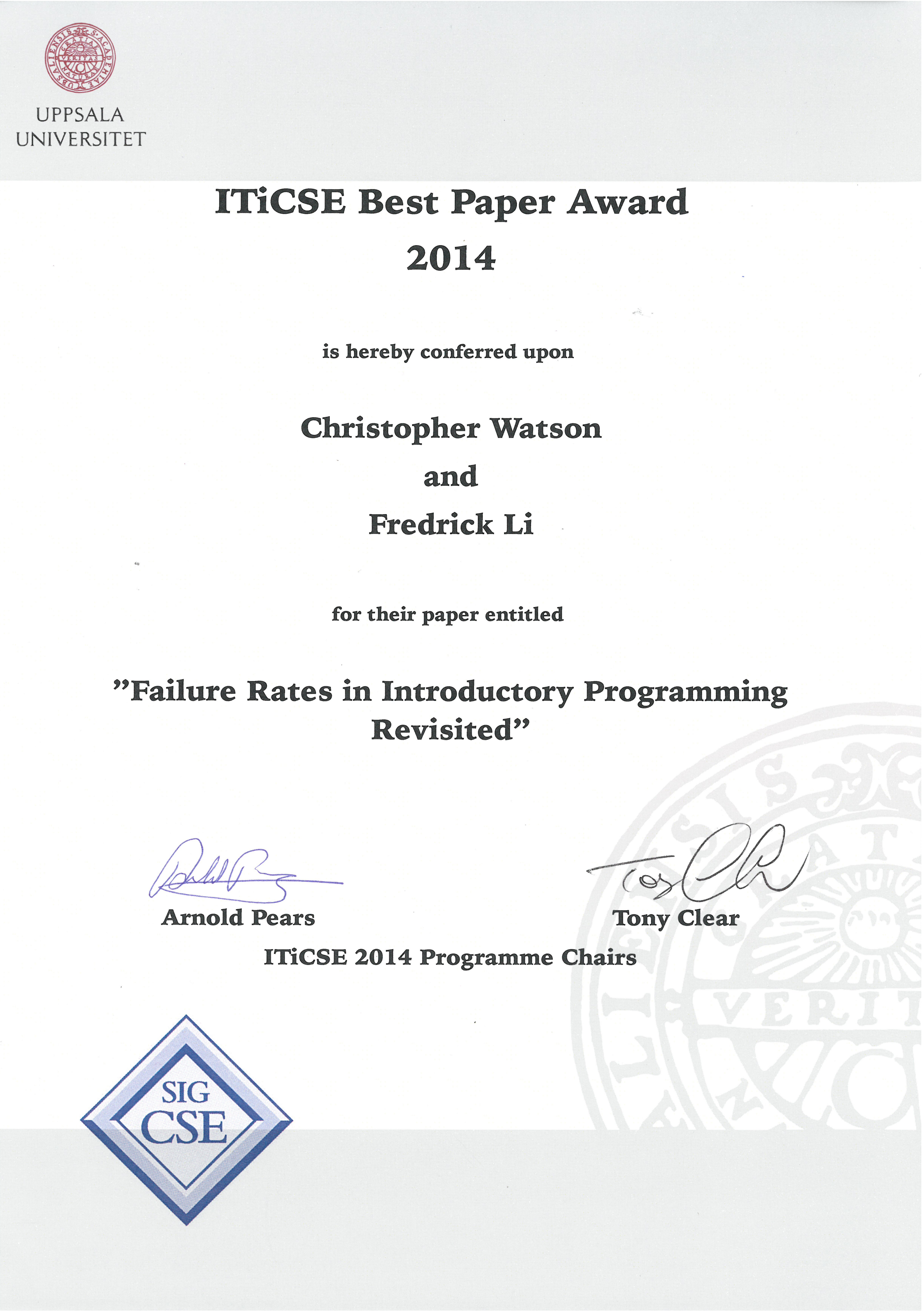
While we have not yet covered looping over code in the lectures, you may find it useful to be able to repeat segments of code using a loop. An example of a for loop that can be included in a method is shown below. Before using it, make sure you understand each part:

```
    for(int i=0; i < upperLimit; i++) {
    // body of for loop
    }
```

Appendix C

Paper Awards

C.1 ACM ITiCSE 2014



UPPSALA
UNIVERSITET

ITiCSE Best Paper Award
2014

is hereby conferred upon

Christopher Watson

and

Fredrick Li

for their paper entitled

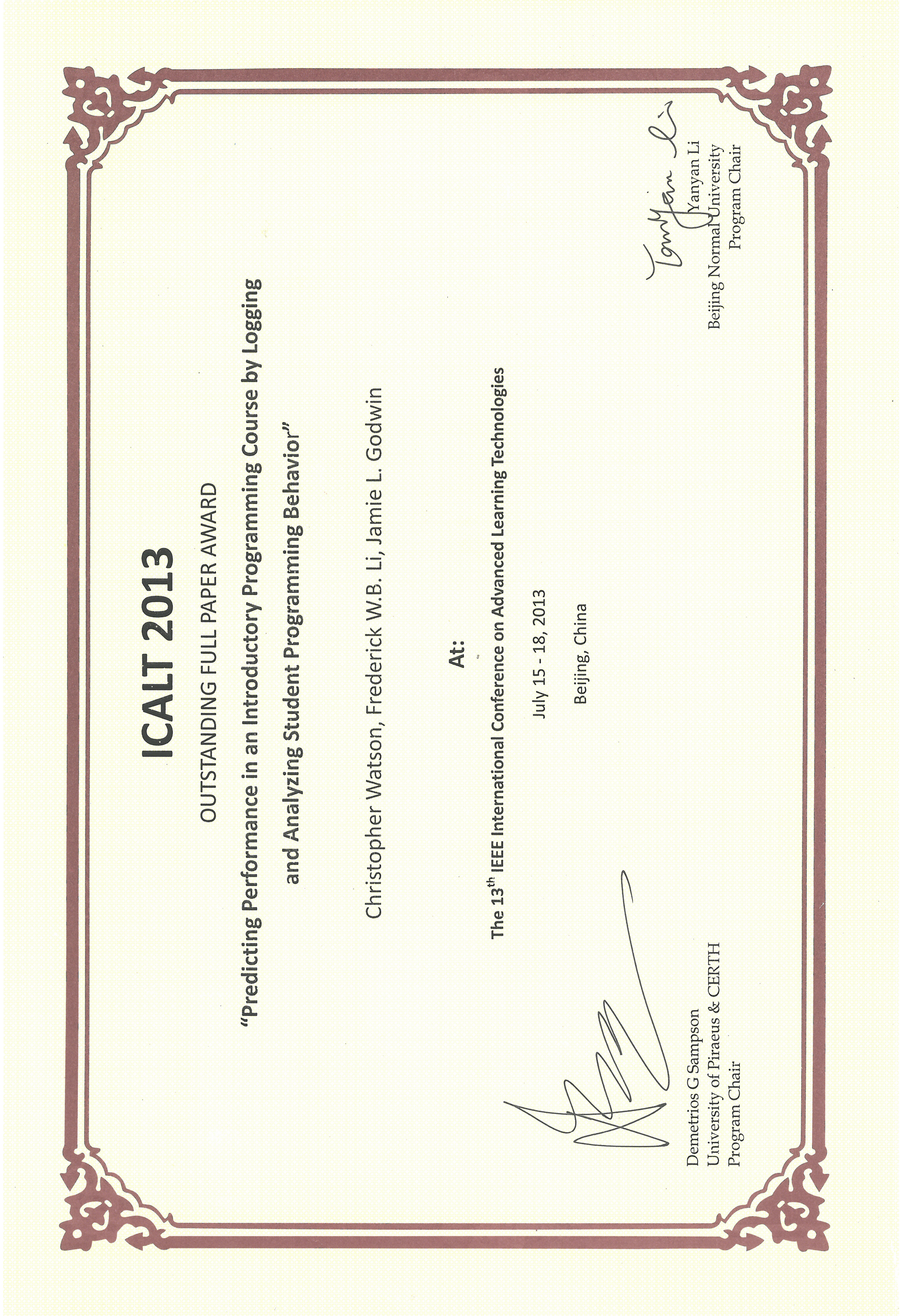
”Failure Rates in Introductory Programming Revisited”

Arnold Pears **Tony Clear**

ITiCSE 2014 Programme Chairs

SIG
CSE

C.2 IEEE ICALT 2013



ICALT 2013

OUTSTANDING FULL PAPER AWARD

"Predicting Performance in an Introductory Programming Course by Logging and Analyzing Student Programming Behavior"

Christopher Watson, Frederick W.B. Li, Jamie L. Godwin

At:

The 13th IEEE International Conference on Advanced Learning Technologies

July 15 - 18, 2013

Beijing, China

Demetrios G Sampson
University of Piraeus & CERTH
Program Chair

Yanyan Li
Beijing Normal University
Program Chair

Appendix D

Publications

The publication which arose from this work can mostly be accessed via Google Scholar:

- https://scholar.google.co.uk/citations?user=yK64l0cAAAAJ

Bibliography

[1] J. Allert. Learning style and factors contributing to success in an introductory computer science course. In *Advanced Learning Technologies, 2004. Proceedings. IEEE International Conference on*, pages 385–389, Aug 2004.

[2] A. Allevato and S. H. Edwards. Discovering patterns in student activity on programming assignments. In *Proceedings of the 2010 American Society for Engineering Education Southeastern Section Conference*, ASEE '10, pages 158–169. ASEE, 2010.

[3] C. A. Alspaugh. Identification of some components of computer programming aptitude. *Journal for Research in Mathematics Education*, 3(2):89–98, 1972.

[4] M. Anderson and C. Gavan. Engaging undergraduate programming students: Experiences using lego mindstorms nxt. In *Proceedings of the 13th Annual Conference on Information Technology Education*, SIGITE '12, pages 139–144, New York, NY, USA, 2012. ACM.

[5] R. M. Arkin and G. M. Maruyama. Attribution, affect, and college exam performance. *Journal of Educational Psychology*, 71(1):85–93, Feb. 1979.

[6] H. S. Austin. Predictors of pascal programming achievement for community college students. *SIGCSE Bull.*, 19(1):161–164, Feb. 1987.

[7] R. J. Barker and E. A. Unger. A predictor for success in an introductory programming class based upon abstract reasoning development. *SIGCSE Bull.*, 15(1):154–158, Feb. 1983.

[8] R. Bauer, W. A. Mehrens, and J. F. Vinsonhaler. Predicting performance in a computer programming course. *Educational and Psychological Measurement*, 28(4):1159–1164, 1968.

[9] R. F. Baumeister, J. D. Campbell, J. I. Krueger, and K. D. Vohs. Does high self-esteem cause better performance, interpersonal success, happiness, or healthier lifestyles? *Psychological Science in the Public Interest*, 4(1):1–44, 2003.

[10] R. F. Baumeister, J. D. Campbell, J. I. Krueger, and K. D. Vohs. Exploding the self-esteem myth. *Scientific American*, 292(1):84–91, 2005.

[11] P. Bayman and R. E. Mayer. A diagnosis of beginning programmers' misconceptions of basic programming statements. *Commun. ACM*, 26(9):677–679, Sept. 1983.

[12] T. Beaubouef and J. Mason. Why the high attrition rate for computer science students: Some thoughts and observations. *SIGCSE Bull.*, 37(2):103–106, June 2005.

[13] J. Bennedsen and M. E. Caspersen. Failure rates in introductory programming. *SIGCSE Bull.*, 39(2):32–36, June 2007.

[14] S. Bergin and R. Reilly. The influence of motivation and comfort-level on learning to program. In *Proceedings of the PPIG*, volume 17, pages 293–304, 2005.

[15] S. Bergin and R. Reilly. Programming: Factors that influence success. *SIGCSE Bull.*, 37(1):411–415, Feb. 2005.

[16] S. Bergin, R. Reilly, and D. Traynor. Examining the role of self-regulated learning on introductory programming performance. In *Proceedings of the First International Workshop on Computing Education Research*, ICER '05, pages 81–86, New York, NY, USA, 2005. ACM.

[17] A. J. Biamonte. Predicting success in programmer training. In *Proceedings of the Second SIGCPR Conference on Computer Personnel Research*, SIGCPR '64, pages 9–12, New York, NY, USA, 1964. ACM.

[18] R. Biddle and E. Tempero. Java pitfalls for beginners. *SIGCSE Bull.*, 30(2):48–52, June 1998.

[19] R. Bornat, S. Dehnadi, and Simon. Mental models, consistency and programming aptitude. In *Proceedings of the Tenth Conference on Australasian Computing Education - Volume 78*, ACE '08, pages 53–61, Darlinghurst, Australia, Australia, 2008. Australian Computer Society, Inc.

[20] N. Bosch, D. Towell, and J. Homer. Characterization of cs1 student programming. In *Proceedings of the 2012 International Conference on Frontiers in Education: Computer Science and Computer Engineering*, FECS '12. World Congress in Computer Science, Computer Engineering, and Applied Computing, 2012.

[21] B. D. Boulay. Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1):57–73, 1986.

[22] J. Boustedt, A. Eckerdal, R. McCartney, J. E. Moström, M. Ratcliffe, K. Sanders, and C. Zander. Threshold concepts in computer science: Do they exist and are they useful? *SIGCSE Bull.*, 39(1):504–508, Mar. 2007.

[23] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543 – 554, 1983.

[24] N. C. Brown and A. Altadmri. Investigating novice programming mistakes: Educator beliefs vs. student data. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, pages 43–50, New York, NY, USA, 2014. ACM.

[25] N. C. C. Brown, M. Kölling, D. McCall, and I. Utting. Blackbox: A large scale repository of novice programmers' activity. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 223–228, New York, NY, USA, 2014. ACM.

[26] K. B. Bruce. Controversy on how to teach cs 1: A discussion on the sigcse-members mailing list. *SIGCSE Bull.*, 37(2):111–117, June 2005.

[27] D. Budgen, A. J. Burn, O. P. Brereton, B. A. Kitchenham, and R. Pretorius. Empirical evidence about the uml: A systematic literature review. *Softw. Pract. Exper.*, 41(4):363–392, Apr. 2011.

[28] P. Byrne and G. Lyons. The effect of student attributes on success in programming. *SIGCSE Bull.*, 33(3):49–52, June 2001.

[29] V. Campbell and M. Johnstone. The significance of learning style with respect to achievement in first year programming students. In *Proceedings of the 2010 21st Australian Software Engineering Conference*, ASWEC '10, pages 165–170, Washington, DC, USA, 2010. IEEE Computer Society.

[30] C. K. Capstick, J. D. Gordon, and A. Salvadori. Predicting performance by university students in introductory computing courses. *SIGCSE Bull.*, 7(3):21–29, Sept. 1975.

[31] N. Card. *Applied Meta-analysis for Social Science Research*. Methodology in the social sciences. Guilford Press, 2012.

[32] M. E. Caspersen and J. Bennedsen. Instructional design of a programming course: A learning theoretic approach. In *Proc. Int. Workshop on Computing Education Research*, ICER '07, pages 111–122, 2007.

[33] M. E. Caspersen, K. D. Larsen, and J. Bennedsen. Mental models and programming aptitude. *SIGCSE Bull.*, 39(3):206–210, June 2007.

[34] A.-M. Cazan. Psychometric properties of cognitive self regulation mslq scales. *Romanian Journal of Experimental Applied Psychology*, 2:49–57, 2011.

[35] R. Cellan-Jones. Bbc news: Programming project comes to primary schools. http://www.bbc.co.uk/news/technology-17740143, 2013. Last Modified: 2013-04-17, Last Accessed: 2013-11-19.

[36] A. T. Chamillard and D. Karolick. Using learning style data in an introductory computer science course. *SIGCSE Bull.*, 31(1):291–295, Mar. 1999.

[37] C.-L. Chen and J. M.-C. Lin. Learning styles and student performance in java programming courses. In *Proceedings of the 2011 International Conference on Frontiers in Education: Computer Science and Computer Engineering*, FECS '11. World Congress in Computer Science, Computer Engineering, and Applied Computing, 2011.

[38] D. Clark, C. MacNish, and G. F. Royle. Java as a teaching language - opportunities, pitfalls and solutions. In *Proceedings of the 3rd Australasian Conference on Computer Science Education*, ACSE '98, pages 173–179, New York, NY, USA, 1998. ACM.

[39] F. Coffield, D. Moseley, E. Hall, K. Ecclestone, et al. Should we be using learning styles?: What research has to say to practice. 2004.

[40] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. L. Erlbaum Associates, 1988.

[41] H. Cooper, L. Hedges, and J. Valentine. *The Handbook of Research Synthesis and Meta-Analysis*. Russell Sage Foundation, 2009.

[42] L. S. Corman. Cognitive style, personality type, and learning ability as factors in predicting the success of the beginning programming student. *SIGCSE Bull.*, 18(4):80–89, Dec. 1986.

[43] S. Coughlan. Curriculum changes to catch up with world's best, 2013. Retrieved Nov. 19, 2013 from *http://bbc.in/1ipPLNF*.

[44] D. Crookes. Educators call for reform in how programming is taught in schools, 2013. Retrieved Nov. 19, 2013 from *http://ind.pn/1evOJgl*.

[45] C. Daly and J. Waldron. A place for c. *Journal of Computing Sciences in Colleges*, 10(2):192–200, Nov. 1994.

[46] C. Daly and J. Waldron. Assessing the assessment of programming ability. *SIGCSE Bull.*, 36(1):210–213, Mar. 2004.

[47] C. Dancey and J. Reidy. *Statistics Without Maths for Psychology: Using SPSS for Windows*. Statistics Without Maths for Psychology: Using SPSS for Windows. Prentice Hall, 2004.

[48] S. Davies, J. A. Polack-Wahl, and K. Anewalt. A snapshot of current practices in teaching the introductory programming sequence. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, pages 625–630, New York, NY, USA, 2011. ACM.

[49] S. Dehnadi and R. Bornat. The camel has two humps (working title). *Middlesex University, UK*, 2006.

[50] P. Denny, A. Luxton-Reilly, and B. Simon. Evaluating a new exam question: Parsons problems. In *Proceedings of the Fourth International Workshop on Computing Education Research*, ICER '08, pages 113–124, New York, NY, USA, 2008. ACM.

[51] P. Denny, A. Luxton-Reilly, and E. Tempero. All syntax errors are not equal. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '12, pages 75–80, New York, NY, USA, 2012. ACM.

[52] S. Dey and L. R. Mand. Effects of mathematics preparation and prior language exposure on perceived performance in introductory computer science courses. *SIGCSE Bull.*, 18(1):144–148, Feb. 1986.

[53] M. do Nascimento, A. Mendona, D. Guerrero, and J. C. A. De Figueiredo. Teaching programming for high school students: A distance education experience. In *Frontiers in Education Conference (FIE), 2010 IEEE*, pages F1J–1–F1J–6, Oct 2010.

[54] T. Dybå and T. Dingsøyr. Empirical studies of agile software development: A systematic review. *Inf. Softw. Technol.*, 50(9-10):833–859, Aug. 2008.

[55] A. Eckerdal, R. McCartney, J. E. Moström, M. Ratcliffe, and C. Zander. Can graduating students design software systems? *SIGCSE Bull.*, 38(1):403–407, Mar. 2006.

[56] A. Eckerdal and M. Thuné. Novice java programmers' conceptions of "object" and "class", and variation theory. *SIGCSE Bull.*, 37(3):89–93, June 2005.

[57] S. H. Edwards, J. Snyder, M. A. Pérez-Quiñones, A. Allevato, D. Kim, and B. Tretola. Comparing effective and ineffective behaviors of student programmers. In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop*, ICER '09, pages 3–14, New York, NY, USA, 2009. ACM.

[58] A. Ehlert and C. Schulte. Comparison of oop first and oop later: First results regarding the role of comfort level. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '10, pages 108–112, New York, NY, USA, 2010. ACM.

[59] J. B. Fenwick, Jr., C. Norris, F. E. Barry, J. Rountree, C. J. Spicer, and S. D. Cheek. Another look at the behaviors of novice programmers. *SIGCSE Bull.*, 41(1):296–300, Mar. 2009.

[60] A. J. Fernndez Leiva and A. C. Civila Salas. Practices of advanced programming: Tradition versus innovation. *Computer Applications in Engineering Education*, 21(2):237–244, 2013.

[61] L. Ficocelli and D. Gregg. Coping with java as the core cs educational language: An evolutionary experience. In *Proceedings of the 16th Western Canadian Conference on Computing Education*, WCCCE '11, pages 38–42, New York, NY, USA, 2011. ACM.

[62] A. E. Fleury. Parameter passing: The rules the students construct. *SIGCSE Bull.*, 23(1):283–286, Mar. 1991.

[63] G. Glass, B. McGaw, and M. Smith. *Meta-analysis in social research.* Sage Library of Social Research. Sage Publications, 1981.

[64] A. Gomes and A. J. Mendes. Learning to program-difficulties and solutions. In *International Conference on Engineering Education*, volume 2007 of *ICEE '07*. iNEER, 2007.

[65] A. Goold and R. Rimmer. Factors affecting performance in first-year computing. *SIGCSE Bull.*, 32(2):39–43, June 2000.

[66] H. Guo. Visual material and learning aids in teaching object oriented programming. In *Proceedings of the 7th Annual ICS HE Academy Conference*, pages 64–69, 2006.

[67] M. Guzdial. Why is it so hard to learn to program? In *Making Software: What Really Works and Why We Believe It*, pages 111–124. O'Reilly Media, 2010.

[68] M. Guzdial and E. Soloway. Teaching the nintendo generation to program. *Commun. ACM*, 45(4):17–21, Apr. 2002.

[69] S. Hadjerrouit. Java as first programming language: A critical evaluation. *SIGCSE Bull.*, 30(2):43–47, June 1998.

[70] D. Hagan and S. Markham. Does it help to have some programming experience before beginning a computing degree program? *SIGCSE Bull.*, 32(3):25–28, July 2000.

[71] L. Hedges and I. Olkin. *Statistical Methods for Meta-analysis*. Academic Press, 1985.

[72] J. Helminen, P. Ihantola, and V. Karavirta. Recording and analyzing in-browser programming sessions. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research*, Koli Calling '13, pages 13–22, New York, NY, USA, 2013. ACM.

[73] J. Helminen, P. Ihantola, V. Karavirta, and L. Malmi. How do students solve parsons programming problems?: An analysis of interaction traces. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research*, ICER '12, pages 119–126, New York, NY, USA, 2012. ACM.

[74] J. W. Henry, M. J. Martinko, and M. A. Pierce. Attributional style as a predictor of success in a first computer science course. *Computers in Human Behavior*, 9(4):341–352, 1993.

[75] Higher Education Funding Council. Data about demand and supply in higher education subjects, 2013. Retrieved Jan. 5, 2013 from *http://bit.ly/19WRiLi*.

[76] A. Hoskey and P. S. M. Maurino. Beyond introductory programming: Success factors for advanced programming. *Information Systems Education Journal*, 9(5):61–70, 2011.

[77] T. R. Hostetler. Predicting student success in an introductory programming course. *SIGCSE Bull.*, 15(3):40–43, Sept. 1983.

[78] J. E. Hunter and F. L. Schmidt. *Methods of Meta-Analysis: Correcting Error and Bias in Research Findings.* SAGE Publications, 2004.

[79] J. Jackson, M. Cobb, and C. Carver. Identifying top java errors for novice programmers. In *Frontiers in Education, 2005. FIE '05. Proceedings 35th Annual Conference*, pages T4C–T4C, Oct 2005.

[80] S. J. Jacobs. Cognitive predictors of success in computer programmer training. In *Proceedings of the Eleventh Annual SIGCPS Computer Personnel Research Conference*, SIGCPR '73, pages 98–110, New York, NY, USA, 1973. ACM.

[81] M. C. Jadud. A first look at novice compilation behaviour using bluej. *Computer Science Education*, 15(1):25–40, 2005.

[82] M. C. Jadud. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the Second International Workshop on Computing Education Research*, ICER '06, pages 73–84, New York, NY, USA, 2006. ACM.

[83] T. Jenkins. On the difficulty of learning to program. In *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences*, volume 4, pages 53–58, 2002.

[84] S. Jones and G. Burnett. Spatial ability and learning to program. *Human Technology: An Interdisciplinary Journal on Humans in ICT Environments*, 4(1):47–61, 2008.

[85] M. Jorgensen and M. Shepperd. A systematic review of software development cost estimation studies. *IEEE Trans. Softw. Eng.*, 33(1):33–53, Jan. 2007.

[86] L. C. Kaczmarczyk, E. R. Petrick, J. P. East, and G. L. Herman. Identifying student misconceptions of programming. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, pages 107–111, New York, NY, USA, 2010. ACM.

[87] S. Katz, D. Allbritton, J. Aronis, C. Wilson, and M. L. Soffa. Gender, achievement, and persistence in an undergraduate computer science program. *SIGMIS Database*, 37(4):42–57, Nov. 2006.

[88] C. M. Kessler and J. R. Anderson. Learning flow of control: Recursive and iterative procedures. *Hum.-Comput. Interact.*, 2(2):135–166, June 1986.

[89] M. Kölling. The problem of teaching object-oriented programming, part 1: Languages. *Journal of Object-oriented programming*, 11(8):8–15, 1999.

[90] A. W. Kosner. Forbes: Can obama convince high schools to teach kids to code? http://www.forbes.com/sites/anthonykosner/2013/02/14/can-obama-convince-high-schools-to-teach-kids-to-code/, 2013. Last Modified: 2013-02-14, Last Accessed: 2013-11-19.

[91] R. J. Koubek, W. K. LeBold, and G. Salvendy. Predicting performance in computer programming courses. *Behaviour & Information Technology*, 4(2):113–129, 1985.

[92] J. Kurhila and A. Vihavainen. Management, structures and tools to scale up personal advising in large programming courses. In *Proceedings of the 2011 Conference on Information Technology Education*, SIGITE '11, pages 3–8, New York, NY, USA, 2011. ACM.

[93] B. L. Kurtz. Investigating the relationship between the development of abstract reasoning and performance in an introductory programming class. *SIGCSE Bull.*, 12(1):110–117, Feb. 1980.

[94] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen. A study of the difficulties of novice programmers. *SIGCSE Bull.*, 37(3):14–18, June 2005.

[95] W. W. F. Lau and A. H. K. Yuen. Promoting conceptual change of learning sorting algorithm through the diagnosis of mental models: The effects of gender and learning styles. *Comput. Educ.*, 54(1):275–288, Jan. 2010.

[96] W. W. F. Lau and A. H. K. Yuen. Modelling programming performance: Beyond the influence of learner characteristics. *Comput. Educ.*, 57(1):1202–1213, Aug. 2011.

[97] R. R. Leeper and J. L. Silver. Predicting success in a first programming course. *SIGCSE Bull.*, 14(1):147–150, Feb. 1982.

[98] S. Letovsky. Cognitive processes in program comprehension. *Journal of Systems and Software*, 7(4):325 – 339, 1987.

[99] S. P. Levy. Computer language usage in cs1: Survey results. *SIGCSE Bull.*, 27(3):21–26, Sept. 1995.

[100] R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä, B. Simon, and L. Thomas. A

multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bull.*, 36(4):119–150, June 2004.

[101] R. Lister and J. Leaney. Introductory programming, criterion-referencing, and bloom. *SIGCSE Bull.*, 35(1):143–147, Jan. 2003.

[102] R. Lister, B. Simon, E. Thompson, J. L. Whalley, and C. Prasad. Not seeing the forest for the trees: Novice programmers and the solo taxonomy. *SIGCSE Bull.*, 38(3):118–122, June 2006.

[103] R. J. Little. A test of missing completely at random for multivariate data with missing values. *Journal of the American Statistical Association*, 83(404):1198–1202, 1988.

[104] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. *J. Syst. Softw.*, 7(4):341–355, Dec. 1987.

[105] M. Lopez, J. Whalley, P. Robbins, and R. Lister. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the Fourth International Workshop on Computing Education Research*, ICER '08, pages 101–112, New York, NY, USA, 2008. ACM.

[106] P. A. Luker. There's more to oop than syntax! *SIGCSE Bull.*, 26(1):56–60, Mar. 1994.

[107] L. Ma, J. Ferguson, M. Roper, and M. Wood. Investigating the viability of mental models held by novice programmers. *SIGCSE Bull.*, 39(1):499–503, Mar. 2007.

[108] R. Mancy and N. Reid. Aspects of cognitive style and programming. In *Proceedings of the Sixteenth Annual Workshop of the Psychology of Programming Interest Group (PPIG)*, pages 1–9, 2004.

[109] L. Mannila and M. de Raadt. An objective comparison of languages for teaching introductory programming. In *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006*, Baltic Sea '06, pages 32–37, New York, NY, USA, 2006. ACM.

[110] S. Maxwell and H. Delaney. *Designing experiments and analyzing data: a model comparison perspective.* Wadsworth Pub. Co., 1990.

[111] R. E. Mayer. The psychology of how novices learn computer programming. *ACM Comput. Surv.*, 13(1):121–141, Mar. 1981.

[112] R. E. Mayer, J. L. Dyck, and W. Vilberg. Learning to program and learning to think: What's the connection? *Commun. ACM*, 29(7):605–610, July 1986.

[113] L. J. Mazlack. Identifying potential to acquire programming skill. *Commun. ACM*, 23(1):14–17, Jan. 1980.

[114] L. McAuley, B. Pham, P. Tugwell, and D. Moher. Does the inclusion of grey literature influence estimates of intervention effectiveness reported in meta-analyses? *The Lancet*, 356(9237):1228–1231, Oct. 2000.

[115] L. P. McCoy and J. K. Burton. The relationship of computer programming and mathematics in secondary students. *Comput. Sch.*, 4(3-4):159–166, Oct. 1987.

[116] L. P. McCoy and M. A. Orey, III. Computer programming and general problem solving by secondary students. *Comput. Sch.*, 4(3-4):151–157, Oct. 1987.

[117] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '01, pages 125–180, New York, NY, USA, 2001. ACM.

[118] C. McDowell, B. Hanks, and L. Werner. Experimenting with pair programming in the classroom. *SIGCSE Bull.*, 35(3):60–64, June 2003.

[119] C. McDowell, L. Werner, H. E. Bullock, and J. Fernald. Pair programming improves student retention, confidence, and program quality. *Commun. ACM*, 49(8):90–95, Aug. 2006.

[120] A. J. Mendes, A. Gomes, A. Cardoso, and L. Paquete. Increasing student commitment in introductory programming learning. In *Proceedings of the 2012 IEEE Frontiers in Education Conference (FIE)*, FIE '12, pages 1–6, Washington, DC, USA, 2012. IEEE Computer Society.

[121] J. Meyer and R. Land. *Threshold concepts and troublesome knowledge: linkages to ways of thinking and practising within the disciplines.* University of Edinburgh, 2003.

[122] J. H. Meyer and R. Land. Threshold concepts and troublesome knowledge (2): Epistemological considerations and a conceptual framework for teaching and learning. *Higher Education*, 49(3):373–388, 2005.

[123] K. Mierle, K. Laven, S. Roweis, and G. Wilson. Mining student cvs repositories for performance indicators. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005.

[124] J. Milic. Predictors of success in solving programming tasks. *The Teaching of Mathematics*, 12(1):25–31, 2009.

[125] I. Miliszewska and G. Tan. Befriending computer programming: A proposed approach to teaching introductory programming. *Informing Science: International Journal of an Emerging Transdiscipline*, 4(1):277–289, 2007.

[126] I. Milne and G. Rowe. Difficulties in learning and teaching programming & views of students and tutors. *Education and Information Technologies*, 7(1):55–66, Mar. 2002.

[127] R. P. Mody. C in education and software engineering. *SIGCSE Bull.*, 23(3):45–56, Sept. 1991.

[128] J. E. Moström, J. Boustedt, A. Eckerdal, R. McCartney, K. Sanders, L. Thomas, and C. Zander. Concrete examples of abstraction as manifested in students' transformative experiences. In *Proceedings of the Fourth International Workshop on Computing Education Research*, ICER '08, pages 125–136, New York, NY, USA, 2008. ACM.

[129] J. E. Moström, J. Boustedt, A. Eckerdal, R. McCartney, K. Sanders, L. Thomas, and C. Zander. Computer science student transformations: Changes and causes. *SIGCSE Bull.*, 41(3):181–185, July 2009.

[130] C. Murphy, G. Kaiser, K. Loveland, and S. Hasan. Retina: Helping students and instructors based on observed programming activities. *SIGCSE Bull.*, 41(1):178–182, Mar. 2009.

[131] J. J. Mussio and M. W. Wahlstrom. Predicting performance of programmer trainees in a post-high school setting. In *Proceedings of the Ninth Annual SIGCPR Conference*, SIGCPR '71, pages 26–46, New York, NY, USA, 1971. ACM.

[132] P. R. Newsted. Grade and ability predictions in an introductory programming course. *SIGCSE Bull.*, 7(2):87–91, June 1975.

[133] C. Norris, F. Barry, J. B. Fenwick Jr., K. Reid, and J. Rountree. Clockit: Collecting quantitative data on how beginning software developers really work. *SIGCSE Bull.*, 40(3):37–41, June 2008.

[134] R. H. Nowaczyk. The relationship of problem-solving ability and course performance among novice programmers. *Int. J. Man-Mach. Stud.*, 21(2):149–160, Nov. 1984.

[135] R. D. Pea. Language-independent conceptual b̈ugsïn novice programming. *Journal of Educational Computing Research*, 2:25–36, 1986.

[136] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson. A survey of literature on the teaching of introductory programming. *SIGCSE Bull.*, 39(4):204–223, Dec. 2007.

[137] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19(3):295 – 341, 1987.

[138] D. N. Perkins, C. Hancock, R. Hobbs, F. Martin, and R. Simmons. Conditions of learning in novice programmers. *Journal of Educational Computing Research*, 2(1):37–55, 1986.

[139] D. N. Perkins and R. Simmons. Patterns of misunderstanding: An integrative model for science, math, and programming. *Review of Educational Research*, 58(3):pp. 303–326, 1988.

[140] A. Petersen, M. Craig, and D. Zingaro. Reviewing cs1 exam question content. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, pages 631–636, New York, NY, USA, 2011. ACM.

[141] C. Piech, M. Sahami, D. Koller, S. Cooper, and P. Blikstein. Modeling how students learn to program. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 153–160, New York, NY, USA, 2012. ACM.

[142] N. Pillay and V. R. Jugoo. An investigation into student characteristics affecting novice programming performance. *SIGCSE Bull.*, 37(4):107–110, Dec. 2005.

[143] P. Pintrich and A. Zusho. Student motivation and self-regulated learning in the college classroom. In R. Perry and J. Smart, editors, *The Scholarship of Teaching and Learning in Higher Education: An Evidence-Based Perspective*, pages 731–810. Springer Netherlands, 2007.

[144] M. Pizka and M. Broy. Success and failure of 1000 first semester cs students. In *IASTED International Conference on Computers and Advanced Technology in Education including the IASTED International Symposium on Web-Based Education*, pages 771–776, 2003.

[145] L. Porter, M. Guzdial, C. McDowell, and B. Simon. Success in introductory programming: What works? *Commun. ACM*, 56(8):34–36, Aug. 2013.

[146] A. Ralston, E. D. Reilly, and D. Hemmendinger, editors. *Encyclopedia of Computer Science*. John Wiley and Sons Ltd., Chichester, UK, 4th edition, 2003.

[147] V. Ramalingam, D. LaBelle, and S. Wiedenbeck. Self-efficacy and mental models in learning to program. *SIGCSE Bull.*, 36(3):171–175, June 2004.

[148] S. Ranjeeth. The impact of learning styles on the acquisition of computer programming proficiency. *Journ. Alternation*, 18(1):336–535, 2011.

[149] S. W. Raudenbush. Analyzing effect sizes: Random-effects models. *The Handbook of Research Synthesis and Meta-Analysis*, 2:295–316, 2009.

[150] C. Riener and D. Willingham. The myth of learning styles. *Change: The magazine of higher learning*, 42(5):32–35, 2010.

[151] R. S. Rist. Knowledge creation and retrieval in program design: A comparison of novice and intermediate student programmers. *Hum.-Comput. Interact.*, 6(1):1–46, Mar. 1991.

[152] A. Robins. Learning edge momentum: a new account of outcomes in cs1. *Computer Science Education*, 20(1):37–71, 2010.

[153] A. Robins, J. Rountree, and N. Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137–172, 2003.

[154] M. M. T. Rodrigo and R. S. Baker. Coarse-grained detection of student frustration in an introductory programming course. In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop*, ICER '09, pages 75–80, New York, NY, USA, 2009. ACM.

[155] M. M. T. Rodrigo, R. S. Baker, M. C. Jadud, A. C. M. Amarra, T. Dy, M. B. V. Espejo-Lahoz, S. A. L. Lim, S. A. Pascua, J. O. Sugay, and E. S. Tabanao. Affective and behavioral predictors of novice programmer achievement. *SIGCSE Bull.*, 41(3):156–160, July 2009.

[156] R. Rosenthal. *Meta-Analytic Procedures for Social Research*. Applied Social Research Methods. SAGE Publications, 1991.

[157] R. Rosenthal and D. B. Rubin. Meta-analytic procedures for combining studies with multiple effect sizes. *Psychological bulletin*, 99(3):400–406, May 1986.

[158] J. Rountree and N. Rountree. Issues regarding threshold concepts in computer science. In *Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95*, ACE '09, pages 139–146, Darlinghurst, Australia, Australia, 2009. Australian Computer Society, Inc.

[159] M. Sall. *Variables Predicting Achievement in Introductory Computer Science Courses [microform]*. Thesis: M.A. Thesis (M.A.)–University of British Columbia, 1989.

[160] R. Samurcay. The concept of variable in programming: Its meaning and use in problem-solving by novice programmers. *Studying the novice programmer*, 9:161–178, 1989.

[161] V. L. Sauter. Predicting computer programming skill. *Comput. Educ.*, 10(2):299–302, Apr. 1986.

[162] D. A. Scanlan. Variables and cognitive factors predicting intermediate-level programming success: A preliminary study. *Computer Science Education*, 1(4):347–354, 1990.

[163] C. Schulte, T. Clear, A. Taherkhani, T. Busjahn, and J. H. Paterson. An introduction to program comprehension for computer science educators. In *Proceedings of the 2010 ITiCSE Working Group Reports*, ITiCSE-WGR '10, pages 65–86, New York, NY, USA, 2010. ACM.

[164] J. Sheard, A. Carbone, S. Markham, A. J. Hurst, D. Casey, and C. Avram. Performance and progression of first year ict students. In *Proceedings of the Tenth Conference on Australasian Computing Education - Volume 78*, ACE '08, pages 119–127, Darlinghurst, Australia, Australia, 2008. Australian Computer Society, Inc.

[165] S. Shuhidan, M. Hamilton, and D. D'Souza. Instructor perspectives of multiple-choice questions in summative assessment for novice programmers. *Computer Science Education*, 20(3):229–259, 2010.

[166] V. J. Shute. Who is likely to acquire programming skills? *Journal of Educational Computing Research*, 7(1):1–24, 1991.

[167] V. J. Shute. Focus on formative feedback. *Review of Educational Research*, 78(1):153–189, 2008.

[168] Simon, Q. Cutts, S. Fincher, P. Haden, A. Robins, K. Sutton, B. Baker, I. Box, M. de Raadt, J. Hamer, M. Hamilton, R. Lister, M. Petre, D. Tolhurst, and J. Tutty. The ability to articulate strategy as a predictor of programming skill. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52*, ACE '06, pages 181–188, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[169] Simon, S. Fincher, A. Robins, B. Baker, I. Box, Q. Cutts, M. de Raadt, P. Haden, J. Hamer, M. Hamilton, R. Lister, M. Petre, K. Sutton, D. Tolhurst, and J. Tutty. Predictors of success in a first programming course. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52*, ACE '06, pages 189–196, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[170] R. H. Sloan and P. Troy. Cs 0.5: A better approach to introductory computer science for majors. *SIGCSE Bull.*, 40(1):271–275, Mar. 2008.

[171] E. Soloway, J. Bonar, and K. Ehrlich. Cognitive strategies and looping constructs: An empirical study. *Commun. ACM*, 26(11):853–860, Nov. 1983.

[172] J. Sorva. Reflections on threshold concepts in computer programming and beyond. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 21–30, New York, NY, USA, 2010. ACM.

[173] J. Spacco, D. Fossati, J. Stamper, and K. Rivers. Towards improving programming habits to create better computer science course outcomes. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '13, pages 243–248, New York, NY, USA, 2013. ACM.

[174] J. Spacco, J. Strecker, D. Hovemeyer, and W. Pugh. Software repository mining with marmoset: An automated programming project snapshot and testing system. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005.

[175] T. D. Spector and S. G. Thompson. The potential and limitations of meta-analysis. *Journal of Epidemiology and Community Health*, 45(2):89–92, June 1991.

[176] J. C. Spohrer and E. Soloway. Novice mistakes: Are the folk wisdoms correct? *Commun. ACM*, 29(7):624–632, July 1986.

[177] J. G. Spohrer and E. Soloway. Analyzing the high frequency bugs in novice programs. In *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*, pages 230–251, Norwood, NJ, USA, 1986. Ablex Publishing Corp.

[178] M. V. Stein. Mathematical preparation as a basis for success in cs-ii. *J. Comput. Sci. Coll.*, 17(4):28–38, Mar. 2002.

[179] M. Storey. Theories, methods and tools in program comprehension: past, present and future. In *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, pages 181–191, May 2005.

[180] A. Subramanian and K. Joshi. Computer aptitude tests as predictors of novice computer programmer performance. *Journal of Information Technology Management*, 7(1-2):31–41, 1996.

[181] E. S. Tabanao, M. M. T. Rodrigo, and M. C. Jadud. Identifying at-risk novice java programmers through the analysis of online protocols. In *Philippine Computing Science Congress*, 2008.

[182] E. S. Tabanao, M. M. T. Rodrigo, and M. C. Jadud. Predicting at-risk novice java programmers through the analysis of online protocols. In *Proceedings of the Seventh International Workshop on Computing Education Research*, ICER '11, pages 85–92, New York, NY, USA, 2011. ACM.

[183] M. M. Teague. *Pedagogy of introductory computer programming: a people-first approach.* PhD thesis, Queensland University of Technology, 2011.

[184] D. Tolhurst, B. Baker, J. Hamer, I. Box, R. Lister, Q. Cutts, M. Petre, M. de Raadt, A. Robins, S. Fincher, Simon, P. Haden, K. Sutton, M. Hamilton, and J. Tutty. Do map drawing styles of novice programmers predict success in programming?: A multi-national, multi-institutional study. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52*, ACE '06, pages 213–222, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[185] V. J. Traver. On compiler error messages: what they say and what they mean. *Advances in Human-Computer Interaction*, 2010, 2010.

[186] US Bureau of Labor Statistics. Computer and information technology occupations, 2013. Retrieved Nov. 19, 2013 from *http://1.usa.gov/1a63neI*.

[187] I. Utting, N. Brown, M. Kölling, D. McCall, and P. Stevens. Web-scale data gathering with bluej. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research*, ICER '12, pages 1–4, New York, NY, USA, 2012. ACM.

[188] A. Venables, G. Tan, and R. Lister. A closer look at tracing, explaining and code writing skills in the novice programmer. In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop*, ICER '09, pages 117–128, New York, NY, USA, 2009. ACM.

[189] P. Ventura and B. Ramamurthy. Wanted: Cs1 students. no experience required. *SIGCSE Bull.*, 36(1):240–244, Mar. 2004.

[190] P. R. Ventura. Identifying predictors of success for an objects-first cs1. *Computer Science Education*, 15(3):223–243, 2005.

[191] W. Viechtbauer. Conducting meta-analyses in r with the metafor package. *Journal of Statistical Software*, 36(3):1–48, 2010.

[192] A. Vihavainen. Predicting students' performance in an introductory programming course using data from students' own programming process. In *Advanced Learning Technologies (ICALT), 2013 IEEE 13th International Conference on*, pages 498–499, July 2013.

[193] A. Vihavainen, J. Airaksinen, and C. Watson. A systematic review of approaches for teaching introductory programming and their influence on success. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, pages 19–26, New York, NY, USA, 2014. ACM.

[194] A. Vihavainen, M. Luukkainen, and J. Kurhila. Using students' programming behavior to predict success in an introductory mathematics course. In *Proceedings of The 2013 International Conference on Educational Data Mining*, EDM '13, pages 300–303. International Educational Data Mining Society, 2013.

[195] X. Wang. A practice of bilingual teaching in a big-scale class: How we teach programming bilingually in a big class. In *Computer Science Education (ICCSE), 2012 7th International Conference on*, pages 1727–1731, July 2012.

[196] C. Watson. *An Exploration of Traditional and Data Driven Predictors of Programming Performance.* PhD thesis, Durham University, 2014.

[197] C. Watson and F. W. Li. Failure rates in introductory programming revisited. In *Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education*, ITiCSE '14, pages 39–44, New York, NY, USA, 2014. ACM.

[198] C. Watson, F. W. Li, and J. L. Godwin. No tests required: Comparing traditional and dynamic predictors of programming success. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 469–474, New York, NY, USA, 2014. ACM.

[199] C. Watson, F. W. B. Li, and J. L. Godwin. Bluefix: Using crowd-sourced feedback to support programming students in error diagnosis and repair. In *Proceedings of the 11th International Conference on Advances in Web-Based Learning*, ICWL'12, pages 228–239, Berlin, Heidelberg, 2012. Springer-Verlag.

[200] C. Watson, F. W. B. Li, and J. L. Godwin. Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *Proceedings of the 2013 IEEE 13th International Conference on Advanced Learning Technologies*, ICALT '13, pages 319–323, Washington, DC, USA, 2013. IEEE Computer Society.

[201] B. Weiner. An attributional theory of achievement motivation and emotion. *Psychological Review*, 92(4):548–573, Oct. 1985.

[202] L. H. Werth. Predicting student performance in a beginning computer science class. *SIGCSE Bull.*, 18(1):138–143, Feb. 1986.

[203] K. L. Whipkey. Identifying predictors of programming skill. *SIGCSE Bull.*, 16(4):36–42, Dec. 1984.

[204] G. White and M. Sivitanides. An empirical investigation of the relationship between success in mathematics and visual programming courses. *Journal of Information Systems Education*, 14(4):409–416, 2003.

[205] S. Wiedenbeck. Factors affecting the success of non-majors in learning to program. In *Proceedings of the First International Workshop on Computing Education Research*, ICER '05, pages 13–24, New York, NY, USA, 2005. ACM.

[206] S. Wiedenbeck, V. Ramalingam, S. Sarasamma, and C. Corritore. A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers*, 11(3):255 – 282, 1999.

[207] S. Wiedenbeck, X. Sun, and T. Chintakovid. Antecedents to end users' success in learning to program in an introductory programming course. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, VLHCC '07, pages 69–72, Washington, DC, USA, 2007. IEEE Computer Society.

[208] R. Wilcox. *Introduction to Robust Estimation and Hypothesis Testing*. Statistical modeling and decision science. Elsevier/Academic Press, 2005.

[209] T. C. Willoughby. Computer programmer aptitude battery: Validation study. *SIGCPR Comput. Pers.*, 2(3):6–9, Sept. 1971.

[210] B. C. Wilson and S. Shrock. Contributing to success in an introductory computer science course: A study of twelve factors. *SIGCSE Bull.*, 33(1):184–188, Feb. 2001.

[211] T. D. Wilson, M. Damiani, and N. Shelton. *Improving the Academic Performance of College Students with Brief Attributional Interventions*. Educational Psychology. Academic Press, San Diego, 2002.

[212] L. E. Winslow. Programming pedagogy&mdash;a psychological overview. *SIGCSE Bull.*, 28(3):17–22, Sept. 1996.

[213] J. M. Wolfe. Wolfe programming aptitude test (school edition). In *Proceedings of the Ninth Annual SIGCPR Conference*, SIGCPR '71, pages 180–185, New York, NY, USA, 1971. ACM.

[214] Yulia and R. Adipranata. Teaching object oriented programming course using cooperative learning method based on game design and visual object oriented environment. In *Education Technology and Computer (ICETC), 2010 2nd International Conference on*, volume 2, pages V2–355–V2–359, June 2010.

[215] C. Zander, J. Boustedt, R. McCartney, J. E. Moström, K. Sanders, and L. Thomas. Student transformations: Are they computer scientists yet? In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop*, ICER '09, pages 129–140, New York, NY, USA, 2009. ACM.

[216] I. A. Zualkernan, J. Allert, and G. Z. Qadah. Learning styles of computer programming students: A middle eastern and american comparison. *IEEE Trans. on Educ.*, 49(4):443–450, Nov. 2006.

[217] S. Zweben. Computing degree and enrollment trends. *Computing Research Association*, 2011.